Representing Conditional Branches for High-Level Synthesis Applications

Minjoong Rim and Rajiv Jain
Department of Electrical and Computer Engineering
University of Wisconsin
Madison, WI 53706

Abstract
In this paper we outline a new representation of behavioral specification for high-level synthesis applications. The main features of our representation are: correct handling of conditional branches, capability to tradeoff between control-select and data-select forms, keeping minimum necessary precedence relationships, correct representation of all conditional actions, simplified mutual exclusion testing and correct determination of bit-widths and value transfers. The representation is simple and can be easily generated automatically.

1 Introduction
High-level synthesis research is targeted at automatically producing register-transfer level (RTL) designs. High-level synthesis programs accept a behavioral specification either in a hardware descriptive language [1] [13], or a programming language [3] [15] or a data flow graph [5] [8] [9] [10] [11]. In the first two cases, the input specification is compiled into one of the several internal representations [9] [12] [18]. An RTL design can only be as good as its input specification can permit it to be, and a fundamental improvement in representing the specification which allows better exploitation of parallelism and pipelining can have a major impact on the final design. The basic issue addressed in this paper is the representation of conditional branches. Despite the wide use of data flow graphs (DFGs) there is no consensus on representation of conditional branches. In this paper we focus on such a data flow graph representation which can be used by next generation high-level synthesis tools to implement better and more efficient RTL designs.

To provide good resource sharing among operations in data flow graphs with conditional branches, mutual exclusion testing must be done. Several algorithms which perform mutual-exclusion test have been reported in literature [9] [17]. In this paper, we will see that these algorithms do not always provide a correct solution, and a new algorithm which overcomes the deficiencies of these algorithms is presented.

The paper is organized as follows. In Section 2 we address the problem of representing conditional branches. The existing methods and their shortcomings are mentioned. Next, we present a new representation which overcomes these shortcomings. In Section 3 some problems with conventional methods for mutual exclusion testing are given and a new algorithm is proposed. The paper concludes with a brief discussion on research in progress.

2 Representing Conditional Branches
The following terms are used in this paper. Two nodes in a DFG are mutually exclusive if during any single execution at most one of these nodes is necessarily performed. A node is unconditional if it does not belong to any conditional branch else it is conditional. The true conditional path of a conditional branch refers to path(s) which are activated with true condition. The false conditional path of a conditional branch is defined similarly. The assumption of a conditional branch having two-way branches is used for clarity of presentation, and our results can be easily extended to multi-way branches.

2.1 Existing Conditional Branch Representations
Existing representations for conditional branches can be broadly classified into two types: control select (C-select) and data select (D-select) [5]. In C-select representation (Figure 1a) condition selection is performed before any conditional operation is executed and a conditional branch selected. Since alternate branches of a conditional branch are mutually exclusive, there is no resource conflict between operations in different branches and all mutually exclusive operations of the same type can share resources. In Figure 1a, one module can perform the two operations without any resource conflict (assuming, of course, that these operations are of the correct type). In C-select representation the decision as to which operations share resources is postponed until scheduling or binding is done. Due to the absence of information on sharing resources a

1The words node and operation are used interchangeably.

Figure 1: Conditional representations
priori, the precedence relationships between condition selection and execution paths must exist in the representation. As we will see later, some of these precedence relationships may be unnecessary and result in inferior schedules.

In D-select representation (Figure 1b) all conditional branches are executed separately in parallel and correct data values are selected at the end of the conditional branch [12]. This representation may result in fast execution since conditional operations can be executed before the condition selection is made. One drawback with this representation is that mutually exclusive operations cannot share resources, resulting in expensive designs. The use of D-select representation, however, may be preferable to C-select in certain cases, for example, when conditional branches do not contain any mutually exclusive operations due to type conflict. Another drawback with the D-select representation is that some conditional actions cannot be represented [5]. This is because the correct conditional path is chosen by the selection of output values and some conditional actions do not have an output (Figure 5a, 5c).

We say that a node is scheduled in a C-select manner with respect to a conditional branch if it is scheduled after the condition selection for the conditional branch is scheduled. Otherwise, it is scheduled in a D-select manner implying that it cannot share resources with any of conditional nodes in the branch. In order to produce good designs, the scheduler must decide which operations must be scheduled in a C-select manner and which operations are to be scheduled in a D-select manner. Suppose a conditional branch has a pair of mutually exclusive operations. If they are scheduled in the same timestep, a D-select representation will prevent them from sharing resources. This may produce an inferior schedule as compared to a schedule produced using the C-select representation. However, when they are scheduled in different time-steps, C-select representation may delay scheduling of operations until the condition selection is made, resulting in a slower schedule.

Another widely used representation is the distribute-join [9] [10] of USC ADAM system [4] [6] (Figure 2a), which is a C-select type and does not permit dynamic tradeoff between C-select and D-select forms. One problem with the distribute-join representation occurs while representing precedence relationships. The distribute-join representation enforces unnecessary precedence relationships which may mislead the scheduler into producing inferior designs. In distribute-join representation, all out-going edges must go through the join node. Suppose, in Figure 2a, if

\[
\begin{align*}
\text{if } (a > 0) \{ & \text{b} = d + e; \\
& c = e + f; \\
\} \\
\text{else} \{ & \text{bb} = g + g; \\
& c = bb + h; \\
\}
\end{align*}
\]

values are available before c is computed, thus enabling operations requiring b and/or bb value(s) to be ready for scheduling. However, these ready operations must wait until the join operation is scheduled, which in turn, can only be scheduled after c becomes available. This prevents scheduling of the ready operations in parallel or before c value is evaluated. Thus, the use of distribute-join representation may delay execution of ready operations which may result in an inferior schedule.

This disadvantage can be easily overcome by splitting distribute and join nodes. The representation in [17] uses fork-join representation which is similar to the distribute-join representation with split distribute and join nodes resulting in too many special nodes. It retains the fork node which is similar to the distribute node. The one drawback of the use of fork or distribute nodes is that some conditional actions which do not have inputs (Figure 5b, 5c) cannot be represented without adding dummy inputs.
2.2 A New Representation

We now present a new representation which overcomes the above mentioned disadvantages. Examples of our representation are given in Figures 2b, 3 and 4. In our representation we have three types of edges which represent different precedence relationships. A value transfer between two operations implies that the value producing (predecessor) operation should be completed before the value consuming (successor) operations start execution. A precedence relationship due to a value transfer is named strong precedence relationship, and is depicted by an arrow called an $S$-edge. Some precedence relationships which do not carry values and are required for correct execution of the specification are called pure precedence relationships. Pure precedence relationships are used for special cases illustrated later, and can also be used by the designer to impose constraints, such as timing constraints or loop dependencies. The edge representing pure precedence relationship is called a $P$-edge and is denoted by a bold arrow. The third type of relationship is not a precedence relationship in a strict sense in that it can be violated for correct execution. We call this a weak precedence relationship and its sole use is to identify correct conditional paths. The edge representing a weak precedence relationship is called a $W$-edge and is denoted by a broken arrow. The $W$-edge carries an attribute $T$ or $F$ denoting whether this edge is in the true conditional path or the false conditional path respectively. The relationships between a node and its predecessors is determined by the type of edges connecting the node and a predecessor. Thus, $S$-predecessors, $P$-predecessors, and $W$-predecessors are immediate predecessors connected by $S$-, $P$-, and $W$-edges respectively.

The reason for distinguishing the three types of edges is as follows. During mutual exclusion testing, $P$-edges are not required since they do not relate to control sequence. During scheduling, $W$-edges are not required since they do not represent real precedence relationships. During binding, we only use $S$-edges, which represents value transfers. Most conventional representations intermingle these different edges which restricts the design of efficient and/or correct algorithms for mutual exclusion testing, scheduling or binding.

A conditional branch is represented by one CB (conditional branch begin) node and several CE (conditional branch end) nodes, one for each value being transferred out of the conditional branch. A CB node selects and activates the correct conditional execution path in the conditional branch according to the condition selection. Boolean operations which determine control flow are explicitly represented. A CB node represents the control for the conditional branch. Edges emanating from a CB node represent weak precedence. No value transfer occurs between a CB node and any of its successor nodes. All S-edges incident to operations in a conditional branch go directly to the operations without going through the CB node.

A CE node marks the termination of a path in the conditional branch. Each value being transferred out of a conditional branch must go through a CE node. This allows the scheduler to schedule the CE nodes in different time-steps eliminating unnecessary precedence relationships mentioned earlier. In Figure 2b operations requiring $b$ or $bb$ can be scheduled independent of when $c$ is produced. Also value transfers are clearly represented since inputs and output of each CE node represent the same variable. This also allows the correct computation of bit-width requirements. The number of inputs to a CE node is always two since there are two input alternatives (true or false) to choose from.

Our representation allows tradeoff between C-select and D-select during scheduling. In C-select representation condition selection precedes initiation of the conditional execution paths. This, however, is not essential to the correct execution of the program, since initiation of the conditional execution path without condition selection only eliminates possibilities of combining some
mutually exclusive operations. In our representation, W-edges emanating from a CB node represents a weak precedence which can be violated, permitting operations in a conditional branch to be scheduled before the CB node. Operations scheduled before or in parallel with the CB node are executed in a D-select manner (time-step 1 of Figure 4). Operations scheduled after the CB node are treated in C-select manner (time-step 2 of Figure 4). In any case, a CE node cannot be scheduled before the corresponding CB node which is taken care of by a pure precedence relationship between the CB and the CE nodes. Our representation can handle special conditional actions shown in Figure 5. We summarize our representation with the construction rules.

1. Construct nodes representing operations.
2. For each conditional branch, construct a CB node.
3. For each output value of a conditional branch, construct a CE node. Since there are always two possibilities (true or false) according to condition selection, the CE node has two input values and one output value.
4. All value transfers are represented by S-edges.
5. If a node in a conditional branch has no S-predecessor in the same innermost conditional branch, then add a W-edge from the CB node to the node in question. The W-edge has an attribute true or false indicating the true or false conditional path.
6. P-edges are inserted between a CB node and all its corresponding CE nodes. P-edges may also be added for special actions such as those illustrated in Figures 5a and 5c.

3 Mutual Exclusion Testing

Having discussed our representation we now address the issue of determining mutually exclusive operations. This is done in two steps. First, the nodes are tagged and next, mutually exclusive operations are determined. Since resource sharing among mutually exclusive operations can only be determined during scheduling, the latter step is deferred until scheduling is in progress. Details on how to tag the nodes and modify the tags during scheduling to reflect resource sharing are discussed in this section. We begin by surveying existing algorithms for determining mutual exclusivity and their shortcomings.

3.1 Existing Algorithms

Two well-known algorithms for mutual exclusion testing are the node coloring algorithm by Park [9] and the condition vector method by Wakabayashi [17]. These algorithms, however, cannot handle certain types of conditional branches correctly. In Figure 6, node n1 is mutually exclusive to node n5 but not mutually exclusive to node n3. The algorithms of Park and Wakabayashi incorrectly indicate that nodes n1 and n3 are mutually exclusive and can share resources. This error can result in a design which will not meet the input specification. If BOOL3 was replaced by BOOL2 in CB3, then n1 becomes mutually exclusive to n4 as well. This shows that mutual exclusion depends upon the boolean variable selecting the conditional branch. For mutual exclusion testing we have to consider condition selections which activate the conditional branches and not the conditional branches themselves [16]. Further, the test for mutual exclusion should not depend on the order of nested conditional branches.

As mentioned before our representation allows trade-off between C-select and D-select forms during or prior to scheduling. This tradeoff can complicate mutual exclusion testing. In Figure 7, since conditional branch CB1 is not initiated in time-step 1, node n1 is not mutually exclusive to nodes n3 or n4. Such a case can eas-
Figure 7: Handling tradeoffs between C-select and D-select forms

ily occur if BOOL2 and BOOL3 are evaluated before BOOL1. Wakabayashi's representation cannot handle such cases even though tradeoff between C-select and D-select forms is allowed in their representation. This is because Wakabayashi implicitly assumes that for any two nested conditional branches the condition selection of the inner conditional branch is always done after the outer one.

3.2 A New Algorithm

We now present a new and simple algorithm for mutual exclusion detection which overcomes the above problems. The tag of a conditional operation is different if it is executed in a C-select or D-select manner for some fixed conditional branch. Before scheduling, the tag of an operation is determined assuming that the operation is executed in a C-select manner for every conditional branch, and the final tag is determined during scheduling. Examples of tags are given in Figures 7.4, 7.6 and 7.7.

Our algorithm assigns a tag \(^2\) to each operation in the data flow graph. The tag of a node represents the condition selections which activates the node. A tag is an unordered set of elements representing condition selection of the conditional branches. Each element contains two fields: the condition field and the selection field. The condition field represents the boolean values of the conditional branch. The selection field represents either true or false decision which activates the node. In this paper, we use an integer for the condition field and T and F for the selection field. A unique integer is used for each distinct boolean value incident to a CB node.

For example, a tag of (1T 2F) means that the node is activated if the condition 1 is true and condition 2 is false. Since unconditional operations are activated regardless of any condition selection, they have an empty tag denoted by ( ).

If a node is activated with the true selection of a certain condition and another node is activated with the false selection of the same condition, they are mutually exclusive. Thus, two nodes with tags (1T 2F) and (2T 3T) are mutually exclusive. For correct design we use an integer for the condition field and T for the selection field. A unique integer is assigned to each distinct boolean value incident to a CB node. This guarantees that two nodes are mutually exclusive. For example, nodes with tags (1F) and (2T) respectively are not mutually exclusive without additional information. Of course, there exist cases in which they could be mutually exclusive. If condition 1 is (a + b == 0) and condition 2 is ((a == 0) & (b == 0)), they become mutually exclusive. If these types of cases can be detected during data flow graph generation stage, then we can take advantage of them. To summarize, the mutual exclusion test is:

- If the tags of two nodes have an element with same condition fields and different selection fields, then they are mutually exclusive.

We now present a tagging algorithm which will tag the nodes such that the mutual exclusion test can be correctly applied. \(^3\)

Node Tagging Algorithm (before scheduling)

1. A unique integer is assigned to each distinct boolean value incident to a CB node.
2. The tag \(tag_i\) of a node \(node_i\) is determined as follows:
   - (a) if \(node_i\) does not have S- or W-predecessors, \(tag_i\) is empty (node is unconditional).
   - (b) if \(node_i\) is a CE node, \(tag_i\) is the same as its corresponding CB node.
   - (c) if \(node_i\) does not have a W-predecessor its \(tag_i\) is the longest tag of its S-predecessors.
   - (d) \(tag_i\) is the same as its W-predecessor with one more element, whose condition field is determined by the boolean value incident to the CB node and whose selection field is determined by the true or false attribute of the W-edge.

When a node is scheduled before or in parallel with a CB node containing this node (that is, in a D-select fashion), its tag must be changed. In this case, the node is not dependent on the boolean value incident to the CB node any more, and the element in the tag

\(^2\)A tag is similar to condition cube in [7] and [16].

\(^3\)After nodes are tagged, we may perform backward traversing to correct bad programming style using the fact that a node is necessary to be executed only if its outputs are necessary.
corresponding to the CB node is eliminated. Examples are given in Figure 7. Notice that the set of all pairs of mutually exclusive operations after scheduling is a subset of that before scheduling.

Having considered mutual exclusion for operations, we now turn our attention to determining mutually exclusive values which can share registers. A value produced in one time-step and used in another time-step must be stored in a register. We define a \( R(\text{register}) \)-value to be a value which needs to be stored in a register over a time-step. Note that if an S-edge crosses several time-steps, then it has several R-values, one for each time-step. If two R-values are mutually exclusive, then only one register is required to store them. If an S-edge is connected to only one node, it is either an input or an output edge in the data flow graph and the corresponding R-values cannot be mutually exclusive to any other R-value. Thus, our interest lies in the S-edges connecting two nodes. Before scheduling, the source and the sink node are tagged assuming they are C-select. A value corresponding to an S-edge is required if and only if both the sink and the source nodes are activated. The condition for which both nodes are activated can be easily obtained. Of the sink and source nodes, let \( \text{node}_1 \) be the node with longer tag and the other node be \( \text{node}_2 \). Then, we can easily show that the tag of \( \text{node}_1 \) must be a super set of the tag of \( \text{node}_2 \), and if \( \text{node}_1 \) is activated, then \( \text{node}_2 \) must also be activated and the value corresponding to the S-edge is required.

**Edge Tagging Rule (before scheduling)**

- The tag of an S-edge is the longer tag of its source and sink nodes.

During or after scheduling the tags of R-values are determined using the S-edges and scheduling information in a similar way as the nodes. Of course, tags of different R-values of the same S-edge can differ indicating that a certain part of the S-edge can be shared and the other part cannot be shared. The test for mutual exclusion is the same as for operation. Examples of R-value tags after scheduling are shown in Figure 8.

### 4 Conclusion and Future Work

In this paper we have presented a new representation for conditional branches and a new algorithm for mutual exclusion testing. The new representation handles conditional branches correctly and satisfies scheduling as well as binding requirements. The new algorithm for mutual exclusion testing is simple and powerful, and it has been coded and verified with several conditional data flow graphs. Currently, we are developing scheduling algorithms which can take advantage of this representation. We have extended the ideas presented in this paper to representing loops as well.

### References