THE ILSP BEHAVIORAL DESCRIPTION LANGUAGE AND ITS GRAPH REPRESENTATION FOR BEHAVIORAL SYNTHESIS

Masayasu Odani
Sun Young Hwang
Tom Blank
Tom Rokicki

Technical Report: CSL-TR-88-350

March 1988

This work was supported by Defense Advanced Research Projects Agency under Contract No. MDA 903-83-C-0335, and partly by Toshiba Corporation.

THE ILSP BEHAVIORAL DESCRIPTION LANGUAGE AND ITS GRAPH REPRESENTATION FOR BEHAVIORAL SYNTHESIS

Masayasu Odani, Sun Young Hwang, Tom Blank, and Tom Rokicki

Technical Report: CSL-TR-88-350

March 1988

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 943054055

Abstract

This report describes the ILSP behavioral description language and its internal representation employed in the *Hermod* behavioral synthesis system. Using combined control/data flow graph (C/DFG) as an intermediate representation, Hermod generates hardware modules and their interconnection from behavioral descriptions. Hex-mod is included in an integrated environment for hardware simulation and synthesis system under development at Stanford University. The functional models written in ILSP can be simulated on the THOR logic/functional/behavioral simulator without translation. After proper verification of its behavior, an ILSP model can be input to the synthesizer for compilation into an RT-level description.

This report consists of two parts: the specification of the ILSP language and its graph representation.

Key Words and Phrases: behavioral description language, behavioral synthesis, structural synthesis, control and data flow graph, register-transfer level description, functional simulation.

The ILSP Behavioral Description Language

1. Overview

This document provides the features and syntax of the ILSP(Input Language for Synthesis Program), which is used to describe the behavior of functional modules. Compared to the ISPS (Instruction Set Processor Specifications) which describes hardware at register-transfer level, the ILSP description is purely procedural or behavioral. Her-mod is included in an integrated environment for hardware simulation and synthesis system under development at Stanford University. The functional models written in ILSP can be simulated on the THOR logic/functional/behavioral simulator without translation. After proper verification of its behavior, an ILSP model can be pipelined to the synthesizer for compilation into an RTL-level description.

Based on the C-language, ILSP has conditional ***(if-then-else*** and ***switch),*** and loop ***(while-*** and do-loop) control constructs. It also allows explicit specification of the actual hardware interface to the outside world. Many features of the C language considered redundant or unnecessary for behavioral representation of a hardware module are omitted in ILSP. For instance, only integer type variables are supported, and parameter passing is handled through interface declarations in the ILSP procedure declaration section.

The overall features of ILSP are presented in the first three sections, followed by the ILSP syntax. The differences from the C language are presented in each section.

2. Lexical Conventions

There are five classes of tokens: identifiers, keywords, constants, operations, and delimiters (like '(' or ' {'). Blanks, tabs, newlines, and comments are used to separate tokens.

2.1 Comments

The characters '/*' introduce a comment, which ends with the characters '*/'.

2.2 Identifiers

An identifier consists of a sequence of letters and digits. The identifier must start with a letter. The underscore '_' counts as a letter. Upper and lower case letters are considered different. There are no restrictions on the length of the identifier.

2.3 Keywords - Reserved Words

The following identifiers are reserved for use as keywords. The keyword ***extern*** is not supported for global variables in ILSP; instead, they should be explicitly declared in the interface declaration sections.

int	***GRP***	SIG	***BUS***
IN_LIST	*OUT_LIST*	*ST_LIST*	*ENDLIST*
if	***else***	***switch***	***case***
default	***break***	*do*	***while***
ONE	***ZERO***	***FLOAT***	***UNDEF***
MODEL	***return***	*LSBO*	***MSBO***

2.4 Constants

There are two types of constants: integer constants and reserved constants. An integer constant consists of a sequence of digits. Only decimal expressions are allowed.

The reserved constants ***are ONE, ZERO, FLOAT,*** and ***UNDEF. They are*** used to represent the value of a signal line in the THOR simulation system. The objects declared as SIG or ***GRP*** in the interface declaration sections are allowed to have one of these values.

3. Attributes of an Object

ILSP supports three fundamental types of objects. The objects declared as integers *(int)* are local variables to the procedure in which they are declared. The objects declared as signals (SIG) are one-bit-signal variables and those declared as groups ***(GRP*** or ***BUS)*** are multiple-bit-signal variables. The bit width and bit order are specified for GRP-type objects (***LSB0* -** lowest indexed bit is the least significant bit or ***MSBO* -** lowest indexed bit is the most significant bit).

The SIG- and GRP-type objects have one of three types of io-attributes: input, output, and state. The value of the input variable (declared in **IN-LIST** section) is taken from the outside of the procedure, and that of the output variable (declared in **OUT-LIST** section) is brought to the outside of the procedure. The state variables (declared in **ST_LIST** section) correspond to the static integers of a C procedure. They are local to the procedure, but retain their values upon reentry to the procedure even after the control leaves from the procedure.

The SIG- and GRP-type objects are the abstract representations of ***registers*** in the hardware representation. An integer object may be realized by a register in the hardware, or just as a wire for signal flow depending on its usage and lifetime.

4. Behavioral Model

Behavioral models consist of a procedure declaration section and a procedural body. The constructs are almost the same as in the C language. The differences are:

1. The parameter passing mechanism in a procedure call is handled through the interface mechanism supported in ILSP. That is, the input and output parameters are declared in the interface declaration sections **(IN-LIST** and **OUT-LIST** declarations) unlike C procedures.
2. A return statement is allowed only at the end of a procedure. No expressions are allowed after a return statement. Instead, a procedure can return values by assigning values to the variables declared in the **OUT-LIST** section.

The syntax for a behavioral model or procedure is presented as follows:

model:
        procedure-declaration procedure-body
        ;

procedure-declaration:
        procedure-identifier '(' ')'
        | MODEL '(' *procedure-identifier* ')'
        ;

procedure-identifier:
        identifier

procedure-body:
        '{' *declaration statement-list return* ';' '}'

5. Declarations

Identifier declarations specify their attributes including the interface signals to the outside of the procedure, the state and local variables. For a group of signals, if the bit order is not specified, the signal with the lowest index is considered as the least significant bit (*LSB0*).

declaration:
        signal-declaration integer-declaration
        ;

signal-declaration:
        input-declaration output-declaration state-declaration
        ;

input-declaration:
        IN-LIST *signal-list* ENDLIST ';'
        ;

output-declaration:
        OUT-LIST *signal-list* ENDLIST ';'

state-declaration:
        /* *empty* */
        | ST-LIST *signal-list* ENDLIST ';'

```
signal-list:
        signal-list  signal
      I signal


signal:
        SIG '(' identifier ')' ';'
      I GRP '(' group-signal-identifier ',' bit-width bit-order ')' ';'


group-signal-identifier:
        identifier
      ,

bit- width:
        constant


bit-order:
        /* empty */
      I ',' LSBO
      I ',' MSBO
      ,

integer-declaration:
        /* empty */
      I int identifier-list ';'
      ;

identifier-list:
        identifier
      I identifier-list ',' identifier
      ,
```

6. Statements

Most statements in the C language are supported except that

1. ILSP allows procedure calls that return more than one value. ***"(receiver-list) = procedure-call-expression;"*** form is used for procedure calls that return multiple outputs and distributes the values to the variables and group signals in ***the receiver-list.***
2. A ***break*** statement is allowed only in a switch statement to eliminate abrupt loop exits.

statement-list:
    statement
  | *statement-list statement*

statement:
    '{' *statement-list* '}'
  | *assignment* ';'
  | if '(' *expression* ')' *statement*
  | if '(' *expression* ')' *statement else statement*
  | while '(' *expression* ')' *statement*
  | do *statement* while '(' *expression* ')' ';'
  | switch '(' *expression* ')' '{' *case-statement-list default-statement* '}'

assignment:
    increment-expression
  | *receiver* '=' *expression*
  | '(' *receiver-list* ')' '=' *procedure-call-expression*

receiver-list:
    receiver
  | *receiver-list* ',' *receiver*
  ;

receiver:
    identifier
  | *group-signal-identifier* '[' *range* ']'
  ;

case-statement-list:
    case-statement
  | *case-statement-list case-statement*
  ,

case-statement:
    case *constant* ':'
  | case *constant* ':' break ';'
  | *case constant* ':' *statement-list* break ';'
  ,

default-statement:
    default ':' break ';'
  | default ':' *statement-list* break ';'
  ;

7. Expressions

Expressions used in ILSP are again based on the C language. General data structures including pointers and arrays are not allowed. Array structure is allowed only to represent a group of signals. The differences from the C language in expressions are summarized as follows:

1. ILSP does not allow integer arrays, structures, or pointers. An exception is that a pointer is passed to a subprocedure as an argument for a GRP-type object in a procedure call.
2. ILSP uses the array structures for group signals to specify bit position. A GRP-type object followed by a range in square brackets specifies a portion of group signals. The expression **x[]** implies the entire signal group of x will be treated as an integer. The expression **x[3]** represents the signal value of the third bit of x, and **x[7:4]** means that the partial signal group between the seventh bit and fourth bit of x is treated as an integer.
3. ILSP allows increment expressions (++ and --) for integer variables only.

```
expression:
        primary-expression
      I mar-y-expression
      I increment-expression
      | binary-expression
      ;
```

7.1 Primary Expressions

```
primary-expression:
        identifier
      Iconstant
      I reserved-constant
      I group-signal-identifier '[' range ']'
      I '(' expression ')'
      I procedure-call-expression


procedure-call-expression:
        procedure-identifier '(' expression-list ')'
      ;

expression- list:
        expression
      I expression-list ',' expression
      ,
range:
        /* empty */
      | constant
      | constant ':' constant
```

```
reserved-constant:
        ZERO
      | ONE
      | UNDEF
      | FLOAT
```

7.2 Unary and Increment Operators

```
unary-expression:
        '-' expression
      | '!' expression
      | '~' expression
```

```
increment-expression:
        '++' identifier
      | '--' identifier
      | identifier '++'
      | identifier '--'
```

7.3 Binary Operators

```
binary-expression:
        expression '&&' expression
      | expression '||' expression
      | expression '&' expression
      | expression '|' expression
      | expression '^' expression
      | expression '>>' expression
      | expression '<<' expression
      | expression '==' expression
      | expression '!=' expression
      | expression '<' expression
      | expression '<=' expression
      | expression '>' expression
      | expression '>=' expression
      | expression '+' expression
      | expression '-' expression
      | expression '*' expression
      | expression '/' expression
```

| *expression* '%' *expression*

8. Examples

The following ILSP procedure describes the functional module that takes two input groups of signals, **in** and **enable,** and calculates the factorial of the value of **in,** setting the signal group **out** and the signal line **valid.**

```
sum()
{
    IN-LIST                     /* declare input ports */
        SIG( enable );
        GRP( in, 8 );
    ENDLIST;

    OUT-LIST                    /* declare output ports */
        SIG( valid );
        GRP( out, 16 );
    ENDLIST;

    int r, s;                   /* declare local variables */

    valid = 0;
    if ( enable ) {

        r = in[];
        s = 0;
        while (r >= 0) {
            s = s + r ;
            r --;
        }

        out[] = s;
        valid = 1;              /* set the flag */
    }
    return;
}
```

The objects declared in the IN-LIST section represent input signals or ports from external. The object **enable represents** one signal line, and **in** represents a group of signals consisting of 8 signal lines. Likewise the objects in the OUT-LIST section represent output signals or ports set by the procedure. The statement "**r** = **in**[]" means that the signals grouped as **in** are packed into an integer (**r**). The statement "out[] = s" sets the group of output signals, **out,** by unpacking the integer (s).

Another example is given below. This procedure describes an ILSP functional model that takes two integers and produces the greatest common divisor (GCD) of those two integers. The procedure *swap*() swaps two integers.

```
gcd()
{
    IN-LIST
        SIG( enable );
        GRP( a, 16);
        GRP( b, 16 );
    ENDLIST;

    OUT-LIST
        SIG( valid );
        GRP( out, 16 );
    ENDLIST;

    int aa, bb;

    valid=O;
    if (enable) {
        aa = a[] ;
        bb = b[] ;
        while ( aa != bb ) {
            while ( bb > aa )
                bb = bb - aa;
            ( aa, bb ) = swap( aa, bb );
        }

        out[] = aa;
        valid = 1;
    }
    return;
}
```

Graph Representation of ILSP Procedures

1. Overview

In the behavioral synthesis process, a behavioral representation is translated into an intermediate representation in graph form, which is subsequently transformed and translated into a structural description [4]. In ***Her-mod,*** a graph representation is chosen that reflects both the program sequencing and the data flow in the program (Control and Data Flow Graph). One of the advantages of combining those two aspects is that data dependencies that cannot be established in data flow analysis can be inferred by the control sequencing in the graph [3].

Data and control operations are represented as nodes in the graph. Data and control transfers between nodes are represented by arcs. This document describes the specification of the control and data flow graph (C/DFG) which is produced by parsing the ILSP program. The C/DFG consists of several data flow subgraphs corresponding to basic blocks of the original ILSP program, each of which consists of straight line code and control nodes connecting them. The graph shows not only the dependency or parallelism of each operation but also the global control and data flow. It also provides a design representation capable of supporting design analyses, design decisions, and design transformations.

Her-mod supports a menu-driven user interface, displaying the control and data flow graph on the workstation screen and allowing the user to control state binding and resource sharing through a graphical interface. The C/DFG also allows hierarchical design by incorporating a procedure-call node. A procedure-call node representing some functional module can be specified by another graph.

In this document, wc present the C/DFG as an intermediate representation of the ILSP program, and describe the C/DFG construction. Examples are provided.

2. C/DFG Structure

The C/DFG consists of several types of nodes and edges. A node can represent data, an abstract operation (a logical or arithmetic operation or a procedure call), and a control construct. An edge can be a control edge representing control sequencing in the ILSP procedure, or a data edge representing data usage or data flow, depending on the type of the nodes connected by it. In this section, types and attributes of the node and edge are described.

2.1 Nodes of C/DFG

There are five types of nodes: data, constant, operation, control, and temporary nodes. Each node has one or more input ports and output ports to which in-coming and out-going edges are connected. Each port is identified by its io-attribute and port-id. Data, constant, temporary, and operation nodes (together with directed edges into and out of the nodes) reflect the data flow and data dependency, while control nodes are used for control sequencing and to mark the boundary of basic blocks.

2.1 .1 Data/Constant/Temporary Nodes

A data (or constant) node corresponds to an identifier (or constant) in the ILSP program. Basically, for each appearance of an identifier in the ILSP program, there is a data node in the graph corresponding to the identifier. Temporary nodes are used to represent the data produced by an operation and used by another operation or control node. Each of these nodes has one input port and one output port Data and constant nodes are graphically represented by circles with their symbolic names inside. Temporary nodes are represented by filled circles.

2.1.2 Operation Nodes

An operation node corresponds to an abstract operation in the ILSP program. A procedure call is considered an operation with multiple inputs and outputs. There are three types of operation nodes: binary operation, **unary** operation, and procedure call nodes. Figure 1 shows the graph representations of the operation nodes of the C/DFG.

1. **Binary operation nodes:** A binary operation node takes two inputs and produces one output. Binary operation nodes have two input ports, ***RIGHT*** and ***LEFT*,** and one output port. The bit width of each port can be different. Table 1 shows the binary operations implemented in Her-mod.
2. **Unary operation nodes:** A unary operation node takes one input and produces one output (both have port-id of 0). The input and output ports have the same bit width. Her-mod supports the following unary operations: "~" (one's complement), "-" (negation), and "!" (logical negation).

3. **Procedure call nodes:** A procedure call is an extension of a normal binary or unary operation. It may take more than two inputs and produce one or more outputs. Thus, procedure-call nodes have multiple input ports and output ports. Each port is identified by the port-id of a negative integer, i.e., the input port whose port-id is ***-n*** ***(n*** is a positive integer) corresponds to the ***n-th*** input argument to the procedure. The same rule is applied to the output ports. The procedure-call node in Figure 1 (c) takes ***n*** arguments and returns ***m*** values to caller.

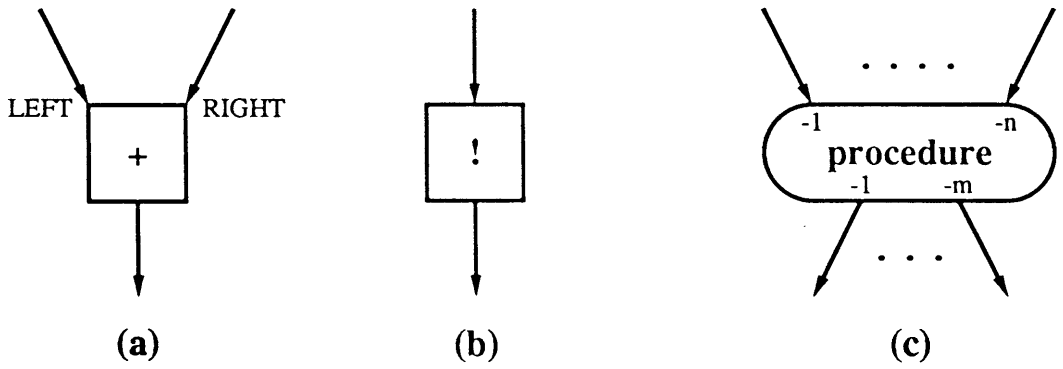


Figure 1: Representation of operation nodes in the C/DFG
(a) Binary operation node (b) Unary operation node (c) Procedure call node

operation type	**operations**
Arithmetic	**"+", "-", "*", "/", " %"**
Relational	**">", ">=", "<", "<=", "==", "!="**
Bitwise Op.	**"&" (AND), "|" (OR), "^" (XOR), ">>", "<<"**
Logical	**"&&" (AND), "||" (OR)**

Table 1: Binary operations

2.1.3 Control Nodes

A control node marks the beginning or end point of a basic block. In other words, a basic block starts and ends with a control node. There are seven different control nodes in ILSP, whose representations are shown in Figure 2.

1. **start node:** The start node represents the beginning of the whole graph, thus there should be only one start node in the graph. The start node has no input port but has one output port. In the C/DFG, the start node has out-going edges that go to the data nodes corresponding to constants, input signals, and state variables declared in the ILSP procedure.
2. **end node:** The end node represents the end of the graph, thus there should be only one end node at the bottom of the C/DFG. The end node has one input port that takes the edges from the data nodes corresponding to the output signals and the state variables that are defined but not killed (not assigned a new value) in the basic block ending with the end node.
3. **fork node:** A *fork* node is used in the if-then-else statement and marks the beginning of the subgraph corresponding to the body of the conditional statement. The fork node has two input ports. One port takes edges from the data nodes which are used after the fork node, i.e., in the body of the conditional statement. The other port takes an edge from the node which produces the control data of the fork node. The fork node has two output ***ports: TRUE*** and ***FALSE*** ports. These ports are the beginnings of the subgraphs corresponding to then-part and else-part, respectively.
4. **sfork node: A** *sfork* node marks the beginning of a switch-statement. It has two input ports, whose connections are same as those of a fork node. Output ports mark the beginnings of the subgraphs representing case statements and default statement.
5. **join node: A** ***join*** node marks the end of an if-then-else or switch statement. It has one input port and one output port. The join node also marks the beginning of a while-loop subgraph.
6. **loop node: A** *loop* node is used as the starting control node for the subgraph corresponding to a do-loop construct. It has one input port and one output port. The function of the loop node is same as the join node in the while-loop construct.
7. **loop-end node: A** *loop-end* node marks the end of the subgraph corresponding to a do-loop construct. It has two input ports whose connections are same as those of fork node, and two output ports, ***TRUE*** and ***FALSE*** ports. The function of the loop-end node is almost same as the fork node. The difference is that the ***FALSE*** port is connected to the loop node by a control edge so that the subgraph between loop node and loop-end node can be repeated.

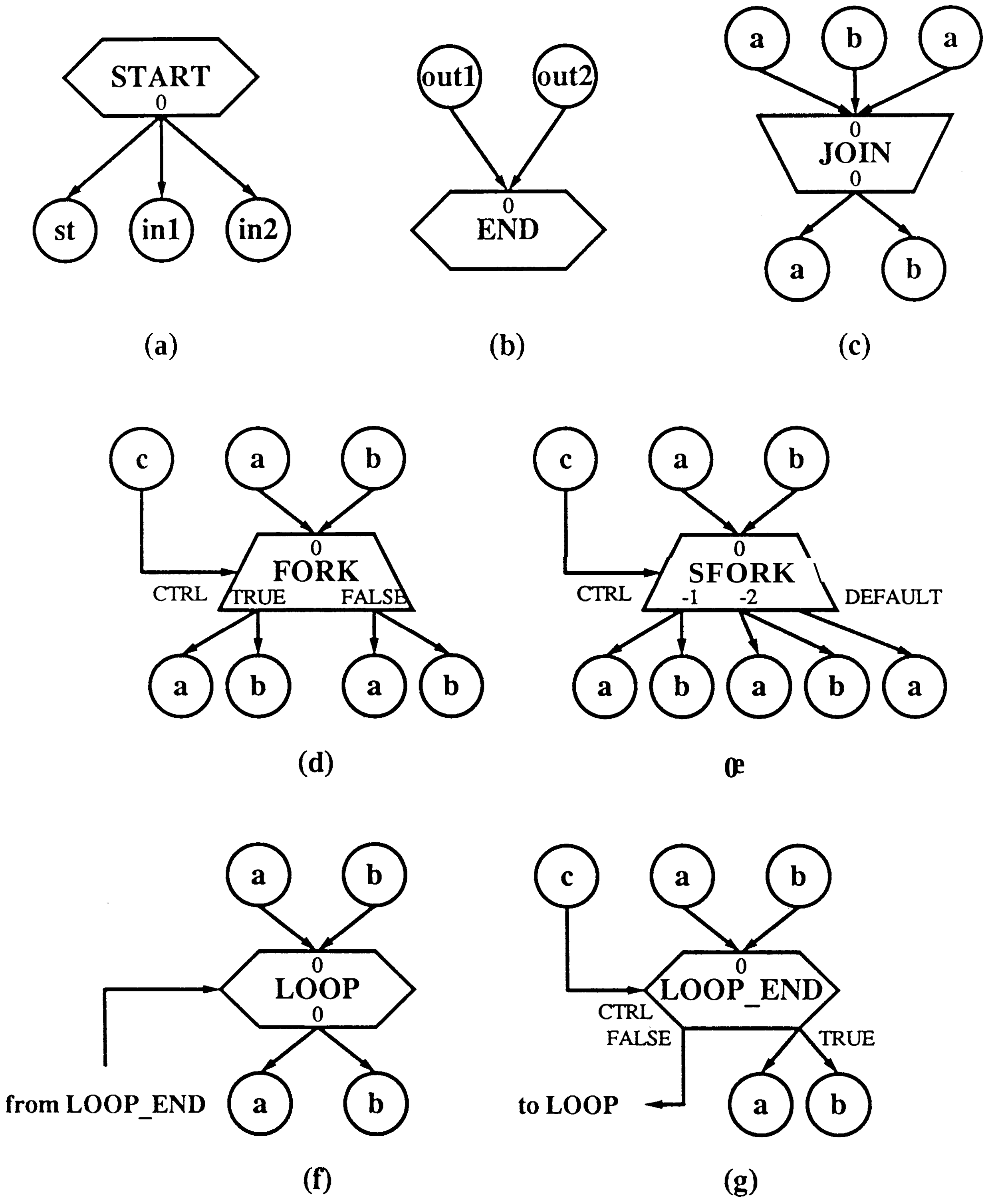


Figure 2: Control node representations in the C/DFG

2.2 Edges of C/DFG

An edge in the graph is a-directed edge which shows the flow of data or control from the initial node to the terminal node. There are two types of edges: data edges and control edges.

2.2.1 Data Edges

Data edges show the data flow in the C/DFG. Depending on the type of the nodes they connect, their implications are different. (Figure 3 shows an example of data edges.)

1. **data node to operation node:** implies the corresponding data is consumed by the operator.
2. **operation node to data node:** implies the result produced by the operation is stored in the variable represented by the data node.
3. **operation node to temporary node:** implies the data produced by the operation is stored temporarily for use by another operation or control node. This construction, for instance, is used for building the subgraph for expressions which have more than one operation.

2.2.2 Control Edges

The implications of control edges are different depending on what types of nodes they connect. The usage of control edges is as follows: (An example is given in Figure 4, where control edges are represented by solid arrows.)

1. **data or temporary node to control node:** implies the value of the data node controls the action of the control node (typically, fork, sfork and loop-end nodes).
2. **control node to data or constant node:** implies the data or constant is used in the basic block headed by the control node.
3. **control node to control node:** This type of control edges represents control transfer.

2.2.3 Edge Attributes

Just as the node of the C/DFG has attributes, each edge of the C/DFG has attributes independent of the edge **type (DATA** or CONTROL). Each edge has one of the following attributes:

1. ***NORMAL:*** This means the edge is neither a ***BACK edge*** nor ***TIMING-CUT edge. (See*** below.)
2. ***BACK:*** A ***BACK*** edge is an edge which constructs a loop in the C/DFG. This edge appears in a subgraph which corresponds to the while-loop or

the do-loop. See while-loop and do-loop constructions in the next section.

3. ***TIMING-CUT:*** A ***TIMING-CUT*** edge has timing-cuts on it. A timing-cut means the synchronization by a clock should be done at the cut point. The data going through the edge will be stored in a register. The timing-cuts creation program (automatic or manual) will choose the proper edges for timing-cut position and makes them ***TIMING-CUT edges.***

4. *B_AND_T*: A ***B-AND-T*** edge is a ***BACK*** edge with timing-cut on it.

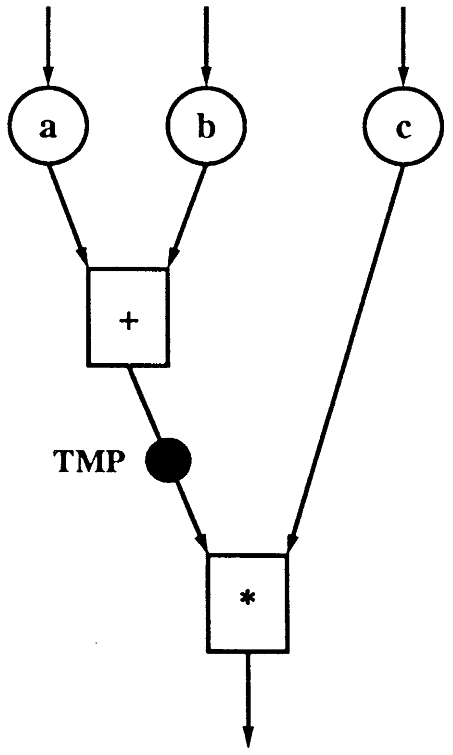


Figure 3: Example of data edges

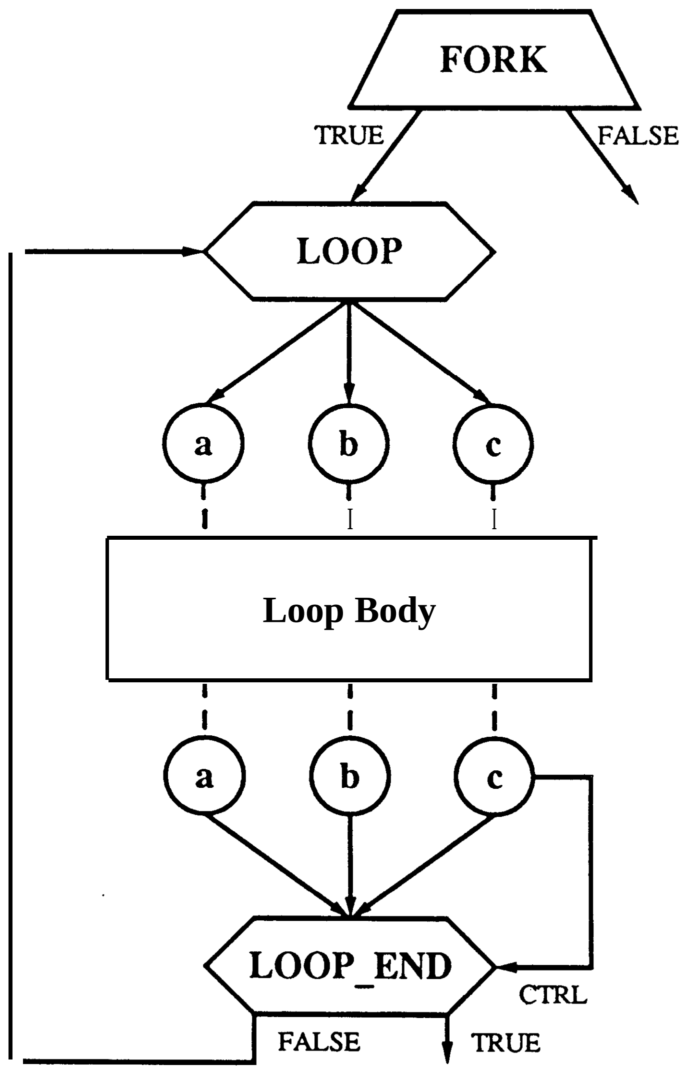


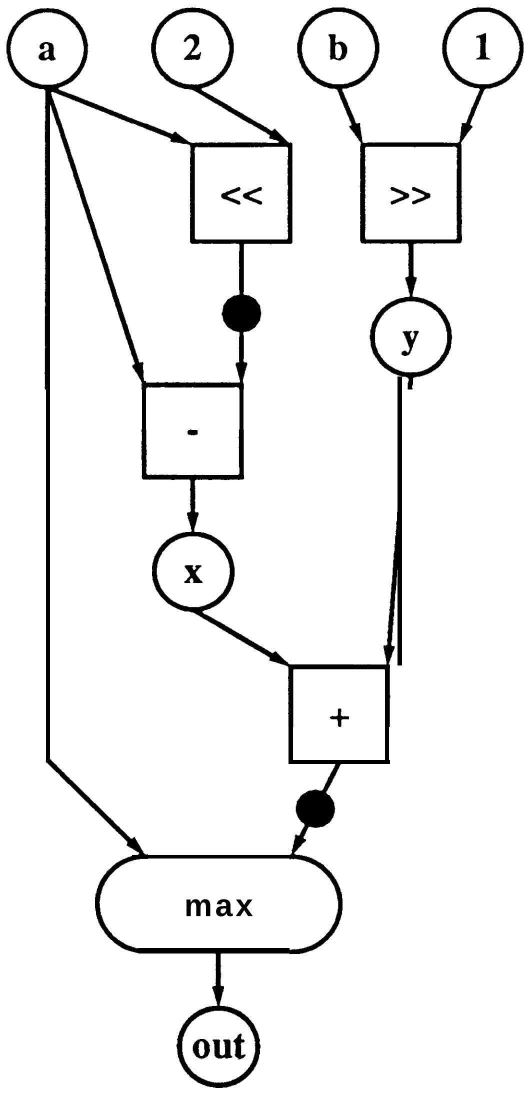
Figure 4: Example of control edges

3. Graph Construction

An ILSP procedure consists of three types of blocks: straight line code, if/switch statements, and while- and do-loops. Graph construction of each type of block is described next.

3.1 Straight Line Code

A block of straight line code consists of expressions and assignments. The subgraph representing straight line code is a directed acyclic graph (DAG). A typical example is presented in Figure 5.



```
x = a[] - a[] << 2 ;
y = b[] >> 1;
out[] = max ( a[], x + y ) ;
```

Figure 5: Straight line code

Usually many optimization and transformation techniques employed in a complier are applied to these subgraphs in a behavioral synthesis systems. Expressions are realized so that the operation nodes take edges from the operand data nodes. Assignment is normally realized by the edge from an operation node to a data node, which means that the result of the operation is stored in the variable represented by the data node. Temporary data nodes are inserted if necessary. Note that temporary nodes are shown for data produced by an operation node and consumed by another operation node. In special cases like x = **a** or x = 0, the assignment operator is used to show the value of the left side variable is replaced by that of the right hand side variable.

Retrieving data from or assigning values to a subset of group signals is allowed in the ILSP syntax. To construct the subgraph which shows such a partial retrieval or assignment, the *fpack* and *funpack* (intrinsic library functions in the THOR simulator [1, 2]) procedure-call nodes are used as shown in Figure 6.

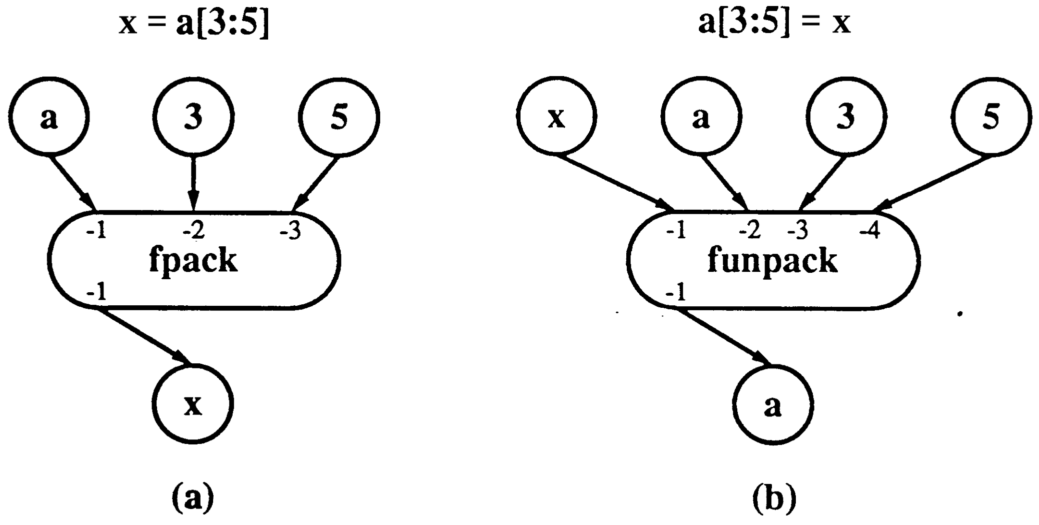
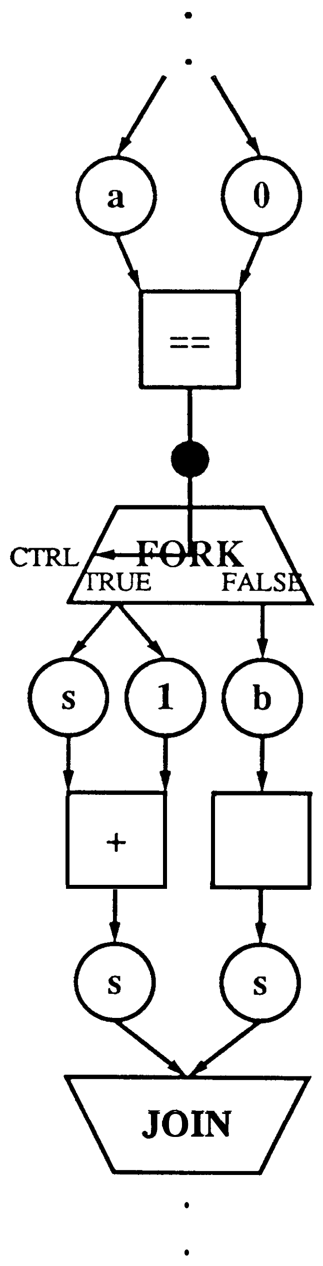


Figure 6: (a) function *fpack* (b) function *funpack*

3.2 Conditional Statements

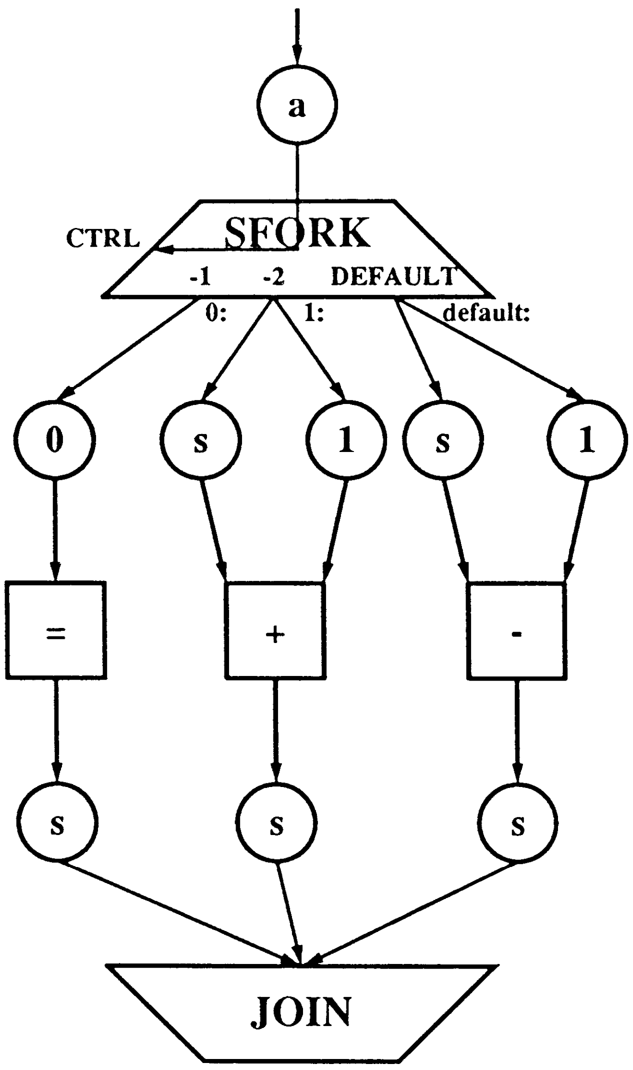
The if-then-else block consists of two sub-blocks corresponding to the then-part and else-part, respectively. The then-part (else-part) sub-block starts from the ***TRUE (FALSE)*** port of a fork node and ends at a join node. The out-going control edges of each port of the fork node show that the terminal nodes of those edges (data nodes or constant nodes) will be used in then-part sub-block or else-part sub-block. The in-coming edges of the join node show that the corresponding data nodes were defined but not killed in either sub-block. The subgraph corresponding to the conditional expression of the if statement is inserted in the basic block which ends at the join node. The fork node takes several in-coming edges whose initial nodes were defined in the basic block but not killed. An example is illustrated in Figure 7.



```
if ( a[] == 0 )
        s[] = s[] + 1 ;
else
        s[] = b[] ;
```

Figure 7: If-then-else construction

A switch statement is converted into a block which has several sub-blocks. Each of these sub-blocks corresponds to a ***case*** statement or *the* ***default*** statement of the original switch statement. The sub-block corresponding to the n-th case (or default) statement of the switch statement ***starts*** from the ***n-th*** port (or ***DEFAULT)*** port of a sfork node and ends at **a** *join* node. The meaning of in-coming and out-going edges of the sfork node and the join node is the same as in the if-then-else block. Figure 8 shows the graph construct of a switch statement.

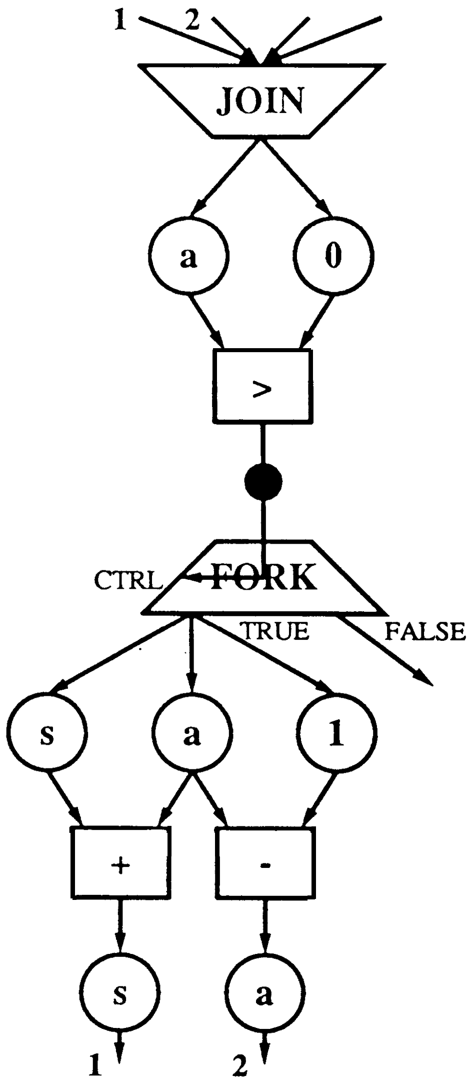


```
switch ( a[] ) {
    case 0 :
        s[] = 0 ;
        break ;
    case 1 :
        s[] = s[] + 1 ;
        break ;
    default :
        s[] = s[] - 1 ;
    b r e a k  ;
```

Figure 8: Switch construction

3.3 While- and Do-Loop Constructs

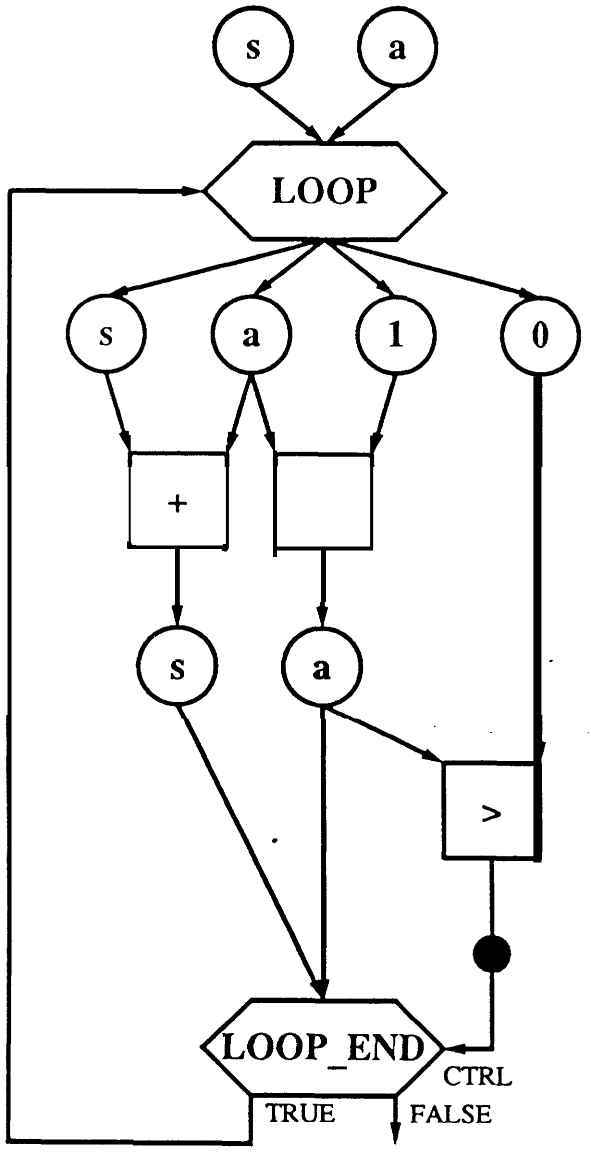
A while-loop subgraph is also constructed with fork and join nodes. In this subgraph, the join node is followed by the subgraph corresponding to the conditional expression of the while statement, which is connected to the ***CONTROL*** port of the fork node. The subgraph which corresponds to the loop body ***starts*** from the ***TRUE*** port of the fork node, and ends at the join node through ***BACK*** edges in order to show this block will be iteratively executed. Figures 9 shows a while-loop construct.



```
while ( a > 0 ) {
      s = s + a ;
      a -- ;
}
```

Figure 9: While-loop construction

A do-loop subgraph consists of one block which starts from a loop node and ends at a loop-end node. The subgraph of conditional expression is also included in this subgraph. The loop-end node has two ports: the ***TRUE*** port connected to the loop node by a ***BACK*** edge, and the ***FALSE*** port from which a new basic block begins after the do-loop statement. An example is given in Figure 10.



```
do {
      s = s + a ;
     a -- ;
} while ( a > 0 ) ;
```

Figure 10: Do-loop construction

4. Examples

Combining the constructs explained in the previous section, the graph representations of two procedures are shown, one for calculating the factorial of a given integer (Figure 11) and the other for finding the gcd (greatest common divisor) of two integers (Figure 13).

```
sum()
{
    IN-LIST                          /* declare input ports */
        SIG( enable );
        GRP( in, 8 );
    ENDLIST;

    OUT-LIST                         /* declare output ports */
        SIG( valid );
        GRP( out, 16 );
    ENDLIST;

    int r, s;                        /* declare local variables */

    valid = 0;
    if ( enable ) {

        r = in[];
        s = 0;
        while (r >= 0) {
            s = s + r;
            r --;
        }

        out[] = s;
        valid = 1;                   /* set the flag */
    }
    return;
}
```

Figure 11: An ILSP procedure calculating the factorial of a given integer

Figure 12 shows the C/DFG for this procedure. Here, relational operator nodes are represented by triangles to differentiate them from the other operational nodes.

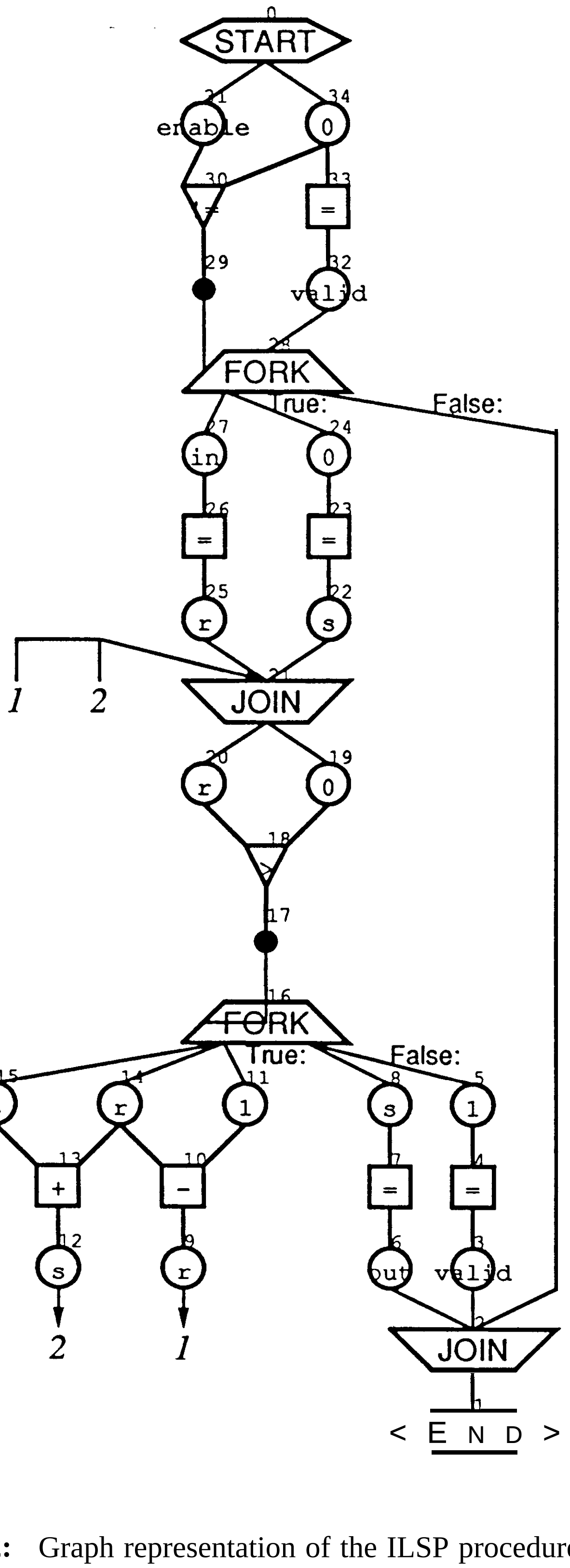
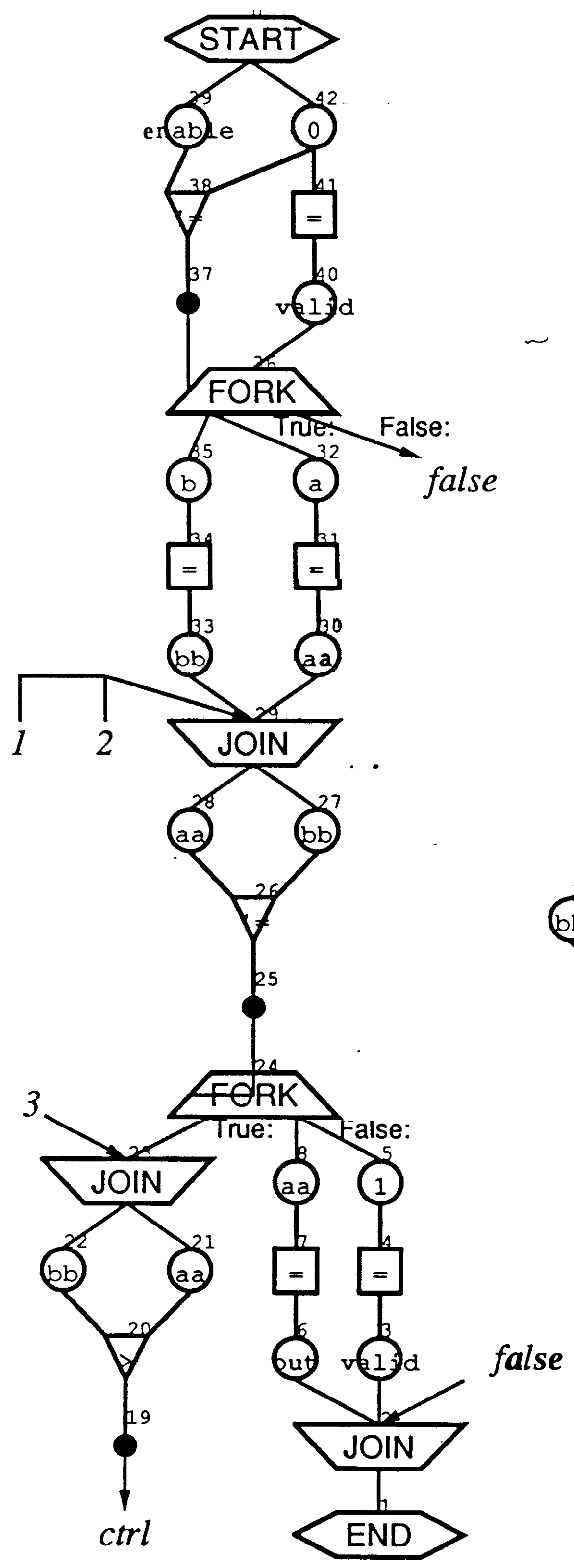


Figure 12: Graph representation of the ILSP procedure in Figure 11

Figure 13 shows the procedure that finds the gcd of two given integers, and Figure 14 shows the C/DFG for this procedure.

```
gcd()
{
        IN-LIST
                SIG( enable );
                GRP( a, 16 );
                GRP( b, 16 );
        ENDLIST;

        OUT_LIST
                SIG( valid );
                GRP( out, 16 );
        ENDLIST;

        int aa, bb;

        valid = 0 ;
        if (enable) {
                aa = a[] ;
                bb = b[] ;
                while ( aa != bb ) {
                        while ( bb > aa )
                                bb = bb - aa;
                        ( aa, bb ) = swap( aa, bb );
                }

                out[] = aa;
                valid = 1;
        }
        return;
}
```

Figure 13: An ILSP procedure calculating the greatest common divisor



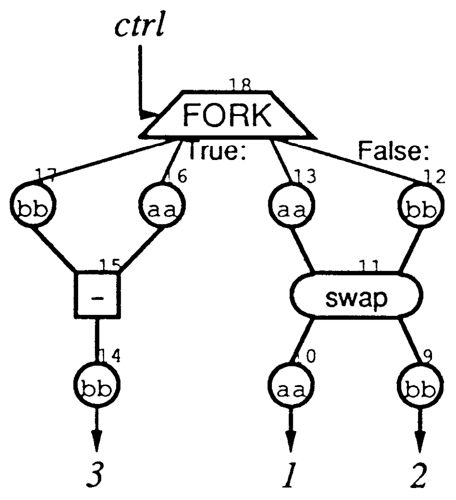


Figure 14: Graph representation of the ILSP procedure in Figure 13

A. Graph Data Structures

The Hermod program is implemented in C++, an objected oriented programming language. The internal representations of the C/DFG are described in this section. The C/DFG is implemented around three linked lists, *node-list, edge-list,* and *sub-graph.*

A.1. Node Data Structure

The struct *node-fist* holds the information of the nodes in the C/DFG. The following data structure is used for the data and control nodes.

```
struct node-list {
    int type ;                        /* DATA, CONTROL, OP, CONST, or TMP */
    int attr ;                        /* node attribute */
    struct caselabel * cl;            /* pointer to case label table */
    struct symtable * ps ;            /* pointer to the symbol table. */
    struct edge-list * pe ;           /* pointer to the edges connected to it */
    struct node-list * next ;         /* pointer to the next entry */
    int in ;                          /* in-degree */
    int out ;                         /* out-degree */
    void*pwl;                         /* work area 1 */
    void * pw2 ;                      /* work area 2 */
    class NList * pN ;                /* pointer to the object for drawing */
    struct hardware * hw ;            /* pointer to the hardware information */
    int num ;                         /* node number */
    struct basicblock * bb ;          /* basic block, for control node */
    int dfn ;                         /* data flow number, for operation node */
    int tclevel ;                     /* timing-cut level */
    class HUnit * hu ;                /* pointer to the associated hardware unit */
} ;
```

The meaning of each field in the struct *node-list* is briefly described.

- **int type -** shows the node type, values.
- **int attr -** shows the attribute of the node. The attribute, for example, means kind of operation for operation nodes, kind of control for control nodes, and constant value for constant nodes.
- **struct symtable *cl -** is a pointer to the mapping table between port number and case label (case constant). This pointer is valid only for sfork node. This pointer is *NULL* for other types of nodes.
- **struct symtable *ps -** is a pointer to the entry in the symbol table. This pointer is meaningful for data nodes, temporary nodes and the procedure call nodes. This pointer is NULL for the other types of nodes.
- **struct edge list *pe -** is a pointer to the first entry of the edges connected

to this node.

- **struct node list *next -** is a pointer to the next entry of the node-list.
- **int in -** is a number of in-coming edges.
- **int out -** is a number of out-going edges.
- **void *pw1, void *pw2 -** are used as working storages.
- **class NList *pN -** is a pointer to the object for drawing. This pointer is set by the drawing procedure.
- **struct hardware *hw -** is a pointer to the hardware information assigned the node.
- **int num -** is a node identification number. This number is zero or positive integer. See next section for detail.
- **struct basicblock *bb, int dfn -** are set by the data flow analysis program to indicate which basic block this node belongs to.
- **int tclevel -** represents the timing-cut level.
- **class HUnit *hu -** is a pointer to the associated hardware unit in the data path built from this C/DFG.

Node Identification Number:

The node identification number is used not only for identifying each node but also for drawing on the screen. The graph drawing routine draws and displays the graph based on the node identification number. The following numbering rules are applied to draw graphs.

The Node Numbering Rules:

1. The start node should have the node identification number 0.
2. If node ***n1*** has an out-going edge to node ***n2***, the node identification number of ***n1*** should be greater than that of the node ***n2***.
   Exception: This rule does not apply to the start node.
3. If node ***nl*** is in the subgraph corresponding to the then-part of an if-statement and node ***n2*** is in the subgraph corresponding the else-part, then the node identification number of the node ***n1*** should be greater than that of the node ***n2***.
4. If the node ***n1*** is in the subgraph corresponding to the ***n-th*** case statement of a switch statement and the node ***n2*** is in the subgraph corresponding to the ***m-th*** case statement where ***m > n*** or the default statement of the switch statement, then the node identification number of the node ***n1*** should be greater than that of the node ***n2***.

Figure 15 shows the legal node numbering. Nodes are numbered sequentially from 0 to the number of nodes according to the linking order in ***the node-fist*** maintained by the synthesis system. Thus, the linking order of nodes in ***the node-list*** should follow the above rules.

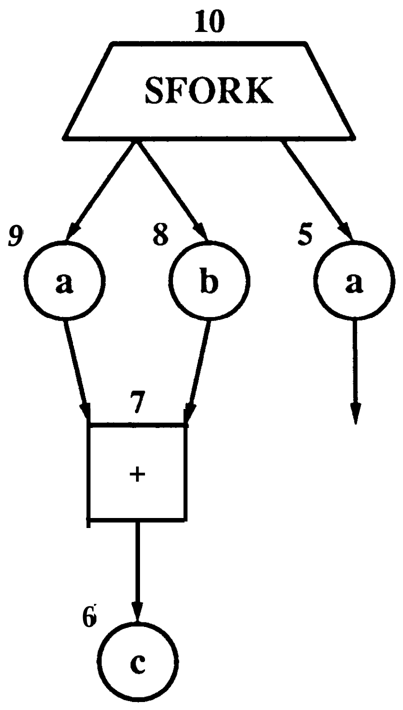


Figure 15: Node numbering

A.2. Edge Data Structure

The struct ***edge-list*** holds the information for edges in the graph.

```
struct edge-list {
    int type ;                          /* edge type */
    int io ;                            /* in-coming or out-going */
    int port1 ;                         /* port of origin node */
    struct node-list * pair ;           /* pointer to the pair node */
    int port2 ;                         /* port of the pair node */
    struct edge-list * mirror ;         /* pointer to the mirror edge. */
    struct edge-list * next ;           /* pointer to the next edge */
    class EList * pE ;                  /* pointer to the object for drawing */
    edgetype et ;                       /* edge type */
    int tcount ;                        /* number of timing-cuts in the edge */
    int tcwork ;                /* work area for the timing-cut functions */
    class List * tcuts ;        /* timing-cuts to which this one belongs */
    int mark.;                  /* used in the automatic timing-cut function */
    int num ;                   /* edge number */
} ;
```

For each node, there is a list of edges associated with it, each of which has the data structure of ***edge-fist.*** The list is a linked list of all the edges connected to the node (both in-comming and out-going edges). The head of the edge list is maintained in the ***pe*** field of the node data structure.

In this graph data structure, ***two edge-list*** data structures are kept to represent each edge. For an edge from node ***nl*** to *n2***,** node ***nl*** sees the edge going to node *n2*, while node *n2* sees it coming from node ***nl.*** They are called mirror edges. (Refer to Figure 16.)

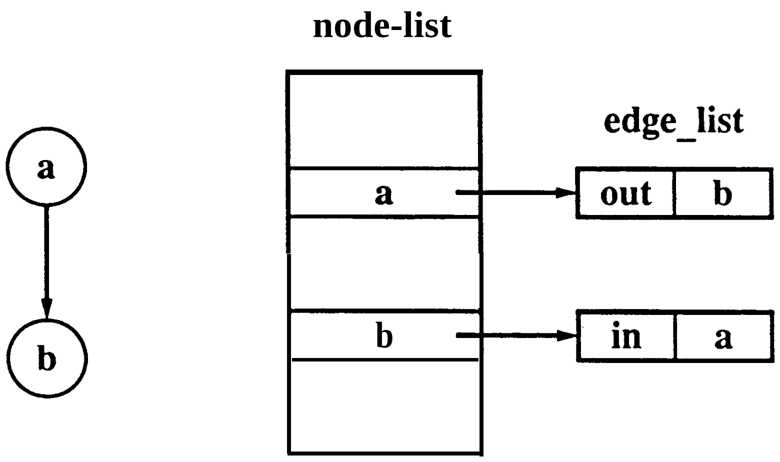


Figure 16: The mirror edge-list

- **int type -** shows the type of edge. It can be either DATA or CONTROL.
- **int io -** indicates the direction of the edge, incoming or out-going.
- **int portl -** shows the port of the initial node of the edge if the edge is out-going. Otherwise, it shows the port of the terminal node.
- **struct node_list *pair -** is the terminal node of the edge if the edge is out-going. Otherwise, it is the initial node of the edge.
- **int port2 -** is the port of the terminal node to which the edge is connected if

the edge is out-going. Otherwise, it is the port of the initial node.

- **struct edge list *mirror -** is a pointer to the mirror edge.
- **struct edge list *next -** is a pointer to **the** next entry of the **edge-list.**
- **class EList *pE -** is the pointer to the object for drawing.
- **edgetype et -** takes the integer value; ***NORMAL, BACK, TIMING-CUT,*** or ***B-AND-T.*** Each type is described in previous sections.
- **int tcount -** is the number of the timing-cuts on the edge.
- **class List *tcuts -** is the linked list of the time-cuts to which the edge belongs.
- **int num -** is the unique id.

A.3. SubGraph Data Structure

The struct *sub_graph* holds the head pointer to the start node of a data flow graph which corresponds to a procedure in an ILSP program.

```
struct sub_graph {
     int num ;                        /* sub_graph identification number */
     struct node-list * start ;       /* pointer to the start node */
     struct sub_graph * next ;        /* pointer to the next entry */
     struct basicblock * bb ;         /* the basic blocks for this graph */
     struct dfntable * trtable ;      /* the data flow translation table,
                                         associate data flow number to graph a node */
     class List * timing-cuts ;       /* linked list of the timing-cuts */
     class GraphData * gd ;           /* pointer to the object for drawing */
     class StatDgrm * sd ;            /* pointer to the state diagram */
     class DataPath * dp ;            /* pointer to the data path graph */
} ;
```

- **struct node list *start -** is the pointer to the start node.
- **struct sub-graph *next -** is the pointer to the next subgraph.
- **struct basicblock *bb, struct dfntable *trtable -** are used in the data flow analysis routine. The table (pointed to by *trtable*) maps a data flow number to a graph node and a symbol table entry which contains information about each identifier.
- **class List *timing cuts -** is the list of timing-cuts imposed on this C/DFG.
- **class GraphData *gd -** is the pointer to the data structures for drawing the C/DFG on the screen.
- **class StatDgrm *sd, class DataPath *dp -** are the pointers to the data

structures that maintain the hardware assignment information.

A.4. Other Data Structures

1) The data structure ***caselabel*** holds a mapping table entry between a port of the sfork control node and a case constant. This is used only for the sfork nodes.

```
struct caselabel {
    int port ;                       /* port number */
    int label ;                      /* case constant */
    struct caselabel * next ;        /* pointer to the next entry */
} ;
```

2) The data structure ***timing-cut*** holds the information on a timing-cut on an edge. Timing-cuts on an edge are kept in the field ***tcuts*** of the structure ***edge-fist,*** and all the timing-cuts in the subgraph are kept at ***the timing-cuts*** field in ***the structure sub_graph***.

```
struct timing-cut (
    int id ;                         /* timing-cut identification number */
    List * edges ;                   /* edges belonging to this timing-cut */
    TimingNode * pN ;                /* pointer to the object for drawing. */
    int level ;                      /* relative position of the timing-cut */
    short drawn ;                    /* flag for checking if already drawn */
} ;
```

3) The basicblock structures maintain the list of nodes belonging to a particular basic block and the information on the usage of data nodes. They also contain state binding and hardware representation information of the subgraph.

4) The class ***List*** is an implementation of the double-linked circular list. It can hold pointers to ***any*** data ***type. The*** classes ***NList, EList, GraphData are*** derived classes of the class List and used for drawing on the screen.

5) The classes ***StatDgrm*** and ***DataPath*** hold the information on the hardware generated from the C/DFG by the synthesis tools. They are tightly coupled with the C/DFG data structure by pointers so that the original C/DFG information can be retrieved easily from these classes.

References

[1] R. Alverson, T. Blank, K. Choi, S.Y. Hwang, A. Salz, L. Soule, and T. Rokicki.
THOR User's Manual: Tutorial and Commands.
Technical Report CSL-TR-88-348, Stanford University, Stanford, CA, January, 1988.

[2] R. Alverson, T. Blank, K. Choi, S.Y. Hwang, A. Salz, L. Soule, and T. Rokicki.
THOR User's Manual: Library Function.
Technical Report CSL-TR-88-349, Stanford University, Stanford, CA, January, 1988.

[3] A. Orailoglu and D.D. Gajski.
Flow Graph Representation.
In ***Proc. 23rd Design Automation Conf.***, pages 503-509. ACM/IEEE, June, 1986.

[4] B.M. Pangrle and D.D. Gajski.
Design Tools for Intelligent Silicon Compilation.
IEEE Trans. Computer-Aided Design of Integrated Circuits and Syst. CAD-6(6):1098-1112, November, 1987.