accepted for publication in
Journal of Systems and Software, 1989

# The *Hermod* Behavioral Synthesis System

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

**Center for Integrated Systems**
**Stanford University**

## Abstract

*Hermod* is an interactive behavioral synthesis program developed at Stanford University. Using a combined control and data flow graph (C/DFG) as an intermediate representation, Hermod generates functional blocks and their interconnection from behavioral descriptions. Hermod supports a menu-driven interface, displaying the control and data flow graph with a set of legitimate *timing-cuts* and its hardware representation. Emphasizing user participation, the system allows the user to control state partitioning and resource sharing through a graphical interface to explore the maximal design space. Written in an object-oriented language C++, Hermod generates a hardware representation in several minutes from a behavioral description of practical size on a VAXstation II/GPX.

**Indexing Terms**: behavioral synthesis, structural synthesis, control and data flow graph, register-transfer level description, design space exploration.

*** Note: Revised Copy for "Journal of Systems and Software".**
**-- 8 June 1988**

# 1. Introduction

Silicon compilation is the process of automatically mapping an abstract design representation to a physical structure [19]. Depending upon the input language, silicon compilers are classified into behavioral compilers (or behavioral synthesizers) and structural compilers. A behavioral synthesizer translates a behavioral description into a structure, creating structural designs consisting of functional blocks and their interconnection. In a behavioral synthesis system, the design is specified by a functional relationship between input and output ports described in a hardware description language [4, 11, 26]. The behavior of output ports is specified in terms of input ports and internal state. The output from a behavioral synthesizer contains hardware modules[1] (*data paths*) required to implement the given behavioral specification, and their scheduling (*control*). [illegible]

The synthesis task includes generation of data paths and their control blocks. Data path synthesis consists of the following subtasks: module bindings, state bindings (control step partitioning), and register and connection bindings. In the module binding process, a functional module is assigned to each abstract operation, and a register is assigned to data carried across state transitions. The functional modules (or registers) can be shared among more than one abstract operation (or variable). Further, the binding process relies on library, which may contain structural modules at several abstraction levels [9]. The module binding process has a great impact on the system cost and performance, since an abstract operation can be realized in many ways. The module binding process is performed before control logic synthesis, even though the process can be iterated later if imposed constraints are not met. State binding implies assignment of each operation to a machine state. Machine states are created depending on the clock period of the system and the delay of hardware modules. Based on the clock period and the number of maximum allowed states, the system assigns each abstract operation to a machine state such that maximum parallelism can be achieved within available hardware modules. Connection bindings imply the interconnection between functional modules and registers, creating the data transfer paths between them. Connections can be implemented using bus structure or multiplexors.

Control synthesis creates the finite state machine that controls the data path units for the proper execution of code sequences. The goal is to define the sequence of the micro-operations and the timing of the control signals to the data path. To generate the control block, the data path must be fully defined, and the required operations must be specified as a linear ordered list of micro-operations which affect either the control flow or data path. Two different design styles have been used for control block implementation: structured and

[1]In this paper, a functional module represents a hardware block which can execute an operation like addition and subtraction, while a hardware module is used to represent a functional module, register, or wire.

custom implementations. In a structured implementation, the next state information and signals for data path selection are encoded in a structured array. This implementation is flexible and easy to modify. A custom implementation uses random logic for efficiency, exploiting the particular features of the data path. Detailed description of various algorithms for control synthesis can be found in [7, 14, 18, 24].

Automatic synthesis from behavioral specifications is an exploratory area for design automation. A number of behavioral compilers have been reported that generate structural descriptions from behavioral level descriptions [6, 8, 14, 16, 17, 20, 23, 25]. Those systems automatically generate a structural description from a behavioral description using the design constraints given by a designer. However, supported design styles are limited in most systems, giving few chances for the designer to change what the system generates. Thus, designer may feel alienated from the system with no choices other than to accept the machine-generated designs, even if he is not satisfied with the output.

This paper describes the Hermod behavioral synthesis program that gives a hardware designer full system control to select the design style he likes through a menu-driven graphical interface. The system is not intended for use with design descriptions which would require thousands of components for realization, but for designs at high abstraction levels where design space exploration is of primary concern in the early stage of design. The Hermod program is included in an integrated environment for hardware simulation and synthesis under development at Stanford University. The functional models written in a behavioral description language *ILSP* [15] can be simulated on the *THOR* logic/functional/behavioral simulator [2] without translation. The behavioral models that have been verified through the THOR simulator are input to Hermod to generate RTL descriptions, which can be again simulated by THOR simulator for verification purposes. Efficient algorithms are incorporated in Hermod for generating timing-cuts for state partitioning, checking consistency after modifications to the system-generated designs, and optimizing through resynthesis.

# 2. Input Language and Internal Representation

## 2.1 ILSP: Behavioral Description Language for Hermod

ILSP (Input Language for Synthesis Program) is used to describe the function or behavior of the hardware to be designed. Based on the C-language, ILSP has conditional (*if* and *switch*), and loop (*while-* and *do-loop*) control constructs, and 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. Compared to the ISPS (Instruction Set Processor Specifications) [4] which describes the behavior and structure of the design at register-transfer and behavioral levels, the ILSP description is purely procedural. The features of ILSP are briefly reviewed next. A detailed description can be found in [15].

**Signal Declarations:** Three fundamental object types are supported. The objects declared as integers are local variables to the procedure in which they are declared. The objects declared in the signal declaration sections (*IN_LIST*, *OUT_LIST* and *ST_LIST* sections) as signals (*SIG*) are one-bit-signal variables and those declared as groups (*GRP* or *BUS*) are multiple-bit-signal variables. The SIG- and GRP-type objects are the abstract representations of *registers* in hardware realization. An integer object may be realized by a register, or just as a wire depending on its usage and lifetime.

**Control Constructs:** Most control constructs in the C language are supported: conditional statements (if- and switch-statements) and loop (while- and do-loop) constructs.

**Expressions and Statements:** Expressions and statements supported in ILSP are a subset of those in the C language. Arrays of complex data structure (like arrays of structure with several data fields) are not supported. Array structure is allowed only to represent a group of signals, which will be realized by registers or memory modules. The differences from the C language in expressions and statements are summarized as follows:

- Structures and pointers are not allowed. An exception is that a pointer is passed to a subprocedure as an argument for a GRP-type object in a procedure call.
- Array structures are used to specify bit position for group signals. 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 the partial signal group between the seventh bit and fourth bit of x treated as an integer.
- Increment expressions (++ and --) are allowed for integer variables only.

- Procedure calls that return one or more values are supported. The *"(receiver-list) = procedure-call-expression;"* form is used for procedure calls that return multiple outputs and distribute the values to the variables and group signals in the *receiver-list*.
- A *break* statement is allowed only in a switch statement to eliminate abrupt loop exits.
- Parameter passing in a procedure call is handled through the hardware interface mechanism. That is, the input and output parameters are declared in the interface declaration sections (*IN_LIST* and *OUT_LIST* declarations), unlike C procedures.
- A return statement is allowed only at the end of 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.

Figure 1 shows a procedure describing the design that takes two groups of signals, **in** and **enable**, and calculates the summation of the value of **in**, setting the signal group **out** and signal line **valid**. The objects declared in the *IN_LIST* section represent input signals or ports. **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).

## 2.2 Graph Representation

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. In Hermod, a graph representation is chosen that reflects both the control sequencing and the data flow in the program. In the graph, a node can represent a data carrier, an abstract operation, or a control construct. An edge can be a control edge representing control sequencing in the behavioral description, or a data edge representing data usage or data flow depending on the types of the nodes connected by it. The graph consists of several data flow subgraphs corresponding to basic blocks of the behavioral description, each of which consists of straight line codes 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 in the model. The C/DFG allows hierarchical design by incorporating a procedure-call node. A procedure-call node representing some hardware block can be specified by another graph. This graph representation is similar to the McFarland's Value Trace [12]. However, unlike VT, nested loop constructs are used in the representation, which is a natural way to handle loops.

There are five types of nodes: data, constant, operation, control, and temporary nodes.

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

    OUT_LIST                    /* declare output ports */
        S̄IG( 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 1:** A behavioral model calculating the summation of a given integer.

Each node has one or more input ports and output ports. Each port is identif 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 dependencies, while control nodes are used for control sequencing and to mark the boundaries of basic blocks.

- *Data/Constant/Temporary Nodes*: A data (or constant) node corresponds to an object (variable or constant) in the behavioral description. Basically, for each appearance of a variable, there is a corresponding data node in the graph. Temporary nodes are used to represent the data produced by an operation and used by another operation or control node.
- *Operation Nodes*: An operation node corresponds to an abstract operation in the description. A procedure call is considered an operation with multiple inputs and outputs.
- *Control Nodes*: A control node shows the beginning and end of a basic block. There are seven different control nodes: start, end, fork (for if-statements), sfork (for switch statements), join, loop, and loop-end nodes.

An ILSP procedure consists of three types of blocks: straight line code, conditional statements, and while- and do-loops.

**Straight Line Code:** A block of straight line code consists of expressions and assignments. Expressions are realized in such a way that operation nodes take edges from 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. Retrieving data from or assigning values to a subset of group signals is allowed. To construct the subgraph showing such a partial retrieval or assignment, the *fpack* and *funpack*[2] procedure-call nodes are used as shown in Figure 2.

[2]These are intrinsic library functions in the THOR simulation system [3]. *Fpack* converts a group signal of specified bit width into an integer and *funpack* converts an integer into a group signal.

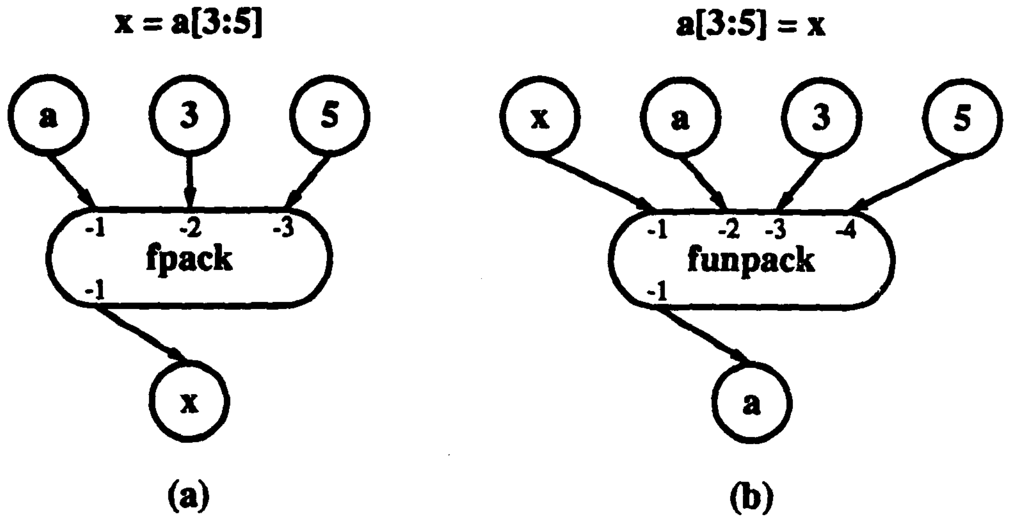

**Figure 2:** Graph representation for the functions (a) *fpack* (b) *funpack*.

**Conditional Statements:** The if 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 subgraph corresponding to the conditional expression is connected to the *CONTROL* port of the fork node.

**While/Do Loop Constructs:** A while-loop subgraph is also constructed with join and fork 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 of the while-loop body starts from the *TRUE* port of the fork node, and ends at the join node through *BACK* edges to show the iterative nature of the block. A do-loop subgraph consists of one block which starts from a loop node and ends at a loop-end node. The output of the conditional expression is connected to the *CONTROL* port of the loop-end node. The loop-end node has two output 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.

Figure 3 shows the graph representation of the procedure shown in Figure 1. Rectangles represent operation nodes that will be mapped into structural components in the module binding process, and circles represent variables and constants (data nodes). Control nodes are represented by trapezoids (fork and join nodes) or hexagons (start and end nodes). The outer fork-join node pair corresponds to the if-statement that is controlled by the value of **enable**. The inner join-fork pair corresponds to the while-loop statement. The control input to the fork node is from the condition expression.

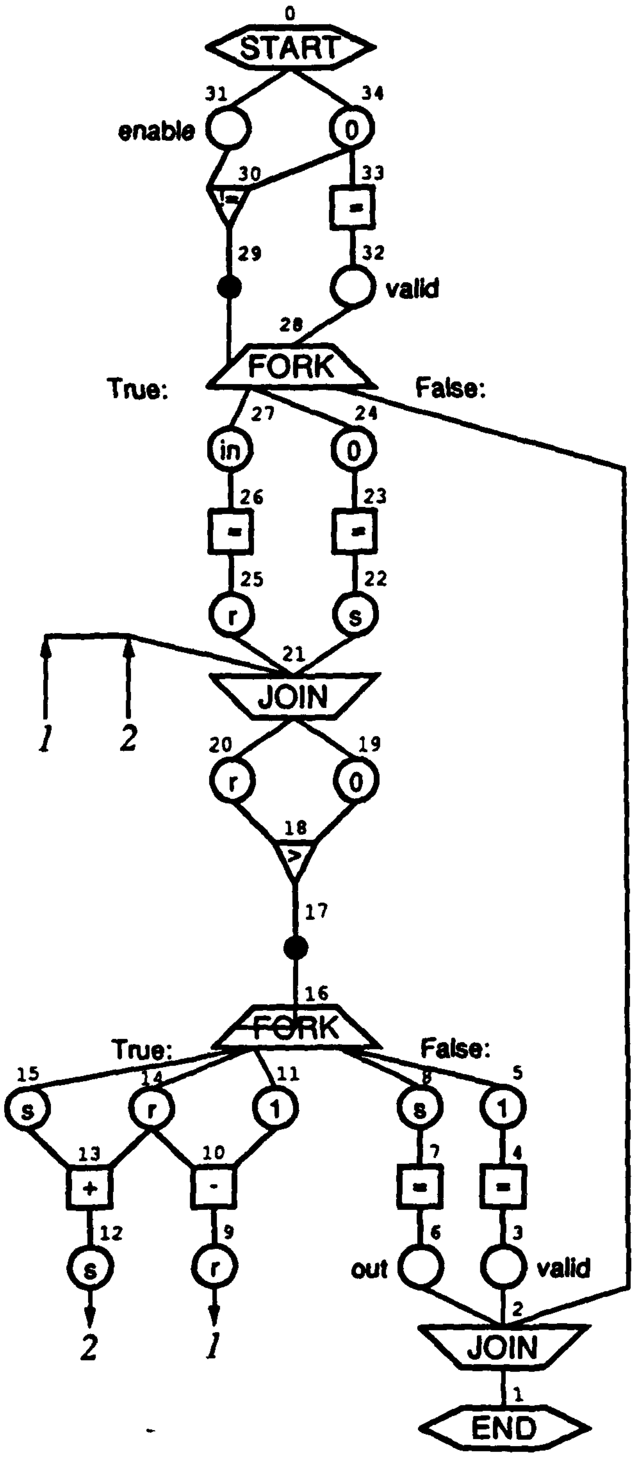

**Figure 3:** Graph representation of the procedure of Figure 1, generated by Hermod.

# 3. The Synthesis Process in Hermod

## 3.1 System Overview

Figure 4 shows an overall picture of Hermod. Taking a procedure describing the desired behavior of a design, Hermod converts it into the intermediate graph form (C/DFG), then builds the hardware realizing the behavior in two passes. In the first pass, the operations in the graph are assigned to machine states and the initial hardware is created by allocating functional modules to the operation nodes, and wires or registers to the arcs. Two graphs are produced as a result of the first pass. One is a data path graph that consists of nodes representing functional modules or data storage and edges representing interconnections. The other is a state transition diagram in which each node corresponds to a machine state and each edge shows a state transition and its conditions. In the second pass, functional modules and registers are merged, if they have no usage conflicts. Those optimization processes can be performed automatically by the system using the information on number of available modules and their functionality, or can be directed by the user through a graphical interface. As a final result, Hermod produces a netlist of the created data path and a control unit specification in the truth table format (*tt* format) for PLA descriptions [5].

## 3.2 Initial Synthesis Process

### 3.2.1 Functional Module Binding

The module binding process transforms the functional block representation provided by the data path allocator to a hardware-bound level of description [9]. In Hermod, the module binding process is embedded in data path synthesis in the initial synthesis process. During initial synthesis, no restrictions are imposed on the number of hardware modules used. The system assigns hardware modules to each node and timing-cut edge. Hardware modules are selected from a module library. Module selection specifies the type, functionality, and other attributes (such as delay, area, control setting, and io-ports) for each abstract operation. A dedicated module is assigned to each abstract operator during the initial synthesis phase. This can exploit the parallelism inherent in the original behavioral description. However, it would be desirable to share functional modules among operators to reduce the overall hardware requirements for implementation. In Hermod, the resource sharing is handles during optimization phase.

### 3.2.2 Control Step Partitioning and State Binding

When the C/DFG is generated, only control and data dependencies are represented in the graph. The state binding process partitions the graph into machine states. Each operation node in the graph is assigned to a particular state. Thus state binding determines the parallelism of the generated design.

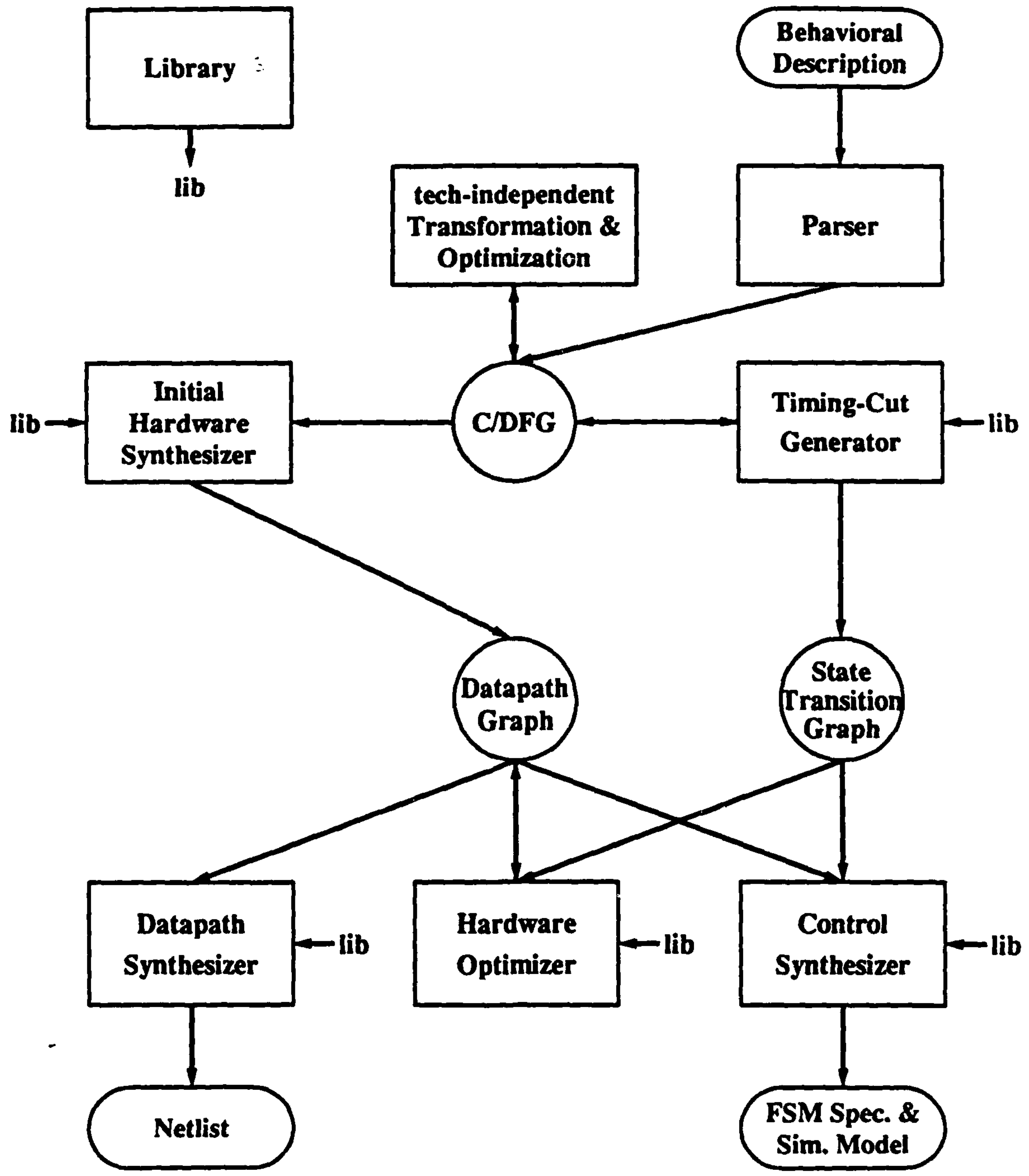

**Figure 4:** System overview.

## *A. Automatic Operation Scheduling Process*

*Timing-cuts* are used in Hermod for control step partitioning. A timing-cut is a set of edges that forms a cut-set of the subgraph corresponding to a basic block of code. A set of timing-cuts divides the graph into several data flow subgraphs each of which forms a machine state. For example, refer to Figure 5. The graph is divided into five subgraphs. Note that states 2 and 4 overlap. It is due to the loop construct in the graph. At state 2, values are assigned to the symbols r and s, which occurs when entering the while-loop. At State 4, condition variable is checked after each iteration of the loop. Timing-cuts are generated depending on the system clock period and the module delay. In this process, Hermod employs the *as soon as possible* (ASAP) scheduling algorithm that schedules each operation in a greedy fashion [16, 25]. It schedules as many operations as possible in each machine state as long as the delay does not exceed the clock period. Timing-cuts are inserted in the following two cases;

- When the delay exceeds the clock period. (Hermod supports chaining and multi-cycling [16].)
- In front of a fork/sfork node (starting node of if/switch statements). In a conditional statement, only one of the branches is executed in the next machine state. To determine which branch should be taken, the previous state must end at the fork/sfork node regardless of the execution delay. This also ~~prevents~~ eliminates the race condition of the loop constructs.

The automatic timing-cut generation process is based on the longest path search. Starting from the start node of the graph, the system calculates the maximum execution delay for each node reachable from the start node until a fork/sfork node or end node is encountered. Then, each node on the path is assigned to a particular machine state according to their maximum delay. The edges connecting operation nodes in different machine states form a timing-cut. When a fork/sfork node is encountered during the search, a timing-cut is created consisting of the edges connected to the node. Then, all the branches from the fork/sfork node are put in a *branch queue*. While the branch queue is not empty, the process retrieves the first entry of the queue and resumes the search. The search continues until another timing-cut or fork/sfork node is encountered. The state binding process determines the values to be stored for use in the next control steps. In the hardware implementation, latches are inserted in the data transfer paths where timing-cuts are generated.

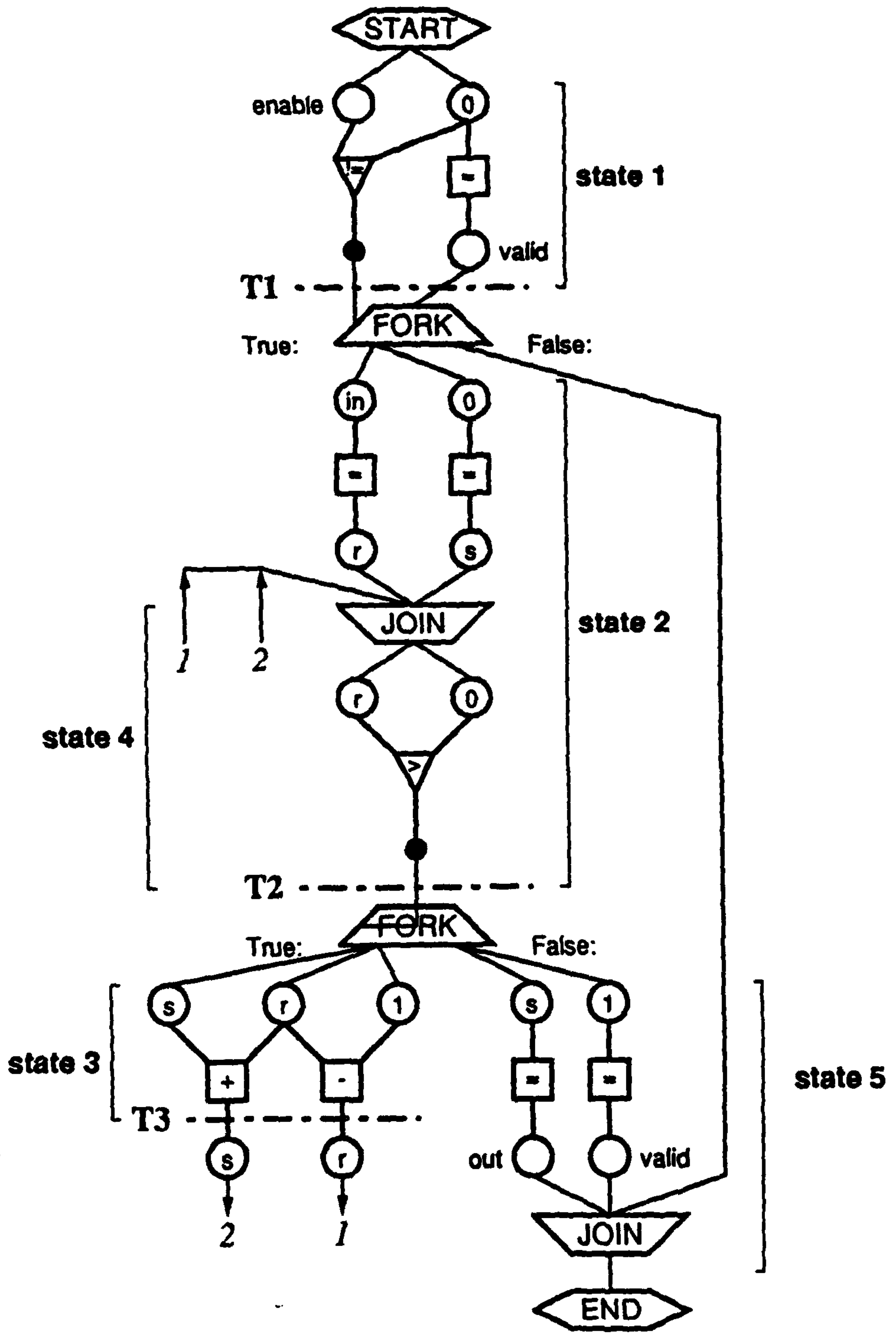

Figure 5: The intermediate graph representation with timing-cuts.

### *B. Manual Timing-cut Modifications*

To maximally utilize hardware resources, modifications are allowed on the C/DFG such as adding, deleting, and moving timing-cuts. Through a graphical interaction tool, the graph is displayed in a window and the user is allowed to pick up the edges to construct a new timing-cut, or pick up an existing timing-cut to add, delete, or move it by clicking the mouse. If the user defines a new set of edges as a timing-cut, the system checks if those edges forms the cut-set of a basic block subgraph. If the changes are legal, that is, they don't violate the behavioral specifications and design constraints, the system accepts the changes. When the user deletes or moves some timing-cuts, the clock period may be changed (become longer) and the execution delay of each state is recalculated. If the maximum delay exceeds the clock period in any state, Hermod asks the user if the clock period may be increased. Those modifications may change the state binding of some operations. Once the modifications are accepted, the system generates a new state transition diagram based on the new state binding, and the new data path.

### *C. Example*

Figure 6 shows the data path for the behavioral description in Figure 1. It is generated using the state binding of Figure 5. The design is obtained by setting the clock period to be the module delay (all the functional modules used here have the same delay). It consists of five machine states, and uses four dedicated functional modules. The registers available in the module library have only reset and load control inputs.

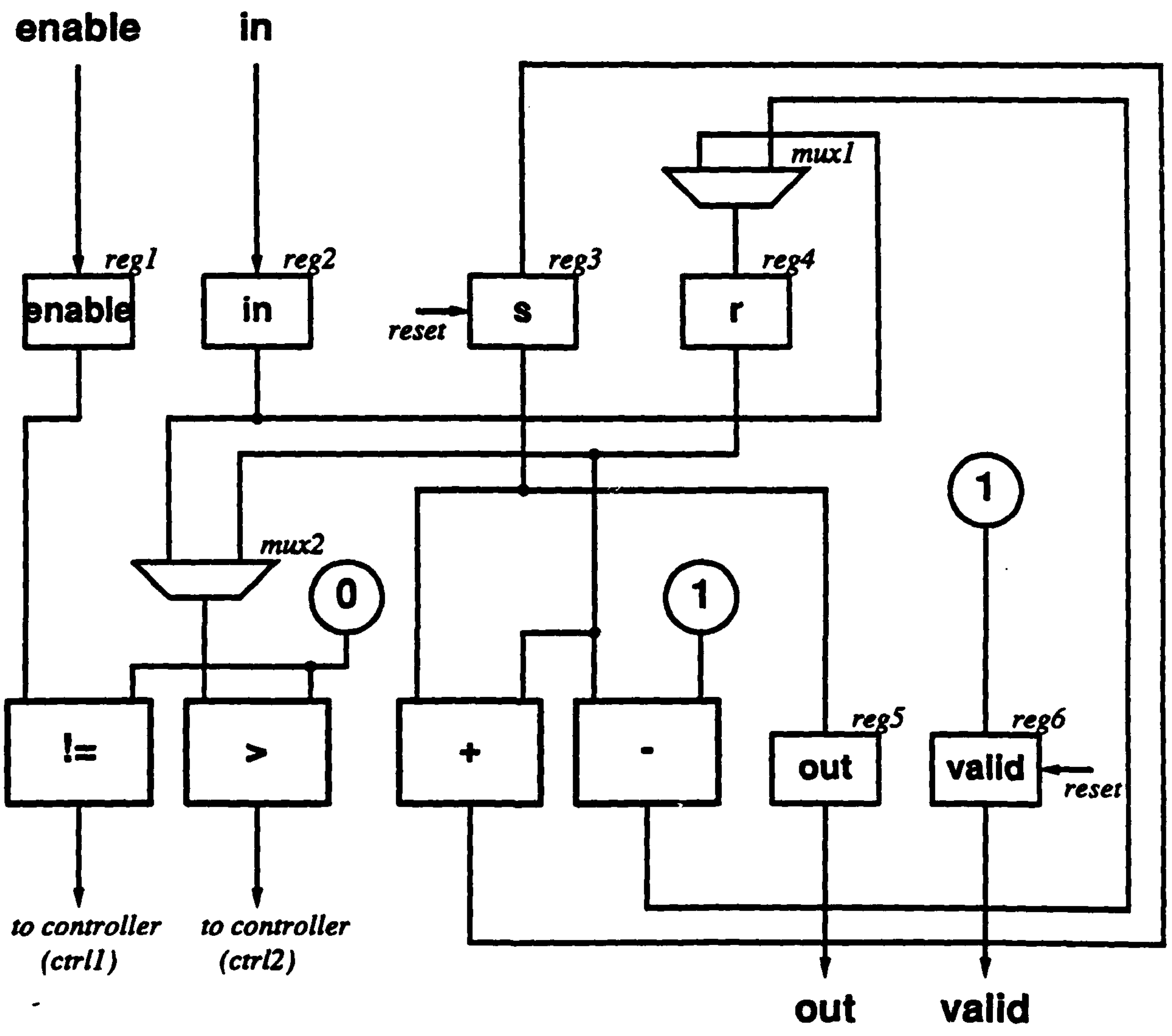

Figure 6: A hardware representation of the graph in Figure 3.

### 3.2.3 Register Binding

Registers are assigned to store the temporary results generated by the operators in a state when these results are used in the following states. The system assigns a register to each edge belonging to a timing-cut, unless the edge comes from a constant node. The system also allocates registers to the data nodes connected to timing-cut edges. The registers allocated to the same variable are merged into one in the optimization phase. Caution must be taken when the same variable appears more than once in a timing-cut edge. For example, in Figure 7, edges *e2* and *e3* are connected to the data nodes representing the same variable *a*. Since the data node connected to the edge *e3* is the last definition of the variable *a* in that state, the register associated with the edge *e3* must be associated with the variable *a*. The register associated with the edge *e2* becomes the temporary register. In order to deal with this problem, the register binding routine first determines the last definition node for each variable in each state. Then it assigns a temporary register to the timing-cut edge connected to the data node which is not the last definition node. In the optimization pass, registers allocated to different symbols that have non-overlapping lifetime are merged.

### 3.2.4 Connection Binding

Allocation of hardware resources (functional modules and registers) implies that there exist physical paths among them. These paths can be obtained by analyzing the paths between the abstract operations. The connection binding routine analyzes the connections of each machine state one by one. First, the connection binding routine extracts the data flow subgraph corresponding to the currently processed machine state. Then, it creates the connections between the functional modules and registers by tracing the edges connected to the operation and data nodes. If an operation uses inputs or produces outputs across the state boundary (e.g, timing-cut), a connection wire is created between the functional modules corresponding to the operation nodes and the register corresponding to the timing-cut edge. Each time a new connection wire is created, the system encodes the currently processed state into the wire data structure for later use during the control generation process.

To allow the maximum sharing of functional modules and registers, multiplexors/busses are created and inserted wherever necessary to transfer data between operands (registers) and operators (functional modules).

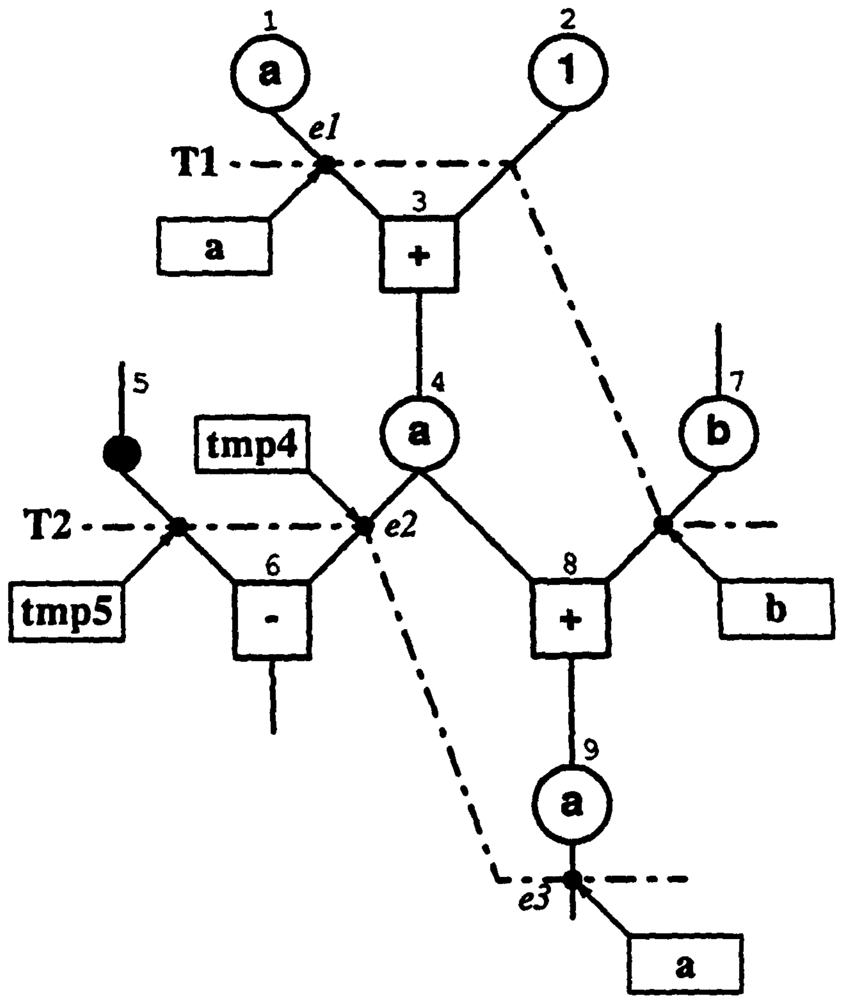

**Figure 7:** Register binding for timing-cuts.

### 3.2.5 Control Generation

Control structure is not specified in the behavioral description. Instead the control block is automatically generated from the description using the information on available resources (library modules) used in the data path synthesis process. The control block can be generated at the same time as the data path. However, it is straightforward to generate the control block after the data path structure is determined, since the timing and sequencing of the control signals are embedded in the state transition diagram and the data path graph. In other words, the control signals are determined by the state binding and resource allocation. The result of data path synthesis (module/state/connection bindings) is a symbolic microcode for control generation. The control model in Hermod is a finite state machine followed by an encoder for generation of control signals for particular data path modules. A finite state machine for the control of the data path in Figure 6 is specified in *tt* format [5] in Figure 8.

## 3.3 The Design Optimization Process

Although the hardware built in the initial synthesis pass realizes the behavior of a given description and satisfies the user-defined timing constraints, it may use more hardware resources than necessary, increasing the wiring and routing complexity. Basically, two hardware modules (functional modules, registers, or connection wires) can be merged if there are no usage conflicts among them in any machine state. Several hardware optimization procedures have been proposed based on the above principle [25]. In Hermod, optimization is performed separately for functional modules and registers in a pre-defined order. Although reducing the number of those hardware resources is the main concern of the optimization process, the numbers of interconnections (nets) and multiplexers should also be taken into account, because those interconnections usually take a large portion of the realized chip area [13]. The final optimization results including the interconnections and multiplexers depend heavily on the execution order and technique of each optimization process.

The Hermod hardware optimizer takes two inputs generated during initial synthesis, the data path graph and the state transition diagram, and produces an optimized data path. The optimizer consists of a rebinding process for each type of hardware resources: functional modules, registers, and connection wires. (The connection rebinding process is currently under development.) The rebinding processes provide menu-driven graphical user interaction tools so that the user can specify the pre-bindings for several hardware modules. The user can also unbind the old module bindings partially or totally and re-optimize the data path using the remaining bindings. The user is allowed to pick functional modules or registers by clicking mouse on them to force them to be merged or split. The user-directed bindings are checked if they have usage conflicts. Once a rebinding process is done, the system reconstructs the data path using the new binding information to rebuild the data path.

```
.i 5
.o 19
.ibl s2 s1 s0 ctrl1 ctrl2
.ol  s2 s1 s0 reg1_l reg1_r reg2_l reg2_r reg3_l reg3_r
reg4_l reg4_r reg5_l reg5_r reg6_l reg6_r mux1_ctrl mux2_ctrl

000--    0011010-1-10000--
0010-    0000000-10000-1--
0011-    0100000-10000-1--
010-0    10100001010000000
010-1    01100001010000000
011--    10000001010000011
100-0    101000000000000--
100-1    011000000000000--
101--    000000000001010--
.e
```

Figure 8: Control block in *tt* format for the data path in Figure 6.

The rebuilt data path is displayed on the screen so that the user can evaluate the automatic rebinding results.

### 3.3.1 Functional Module Rebinding

Two functional modules in a data path can be merged into one if they are not activated simultaneously in any machine state and they can be realized by a library module. (The second condition is not strict, because a library module that can execute those operations may be added later.) Let $G_c$ be the *usage compatibility graph* consisting of nodes representing the functional modules of the data path and edges connecting the functional modules that satisfy the above conditions. Then, finding the minimum number of functional modules necessary to realize the data path is reduced to clique partitioning problem of the graph $G_c$ [25].

Figure 9 shows the functional module rebinding procedure in Hermod. The procedure is based on the cluster development method.

```
Step 1: Build the usage compatibility graph $G_c$ for the data path graph.
Step 2: Determine the kernel functional modules.
Step 3: While compatibility graph $G_c$ is not empty, do the following:
    Step 3.1: Calculate the cost function for each edge of $G_c$.
    Step 3.2: Choose edge $e$ with minimal cost.
    Step 3.3: Merge two functional modules connected by $e$.
    Step 3.4: Update the graph $G_c$.
Step 4: Generate the new data path.
```

**Figure 9:** Functional module rebinding procedure.

In the procedure, *kernel* functional modules are determined first. The kernel functional modules are the modules that construct the maximum independent set of the graph $G_c$. Let N be the number of the kernel functional modules. Then, N gives the minimum number of the functional modules necessary to realize the data path. Finding the maximum independent set of the given graph is NP-complete [10]. However, the maximum independent set of the graph $G_c$ can be determined using the following heuristics. Suppose the data path is realized by library modules $L_1, L_2, \ldots, L_m$. First, count the number of functional modules realized by the library module $L_i$ in each machine state. Then, for each library module $L_i$, find out the machine state $S_i$ that requires the maximum number of functional modules realized by $L_i$. If more than one state requires the maximum number of the module $L_i$, then one is chosen arbitrarily. Finally, collect the functional modules realized by $L_i$ in $S_i$. The collected functional modules form the maximum independent set of the graph $G_c$ and become the kernel functional modules.

Once the kernel functional modules are determined, the modules other than the kernel modules are merged into one of the kernel modules one by one. The interconnections (wires and multiplexers) are taken into account when functional modules are merged. Among pairs of functional modules that can be merged (connected by edges in the usage compatibility graph), one is selected with minimal cost. The cost function for the edge connecting modules $u_i$ and $u_j$ is

$C(u_i, u_j)$
  = c1 * number of multiplexors required to merge $u_i$ and $u_j$
  + c2 * number of wires added
  - c3 * area reduction due to merging of two modules,

where c1, c2, and c3 are constants determined empirically.

After merging a pair of functional modules $u_i$ and $u_j$, the compatibility graph is updated. If either one of them is a kernel functional module, say $u_i$, then $u_j$ is removed from the graph. Then, the edge ( $u_k$, $u_i$ ) is removed from the graph, unless there was edge ( $u_k$, $u_j$ ) in the compatibility graph before merging. Figure 10 shows the results of automatic functional module rebinding on the initial design of Figure 6. Three functional modules are merged into one, and one three-input multiplexor is added due to the merging. This new configuration is reflected when generating the control block for this data path.

### 3.3.2 Register Rebinding

Two registers can be merged into one if and only if they are not simultaneously *live* at the entry and exit points of any machine state. The register allocation based on this property has been employed in behavioral synthesis systems as well as in optimizing compilers [1]. Tseng and Siewiorek reduced the register optimization problem into the clique partitioning problem [25]. The algorithm is implemented in Hermod supplemented with menu-driven interactive tools. Using the tools, the user is able to interact with the system in the register rebinding process by picking particular registers for merging. Or the user can ask the system to retract a particular merging to further explore the alternatives. Figure 11 shows the results of register rebinding on the partially optimized design of Figure 10. Two registers are saved after register rebinding for this particular example, i.e., registers in and r are merged, as are out and s registers.

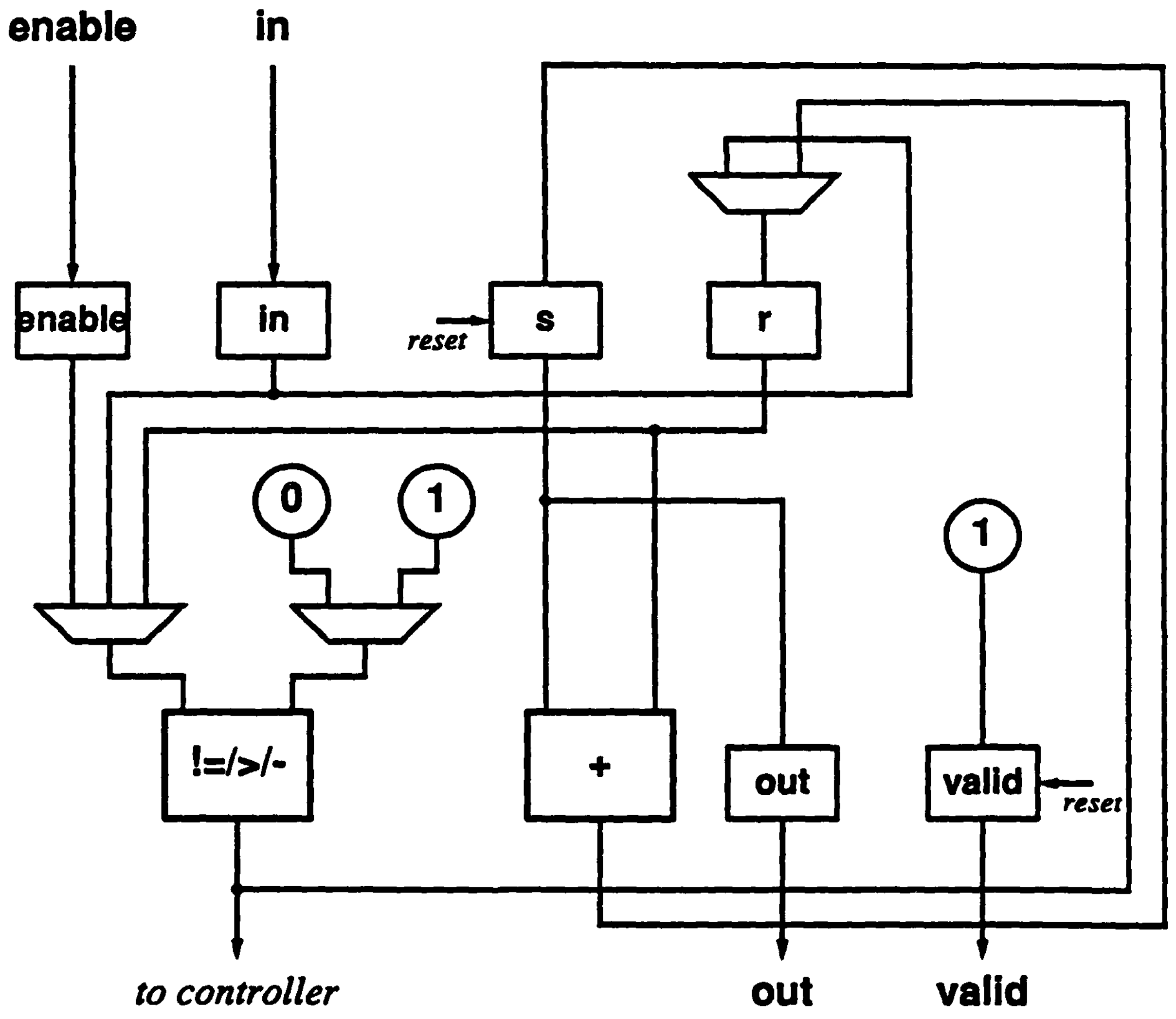

**Figure 10:** Hardware representation of the procedure in Figure 1 after functional module rebinding.

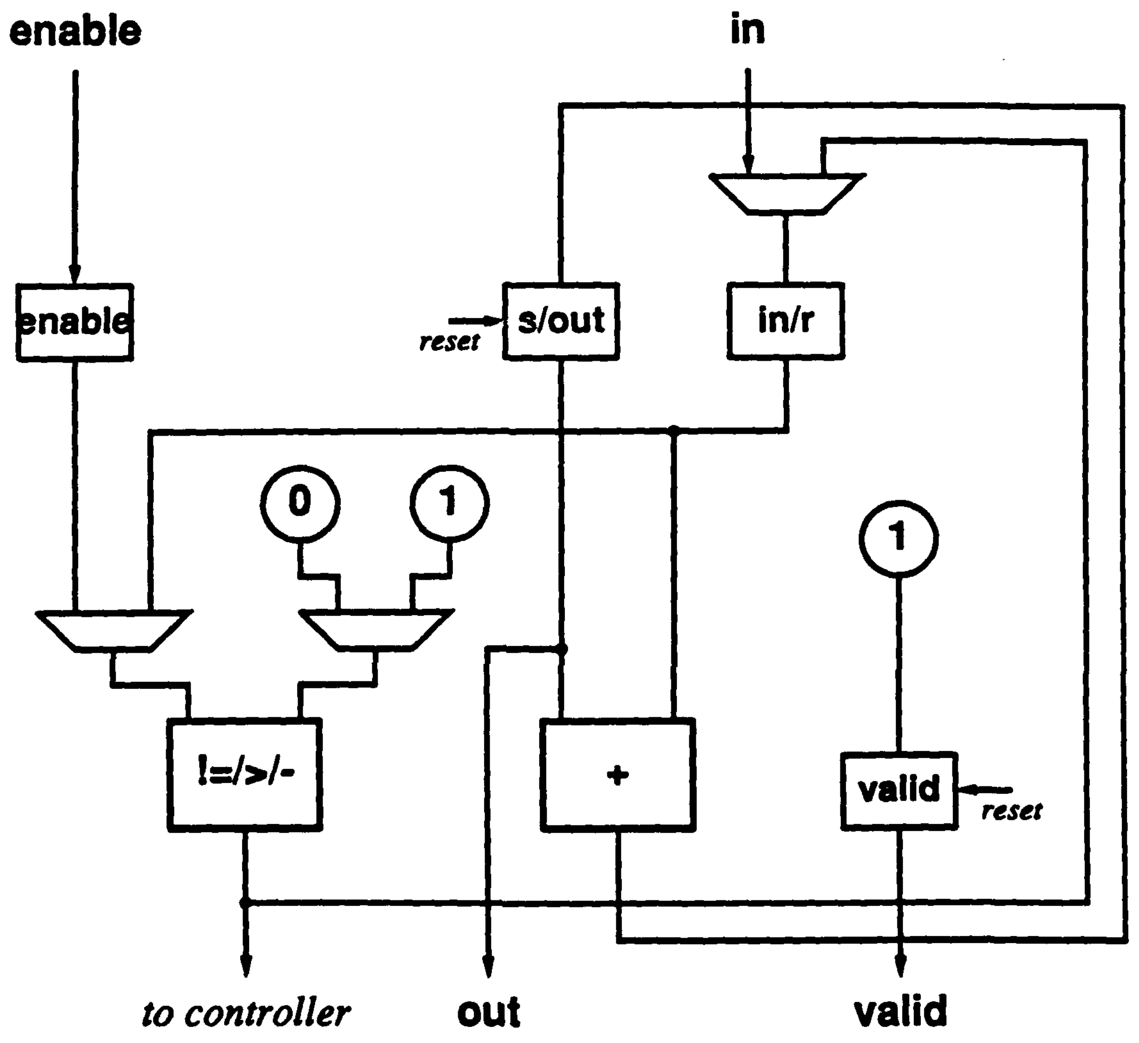

**Figure 11:** Hardware representation after register rebinding on the data path of Figure 10.

# 4. Synthesis Examples

The Hermod behavioral synthesis program is implemented in C++, an object-oriented extension of the C language, and runs on a VAXstation II/GPX under UNIX™. This section shows the synthesis results by Hermod for two CPU chips: FRISC microprocessor, a stack-based 16-bit microprocessor [22], and PDP-8 CPU [21].

## 4.1 FRISC Design

The top-level description of the FRISC processor is given in the Appendix. Here, memory read/write is simulated by the procedures *m_read* and *m_write*. One variable is used as a dummy output of the procedure *m_read*. The data path synthesis routine ignores the dummy variable and no registers are assigned to it, since it is never referenced in the main procedure.

Figure 12 demonstrates the schematic of the data path synthesized by Hermod. The design was done automatically except for slight manual modifications in state bindings to avoid memory access conflicts. In this design, four functional modules (two ALUs, one shifter, and one comparator) and six registers are used. One ALU is used to perform increment and decrement operations for stack pointer S and program counter P. The other ALU performs plus, minus and logical operations. Two registers A and B are used as operand registers for the ALU, while register B is also used as memory buffer. Register I is the instruction register, and register M is used for subroutine calls.

## 4.2 PDP-8 Design

The second example is taken from the ISP behavioral description of the PDP-8 in [21]. Figure 13 shows the data path generated by Hermod. In this design, several 1-bit inverters and gates are used, as well as 12-bit functional modules such as ALU, shifter, and comparators. Those 1-bit logical operators are employed to realize the expressions of the if-constructs. After initial data path is generated by the system, manual optimization tools are invoked to optimize the 1-bit modules in the design, because better optimization results can be obtained by considering the logical meaning and structure of the circuit. The remaining portions of the data path are optimized automatically. It took less than 20 minutes to finish the data path design using manual and automatic optimization tools in Hermod. Of course, the control FSM is automatically generated.

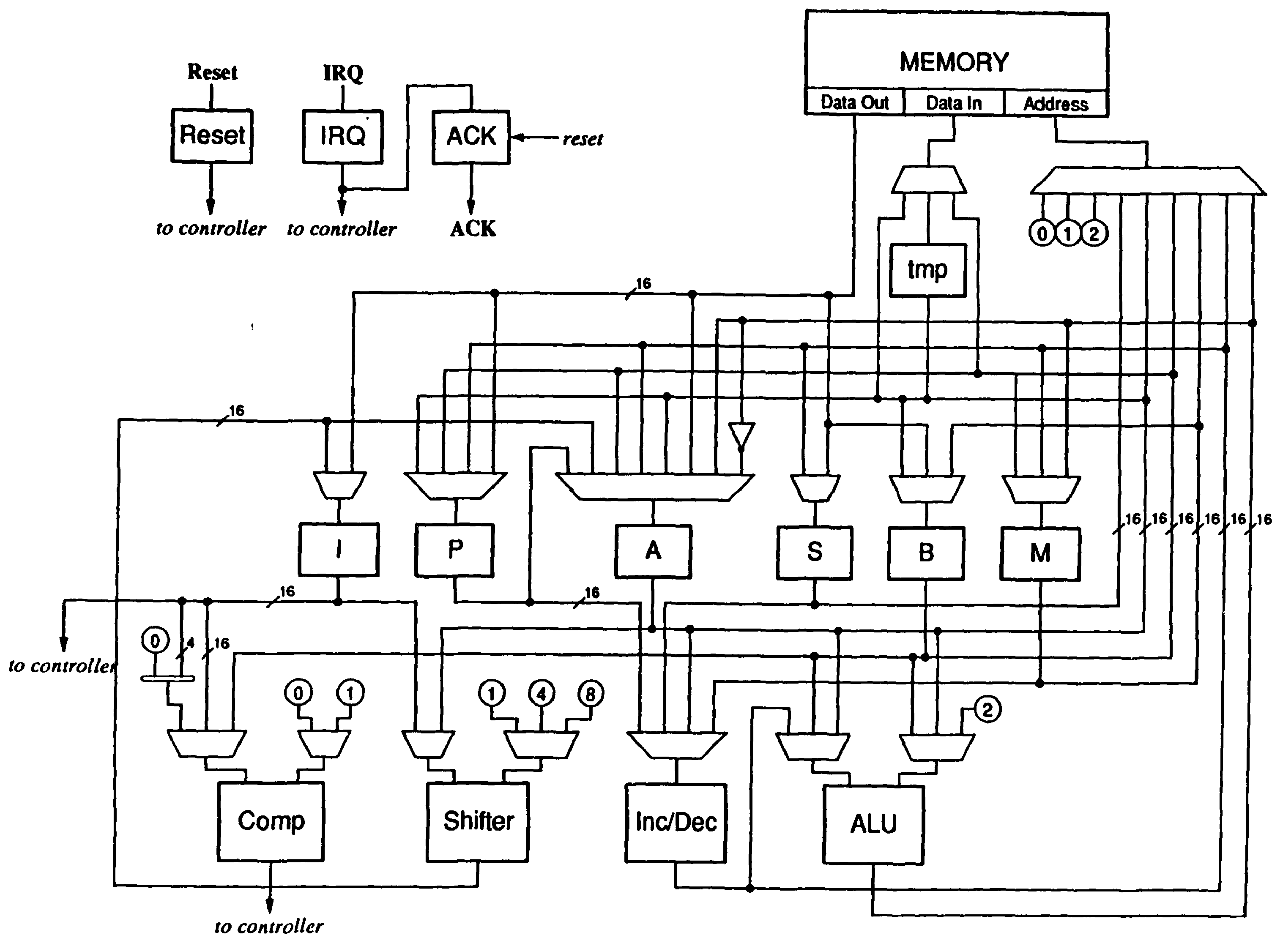

Figure 12: FRISC data path designed automatically by Hermod.

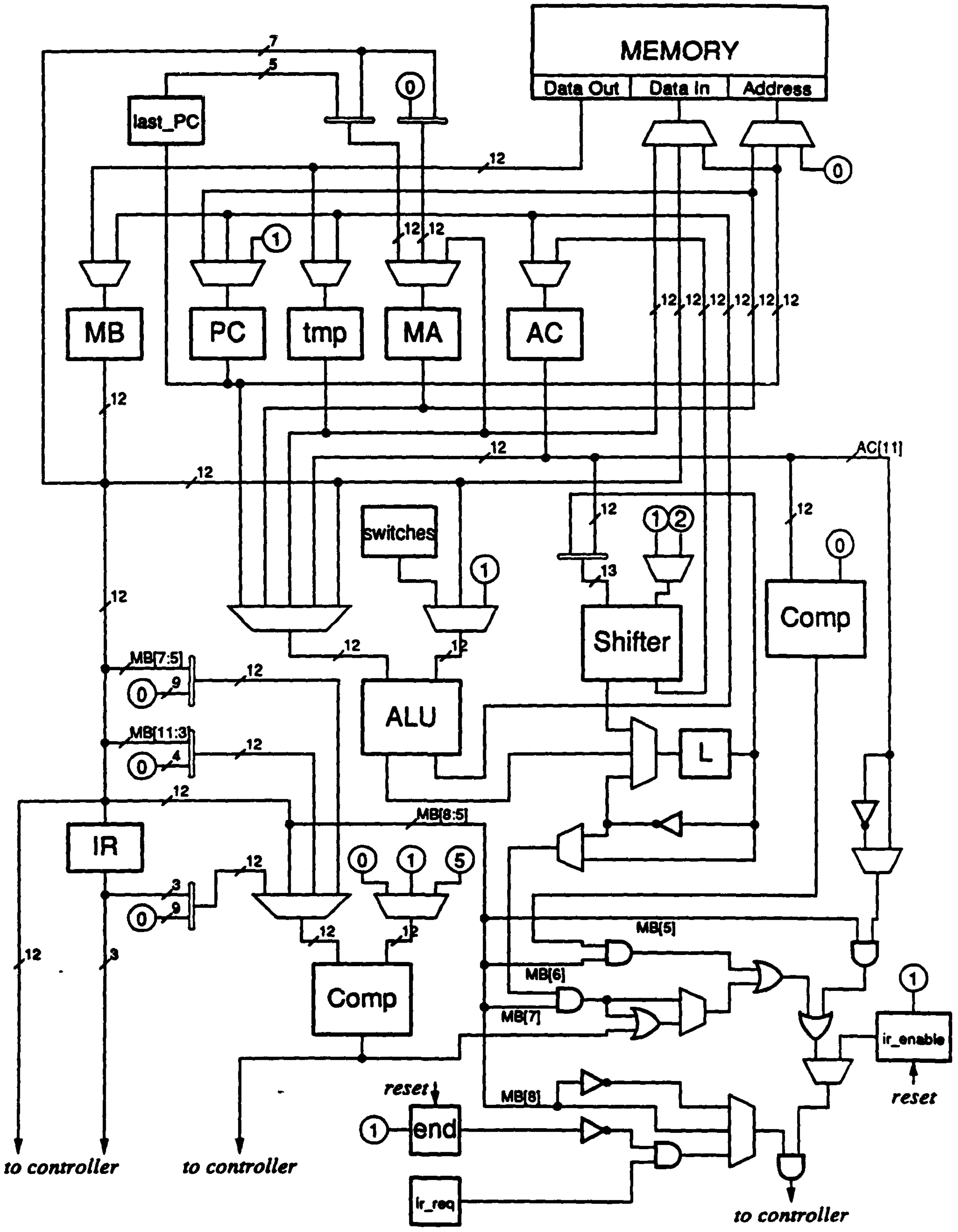

**Figure 13: PDP-8 data path designed automatically by Hermod.**

## 5. Concluding Remarks

Hermod is a behavioral synthesis system developed to provide designers an interactive environment within a graphical frame, thus to equip them with tools for direct control of synthesis process. From behavioral descriptions, Hermod generates and displays the control and data flow graph (with a set of legitimate *timing-cuts*) and its hardware representation. Unlike the other synthesis systems, Hermod allows the designer to control the synthesis process in state partitioning and resource sharing through a menu-driven graphical interface to explore the maximal design space. When the designer wants changes in a machine-generated hardware, the system checks the legitimacy of a user's request, then generates a new hardware representation that results from the changes. This system gives designers a clear view of the synthesis process, and suggests a systematic way to create and modify the designs.

Hermod provides a framework for tool development. It is simple to hook up new tools into the system to upgrade its synthesis and verification capabilities. Hermod is not intended for use with design descriptions which would require thousands of components for direct hardware realization. Instead, the system can be effectively used for design descriptions at high abstraction levels in the early stage of design, where (1) the desired behavior is normally described in a hierarchical fashion and (2) design space exploration is of primary concern. Further, it has no predefined data paths, thus does not impose any particular design style for hardware generation.

Although Hermod has some limitations in its capabilities in the current implementation, it can be extended without major modifications in the program. Future extension will include the interconnection hardware optimization and development of more tools for intelligent hardware synthesis.

## Acknowledgement

This material is based upon work supported partly by ONR/DARPA under Contract N00014-87-K-0828 and partly by Toshiba Corporation. The authors would like to thank Professor Giovanni De Micheli for his helpful discussion and suggestions. Finally, the authors are grateful to the referees for their valuable comments.

## References

[1] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1979.

[2] 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, Calif. , January 1988.

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

[4] M. R. Barbacci, "Instruction Set Processor Specifications (ISPS): The Notation and its Applications", *IEEE Trans. Computers*, Vol. C-30, No. 1, January 1981, pp. 24-40.

[5] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, MA, 1984.

[6] R. Camposano, "Synthesis Techniques for Digital Systems Design", in *Proc. 22nd Design Automation Conference*, ACM/IEEE, June 1985, pp. 475-481.

[7] G. De Micheli, "Synthesis of Control Systems", in *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, G. De Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti, (editor), Martinus Nijhoff Publishers, 1987, pp. 327-364.

[8] S. W. Director, A. C. Parker, D. P. Siewiorek, and D. E. Thomas, "A Design Methodology and Computer Aids for Digital VLSI Systems", *IEEE Trans. Circuits and Systems*, Vol. CAS-28, No. 7, July 1981, pp. 634-645.

[9] E. Dirkes, *A Module Binder for the CMU-DA System*, Technical Report CMUCAD-85-43, Carnegie-Mellon Univ., Pittsburgh, PA , May 1985.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman and Company, San Francisco, Calif., 1979.

[11] D. D. Hill, *Language and Environment for Multi-Level Simulation*, Technical Report 185, Stanford University, Stanford, Calif. , March 1980.

[12] M. C. McFarland, *The Value Trace: A Data Base for Automated Digital Design*, Master's Thesis, Carnegie-Mellon Univ., Pittsburgh, PA, December 1978.

[13] M. C. McFarland, "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions", in *Proc. 23rd Design Automation Conference*, ACM/IEEE, June 1986, pp. 474-480.

[14] A. W. Nagle, R. Cloutier, and A. C. Parker, "Synthesis of Hardware for the Control of Digital Systems", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems.*, Vol. CAD-1, No. 4, October 1982, pp. 201-212.

[15] M. Odani, S. Y. Hwang, T. Blank, and T. Rokicki, *The ILSP Behavioral Description*

*Language and its Graph Representation for Behavioral Synthesis*, Technical Report CSL-TR-88-350, Stanford University, Stanford, Calif., March 1988.

[16] B. M. Pangrle and D. D. Gajski, "Design Tools for Intelligent Silicon Compilation", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems.*, Vol. CAD-6, No. 6, November 1987, pp. 1098-1112.

[17] N. Park and A. Parker, "Sehwa: A Program for Synthesis of Pipelines", in *Proc. 23rd Design Automation Conference*, ACM/IEEE, June 1986, pp. 454-460.

[18] A. C. Parker, "Automated Synthesis of Digital Systems", *IEEE Design and Test of Computers*, Vol. 1, No. 4, November 1984, pp. 75-81.

[19] A. C. Parker and S. Hayati, "Automating the VLSI Design Process Using Expert Systems and Silicon Compilation", *Proceedings of IEEE*, Vol. 75, No. 6, June 1987, pp. 777-785.

[20] P. G. Paulin, J. P. Knight, and E. F. Girczyc, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis", in *Proc. 23rd Design Automation Conference*, ACM/IEEE, June 1986, pp. 263-270.

[21] D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, NY, 1982.

[22] J. R. Southard, "MacPitts: An Approach to Silicon Compilation", *IEEE Computer*, Vol. 16, No. 12, December 1983, pp. 74-82.

[23] H. Trickey, "Flamel: A High Level Hardware Compiler", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems.*, Vol. CAD-6, No. 2, March 1987, pp. 259-269.

[24] C. J. Tseng, A. M. Prabhu, C. Li, Z. Mehmood, and M. M. Tong, "A Versatile Finite State Machine Synthesizer", in *Proc. Int. Conf. Computer-Aided Design*, IEEE, November 1986, pp. 206-209.

[25] C. J. Tseng and D. P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems.*, Vol. CAD-5, No. 3, July 1986, pp. 379-395.

[26] R. Waxman, "Hardware Design Languages for Computer Design and Test", *IEEE Computer*, Vol. 19, No. 4, April 1986, pp. 90-97.

## A. Behavioral Description of FRISC

```
/*
 *    FRISC - 16-bit Microprocessor
 */

#define CPUSIZE 16

frisc()
{
    IN_LIST
        SIG ( RESET ) ;
        SIG ( IRQ ) ;
    ENDLIST ;

    OUT_LIST
        SIG ( IACK ) ;
    ENDLIST ;

    ST_LIST
        GRP ( P , CPUSIZE ) ;
        GRP ( S , CPUSIZE ) ;
        GRP ( M , CPUSIZE ) ;
        GRP ( A , CPUSIZE ) ;
        GRP ( B , CPUSIZE ) ;
        GRP ( I , CPUSIZE ) ;
    ENDLIST ;
    int dum , dum2 , T ;                    /* local variables */

    IACK = IRQ;

    if (RESET)
    {
        P[] = m_read ( 0 ) ;
        S[] = m_read ( 1 ) ;
    }
    else if (IRQ)                           /* Interrupt request? */
    {
        S[] = S[] + 1;
        dum = m_write ( S , B ) ;
        S[] = S[] + 1;
        dum2 = m_write ( S , A ) ;
        B[] = M[];
        A[] = P[];
        M[] = S[] + 1;
        P[] = m_read ( 2 ) ;
        IACK  = 0;
    }

    I[] = m_read ( P ) ;
```

```
P[] = P[] + 1;                                  /* Increment program counter */

while( I[] )
{     /* Until no opcodes left in buffer, decode ops. */
    if (I[3:0] == 1)
    {
      switch (I[7:4]) {
        case 0:                                 /* Nand */
          A[] = ~(A[] & B[]);
          B[] = m_read ( S );
          S[] = S[] - 1;
          break;
        case 1:                                 /* Subtract */
          A[] = B[] - A[];
          B[] = m_read ( S );
          S[] = S[] - 1;
          break;
        case 2:                                 /* Shift right */
          A[] = A[] >> 1;
          break;
      }
      I[] = I[] << 8;                           /* Shift out opcode */
    }
    else                                        /* Normal op code */
    {
      switch (I[4:0]) {
        case 2:                                 /* Constant */
          S[] = S[] + 1;
          dum = m_write ( S , B );
          B[] = A[];
          A[] = m_read ( P );
          P[] = P[] + 1;
          break;
        case 3:                                 /* Get S */
          S[] = S[] + 1;
          dum = m_write ( S , B );
          B[] = A[];
          A[] = S[];
          break;
        case 4:                                 /* Set S */
          S[] = A[];
          B[] = m_read ( S );
          S[] = S[] - 1;
          A[] = B[];
          B[] = m_read ( S );
          S[] = S[] - 1;
          break;
        case 5:                                 /* Get M */
          S[] = S[] + 1;
          dum = m_write ( S , B );
        B[] = A[];
        A[] = M[];
        break;
    case 6:                              /* Load */
        A[] = m_read ( A ) ;
        break;
    case 7:                              /* Store */
        dum = m_write ( B , A ) ;
        A[] = B[];
        B[] = m_read ( S ) ;
        S[] = S[] - 1;
        break;
    case 8:                              /* Go to */
        P[] = A[];
        A[] = B[];
        B[] = m_read ( S ) ;
        S[] = S[] - 1;
        break;
    case 9:                              /* If */
        if ( B[] > 0 )
            P[] = A[];
        A[] = B[];
        B[] = m_read ( S ) ;
        S[] = S[] - 1;
        break;
    case 10:                             /* End */
        A[] = B[];
        B[] = m_read ( S ) ;
        S[] = S[] - 1;
        break;
    case 11:                             /* Mark */
        S[] = S[] + 1;
        dum = m_write ( S , B ) ;
        B[] = A[] ;
        A[] = M[] ;
        M[] = S[] + 2;
        break;
    case 12:                             /* Call */
        T = P[];
        P[] = A[];
        A[] = T;
        break;
    case 13:                             /* Return */
        P[] = B[];
        S[] = M[];
        B[] = m_read ( S ) ;
        S[] = S[] - 1;
        M[] = B[];
        B[] = m_read ( S ) ;
        S[] = S[] - 1;
```

```
                  break;
               case 14:                                /* Add */
                  A[] = B[] + A[];
                  B[] = m_read ( S ) ;
                  S[] = S[] - 1;
                  break;
               case 15:                                /* Increment */
                  A[] = A[] + 1;
                  break;
               default :
                  break ;
            }
            I[] = I[] << 4;                            /* Shift out opcode */
         }
      }
      return ;
}
```