Instruction Selection Using Binate Covering for Code Size Optimization

Stan Liao   Srinivas Devadas
Department of EECS
Massachusetts Institute of Technology
{syliao,devadas}@rle-vlsi.mit.edu

Kurt Keutzer   Steve Tjiang
Advanced Technology Group
Synopsys, Inc.
{keutzer,tjiang}@synopsys.com

http://rle-vlsi.mit.edu/spam

Abstract—We address the problem of instruction selection in code generation for embedded DSP microprocessors. Such processors have highly irregular data-paths, and conventional code generation methods typically result in inefficient code.

Instruction selection can be formulated as directed acyclic graph (DAG) covering. Conventional methods for instruction selection use heuristics that break up the DAG into a forest of trees and then cover them independently. This breakup can result in suboptimal solutions for the original DAG. Alternatively, the DAG covering problem can be formulated as a binate covering problem, and solved exactly or heuristically using branch-and-bound methods.

We show that optimal instruction selection on a DAG in the case of accumulator-based architectures requires a partial scheduling of nodes in the DAG, and we augment the binate covering formulation to minimize spills and reloads. We show how the irregular data transfer costs of typical DSP data-paths can be modeled in the binate covering formulation.

Keywords—code generation, instruction selection, digital signal processors

I. Introduction

An increasingly common micro-architecture for embedded systems is to integrate a microprocessor or microcontroller, a ROM and an ASIC all on a single IC. Such a micro-architecture can currently be found in many diverse embedded systems, e.g., FAX modems, laser printers, and cellular telephones.

The programmable component in embedded systems can be an application-specific instruction processor (ASIP), a general-purpose microprocessor such as the Sparc, a microcontroller such as Intel 8051, or a digital signal processor such as TMS320C25. This paper focuses on the DSP application domain, where embedded systems are increasingly used. Many of these systems use processors from the TMS320C2x, DSP5600x or ADSP families, all fixed-point DSP microprocessors with irregular data-paths.

Code size matters a great deal in embedded systems since program code resides in on-chip ROM, the size of which directly translates into silicon area and cost. Designers often devote a significant amount of time to reduce code size so that the code will fit into available ROM; exceeding on-chip ROM size could require expensive redesign of the entire IC [7]. As a result, a compiler that automatically generates small, dense code will result in a significant productivity gain as well.

We believe that generating the best code for embedded processors will require not only traditional optimization techniques, but also new techniques that take advantage of special architectural features that decrease code size. This paper presents one of our efforts at developing such techniques. We address the problem of *instruction selection* in code generation for embedded DSP microprocessors. We emphasize decreasing code size, although our techniques can also increase execution speed.

Instruction selection can be formulated as directed acyclic graph (DAG) covering. Conventional methods for instruction selection use heuristics that break up the DAG into a forest of trees, which are then covered optimally but independently [1] [3]. Independent covering of the trees may result in a suboptimal solution for the original DAG. Trees, as a *heuristic formulation*, inherently preclude the use of complex instructions in cases where internal nodes are shared. Alternatively, the DAG covering problem can be formulated as a binate covering problem [11], and solved exactly or heuristically using branch-and-bound methods. We present the basic binate covering formulation of instruction selection in Section III. Unlike the heuristic formulation of trees, a good *heuristic procedure* for solving the covering problem is likely to elude the difficulties faced by trees.

The formulation of Section III ignores data transfer costs between nodes in the DAG. This formulation is used to obtain a preliminary instruction selection where pattern DAGs that cover more than two nodes in the given binary DAG are selected. The binary DAG is transformed into a general DAG and a second step of instruction selection taking into account data transfer costs is performed.

In Section IV, we generalize the work of Aho et al. [2] and give a binate covering formulation for optimal code generation for a one-register machine, which takes into account spill and reload costs. Next we provide a formulation in Section V which takes into account irregular data transfer costs under a more general machine model.

II. Motivating Example

Fig. 1 shows a simplified model of the data-path of Texas Instruments' popular TMS320C25 architecture. The TMS320C25 is an accumulator-based machine. In addition to the usual ALU, there is a separate multiplier which takes input from the T register and memory and places the result in the P register. Note that there are no general-purpose registers other than the accumulator.

An important feature in this architecture and other DSP architectures is that certain instructions assume their operands are in specific locations (registers or memory) and deposit their results in specific registers. For example, the `MPY` instruction assumes that the multiplier and multiplicand come from memory and the T register and writes the result into the P register. Another example is the `ADDT` instruction, which adds an operand from the memory, shifted by the amount specified in the T register, to the accumulator.

It is also not unusual to find complex instructions in DSPs. Typical examples include *add-with-shift* (e.g., TMS320C25 `ADD` and `ADDT`) and *multiply-accumulate* (e.g., DSP56000 `MAC`). Utilizing these instructions is essential to generating compact and efficient code. The conventional heuristic of breaking up a DAG into trees prohibits the use of these complex instructions in the case where internal nodes are shared. In addition, this heuristic may introduce unnecessary stores of intermediate values.

Consider the subject DAG and pattern DAGs shown in Fig. 2. Conventional tree-covering will first break up the DAG at node n_3, thereby prohibiting the use of pattern (d). Fig. 3(a) shows the result-

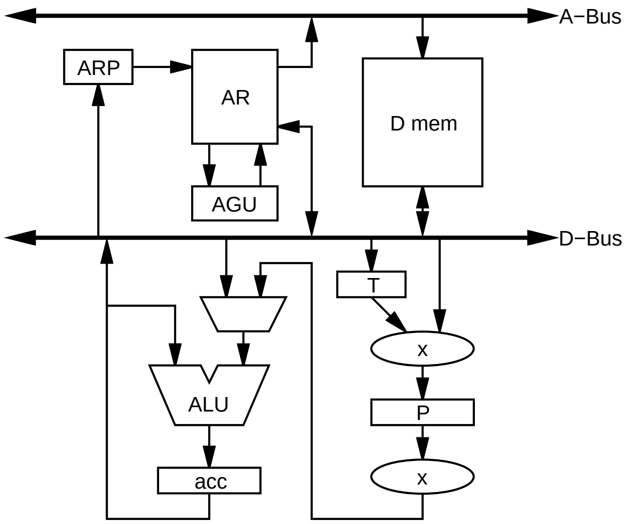


Fig. 1. TMS320C25 data-path (simplified model)

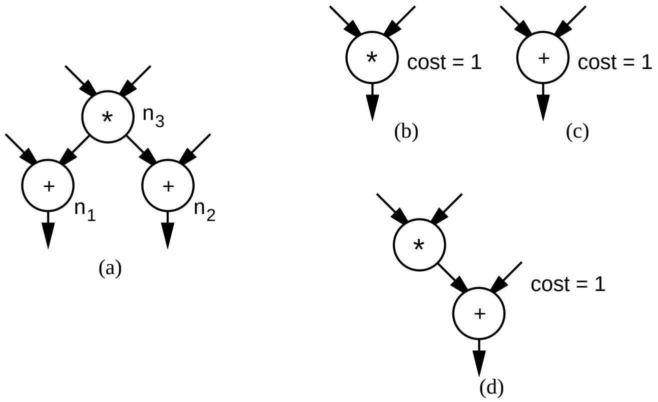


Fig. 2. (a) Subject DAG and (b)–(d) Pattern DAGs

ing tree-cover, and Fig. 3(b) shows the optimal DAG cover. Even if the pattern (d) is not used, tree-covering may still result in inefficient code. For instance, using tree-covering we might first evaluate node n_3, store it into memory, and then evaluate nodes n_1 and n_2. However, with the data-path in Fig. 1, it is possible to let the intermediate result remain in the P register and evaluate n_1 and n_2 using the instruction APAC, which adds the contents of the P register to the accumulator without destroying the former.

In the sequel we will show how to solve both the problem of selecting complex instructions and the problem of data transfers using a binate-covering formulation of instruction selection. The first problem is easily taken into account in the basic DAG covering formulation (Section III). The second problem is tackled in Section IV and Section V.

III. Basic Formulation

The formulation in this section assumes that the target machine is such that data transfers between registers or between registers and memory have zero cost.

A subject DAG corresponds to a basic block in the given program [3]. This subject DAG is covered using pattern DAGs that correspond to individual machine instructions. Each pattern DAG has an associated cost. The DAG covering problem is to cover the subject DAG with a set of pattern DAGs with minimum cost.

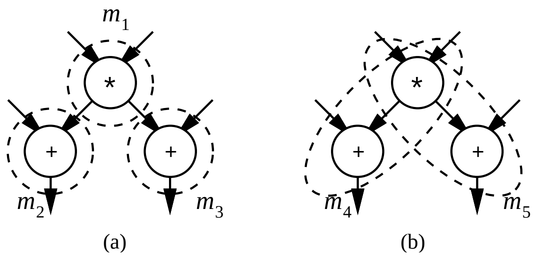


Fig. 3. Two coverings of subject DAG

	m_1	m_2	m_3	m_4	m_5
$m_2 + m_4$	2	1	2	1	2
$m_3 + m_5$	2	2	1	2	1
$\overline{m_2} + m_1$	1	0	2	2	2
$\overline{m_3} + m_1$	1	2	0	2	2

Fig. 4. Covering Matrix

There are three steps associated with DAG covering.

1. All matches of the pattern graphs in the subject graph are generated.
2. A covering matrix is created which expresses the conditions which lead to a legal cover.
3. A cover with minimum cost is obtained using a branch-and-bound algorithm. Alternatively, heuristic methods can be used to find covers with low cost.

Step 1 is a relatively straightforward pattern matching step. A Boolean variable, call it m_i, corresponds to each successful match of a pattern graph in the subject graph G. Let the nodes in the subject graph be $n_j,\ 1 \leq j \leq N$. Each node $n_j \in G$ can be covered by a set of matches $m_{j_1},\ m_{j_2},\ \cdots,\ m_{j_P}$. All possible matches m_1 through m_5 for the example subject DAG are marked in Fig. 3.

Step 2 generates a covering matrix in which each column corresponds to a distinct match m_i. Let there be M columns, $m_i,\ 1 \leq i \leq M$. The rows correspond to disjunctive clauses over the m_i's, and represent covering constraints. In the basic DAG covering formulation, there are two different sets of rows, i.e., disjunctive clauses.

- Rows in the first set represent the different ways that any particular node $n_j \in G$ can be covered using the different matches. For the subject graph of Fig. 2(a) the covering matrix is shown in Fig. 4. The first row in the matrix $(m_2 + m_4)$ corresponds to node n_1 and indicates that node n_1 can be covered either by match m_2 or match m_4, as indicated in Fig. 3. Therefore, we put 1's in the entry corresponding to column m_2 and column m_4 and 2's in other columns. Similarly, the next row indicates that either m_3 or m_5 needs to be selected to cover node n_2. Note that in this first set of rows we only need clauses that cover the *root* nodes, because the selection of a particular match will necessitate the selection of matches that cover nodes connected to inputs (see below).
- Matches are allowed to have nodes internal to the match feed nodes not in the match. This results in a second set of rows. For each match m_i, we have to ensure that the non-leaf inputs to the match are the outputs of other matches. By *non-leaf inputs* we mean *internal nodes* (in contrast to primary inputs) in the DAG that serve as inputs to other nodes. Let the non-leaf inputs to match m_i be $s_{i_1},\ s_{i_2},\ \cdots,\ s_{i_T}$. For each i_k, let W_{i_k} be the set of matches that have s_{i_k} as an output node. W_{i_k} can be viewed as a disjunctive expression over the Boolean variables corresponding to each match.

Selecting match m_i implies that we have to satisfy each of the W_{i_k}. We can write the expression

$$m_i \Rightarrow W_{i_k},\ 1 \leq k \leq T$$

which translates to the clauses

$$(\overline{m_i} + W_{i_1}) \cdot (\overline{m_i} + W_{i_2}) \cdots (\overline{m_i} + W_{i_T}).$$

Each of these clauses corresponds to a distinct row in the covering matrix. Each match m_i generates T additional rows if it has T non-leaf inputs.

In the covering matrix of Fig. 4, the second set of rows correspond to these additional clauses. For match m_2, we have to implement the non-leaf node n_3 as the output of another match. This can be done using match m_1 alone. Therefore, we generate the clause $(\overline{m_2} + m_1)$, corresponding to the third row. Since m_2 is complemented in the clause we put a 0 in the entry corresponding to column m_2. We put a 1 in the entry corresponding to column m_1 and 2's in other columns. The fourth row is generated for match m_3, which if selected would require the selection of m_1.

The cost of a match $cost(m_i)$ is simply the cost of its associated pattern DAG. In Step 3, we select a set of columns from the covering matrix such that the cumulative cost of the columns is minimum, and such that every row either has a 1 in the entry corresponding to a selected column, or has a 0 in the entry corresponding to an unselected column. In our example, we will end up selecting m_4 and m_5 with a minimum total cost of $1 + 1 = 2$; this corresponds to the covering of Fig. 3(b). The reader can verify that selecting m_4 and m_5 satisfies all the disjunctive clauses of Fig. 4.

As an aside note that tree covering methods would not be able to discover the optimal solution of Fig. 3(b) since the subject DAG would be broken up into three trees, which when covered independently would result in the covering of Fig. 3(a) that has a cost of $1 + 1 + 1 = 3$.

This problem is called the binate covering problem because the variables m_i are present in their true and complemented forms. This problem is NP-complete, and has received considerable attention. Exact solutions are given in [5], [9]. These techniques have been improved recently in [6] without compromising optimality. Heuristic methods have been given in [8], [9].

We first solve the binate covering problem with zero data transfer costs and determine matches that use pattern DAGs with more than one operator, e.g., the pattern DAG of Fig. 2(b). The original DAG is modified to reflect the use of complex operators. Thus, the new DAG can have nodes with more than two inputs. A second step of binate covering is performed on the new DAG that accurately models spill and data transfer costs. This step is described in the next two sections.

IV. Data Transfer Costs in One-Register Machines

We focus on one-register machines, or accumulator-based architectures. In such architectures, accumulator spills to memory and reloads from memory can account for a large fraction of the instructions. The binate covering formulation must take this cost into account in order to find an optimal instruction selection.

The major complication in modeling memory spills is that the spilling of values depends on the chosen instruction schedule [10]. However, since we are performing instruction selection we do not as yet know the schedule. We therefore have to both choose the instructions and determine a (partial) schedule of these instructions in binate covering. The partial schedule is determined by adding Boolean variables corresponding to adjacency constraints over pairs of nodes in the DAG that are connected by an edge.

A. Previous Work

In [2] Aho et al. presented optimal code generation algorithms (on DAGs) for two different models of one-register machines:

- Non-commutative machines, in which available operations are:
  1. $a \leftarrow a$ **op** m
  2. $a \leftarrow m$ (load)
  3. $m \leftarrow a$ (store)

  where a denotes the accumulator and m denotes memory.
- Commutative machines, in which available operations are:
  1. $a \leftarrow a$ **op** m
  2. $a \leftarrow m$ **op** a
  3. $a \leftarrow m$ (load)
  4. $m \leftarrow a$ (store)

We find the models above inadequate for the following reasons. First, in our application the given DAG can have ternary or higher-arity operators depending on the complex patterns chosen in the first step of binate covering. Second, the non-commutative model of [2] does not take commutative operators into account. For example, in evaluating the expression $(b + c)$, the value of b must be first loaded to the accumulator and then added with c. However, if b and c are themselves expressions, the accumulator may already contain c immediately before the evaluation of $(b + c)$. Since addition is commutative, adding the accumulator with b is perfectly acceptable. The commutative model, on the other hand, assumes that the first operand of *any* operation can be in memory. In general, machines will have both commutative and non-commutative operators.

We believe the best way to handle commutativity is to treat each operation independently, using a separate pattern for the commutative forms of the operations wherever necessary, rather than assuming commutativity in the machine model.

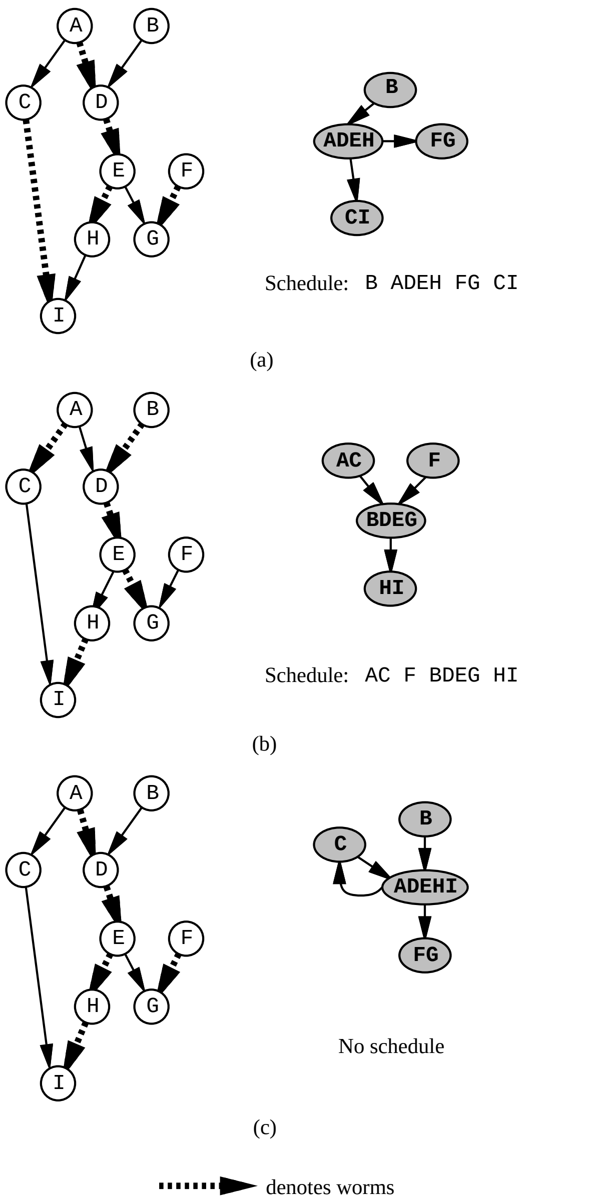
We present a compact binate-covering formulation for the optimal code generation for the non-commutative one-register machine taking into account the commutativity of individual operators in the following sections. The operators can be binary, ternary or higher-arity operators. For ease of exposition we will concentrate on binary operators; however, the techniques generalize to higher-arity operators.

B. Definitions

Let H be a directed graph. A *u-cycle* in H is a set of edges that would form a cycle if the edges were considered undirected. If H contains a u-cycle, it is said to be *u-cyclic*; otherwise, it is *u-acyclic*. We use the terms *d-acyclic*, *d-cyclic*, and *d-cycle* for the case where the directions of the edges are considered.

A *worm* is a directed path in a DAG D such the nodes in the path will appear *consecutively* in the schedule [2]. A *worm-partition* of D is a set of disjoint worms. An edge is said to be *selected* with respect to a worm-partition if it belongs to some worm in the partition. Associated with a worm-partition is a directed graph G. Each node of G corresponds to a worm in D, and there is an edge between nodes w_1 and w_2 of G whenever there is an edge in D between some node of worm w_1 and some node of worm w_2. We can think of deriving G from D (given a worm-partition) by successively merging the nodes that are connected by selected edges (*imploding* the edge).

A worm-partition is said to be *legal* if a valid schedule can be derived from G such that the nodes of each worm appear consecutively in the schedule. Henceforth we shall denote by D the original expression DAG, and by G the induced graph of a worm-partition of D. A sufficient condition for a worm-partition to be legal is that G is d-acyclic [2]. (This condition, however, is not always necessary. See Theorem 4.)

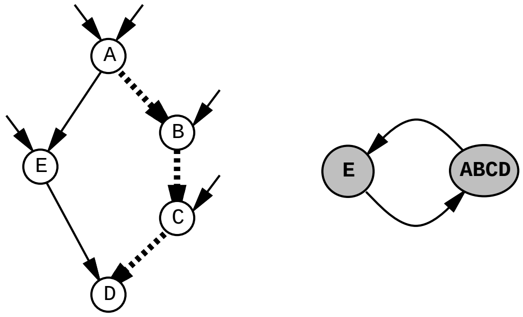


(a)

(b)

(c)

Fig. 5. Worms and schedules. (a) A DAG with its worm-partition and a derived schedule (b) Another worm-partition and schedule (c) An illegal worm-partition



(a) (b)

Fig. 6. (a) Reconvergence in a DAG with a worm (b) A cycle in G due to the worm

Fig. 5 illustrates the concepts of worms and worm-partitions, and their relation to scheduling. The graph with shaded nodes is that of G. In (a) and (b), two different worm-partitions of the DAG, along with their corresponding schedules, are shown. The schedules are derived by scheduling G and then expanding the nodes of G back into nodes of D. Note that in each schedule the nodes in a worm are placed consecutively. In (c), an illegal worm-partition is shown. This partition gives rise to a cycle in G, and no schedule exists that places the nodes in each worm consecutively.

C. Fundamental Adjacency Clauses

Because the selection of an edge indicates that its head-node and tail-node will be placed adjacently in the schedule, the following *fundamental adjacency clauses* (or fundamental clauses) must be satisfied.

> If a node has multiple fanouts, then at most one of the fanout edges may be selected. If a node has multiple fanins, then at most one the fanin edges may be selected.

Clearly the fundamental clauses are necessary for a worm-partition to be legal in any DAG. (Simply stated, in any schedule each node may have at most one immediate predecessor and at most one immediate successor.) Let e_i be a Boolean variable which takes the value of 1 if edge i is selected, and 0 otherwise. The fundamental clauses are, for each node n,

$$\overline{e_i} + \overline{e_j} \qquad (1)$$

for every pair of fanout edges i and j of n, and for every pair of fanin edges i and j of n. We have to satisfy each clause generated above. The cost of all the e_k's is equal to zero. However, not choosing an e_k will imply spilling and reloading and associated costs, as described in Section IV-E.

The following theorem shows that these fundamental clauses are sufficient for u-acyclic DAGs.

Theorem 1: If the subject DAG D is u-acyclic, then the fundamental clauses are sufficient. In other words, any worm-partition that satisfies the fundamental clauses is legal.

Proof: If D is u-acyclic, then selecting an edge and merging the head- and tail-nodes of the edge results in a DAG that remains u-acyclic. By repeating this process we cannot possibly create a u-cycle. Therefore, by merging the nodes according to the selected edges of the worm-partition, no u-cycle (much less a d-cycle) will appear in G. This implies that the worm-partition is legal. ■

If there are u-cycles in D the fundamental clauses become insufficient. A good example is one of reconvergent paths (Fig. 6). Note, on the other hand, that selecting an edge that is not part of any u-cycle in D will not create a d-cycle in G. Thus we only need to focus on writing additional clauses for u-cycles.

D. Clauses for U-Cycles

Since u-cycles in D may lead to d-cycles in G, we need to add clauses that prevent this from happening. Let C be a u-cycle of D, and arbitrarily choose a direction of traversal on C as the forward direction, and label the edges as forward and backward accordingly.

Theorem 2: If all forward edges (or all backward edges) in a u-cycle are selected, then imploding the selected edges will result in a d-cycle. Conversely, if at least one forward edge and at least one backward edge are not selected, then the u-cycle remains d-acyclic after implosion.

Proof: ⇒: If all forward edges in a u-cycle are selected, then in the imploded u-cycle only the backward edges remain. Since

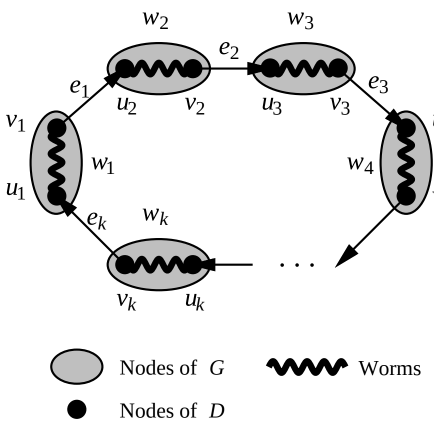


Fig. 7. From a d-cycle of G we can always find a u-cycle in D that produced it

all the edges of the imploded u-cycle are in the same direction, the imploded u-cycle is also a d-cycle.

⇐: If at least one forward edge and at least one backward edge are not selected, then the imploded u-cycle have at least two edges pointing to the opposite directions; hence, the imploded u-cycle remains d-acyclic. ■

Therefore, it is necessary that for each cycle we do not select all edges of the same orientation. Is it possible that, even if the selected edges satisfy this condition for every cycle (including the composite cycles), there is still a d-cycle in G? That is, is it possible that a d-cycle in G arises from another cause than u-cycles in D? The following theorem shows that this is impossible, thereby establishing the sufficiency of this condition.

Theorem 3: If G is d-cyclic, then there exists a u-cycle in D of which all the forward edges (or all backward edges) are selected. Therefore, if the clauses derived from Theorem 2 are satisfied for every u-cycle in D, then G is d-acyclic.

Proof: Let $w_1, w_2, \ldots, w_k$ be the nodes of a d-cycle in G, which also denote the corresponding worms in D. By the definition of G, there exist nodes $v_1 \in w_1$ and $u_2 \in w_2$ such that an edge $e_1 = (v_1, u_2)$ exists between them. Similarly, there exist edges $e_i = (v_i, u_{i+1})$ for $i = 2, \ldots, k-1$, and $e_k = (v_k, u_1)$ (Fig. 7). Since v_i and u_i are nodes in a worm, there is a path between them (in one direction or the other). Denote by P_i the path between v_i and u_i. Now $(P_1, e_1, P_2, e_2, \ldots, P_k, e_k)$ form a u-cycle in D. Furthermore, regardless of the orientations of the paths P_i's, in this u-cycle every edge in the direction opposite to the e_i's is selected (recall all edges of P_i are selected edges). ■

It turns out that we can compactly write clauses to require that at least one forward edge and one backward edge are selected, as follows. Let $f_1, f_2, \ldots, f_k$ be the Boolean variables for the forward edges of a u-cycle in D. The clause

$$\overline{f_1} + \overline{f_2} + \cdots + \overline{f_k} \qquad (2)$$

will ensure that not all of these edges are selected. A similar clause is written for the backward edges. Hence, two clauses for each u-cycle suffice. No new variables are introduced into the formulation merely additional clauses.

One important exception needs to be made regarding self-loops in G (which was not addressed in [2]). Consider the portion of a DAG shown in Fig. 8. If we choose the path A→B→C→D (which are all of the forward edges in this u-cycle), then a d-cycle results in G (in accordance with Theorem 2). However, this d-cycle is a *self-loop*. It can be easily verified that the worm thus chosen is actually legal: the schedule ABCD, in which the nodes of the worm (namely, A–D) appear consecutively, is admissible. This example shows that self-loops do not make the worm-partition illegal. The lemma and the theorem that follow state this formally.

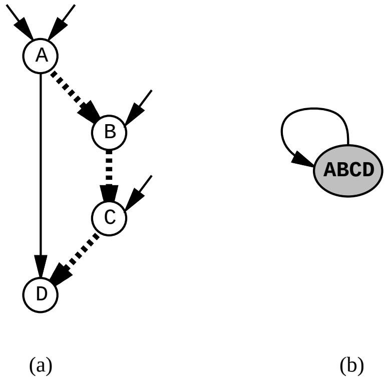


Fig. 8. (a) A portion of a DAG D with a selected worm (b) The induced self-loop in G

Lemma 1: Let C be a self-loop of G. The corresponding u-cycle in D must consist of two reconvergent paths. Furthermore, if u and v are the end-points of the reconvergent paths, one of these paths must be an edge from u to v.

Proof: The loop-edge of C corresponds to a single edge from some node u of the worm to some other node v of the worm. Other than the edge (u, v), there is another path P from u to v (the worm). With respect to D, u must be a predecessor of v (otherwise the DAG would not be d-acyclic). Thus we have the two reconvergent paths: the edge (u, v) and the path P. ■

Theorem 4: If all d-cycles of a worm-partition G are self-loops, then G is legal. Conversely, if a d-cycle of G contains more than one node, then G is illegal.

Proof: Self-loops arise solely from the kind of reconvergent paths described in Lemma 1, with the long path being part of a worm. Thus when we schedule these nodes consecutively, the precedence relation required by the edge (u, v) is not violated. On the other hand, if there are more than one node in a d-cycle of G, scheduling the nodes of one worm consecutively will be unsuccessful because some node of the current worm depends on some node of another worm, which in turn depend on the current worm. ■

In light of Theorem 4, Clauses (2) are not required for self-loops. Instead, a clause consisting of a single variable requiring the edge (u, v) *not* to be selected is prescribed—clearly, choosing the edge (u, v) would lead to a non-trivial d-cycle in G. If, on the other hand, the reconvergent path is itself a single edge (e.g., when an operator takes both operands from the same node), then neither the cycle- nor the reconvergence-clauses is necessary—the fundamental clauses described in Section IV-C ensure that at most one of these edges is selected.

E. Clauses for Reloads and Spills

Depending on where an operation takes its operands from and which edges are selected, spills and reloads may be required between computations. We now describe precisely how to write clauses to activate spills and reloads.

Fig. 9(a) shows a fragment of a DAG. Consider the edge (C,B), whose corresponding Boolean variable is e_2. There are four cases to consider for this edge:

1. Match m_1 is used and $e_2 = 1$. Since m_1 requires its left-operand from the accumulator, and B is immediately after C,

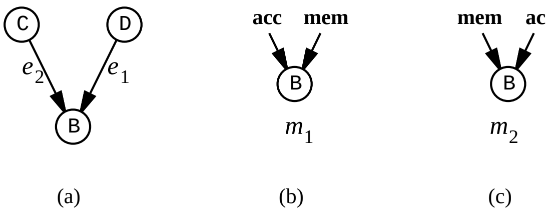


Fig. 9. Spilling and reloading according to adjacency of nodes

no spill on C or reload on the edge (C,B) is necessary.

2. Match m_1 is used and $e_2 = 0$. In this case, a spill on C is required, because a node other than B immediately follows C and destroys the contents of the accumulator, but this value is needed by B later. Also, a reload is necessary immediately before B is scheduled, because m_1 takes its left-operand from the accumulator.
3. Match m_2 is used and $e_2 = 1$. Even though B immediately follows C, a spill is still required because m_2 takes its left-operand from the memory. No reload is necessary.
4. Match m_2 is used and $e_2 = 0$. As in the previous case, only a spill is required.

Let **spill**(C) denote the match that transfers the value of C from the accumulator to the memory immediately after C is computed, and **reload**(C,B) denote the match that loads the value of C from the memory to the accumulator immediately before B is scheduled. We can then express the above conditions by the following clauses:

$$\overline{m_1} + e_2 + \mathbf{spill}(\mathtt{C}) \tag{3}$$

$$\overline{m_1} + e_2 + \mathbf{reload}(\mathtt{C},\mathtt{B}) \tag{4}$$

$$\overline{m_2} + \mathbf{spill}(\mathtt{C}) \tag{5}$$

Similar clauses are prescribed for edge e_1, as well as every other node and all possible matches on it.

Given the DAG with the complex patterns selected, the fundamental clauses, u-cycle clauses and the clauses for reloads and spills are added to the clauses for the node matches. The clauses for the node matches are very simple for a one-register machine, since the only choices for a node are where the inputs come from, as shown in Fig. 9.

V. Data Transfer Costs in Multiple-Register Machines

A. Target Architecture

We will now assume that the target architecture can be modeled conveniently with the $[1, \infty]$ *model* [4]. In the $[1, \infty]$ model each resource class is assumed to have either one element or an infinite number of elements. For resource classes that have more than one element, we will perform a separate pass of storage allocation at or after scheduling, as in [12].

As shown in Section II, the typical fixed point DSP has irregular data-paths, and certain registers have specialized uses. Consequently, at completion of an operation its results may not be available for use by another operation that takes operands from other registers. For example, the MPY operator requires one of its operands to come from the **treg** register and the other from the memory. Thus a *data transfer* is necessary to move the operands to the desired register(s). Tree-covering methodology models the cost of this transfer is modeled by associating a cost with a *unit production*, a production with a non-terminal in each of the left- and right-sides. In this section, we show how to incorporate data transfers into the binate covering formulation.

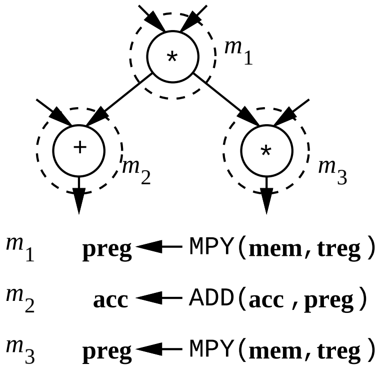

*
m_1
+
m_2
*
m_3
m_1 preg ← MPY(mem, treg)
m_2 acc ← ADD(acc, preg)
m_3 preg ← MPY(mem, treg)

Fig. 10. Data Transfers

B. Example

Consider a fragment of an expression DAG in Fig. 10. The operation covered by match m_3 requires that its left operand come from the memory and its right operand come from the **treg** register. However, the match m_1 produces its result in the **preg** register. A match m_4 that transfers the contents of the **preg** register to memory is required. Hence, we write:

$$\overline{m_3} + \overline{m_1} + m_4 \tag{6}$$

to require the selection of match m_4 in the event that both m_1 and m_3 are selected. Similar clauses are also prescribed for other matches on node 1.

C. Constructing the Clauses

Based on the example in Section V-B, we now describe a general procedure for constructing the clauses necessary for data transfers. We add these clauses to all the clauses summarized at the end of Section IV. For every pair of nodes n_1 and n_2 in the given DAG connected by an edge, for each possible match m_i on n_1 and each possible match m_j on n_2, we will write

$$\overline{m_i} + \overline{m_j} + q_{ij} \tag{7}$$

where matching q_{ij} indicates a transfer of the result of match m_i to the location required by m_j. If m_i results in writing an operand into memory, and m_j requires reading from register **preg**, then q_{ij} will correspond to a match that moves data from memory to **preg**. Similarly, for other moves across different register classes. If m_i writes into a register **acc** and m_j reads from the same register **acc**, q_{ij} is the disjunction of an adjacency constraint between the output node of m_i and the input node of m_j, and a spill/reload match (cf. Section IV-E).

We assume that this adjacency constraint has to be satisfied in the schedule to guarantee a correct data transfer without a spill/reload. Satisfying the adjacency constraint is not necessary for correct data transfer in multiple-register machines; in-between instructions can exist but should write registers other than **acc**. However, relaxing this assumption would require a life-time analysis of registers and a very large number of clauses. After instruction selection and partial scheduling using binate covering an optimized complete schedule can be generated which exploits life-time analysis.

In some cases, due to data-path constraints, it is not possible to move the contents of one location to another location via a single move. For example, suppose there is no direct path from the **preg** register to the memory, and the only way to accomplish the move from **preg** to memory is through the accumulator. In this case, two moves will be required, and q_{ij} will represent the conjunction of the two corresponding matches.

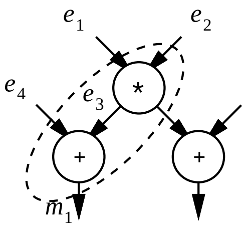


Fig. 11. Matches altering fundamental clauses

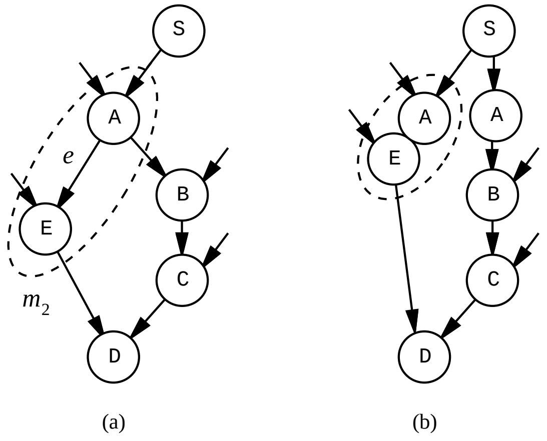


Fig. 12. (a) A u-cycle in a DAG (b) Modified u-cycle

VI. Discussion

We presented a two-pass strategy. The first step selects complex operators, and the second step selects matches that minimize data transfer costs on a transformed DAG. Can both these steps be performed simultaneously by solving a single binate covering problem?

The answer is yes, but the number of clauses in binate covering can become very large. The reason is that the selection of complex operators affects the fundamental adjacency clauses and the clauses for u-cycles.

The fundamental clauses corresponding to the marked edges of the DAG of Fig. 11 are $\overline{e_1} + \overline{e_2}$, and $\overline{e_3} + \overline{e_4}$. However, if match m_1 is selected, then the fundamental clauses should become $\overline{e_1} + \overline{e_2}$, $\overline{e_1}+\overline{e_4}$, and $\overline{e_2}+\overline{e_4}$. This can be incorporated by writing the following clauses: $\overline{e_1} + \overline{e_2}$, $\overline{e_3} + \overline{e_4} + m_1$, $\overline{e_1} + \overline{e_4} + \overline{m_1}$, and $\overline{e_2} + \overline{e_4} + \overline{m_1}$. This has to be done for *each* match which corresponds to a complex pattern that covers any edge of the DAG, in the manner that m_1 covers e_3 in our example above.

If the DAG is u-acyclic the fundamental clauses are sufficient, and the above modification will be enough. However, u-cycle clauses have be modified in the general case. This modification can result in a very large number of clauses, since choosing a complex pattern can change the u-cycles of a DAG. To understand this consider Fig. 12. If match m_2 is selected in Fig. 12(a), then in effect, the DAG is modified to the one shown in Fig. 12(b). There is a new u-cycle beginning from the node S! This means that we have to write clauses corresponding to this new u-cycle when match m_2 is selected, and when it is not. Note that if all u-cycles begin from level 1 nodes in the DAG, i.e., nodes whose inputs are leaves, then no new u-cycles will be introduced due to complex operators. Even if a new u-cycle is not generated, we still have to modify the original u-cycle clauses since edge e is covered by m_2, as in the fundamental clause case.

VII. Summary and Ongoing Work

We have presented a formulation of the instruction selection problem as that of binate covering. This formulation captures data transfer and memory spill costs commonly associated with DSP processors.

Our preliminary experiments indicate that exact binate covering can be applied to small-to-moderate sized basic blocks for the TMS320C25 processor. These optimal solutions are better than those produced by the tree covering heuristic in many cases. For large basic blocks or entire procedures, however, computationally efficient heuristic strategies are required. Two avenues are being explored. First, large basic blocks will be broken into simpler blocks which can be covered using the exact binate covering algorithm. Second, heuristics which restrict the number of matches and therefore clauses in the covering matrix will be investigated.

VIII. Acknowledgements

We thank Richard Rudell and Olivier Coudert for help with the binate covering formulation. This research was supported in part by the Advanced Research Projects Agency under contract DABT63-94-C-0053, and in part by a NSF Young Investigator Award with matching funds from Mitsubishi Corporation.

References

[1] A. Aho and S. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23:488–501, July 1976.

[2] A. Aho, S. Johnson, and J. Ullman. Code generation for expressions with common subexpressions. *Journal of the ACM*, pages 146–160, January 1977.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[4] G. Araujo and S. Malik. Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures. In *Proceedings of 1995 International Symposium on System Synthesis*, 1995.

[5] R. K. Brayton and F. Somenzi. Boolean Relations and the Incomplete Specification of Logic Networks. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 316–319, November 1989.

[6] O. Coudert and J-C. Madre. New Ideas for Solving Covering Problems. In *Proceedings of the 32nd Design Automation Conference*, pages 641–646, June 1995.

[7] J. G. Ganssle. *The Art of Programming Embedded Systems.* San Diego, CA: Academic Press, Inc., 1992.

[8] J. Gimpel. The Minimization of TANT Networks. *IEEE Transactions on Electronic Computers*, EC-16(1):18–38, February 1967.

[9] A. Grasselli and F. Luccio. A Method for Minimizing the Number of Internal States in Incompletely Specified Machines. *IEEE Transactions on Electronic Computers*, EC-14(3):350–359, June 1965.

[10] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Code Optimization Techniques in Embedded DSP Microprocessors. In *Proceedings of the 32nd Design Automation Conference*, pages 599–604, June 1995.

[11] R. Rudell. Logic Synthesis for VLSI Design. In *U. C. Berkeley, ERL Memo 89/49*, April 1989.

[12] A. Sudarsanam and S. Malik. Memory Bank and Register Allocation in Software Synthesis for ASIPs. In *Proceedings of the International Conference on Computer-Aided Design*, 1995 (this volume).