Functional Synthesis of Digital Systems with TASS

Said Amellal and Bozena Kaminska

Abstract— **Synthesizing a digital system from a functional description is a complex process requiring the solution of various different problems. TASS (Tabu Search Synthesis System) is a functional synthesis system made up of interdependent modules based on new and more efficient algorithms. First, a control and data flow graph model for system representation is developed and presented. This model generates a single graph representing both the data and control flows of a VHDL behavioral description. A new representation of conditional branches and a mutual exclusion testing procedure offering optimized resource sharing and critical path reduction possibilities have been developed. This graph model is an environment used for various synthesis needs starting from high-level transformations to FSM synthesis for controller implementation. A new mathematical formulation of the scheduling problem is developed using a new approach based on penalty weights. This approach avoids the inflexibility of the ILP formulations developed in related works where the functional unit performing each type of operation is fixed prior to scheduling. The Tabu Search technique, which has been effective in finding optimal solutions for many types of large and difficult combinatorial optimization problems, has been adapted for this purpose. This technique, which performs an intelligent and fast solution space exploration, combined with an effective mathematical formulation makes the scheduling algorithm presented here very powerful.**

I. Introduction

THE FUNCTIONAL SYNTHESIS of a digital system is the realization of a register transfer level (RTL) description from the functional specification of the system. This process is also called *high-level synthesis* (HLS) because the system is synthesized from a high-level (algorithmic) description, or *architectural synthesis* because the process consists in synthesizing an architecture of a system from a description of the functions to be performed. The functional specification (or *behavioral description*) describes the function to be performed by the system in an algorithmic form. This is done using a hardware description language (HDL) like VHDL, ISPS of Hardware C. The RTL structure produced by an HLS tool is a set of interconnected components that realize the specified behavior of the system. This structure includes storage units, functional units, multiplexers and/or buses, as well as hardware to control the operations and data transfers between these units. The interconnections of these hardware units form a network called a *data path*. The control hardware forms the *controller*, which is usually synthesized as a finite-state machine (FSM). In Fig. 1(a) is the behavioral description of the modulo-3 divider example taken from [31], (b) is the control unit and (c) is the data path implementation.

Manuscript received June 23, 1993; revised November 29, 1993. This paper was recommended by Associate Editor R. Camposano.

The authors are with Ecole Polytechnique de Montréal, Electrical Engineering Department, Montréal, PQ Canada H3C 3A7.

IEEE Log Number 9215138.

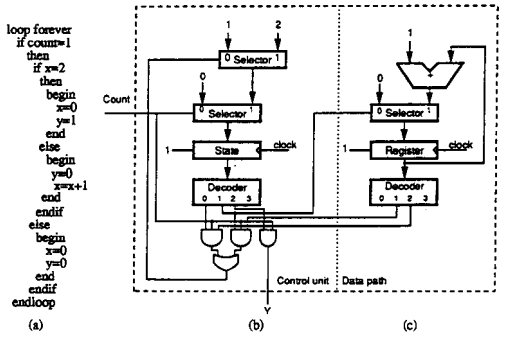

Fig. 1. Data path and controller for the modulo-3 divider example.

The first step in functional synthesis is to translate the algorithmic description into an internal representation which is usually a graph-based representation where the data flow and the control flow implied by the specification are represented. The representation using a single graph where both the control flow and the data flow are implemented is called a *control and data flow graph* (CDFG). A second type of representation shows the data flow and the control flow in separate graphs called the *data flow graph* (DFG) and the *control flow graph* (CFG) respectively. The type of representation used has an important impact on the final design. At this level, some compiler-like optimizations, called high-level transformation [31], can be performed.

The next two steps are the most important in the synthesis process: scheduling and allocation. In scheduling, each operation of the CDFG is assigned to a control step where it is executed. A *control step*, also called a *time partition*, is the sequencing unit in a synchronous system and also corresponds to a clock cycle. Allocation consists in assigning functional units to operations, storage units (registers and register files) to values, and buses and/or multiplexers to data transfers derived from the control and data flows. Finally, based on the way that operations are scheduled and the hardware units allocated, a control unit is synthesized to synchronize the execution of operations.

Many different scheduling and allocation techniques have been developed. The performance of a scheduling and/or allocation algorithm can be evaluated using the following interdependent quality metrics:

- The quality of the solution produced; an optimal or a suboptimal design.
- The complexity of the algorithm; the CPU runtime.
- The solution space exploration.
- The possibility of handling large applications efficiently.

0278-0070/94$04.00 © 1994 IEEE

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.

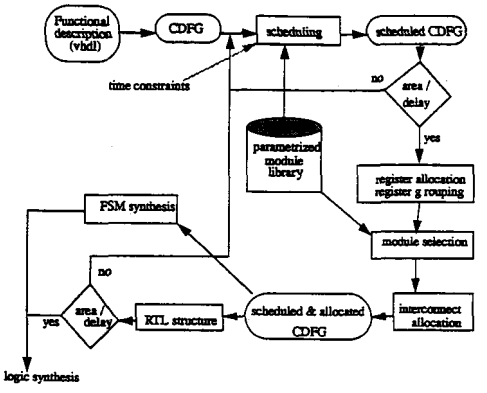

Fig. 2. Synthesis process with TASS.

- The controllability of the synthesis process through designer constraints: area, delay, design rules, power consumption, testability, etc.
- The possibility of predicting, with a maximum degree of accuracy, the previous parameters at a high level.

Most of the exiting synthesis systems satisfy some, but not all, of these requirements. This will be illustrated with examples in the following sections.

TASS is a synthesis system starting from a VHDL functional specification that produces an RTL structure. TASS is derived from the TAbu search Synthesis System. The Tabu Search optimization technique, [11] and [12], has been adapted and used in TASS. The synthesis process with TASS is described in Fig. 2. The functionality of the circuit to be synthesized is described using a VHDL subset [28]. After compilation, this description is translated into a graph-based representation where both the control flow and the data flow are implemented in a single graph. We obtain a control and data flow graph that is used for various synthesis needs (Fig. 3). This representation is very flexible. Design constraints concerning delay, timing or hardware to be used can be incorporated, and some high-level transformations are applied directly to this graph. A control and data flow graph model has been developed for this purpose (Section II). The goal in developing this model was to perform an efficient implementation of the control flow, which makes the model very powerful for synthesizing with control-dominated applications. With TASS, time-constrained scheduling is performed using a scheduling algorithm based on a recent mathematical optimization technique called Tabu Search (Section III).

The mathematical formulation of the scheduling problem is developed using a new approach which overcomes some insufficiencies of the previous algorithms (Section III). Efficient and fast solution space exploration is performed using the Tabu Search technique adapted to the scheduling problem. When a satisfactory schedule has been produced, hardware (storage, functional units and interconnects) is allocated to the graph. A scheduled and allocated graph is thus obtained, from which an RTL structure is extracted, and an FSM synthesized in order to generate the controller.

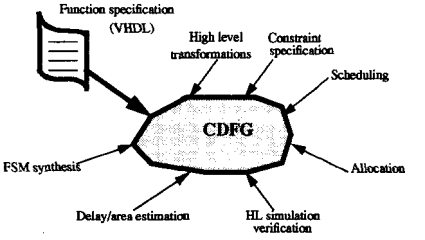

Fig. 3. The CDFG environment.

TASS has the following characteristics which make it different from existing synthesis systems and more powerful:

- The system representation is based on a CDFG model where the control flow is implemented using more powerful procedures than existing ones. This is shown in Section II.
- TASS uses a single CDFG for different synthesis needs. This allows an optimized, efficient and flexible representation, resulting in a low-synthesis CPU runtime, as well as an optimized design due to the availability of global informations in a single graph.
- The scheduling algorithm uses a new mathematical formulation which is solved using the Tabu Search (TS) technique adapted to this problem. Tabu Search is known as an efficient optimization technique that has been adapted and used successfully with a various classical optimization problems [13].

Many experimentations have been performed with TASS. Large benchmarks have been used (Section IV). These experiments have shown that TASS gives optimized designs and it is better than existing synthesis algorithms, especially with applications including a great deal of control, like processors, controllers, decoders, etc.

The next section presents the control and data flow graph model developed and used by the TASS system. The scheduling algorithm exploiting the advantages of this model is presented in Section III. Section IV includes some experimental results obtained with TASS using HLS benchmarks, in order to demonstrate the efficiency of the TASS system.

II. CDFG Model for Behavioral Description

2.1. Introduction

High-level synthesis of digital systems is a process requiring the solution of various different problems. The research is focused on producing a Register Transfer Level (RTL) structure from a behavioral description. Many published papers have addressed the main tasks of behavioral synthesis, which are operation scheduling and resource allocation. However, important problems appear when synthesizing with practical applications. To overcome this difficulty, we propose a complete solution starting with a behavioral specification for achieving an RTL structure. In this section, a Control and Data Flow Graph (CDFG) model for the internal representation of system behavior is presented. Different internal representations

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.

are used in related works and there is no concensus on how to represent internally the system behavior. Our CDFG model offers a solution to this problem since it constitutes a simple and effective representation when used with either operation-dominated applications or control-dominated applications. In fact, the control constructs are converted into graph blocks and can easily be handled for synthesis purposes, as well as for behavioral verification.

Many of the earliest synthesis systems use separate data and control flow representations (examples are [35], [36]). A graph is generated for the data flow, and conditional constructs, loops and procedure calls are represented in a separate control graph. This involves a high degree of redundancy, and, although relatively easy to implement, is more complex to handle. Furthermore, this representation limits the synthesis algorithms (graph optimization, scheduling, allocation) in their search space, increasing the risk of staying with suboptimal solutions.

The graph representations combining both the data and the control flows are used in [1]–[3]. In [1], the control dependencies are expressed, but only the data dependencies are taken into account, and the control flow is not exploited. The loops are represented using branch and merge nodes. A branch-merge loop construct is made for each variable occurring in the loop (test and body). This results in a complex subgraph with a high degree of redundancy of branch and merge nodes. The same representation is used in [2]. The Sprite Input Language (SIL) [3] uses a single signal flow graph. This language is more confined to DSP applications.

The common deficiency of the above-mentioned representations is that they do not exploit the possibilities of the potential resource sharing between conditional operations and the critical path minimization with conditional branches.

In our model, the algorithmic hardware description is made using a VHDL subset. This language is gaining wide acceptance in the hardware description community as a standardized platform for hardware descriptions. Since VHDL is not designed for synthesis purposes exclusively, a subset is defined [28] that excludes some constructs that are used only for simulation and others that are hard to synthesize or cannot be synthesized at all. Examples are recursive subprograms, I/O file handling, the synchronization of operations used in simulation, etc.

The control and data flow graph generated by our model makes up a data environment used by different synthesis procedure (Fig. 3). The main characteristics of our CDFG model are the following:

- The data and control flows are represented in the same graph. Consequently, both the data path and the controller can be synthesized using the same graph.
- The representation of the control flow due to the control constructs is simple and the whole graph is easily generated automatically.
- Combining data flow and control flow in the same graph allows the generation of an optimized CDFG, eliminating the extra dependency representations (Section 2.5).
- The CDFG model performs the representation of a VHDL subset behavioral description without imposing any restrictions on the scheduling task except the data and control dependencies needed for correct execution of the graph. This results in better exploration of the design space.
- The problem of resource sharing among mutually exclusive operations of a CDFG with conditional branches is solved in this model. A numbering procedure for branches in the graph and a function testing mutual exclusion are developed. It is shown (Section 2.3) that these procedures are more efficient than the previous ones.
- Once the behavioral description with VHDL has been converted into a CDFG representation, the graph can be extended by explicitly incorporating the designer constraints into the graph. The constraints concern either the delay of a node or a resource to be used.

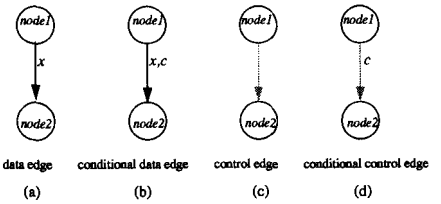

Fig. 4. Edge types.

The next section includes the basic definitions of the elements of the CDFG model. Section 2.3 addresses one of the most important problems in graph representation for synthesis: representation of the conditional branches and mutual exclusion testing. The next section deals with the loop construct implementation. Section 2.5 explains how the redundancy of dependency representations are avoided in order to generate a simple and optimal graph that is easy to handle with the synthesis tool.

2.2. Basic Concepts

The control and data flow graph is a graph $G = (V, E)$, where V is the set of nodes and E the set of directed edges. A node from the set $V = \{v1, v2, \cdots, vn\}$ can be either an operation node like $+, -, *$ or a control node such as *branch, loop, endloop*.

An edge represents the transfer of value from one node to another or a control involving an execution order between two nodes. Thus, two types of edges are used: *the data dependency edge* and *the control dependency edge*. We have introduced *conditional edges* where the data or the control dependency is taken into account only if the associated condition is true. In this case, the edge is weighted by this condition. The conditional dependency is used in conditional and iterative blocks (Sections 2.3 and 2.4). Fig. 4 shows the representations of different types of data and control edges. *Node 2* in Fig. 4(a), uses the result x of *node 1*. In Fig. 4(b), *node 2* is a conditional successor of *node 1* ; *node 2* has to wait for the result of *node 1* only if condition c is true. In Fig. 4(c), *node 2* can be executed only if *node 1* is completed. In Fig. 4(d), *node 2* has to wait for the execution of *node 1* only if condition c is true. It is obvious that if a node is using the result of another node, the last one has to provide a control for the first one. So, a data

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.

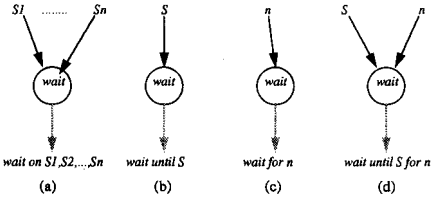

Fig. 5. Representations of different forms of the *wait* statement.

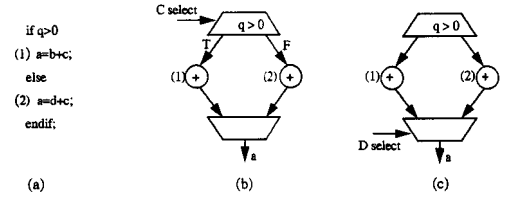

Fig. 6. Conditional branch representation types.

dependency also includes an implicit control dependency. A control dependency is then dominated by a data dependency.

The model uses the token passing semantic [37]. The data and control flow is modeled as a token passing. The execution of a node is completed in two phases: the consuming of the tokens available at the incoming edges and the generation of a unique token whose instances are propagated through the outgoing edges. A token available at an incoming edge models the availability of a value (or control) provided by the origin node. Depending on the edge type (data or control dependency), the presence of token means the availability of an instance of the value needed or the completion of the predecessor node. A node can be executed as soon as a token is available at each incoming edge. It is obvious that if the incoming edge is conditional, the token is needed only if the associated condition is true.

In order to keep the number of executed instances of a node (corresponding to the number of times the incoming tokens are consumed by the node), a token index (ti parameter) is associated with each node. This parameter is used in cyclic blocks to implement the loop constructs.

2.2.1 A Graph Node: The basic types of nodes used in the graph are discussed. They belong to four different classes: operation nodes, input/output nodes, synchronization nodes and, finally, control nodes.

The operations can be arithmetic, like $+, -, *, /, ++$, Boolean, like *And, Or, Nor, Xor*, relational, like $>, <, \leq$, or can be extended to a more complex function. In the last case, the execution of the node might involve the execution of a subgraph, and is used to represent the procedure call in the behavioral description. Each node holding an operation of a certain type might have a different execution delay depending on how the node is scheduled and the hardware used to perform the specific type of operation. The CDFG model does not impose any restriction on the scheduling algorithm except the precedence relationships specified for correct execution of the graph.

The input operation is accomplished by a *get* node and the output by a *put* node. The *get* node uses an input port and the value received is considered to be the result of the operation and the *put* node uses an output port and the value transmitted is considered to be the input of the operation.

Synchronization nodes are used to represent the various forms of the VDHL *wait* statement: "*wait on signal-list*," "*wait until condition* " and "*wait for time-expression* ". In the last case, the time expression is converted into an equivalent number of clock cycles. A mixture of the last two *wait* constructs is allowed (for example: *wait until* s *for* 10 ns). The parameters of the *wait* statements are considered as the inputs for the *wait* node, and all the subsequent nodes in the graph are synchronized after the *wait* node by means of control edges. Note that depending on the edges already created, only a few control edges have to be added in order to implement the *wait* semantic (Section 2.5). Fig. 5 shows the representations of the various forms of the *wait* statement.

The control nodes are used to represent the control flow involved in the use of conditional branches and loops. The *if/case* statement is translated into a *branch* node and the *endif/endcase* into a *merge* node. Details on the semantics of these nodes are given in Section 2.3. A loop is represented using a *loop* and *endloop* nodes combined with the conditional branch nodes. Section 2.4 gives details on the implementation of loop constructs.

2.2.2 An Object: There are three classes of objects handled by the nodes: variables, signals and constants. The variable types supported are scalar types, bit and bit vectors and enumeration types (predefined or user-defined). A signal can be either internal to the process or associated with an external port.

2.3. Conditional Branches and Mutual Exclusion Testing

In this section, two problems are discussed: conditional branch representation and mutual exclusive testing. The modeling of the conditional branches (CB's) has a significant effect on the quality of the final design. A good representation should contribute to the optimization of the area and the delay of the design. For the most part, this representation has to provide the possibility of exploiting potential resource sharing between mutually exclusive operations, as well as to reduce the critical path.

2.3.1 Representation of Conditional Branches: Existing representations are classified into two types [32]: the control select (C-select) representation and the data select (D-select) representation. In the C-select representation (Fig. 6(b)), the condition evaluation is performed first, then the right CB is selected. This representation has the advantage of allowing resource sharing between mutually exclusive operations. In the example in Fig. 6(b), an adder can be allocated for both operations, (1) and (2), because they are never both executed. However, if we suppose that there is an adder available before condition evaluation, then this functional unit cannot be used to execute (1) and (2) in order to reduce the overall delay, because the value of the condition is not yet known. In the D-select representation, all the conditional branches

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.

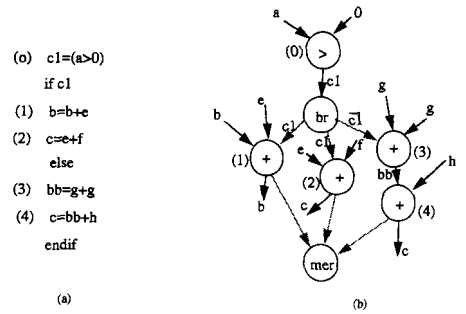

Fig. 7. Conditional branch representation.

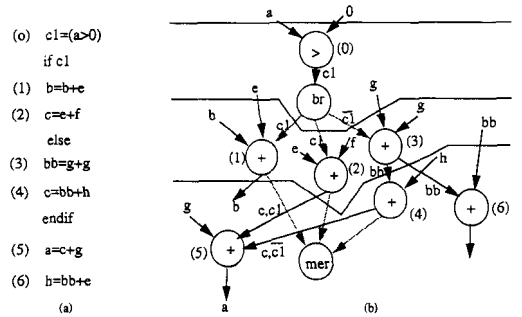

Fig. 8. Selection of a value from a CB.

are executed separately in parallel and correct data values are selected at the end of the conditional branches. The conditional operations can be executed before the condition is evaluated. This representation may sometimes result in a fast execution. Suppose that operation (2) in Fig. 6(c) is a subtraction and there is no existing functional unit performing both (1) and (2), then the two operations are executed in parallel by separate units assumed to have been allocated already. This may reduce the critical path. However, when the two operations are of the same type (as in Fig. 6(c)), two adders are required because they are not considered mutually exclusive and cannot share hardware.

The conclusion is that the C-select representation may be preferable to the second representation in certain cases, but in other cases, the use of the D-select form may result in a faster and cheaper design. So, in order to produce a good design, the scheduler must have the possibility to make a tradeoff between the two forms of representations. Additional advantages of this tradeoff are illustrated in [4] and [6].

A well-known representation of the CB is the *Distribute-Join* representation, which is of the C-select type used in the ADAM system [33]. However, there is no possible tradeoff between the C-select and the D-select forms. In the CMUDA system [34], transformations between C-select and D-select are allowed, but the dynamic tradeoff during scheduling is not possible. A recent and more interesting representation is presented in [4]. Each conditional path starts with a single conditional branch-begin node (CB node) and finishes with its own conditional branch-end node (CE node). The tradeoff between the C-select and D-select forms is possible when scheduling. However, a CE node has to be added for each value being transferred out of a conditional branch, which may result in a complex graph representation.

In our representation, the conditional branches are represented using *branch/merge* nodes that are only delimiters and do not involve any delay constraints. The dynamic tradeoff between the C-select and the D-select forms is provided for our scheduling algorithm using the conditional precedence edges introduced above. This representation results in a simple graph which takes advantage of providing area and delay optimization possibilities.

2.3.2 Conditional Blocks: In our representation, the translation of an *if/case* statement generates a *branch* node and a *merge* node (Fig. 7). The *branch* node has incoming edges transporting the branching exclusive conditions. The outgoing edges are conditional control edges selecting the conditional branch with a condition having the value true. Depending on the input conditions, the token generated from the execution of the branch node is only propagated through one branch. Using control edges, the conditional branches converge into a *merge* node indicating the end of each branch. The subgraph constituted thereby is called a *conditional block*, where the *branch* node is the entrance to the block and the *merge* node is the exit.

The whole control and data flow graph is made up of interconnected blocks of two types: nonconditional blocks and conditional blocks (Fig. 10). A nonconditional block is a section of the graph that contains no conditional branches.

In our representation, all the nodes executed under the same condition are in the same conditional branch. In Fig. 7(b), operations (1) and (2) are executed under the same condition $c1$ and belong to the same conditional branch. But, how is the right c value selected from the CB's? Note that a value from a conditional branch can be used as soon as it is produced. In Fig. 8(b), the right c value used by node (5) is selected using conditional data edges weighted with both the c and the condition $c1$ under which this value is selected. Operation (5) is scheduled just after its predecessor (4). Similarly, if $\overline{c1}$ is true, then the value *bb* produced by the conditional branch is selected for use by node (6), otherwise ($\overline{c1}$ false), node (6) uses the value *bb* coming in from the outside of the conditional block.

The representation in Fig. 7 is of the C-select type. The D-select representation is also allowed. In fact, a conditional node can be executed before the selection condition is evaluated. In Fig. 9(b), selection node (0) is scheduled after conditional operations (1), (2), and (3), which are executed in parallel (corresponding to the D-select representation). This scheduling is advantageous when operations (1), (2), and (3) do not share functional unit due to type conflict.

Before scheduling, the representation type in Fig. 9 that offers more scheduling freedom is used. The scheduler determines if it is advantageous to assign the selection node before or in parallel with the branch node, and take advantage of resource sharing possibilities between mutually exclusive operations or to keep the same representation form. So, the

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.

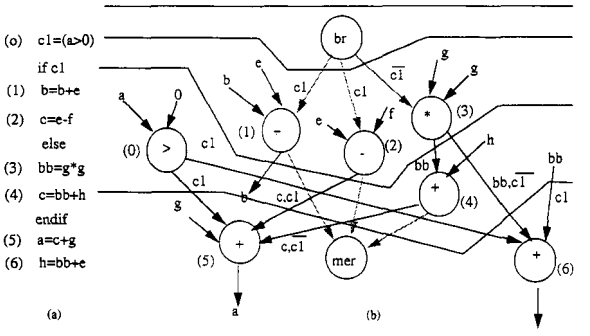

Fig. 9. D-select representation possibility.

scheduling algorithm determines the most advantageous representation type and schedules the operations according to the conditional operation types and the resources available.

2.3.3. Mutual Exclusion Testing: The well-known algorithms for mutual exclusion testing are the node coloring algorithm [5] and the condition vector technique [6]. It has been shown [4] that these algorithms cannot handle certain types of conditional branches correctly, and give incorrect results in some cases. The most recent technique for mutual exclusion testing is the node tagging algorithm [4]. A tag is assigned to each node. However, this tagging handles only one selection condition, so multi-way branches are not allowed. Furthermore, in some cases, the two tags assigned to the two nodes do not have enough information to determine if they are mutually exclusive or not. The pessimistic solution is then considered, supposing that the two nodes are not mutually exclusive. This can result in an inferior design.

Our mutual exclusion testing procedure uses the structure defined in the previous section. New concepts concerning this structure are introduced and they are defined as follows:

- The whole control and data flow graph is represented by a connected graph, called *conditional graph*, made up of the conditional and nonconditional branches. This conditional graph is the skeleton of the CDFG. An example is given in Fig. 10.
- A *global block* groups an unconditional branch and all the subsequent conditional branches.
- A *branch* groups all the nodes executed under the same conditions, if it is conditional. Otherwise, it groups all the nodes that exist before the next conditional branch.
- The *level* of a conditional branch is the number of branch nodes to be executed before reaching the branch incremented by 1, because a non-conditional branch is of level 1.

Note that this graph modeling does not impose execution precedence relationships between branches. It is used only for mutual exclusion testing purpose.

Our mutual exclusion testing procedure uses a branch numbering that is similar to the node coloring used in [5], but implemented and used in different ways. Only branches of the conditional graph are numbered, and each node has the reference of the branch to which it belongs.

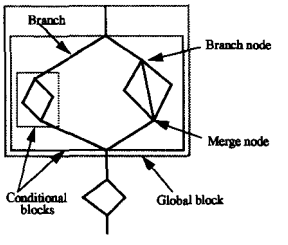

Fig. 10. Conditional graph example.

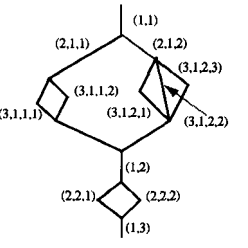

Fig. 11. Example of branch numbering.

The number of a branch in the graph is a dynamic size vector having the following form:

$$B = (n, N_1, N_2, \cdots, N_n)$$

where:

n	the branch level,
N_1	the global block number, and
N_i	for $2 \leq i \leq n$: the number of the branch of level i.

An example of branch numbering is given in Fig. 11.

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.

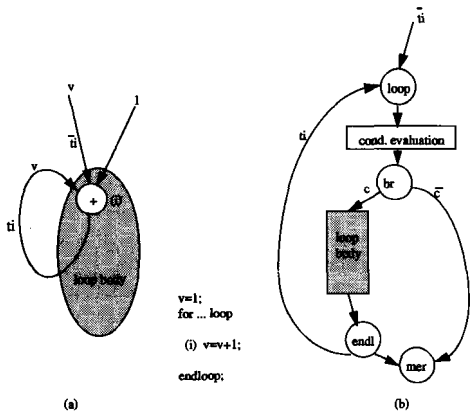

Fig. 12. (a) Data dependency inside a loop. (b) General form of a loop block.

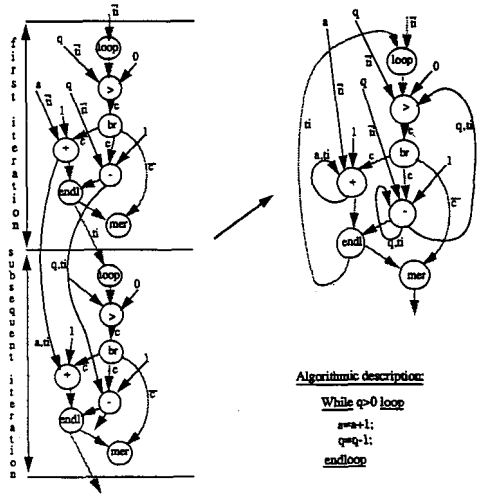

Fig. 13. Loop duplication technique for conditional data dependency.

Using this branch numbering, the testing of the mutual exclusion of nodes is easily performed. Having two nodes, one from a branch i whose number is $B_i = (n, N_1, N_2, \cdots, N_n)$ and the second from a branch j whose number is $B_j = (m, N_1, N_2, \cdots, N_m)$, they are mutually exclusive if:

- B_i and B_j are in the same global block ($N_1 = M_1$), and
- B_i and B_j are in a conditional block (n and $m > 1$), and
- there exist two different conditional branches of same level $k(N_k = M_k)$ having the same root ($N_t = M_t$ for $t < k$).

In Fig. 11, a node of branch (3, 1, 2, 3) is not mutually exclusive with a node from branch (2, 2, 2), because they are not in the same global block. But, the first node is mutually exclusive with a node of branch (2, 1, 1), because there exist the two different CB's (2, 1, 1) and (2, 1, 2) of the same level (2) and having the same root.

2.4. Iterative Blocks

2.4.1. Loop Representation: In order to represent loop constructs, *for...loop* and *while...loop*, a *loop* and *endloop* nodes are used combined with *branch* and *merge* nodes. The *loop* node is an entrance to the loop block and the *endloop* is the convergence node of the loop body. After the entrance of the loop blocks, the loop condition is evaluated. A *branch* node is then used to select either the branch containing the loop body or the empty branch converging to the *merge* node, which is the exit node of the loop block. Note that this implementation takes advantages of the conditional branch representation described above because of the use of the *branch/merge* nodes. An example is the possibility of scheduling an operation of the loop body before or in parallel with the evaluation of the loop condition. The general form of a loop block is given in Fig. 12(b).

Since the iterative block may be executed more than once, the loop semantic is translated in the graph using conditional edges, where the condition concerns the *token index*. As specified in Section 2.2, the *token index* is the number of executed instances of the node. So, at the first iteration, the loop is activated by the control edge weighted with the condition t_i (*token index* equal 0) and in the next iterations the same node is activated by the edge weighted with the condition ti (*token index* > 0) outgoing from the *endloop* node (feedback edge). The proper execution of a loop is then assured.

2.4.2. Conditional Data Dependency Inside a Loop: An iterative block is a cyclic subgraph of the CDFG. The data dependency inside this subgraph is more complex than it is in an acyclic subgraph. The main problem is to implement the data dependency between different instances of a node inside the loop body. In Fig. 12(a), the value v used by the node (i) is from the outside of the loop at the first iteration, but, in the following iterations, the same value v is from the previous iteration, provided from the execution of an instance of the same node (i). This problem is resolved using the conditional edges introduced above. The data edge coming in from outside of loop is weighted by t_i and the feedback edges by ti.

The conditional data dependency inside an iterative block is obtained using the loop duplication technique, an example is given in Fig. 13. The first instance represents the first iteration of the loop and the second instance is any one of the subsequent iterations. Once the loop has been duplicated, the same procedure as for regular data dependency edge creation is used to create the conditional data edges, except that when the value is from the outside the whole block, the edge is weighted by t_i and when the value is from an internal node, the condition ti is associated with the edge. After the creation of the data dependency edges, the two instances of the loop block are merged. Note that before the duplication of the loop the regular data dependency inside the loop body must be performed.

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.

```
Begin /* testing of existing path between two nodes i and j */
  -Provide tokens at the incoming edges of node i
  -Execute node i
  While there are executed nodes and node j is not executed
    do
     -Check for the ready nodes and execute them
    enddo
  if node j is excuted then a path between i and j is detected
End
```

Fig. 14. Token propagation procedure.

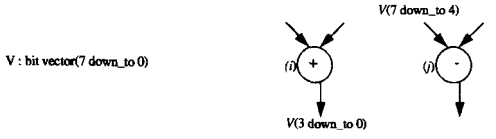

Fig. 15. Bit level data dependency.

2.5. Graph Optimization

The control and data flow graph is generated by first creating the data dependency edges, because these are unavoidable. The control dependencies are created in a second phase. Since a data dependency implicitly includes a control dependency, the former is said to be dominant. Therefore, in order to generate an optimized graph, when a control edge between two nodes is needed, it is physically created only is there is no existing indirect dependency between the two nodes. There is an indirect dependency (or implicit dependency) between two nodes if a path between them can be found in the graph, and a path exists between two nodes i and j, if the execution of i involves the execution of j. Note that, in this procedure, an abstraction of all the conditions associated with conditional edges is done, and they are all supposed to be true.

The test on an existing indirect dependency between two nodes is performed using a token propagation procedure. Starting from the first node, the necessary tokens are provided at the incoming edges. Using an iterative process, all the ready nodes are checked at each iteration and then executed. This process stops when the second node is reached or when no node can be executed. In the first case, a path between the two nodes is detected. The corresponding algorithm is shown is Fig. 14.

This procedure makes it possible to find and avoid any redundancy in the dependency representation. Consequently, extra control edges in the graph are not created. An optimized graph is then generated.

Note also that a bit-level data dependency is implemented in the graph model. In the example in Fig. 15, node (i) has as a result the four first bits of the bit vector V. Node (j) uses the last four bits of the same bit vector. The two nodes are not considered data-dependent. This means that no edge is created between them. It is a characteristic that may reduce the critical path of the graph. This kind of bit handling is very often used in applications like processors and controllers.

The next section presents the scheduling algorithm taking advantages of the graph model presented in this section. The Tabu Search technique adapted to solve the formulated scheduling problem is also presented.

III. Scheduling of a CDFG

3.1. Introduction

Operation scheduling and hardware allocation are the major tasks in high-level synthesis. Scheduling is recognized to be the most important. Operation scheduling determines the cost-speed tradeoffs of the design. Many different scheduling techniques have been reported. Most systems use heuristic algorithms to schedule operations. Our approach consists of a mathematical formulation of the scheduling problem, and uses a recently developed optimization method called Tabu Search that is adapted to solve it. Our approach differs from existing scheduling algorithms in the approach used to develop the mathematical formulation, which is based on penalty weights instead of cost estimation ([15]–[18]), as well as in the optimization method used, which is the tabu search technique adapted to solve the problem. Furthermore, the algorithm used in TASS performs the scheduling of a control and data flow graph where both the data and the control flows are represented in the same graph, generated from a VHDL behavioral description.

Note also that the approach used to develop the mathematical description of the scheduling problem gives a more complete formulation, taking into account almost all the area parameters, without an increase in complexity. This approach avoids the inflexibility implied by the cost function used in previous research ([15]–[18]), where the solution space is restricted by fixing the Functional Unit (FU) performing each type of operation. The advantage of our approach is that, after scheduling, module selection can be performed in order to better optimize the design that is to be generated, as opposed to the previous approach where the modules are fixed prior to scheduling.

Tabu Search (TS) is a metaheuristic procedure developed by Glover for solving combinatorial optimization problems ([11], [12]), which uses heuristics at different levels and avoids the problem of being trapped at locally optimal solutions. TS has been effective in finding optimal solutions for many types of large and difficult combinatorial optimization problems. A partial list of these applications is given in [13]. Tabu Search results show that this method is faster and more efficient than the better known optimization techniques like Simulated Annealing.

In most existing synthesis systems based on the ILP approach, as in [15]–[18], the objective function characterizes the cost of the hardware units needed by the scheduling. The ILP models developed are simplified and sometimes include only the functional units [15]. The area cost of storage, interconnections and control circuitry are often ignored. Otherwise, the formulation becomes complex, and thus cannot be used for sizable applications. Note that the cost function is rigid, and the ALU executing each type of operation must be specified before scheduling. The search space is thus restricted. Better solutions

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.

may be obtained if different ALU's (simple or complex) are available for executing the same type of operation [22]. The area cost may be reduced when different operations are combined for execution by the same FU and when more than one ALU is available to execute the same type of operation.

The formulation developed here is based on penalty weights instead of on cost evaluation. Penalty weights include the real costs of the hardware units available in a given technology and are taken from a library. The penalty weights used in our program make it possible to take into consideration different area parameters of the design to be generated. The number of functional and storage units, as well as the number of interconnections, is optimized by the minimization of an objective function including penalty weights developed for this purpose. The results obtained show that TASS, using this algorithm, generates better designs than existing algorithms.

In Section, 3.2, the detailed formulation of the scheduling problem is presented. Section 3.3 includes the Tabu Search algorithm for scheduling.

3.2. A New Formulation of the Scheduling Problem

In most scheduling methods, like "list scheduling," operations are assigned to control steps one at a time, consequently their results depend strongly on the order of the assignments. The ILP approach to the scheduling problem is a global method. With this method, called transformational, the algorithm starts with an initial scheduling, then nodes are moved in order to minimize a defined objective function indicating the area required by current scheduling. Nodes are assigned simultaneously to control steps. Since nodes are control- and data-interdependent, global assignments generate a better design than list scheduling. The results depend on the objective function defined and on the optimization method used. In the ILP approach, the scheduling problem is stated by a mathematical description and solved using an ILP method.

In our algorithm, the idea in establishing a mathematical formulation to avoid the inflexibility highlighted in the first section is to develop a more complete and efficient formulation taking advantage of the use of real unit costs.

To do this, the objective function to minimize is a sum of penalty weights, defined as follows:

$$f = \sum_{l} W_i.$$

Each W_i weight is a cost-based penalization of a worst assignment of operation nodes requiring the greatest amount of a given hardware source.

Let us consider the following parameters: n is the total number of nodes of the control and data flow graph and k the total number of control steps. i denotes a node, for $1 \leq i \leq n$. The propagation delay of an operation node i is d_i. d_i can be one control step, a partition of a control step (for operation chaining) or more than one control step (for multicycle operations). t_i is the number representing the type of operation i, and c_i the number representing its *equivalence class*. An equivalence class groups all operations than can be executed by the same complex ALU that exists in the library used. We define $C[t_i]$ as the cost of the FU performing an operation of type t_i. The notation $i \sim> j$ means that node j is a successor of node i: if j uses the result of i, we write $i \rightarrow j$ and if i provides a control to j, we write $i - - > j$. y_i is the control step where the node i is assigned, S_i its ASAP (As Soon As Possible) step and L_i its ALAP (As Late As Possible) step.

```
Begin
  -determine the set Nms of operation nodes of
   the same type m assigned to control step s
  -make current the first operation of the set Nms
   while the last element of Nms has not been reached
    do
 check:
     -look for a mutually exclusive node in Nms
      if such a node is found then
     -eliminate the node from Nms
     -goto check
      endif
     -if the number of nodes kept in Nms is
       greater to the maximum found before
       then keep this number
     -make current the next element of Nms
     enddo
   -the maximum number of operations of type m that
    can be executed in the control step s is the number
    kept
End
```

Fig. 16. K_{ms} factor determination.

Now, the scheduling problem is defined as follows: given the number of control steps k,

$$\text{minimize } f(y_1, y_2, \cdots, y_n) = \sum_{i} W_i$$

satisfying constraints:

$$(1) \quad S_i \leq y_i \leq L_i \qquad \text{for } 1 \leq i \leq n$$

$$(2) \quad y_i + d_i \leq y_j \qquad \text{for } (i,j)/i \rightarrow j \qquad 1 \leq i \text{ and } j \leq n$$

where W_i are penalty weights defined as follows.

W_1: Penalization of control steps where a maximum number of same-type operation nodes have to be executed for each type of operation. If po is the total number of operation types:

$$W_1 = \sum_{m=1}^{po} C[m] * \max\{K_{ms}/1 \leq s \leq k\}$$

where K_{ms} is the maximum number of operations of the same type m that can be executed in control step s. Since the operation nodes assigned to the control step s might belong to different conditional branches, some operations can be mutually exclusive. The worst case, where the maximum number of operations have to be executed, is taken into account. This number K_{ms} is determined using the algorithm described in Fig. 16. The basic concept of this algorithm is to consider the operation nodes concerned, and kept in a set Nms one by one, and at each time any mutually exclusive operation with the current operation is removed from the set. The maximum number of operations kept in Nms is the value of K_{ms}.

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.

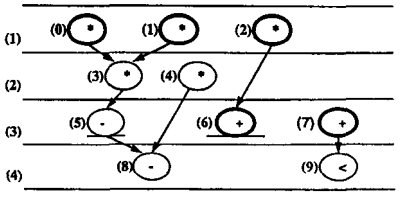

Fig. 17. Examples of W_1 and W_2 penalty weight evaluation.

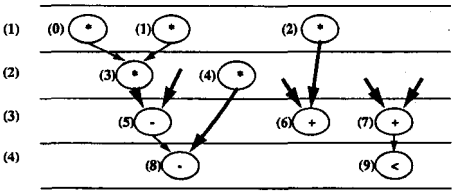

Fig. 18. Example of W_3 and W_4 penalty weight evaluation.

In the example in Fig. 17, there is a maximum of three multiplications assigned to control step (1) and two additions to control step (3). So, the penalization weight W_1 is: $W_1 = 3*$ cost [$*$] $+ 2 *$ cost [+].

W_2: Penalization of control steps where a maximum number of nonmutually exclusive operations of the same equivalence class have to be executed for each equivalence class.

When two nonmutually exclusive operations i and j from the same equivalence class m are assigned to the same control step, two functional units of type t_i and t_j are needed. If they are assigned to different steps, they can be executed by the same FU of type m. The penalization factor is the loss in terms of cost, and is defined as: $C[t_i] + C[t_j] - C[m]$; then, if pc is the total number of equivalence classes of operations, and oc_i denotes operation class number i:

$$W_2 = \sum_{m=oc_1}^{oc_{pc}} \max\{\tilde{K}_{ms}/1 \le s \le k\}$$

where $\tilde{K}_{ms}$ is the sum of the penalization factors corresponding to each of these pairs of operations (i, j), and is easily determined using the mutual exclusion testing procedure described in the previous section.

In Fig. 17, the two operations, (5) and (6), are of the same equivalence class, if we suppose that there exists a functional unit performing both an addition and a subtraction. The W_2 penalization weight has the following value: $W_2 = 1*$(cost[+] + cost [−]−cost[+/−]).

W_3: Penalization to minimize the lifetimes of variables.

When $i \rightarrow j$, the length of time that the output of i must be kept to be used by j contributes to the lifetime of the variable containing this value and is equal to the number of control steps separating i and j. W_3 is a penalization for nodes i and $j/i \rightarrow j$ assigned to control steps separated by more than one step. The penalty factor for each additional step is taken as: $C[R]/k$, where $C[R]$ is the cost of a register R. Let us consider the function $Su_{ij} = 1$ if $i \rightarrow j$, 0 otherwise,

$$W_3 = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (Su_{ji} * (y_j - y_i - d_i) + Su_{ij} * (y_i - y_j - d_j)) * C[R]/k.$$

In Fig. 18, node (6) uses the result of node (2) and node (8) uses the result of node (4). In both cases, the two nodes are separated by one extra control step. So, W_3 has the following value: $W_3 = 2*$ (cost[storage]/4).

W_4: Each data transfer requires a bus. In a bus architecture, the number of buses needed at a control step is the maximum number of distinct input variables used in this step. W_4 is to penalize the control step where the maximum number of distinct variables is used.

Let Z_s be the maximum number of distinct variables used in step s.

$$W_4 = C([bus] * \max\{Z_s/1 \le s \le k\})$$

where $C[bus]$ is the bus cost.

In Fig. 18, a maximum of six values (supposed distinct) are used in control step (3). The value of W_4 is

$$W_4 = 6 * cost[bus].$$

.The formulation given above is flexible due to the composition of the objective function on the separate penalty weights. The scheduling performed is global, the nodes are assigned and moved simultaneously within the control steps. A large search space can be explored and there is no restriction on the operation performed by any fixed FU and, vice versa, the FU performing any fixed operations. Here, the scheduling problem consists in finding the vector $S = (y_1, y_2, \cdots, y_n)$ whose $f(S)$ value is optimum while satisfying the given constraints. The goal is to minimize the hardware resources needed during the allocation, that is the next task performed by TASS.

The mathematical formulation of the scheduling problem proposed here is solved using the Tabu Search technique adapted to this problem, which is presented in the next section.

3.3. Tabu Scheduling

Our algorithm (Fig. 19) performs time-constrained scheduling. Given a maximum number of steps, nodes of the graph are assigned within this limit in such a way to minimize the cost of the hardware resources needed by the scheduling. By modifying the number of control steps, and then rescheduling, different design alternatives may be obtained.

Given an initial scheduling, nodes are moved from one control step to another. A node is moved within the time interval determined according to its ASAP and ALAP control steps and data dependency with other nodes. Thus ASAP and ALAP schedulings are performed first. The initial solution can be one of these schedulings or a randomly generated scheduling. All the Tabu Search elements are used [11]. During each iteration, a set of solutions V^* is generated from the current solution S in the *neighborhood* $N(S)$ of S. The best solution S' of V^* (solution whose f value is minimal) is determined and a *move* from S to S' is allowed, even if

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.

```
begin

 1- initiate short-term memory with given parameters, tabu_size and nbmax.
 2- determine ASAP and ALAP schedulings.
 3- generate initial solution S.
 4- evaluate f, with the generation of a subset V* from the neighborhood N(S).
 5- while fewer than nbmax iterations have passed without
          improvement on the best solution found
   do
    5.1- choose best solution S' from V*
    5.2- if S' has tabu status and
           aspiration level criterion is not satisfied
       then
         -remove S' from V*, goto 5.1.
       else
         -update tabu list and long-term memory.
          if S' is better than the best one found S*
               then keep S', S* <-- S'
         -evaluate f, with the generation of V*.
       endif
   enddo

 6- perform one of the following:

   6.1- restart procedure from the beginning
        with new parameters tabu_size and nbmax.
   6.2- restart procedure from the best scheduling found with or
        without new parameters, tabu_size and nbmax.
   6.3- restart procedure invoking long-term memory and
        eventually new values for tabu_size and nbmax.
   6.4- end procedure.

end
```

Fig. 19. Tabu scheduling algorithm.

$f(S') > f(S)$. The best solution found in all past iterations is kept.

In order to prevent cycling on the same solutions, going back to a solution visited during the last k iterations is not allowed. We introduce a list T of length k, called a *tabu list*, where the last k solutions accepted, called *tabu solutions*, are kept. T is used as follows: whenever a move from S to S' is made, S' is accepted only if it does not exist in T. This condition is called a *tabu restriction*. If S' exists in T, it takes *tabu status* and is rejected. Otherwise, S' is introduced at the end of T and the oldest solution introduced is removed. Going back to S' is forbidden (*tabu*) as long as this solution is in the tabu list. We are limited to the k last iterations because sometimes it is useful to go back to a solution that has been visited and continue the search taking another direction. The iterative process is stopped when *nbmax* iterations have been performed without improving the best solution found. The tabu list size (*tabu_size*) and *nbmax* are important parameters fixed by the user according to the problem addressed.

Tabu restrictions may not be good filters. Interesting solutions may be rejected because they are tabu. An additional feature is therefore introduced to cancel the tabu status of a solution when this may be desirable, which is a set of criteria called *Aspiration Level Conditions*. When these are satisfied, the solution is accepted, even if it has the tabu status.

The *long-term memory* [13] is also exploited. The basic structures of the method constitute a so-called *short-term memory*. This is an optimization phase where an intensified search, guided by the tabu restrictions and the aspiration criterion, is performed. The search may continue. A long-term memory is then invoked. This is a function that records moves taken in the past in order to penalize them in the next phase. The goal is to diversify the search by reaching regions not yet explored. The search can be restarted with an intelligent selection of new regions in the solution space. A description of the proposed Tabu Search algorithm for scheduling is given in Fig. 19.

The following definitions describe the Tabu Search elements adapted to the scheduling in our algorithm.

Neighborhood of Solution S, $N(S)$: A solution S is a vector $(y_1, y_2, \cdots, y_n)$, where n is the total number of nodes and y_i is the control step where the node i is scheduled. A move from S to S' inside $N(S)$ is made by moving a node from its control step in S to another.

During the evaluation of $f(S)$, operation nodes scheduled in control steps where the most important penalty weights (W_1

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.

and W_2) are the highest (the worst scheduled operations called "critical operations") are kept in vector V^*. These operation nodes are the candidates for a move. A transition from solution S to solution S' is made by moving one critical operation i to a control step inside the interval $[S_i, L_i]$ where S_i is the ASAP value of i and L_i is its ALAP value. Any violation of the data or control dependency constraints is rectified by moving one of the nodes concerned in an appropriate way. In this case, a mobility-based priority is used to select the node to move.

Move of an Operation Node: Two options are available: a random move and an attraction force-directed move. The attraction force of an operation i to a control step k is evaluated as follows:

$$\text{force}_{ik} = -3 * N_{t_i k} - 2 * N_{c_i k} - N_k$$

where:

$N_{t_i k}$ is the maximum number of operations of the same type that have to be executed in step k.
$N_{c_i k}$ is the maximum number of operations of the same equivalence class that have to be executed in step k.
N_k is the maximum number of operations that have to be executed in step k.

The operation is moved to the control step for which the attraction force is maximal.

Move Attributes: Keeping the k last solutions in the tabu list may require excessive memory space. Furthermore, testing whether a solution S' is in the tabu list has to be done frequently and this may take a lot of CPU time. So, instead of keeping a whole solution, only the modification characterizing the move is saved in T. A move from S to S' is characterized by a triplet (*node* , i, j), where *node* is the critical operation node moved, i its control step in S and j its control step in S'. The tabu list contains a series of these triplets. The tabu status of a solution corresponds to a move where an operation node is moved from step j to step i when the triplet (*node*, i, j) exists in the tabu list.

Long-term memory: The long-term memory is an $n \times k$ matrix denoted LTM, where n is the total number of nodes and k the number of control steps. Initially, all the elements of the LTM are initialized at 0. During the iterative process, when a node i is moved to a control step j, the long-term memory changes as follows: $LTM_{ij} = LTM_{ij} + 1$. When invoking the long-term memory, after the short-term memory phase, during any $f(S')$ evaluation, and for each node i moved to a control step j, the value λLTM_{ij} is added to f like a penalty weight. LTM_{ij} is the number of times node i is moved to control step j in the previous phase and λ is a penalty factor. By changing parameter, λ it is possible to direct the search "closer to" or "farther from" the regions already explored.

IV. Experimental Results

The TASS system has been implemented and tested using high-level synthesis benchmarks. The programs are written in C++ programming language running on a SPARC station 2. In this section, some experimental results are shown in order to prove the efficiency of the TASS system. TASS was also compared to existing synthesis system. Two classes of examples are considered: operation-dominated examples and control-dominated examples.

system	No.of partitions	No.of buses/mux	ALUs
HAL89	4	3buses/1mux	2(*),1(+),1(-),1(<)
SPLICER	4	6mux	2(*),1(+),1(-),1(<)
TASS	4	6buses	2(*),1(+,-),1(<)
TASS	6	4buses	1(*),1(+,-),1(<)

Fig. 20. The differential equation example.

system	No.of partitions	No.of buses/mux	ALUs
Emerald	4	3buses/2mux	(+,*,OR),(*,-,AND),(/)
SPLICER	4	4buses/1mux	(+,-,AND),(+,OR,*),(/)
DEV_NEW	5	4buses	(+,-,AND),(OR,*,/)
TASS	4	5buses	(+,OR),(*,/),(+,-,AND)
TASS	5	4buses	(+,-),(*,/),(AND,OR)

Fig. 21. The FACET example.

From the first class, we have used standard examples: the differential equation, the FACET example [23] and the algorithm of the fifth-order elliptic wave filter extracted from [19].

Fig. 20 shows the results for the differential equation example compared to related works, [14], [24]. HAL89 [14] and SPLICER [24] found two multipliers, an adder, a subtractor and a comparator, while with TASS, we were able to combine the adders and the subtractors by using a complex ALU from the library that performs both operations. This was possible because of the notion of the equivalence classes of operations used in the search process of an optimized design. With six control steps, only one multiplier, one ALU performing addition and subtraction and one comparator are required. The design was generated after 16 iterations with *tabu-size*= 5 in a CPU time of 2 s.

The results obtained for the FACET example are shown in Fig. 21. In comparison with the results presented in [25], [24], [20], a better distribution of functional units is generated since the multiplication and division, which are of the same equivalence class, are performed by a separate ALU from those of the addition, subtraction and the Boolean operations. With Emerald [25], SPLICER [24] and the Devadas and Newton algorithm based on the Simulated-Annealing technique [20], the addition, multiplication and Boolean operations are allocated to the same unit, which is not very practical because this kind of hardware unit may be expensive to be produced. This design was generated in a CPU time of 2 s with *tabu-size*= 5 and after two iterations. With five control steps, the design generated is simpler and has better operation distribution. Three ALU's are needed, each performing two operations from the same equivalence class.

In order to compare the results obtained with the fifth-order elliptic filter example to related works, we consider that a multiplication takes two cycles and an addition takes

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.

system	HA89	Dev_New	FDLS	ALPS	TASS
adder multiplier	3 2 2 3 2 1	3 2	3 2 3 1	3 2 2 1 3 2 1 1	3 2 2 2 1 3 2 1 1 1
buses	6 - -	-	- -	6 4 4 4	8 6 5 4 4
registers	- - -	-	- -	- - - -	9 8 6 6 6
cycles	17 19 21	17	17 21	17 19 21 28	17 18 21 24 28
CPU time	between 2 & 8 mn	4 mn	between 1&2 mn	within tens of seconds	between 10 and 20 seconds

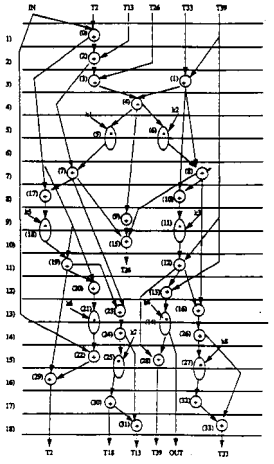

Fig. 22. The fifth-order elliptic filter example.

one cycle to be performed. Results have been compared to related works: the HAL89 system [14], the synthesis system developed by Devadas and Newton (Dev-New) [20] based on the Simulated Annealing method, the force-directed list scheduling (FDLS) algorithm [21] and the ALPS system [18] using the ILP approach. The comparison is summarized in Fig. 22. Design alternatives have been generated for 17, 18, 21, 24, and 28 control steps (Fig. 22(a)). We obtained better scheduling with 18 control steps (Fig. 22(b)): only two adders, two multipliers, six buses, and eight registers are required. This scheduling is generated with *tabu-size* $= 5$, *nbmax* $= 400$ and ALAP scheduling as the first solution.

A reduced area design is generated with 28 control steps. Only one adder, one multiplier, four buses, and six registers are needed. Fig. 23 shows the values of the objective function found in the iterative process of Tabu Search with 17, 18, 24, and 28 control steps. Each point corresponds to a solution. With 17 control steps, the solution space is reduced since 17 is the minimum delay corresponding to the critical path length. The optimal solution $(3(+), 3(*))$ is obtained after few iterations. With 24 and 28 control steps, the solution space is extended. The position of the different points shows that Tabu Search explores a large solution space. With 28 control steps, the best solution is obtained at iteration 214 after 15 s. The value of the objective function at this iteration is 897. Note that the runtime of the various experiments of the example ranges between 10 and 20 s. The results shown here were obtained without invoking the long-term memory. The use of the attraction force-directed move of operations makes the scheduling algorithm faster than other known algorithms in converging to the optimal solution.

From the second class of examples, we present results obtained with small examples: the algorithm of the Greater Common Divider (GCD), also used in [8], and the MAHA example [7] selected due to its complexity and potential resource sharing, and with large examples: the decoder, which is part of the error correction system, and the AM2901 AMD microprocessor slice.

Fig. 24 shows the results of both the CDFG generation and the scheduling for these examples. The Nodes, Conditional blocks, and Loops columns give the size of the problem. The Edges column gives the number of edges generated, and the corresponding CPU times required is given in the next column in order to show the low complexity of the model. These results show that the runtimes are reasonably short, even when the model is used for large examples. During the scheduling tests with these examples, the operation chaining used is 1 and the multiplication is supposed to take two control steps to be performed. The GCD example is scheduled in seven control steps when one subtractor and one comparator are used. This design needs only two buses.

Fig. 25 shows the results for the MAHA example. We compare our results with those produced by the critical-path scheduling algorithm [7] (MAHA), the path-based scheduling algorithm [8] (Path) and the KIM's method [10]. While the same resources are used: one adder and one subtractor, the scheduling produced with our algorithm required only eight control steps with an operation chaining of 1; but with the MAHA algorithm and the path-based scheduling algorithm, a scheduling with eight control steps is only obtained with a chaining of 2. However, when the chaining is 2 and with the same number of units, the scheduling is reduced to six control steps in our case. Two of the three algorithms require eight control steps, and the path-based scheduling algorithm needs nine control steps. Our results were obtained in CPU times ranging from 2 to 4 s.

The next example is a decoder which is part of the error correction system. The decoder receives data in the serial mode, checks the parity and performs the correction if there is any error and if possible. The VHDL functional description of this example has 151 lines. The control and data flow graph generated by the model includes 78 nodes (operations and control nodes), 43 objects (variables, constants and signals), and 159 data and control edges. Fig. 26 is the corresponding conditional graph showing the complexity of the control and the potential resource sharing in this example.

Fig. 27 shows the results obtained with the decoder example. The first column shows the number of control steps. The second column in this table gives the list of the functional units of the design. The next column gives the number of buses and the last column includes both the number of registers and the total number of bits of registers. Here, we suppose that an integer is represented in 32 bits. Note that the number of registers given here concerns only the storage units allocated for the variables. The critical path of the decoder has a length

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.

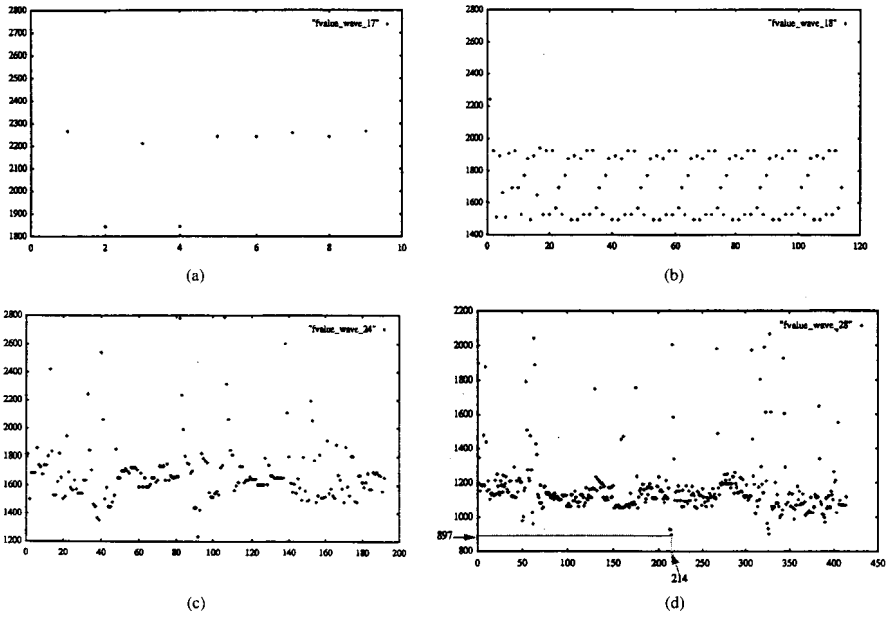

Fig. 23. Variation of the objective function for (a) 17, (b) 18, (c) 24, and (d) 28 control steps.

	Nodes	Cond branc.	Loops	Edges	CPU time sec.	# steps	ALUs	Buses
GCD	16	2	1	27	1	7	1(-), 1(#)	2
MAHA	26	10	0	38	2	8	1(+), 1(-)	4
Decoder	78	15	1	159	15	18	1(+),3(comp) 2(bool)	4
AM2901	155	26	0	516	50	11	1(+),8(comp) 9(bool)	9

Fig. 24. The CDFG generations and scheduling.

	Total steps	Adder	Subtrac	Chaining
TASS	8 6	1 1	1 1	1 2
MAHA	8	1	1	2
Path	9	1	1	2
KIM	8 8	1 1	1 1	1 2

Fig. 25. The MAHA example.

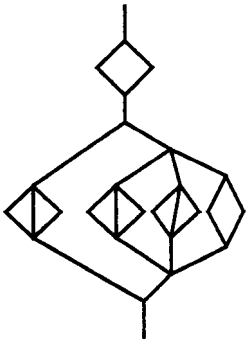

Fig. 26. Conditional graph of the decoder.

# Steps	ALUs	# Buses	#Registers total bits
18	1(+),3(comp) 2(bool)	4	5/55
19	1(+),3(comp) 2(bool)	4	4/54
20	1(+),2(comp) 1(bool)	4	5/55
21	1(+),2(comp) 1(bool)	3	5/55
22	1(+),1(comp) 1(bool)	4	4/54

comp(<,=), bool(Xor,and,not)

Fig. 27. Results with the decoder example.

of 18 control steps. With these 18 control steps, the design generated includes one adder, three comparators, and two Boolean operators. Fig. 28 is a data path implementation of this design. A reduced-area design is generated with 22 control steps. This design includes only one functional unit of each type, four buses and a total of 54 bits of registers. These results were obtained in a CPU runtime ranging from 15 to 20 s.

The last example is the AM2901 four bit microprocessor slice. The VHDL functional description of this benchmark has 248 lines and includes many conditional constructs. The use of the bit-level data dependency described in Section 2.5 has improved the critical path from 15 control steps to 11. Design alternatives are generated and the results are summarized in Fig. 29. The fastest design with 11 control steps includes one adder, eight comparators, and six Boolean operators. An

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.

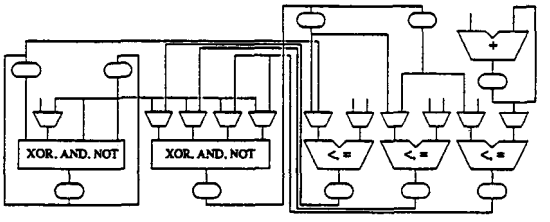

Fig. 28. Data path implementation of the decoder with 18 control steps.

# Steps	ALUs	# Buses total bit	# Registers total bits
11	1(+),8(comp) 6(bool)	9/72	7/42
12	1(+),6(comp) 7(bool)	8/64	8/47
13	1(+),4(comp) 7(bool)	8/64	7/43
16	1(+),3(comp) 6(bool)	5/49	7/42

comp(=), bool(*and,not,or,xor*)

Fig. 29. Results for the AM2901 example.

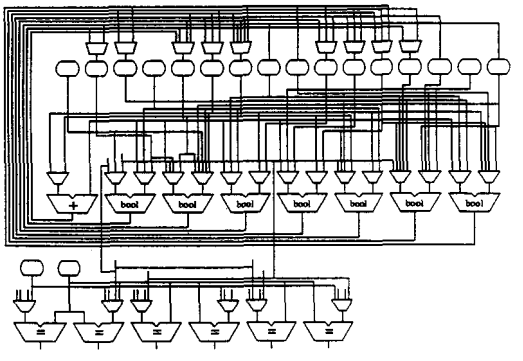

Fig. 30. A data path implementation with 12 cycles.

example of data path implementation with 12 control steps is shown in Fig. 30. Note that if both the comparison and the Boolean operations are combined to be performed by the same functional unit, the number of ALU's will be reduced. Since the buses allocated are of different sizes, according to the bit width of the values transferred which is, in general, between three and five bits, the total number of bits of buses is given in Fig. 29. The registers in this figure are for variable storage only. The results shown are obtained in a CPU time ranging from 40 to 60 s.

V. Conclusion

We have presented the main algorithms used in the TASS synthesis system. In this system, the formulation of the scheduling problem is based on penalty weights avoiding the inflexibility of the formulations developed in related works. Furthermore, a new solution technique using the Tabu Search method is adapted to solve the formulated problem. A new control and data flow graph model is developed. This model performs the conversion of a VHDL behavioral description into a single optimized graph used by our algorithm for scheduling. The use of the conditional dependency edges in the graph has permitted to perform a better implementation of the control constructs than in the related works. Since the graph generated includes both the data and control flows, it is also used for controller synthesis. With the formulation of the scheduling problem developed here, our algorithm performs a global optimization using area costs of hardware resources taken from a given library. This enlarges the design space, resulting in better optimized designs. Note that a testability evaluation of a data path has been incorporated in TASS system [38].

References

[1] G. G. de Jong, "Data flow graph: System specification with the most unrestricted semantics," in *Proc. European Design Autom. Conf.*, 1991, pp. 401–405.
[2] J. T. J. Van Eijndhoven and L. Stok, "A data flow graph exchange standard," in *Proc. European Design Autom. Conf.*, 1992, p. 193–199.
[3] T. Krol, J. V. Meerbergen, C. Niessen, W. Smits, and J. Huisken, "The Sprite input language, an intermediate format for high-level synthesis," in *Proc. European Design Autom. Conf.*, 1991, pp. 186–192.
[4] M. Rim and R. Jain, "Representing conditional branches for high-level synthesis applications," in *Proc. 29th ACM/IEEE Design Autom. Conf.*, 1992, pp. 106–111.
[5] N. Park and A. C. Parker, "Sehwa: A software package for synthesis of pipelines from behavioral specifications," *IEEE Trans. Computer-Aided Design*, pp. 356–370, Mar. 1988.
[6] K. Wakabayashi and T. Yoshimura, "A resource sharing and control synthesis method for conditional branches," in *Proc. Int. Conf. Computer-Aided Design*, 1989.
[7] A. C. Parker, J. Pizzaro, and M. J. Mlinar, "MAHA: A program for data path synthesis," in *Proc. Int. Conf. Computer-Aided Design*, 1988, pp. 52–55.
[8] R. Composano, "Path-based scheduling for synthesis," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 85–93, Jan. 1991.
[9] K. Wakabayashi and H. Tanaka, "Global scheduling independent of control dependencies based on conditions vectors," in *Proc. 29th ACM/IEEE Design Autom. Conf.*, 1992, pp. 112–115.
[10] T. Kim, J. S. Liu, and C. L. Liu, "Scheduling algorithm for conditional resource sharing," in *Proc. Int. Conf. Computer-Aided Design*, 1991, pp. 84–87.
[11] F. Glover, "Tabu Search, Part I," *ORSA J. Computing*, pp. 190–206, 1989.
[12] F. Glover, "Tabu Search, Part II," *ORSA J. Computing*, pp. 4–32, 1989.
[13] F. Glover, "Tabu Search: A tutorial" Center for Applied Artificial Intellegence, Univ. of Colorado, Boulder, Feb. 1990.
[14] P. G. Paulin and J. P. Knight, "Algorithms for high-level synthesis," *IEEE Design & Test*, pp. 18–31, Oct. 1989.
[15] H. Shin and N. S. Woo, "A cost function based optimization technique for scheduling in data path synthesis," in *Proc. Int. Conf. Computer-Aided Design*, 1989, pp. 424–427.
[16] C. A. Papachriston and H. Konuk, "A linear-driven scheduling and allocation method followed by interconnect optimization algorithm," in *Proc. 27th ACM/IEEE Design Autom. Conf.*, 1990, pp. 77–83.
[17] J. Lee, Y. Hsu, and Y. Lin, "A new integer linear programming formulation for the scheduling problem in data path synthesis," in *Proc. Int. Conf. Computer-Aided Design*, 1989, pp. 20–23.
[18] C. T. Hwang, J. H. Lee, and Y. C. Hsu, "A formal approach to the scheduling problem in high-level synthesis," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 464–475, Apr. 1991.
[19] P. Dewilde, E. Deprettere, and R. Nouta, "Parallel and pipelined VLSI implementation of signal processing algorithms," *VLSI and Modern Signal Processing*, S. Y. King, H. J. Whitehouse, and T. Kailath, eds. Englewood Cliffs, NJ: Prentice-Hall, 1985, pp. 257–275.
[20] S. Devadas and R. Newton, "Algorithms for hardware allocation in data path synthesis," *IEEE Trans. Computer-Aided Design*, vol. CAD-7, pp. 768–781, July 1989.
[21] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 661–679, June 1989.

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.

[22] L. Ramachandran and D. D. Gajski, "An algorithm for component selection in performance optimized scheduling," in *Proc. Int. Conf. Computer-Aided Design*, 1991, pp. 92–95.
[23] C.-J. Tseng and D. P. Siewiorek, "FACET: A procedure for the automated synthesis of digital systems," in *Proc. 20th ACM/IEEE Design Autom. Conf.*, 1983, pp. 490–496.
[24] B. M. Pangrle and D. D. Gajski, "Design tools for intelligent silicon compilation," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 1098–1112, Nov. 1987.
[25] C.-J. Tseng, and D. P. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Trans. Computer-Aided Design*, vol. CAD-5, pp. 379–395, July 1986.
[26] M. Jamoussi, B. Kaminska, "Controllability and observability measures for functional-level testability evaluation," in *Proc. 11th VLSI Test Symp.*, 1993, pp. 154–157.
[27] S. Amellal and B. Kaminska, "Scheduling algorithm in data path synthesis using the Tabu Search technique," in *Proc. European Design Autom. Conf.*, Paris, France, Feb. 1993, pp. 398–402.
[28] J. Roy, and R. Vemuri, "Appropriate usage of VHDL: The synthesis Point of Views," Tech. Memo, Univ. of Cincinnati, May 1990.
[29] H. Oudghiri and B. Kaminska, "Global weighted scheduling and allocation algorithms," in *Proc. European Design Autom. Conf.*, 1992, pp. 491–495.
[30] S. Amellal and B. Kaminska, "Scheduling of control and data flow graph," in *Proc. Int. Symp. on Computers and Systems*, Chicago, pp. 1666–1669, May 1993.
[31] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis Introduction to Chip and System Design*. Boston, MA: Kluwer Academic, 1992.
[32] L. Hafer and A. Parker, "A formal method for the specification, analysis, and design of register-transfer level digital logic," *IEEE Trans. Computer-Aided Design*, Jan. 1983.
[33] J. J. Granacki, *et al.*, "The ADAM advanced design automation system: Overview, planner, and natural language interface," in *Proc. 22th Design Autom. Conf.*, 1985.
[34] R. A. Walker and D. E. Thomas, "Behavioral transformation for algorithmic level IC design," *IEEE Trans. Computer-Aided Design*, Oct. 1989.
[35] J. S. Lis and D. D. Gajski, "Synthesis from VHDL," in *Proc. IEEE ICCD*, 1988, pp. 378–381.
[36] B. M. Pangrle, "A Behavioral Compiler for Intelligent Silicon Compiler," Univ. of Illinois at Urbana, Champaign, 1987.
[37] A. L. Davis and R. M. Keller, "Data flow program graphs," *IEEE Trans. Computers*, 1982.
[38] M. Jamoussi, B. Kaminska, and D. Mukhedkar, "A New testability measure: A concept for data flow testability evaluation," in *Proc. 5th Int. Conf. VLSI Design*, Jan. 1992, pp. 239–244.

Said Amellal received the B.S. degree in computer engineering from University of Algiers, Algeria, in 1985. He obtained the M.S. degree in electrical engineering from Ecole Polytechnique de Montréal, Canada, in 1992.

Presently, he is a research assistant at Ecole Polytechnique de Montréal, Electrical Engineering Department. His research interests include functional synthesis of digital systems, design verification, synthesis for testability, and testability evaluation at functional level.

Bozena Kaminska for a photograph and a biography, please see page 358 of the March 1994 issue of this *Transactions*.

Authorized licensed use limited to: Tsinghua University. Downloaded on January 01,2021 at 13:51:00 UTC from IEEE Xplore. Restrictions apply.