Using Minimal Minterms to Represent Programmability

Scott J. Weber and Kurt Keutzer
University of California, Electronics Research Lab, Berkeley
{sjweber, keutzer}@eecs.berkeley.edu

Abstract
We address the problem of formally representing the programmability of a system. We define the programmability of a system as the set of valid execution paths that can be configured statically by software. We formally represent this programmability as a Boolean function. From this representation, we extract a subset of on-set minterms that we call minimal minterms. We prove that these minimal minterms represent the set of smallest schedulable atomic actions of the system, and that we can use a special generator relation to determine if subsets of these actions can be executed in parallel. We also prove that given an arbitrary Boolean function we can extract the minimal minterms and recreate the entire on-set by applying the generator relation to every element of the power set of the set of minimal minterms. Thus, the minimal minterms represent the complete instruction set supported by the system, and the generator relation represents the inherent parallelism among the instructions. Furthermore, we automatically generate the required software development tools and hardware implementation from this representation of programmability. Finally, we show that we can efficiently compute the minimal minterms and apply the generator relation to verify parallel executions on interesting data path systems.

Categories and Subject Descriptors: C.0


Keywords: Instruction set extraction, Boolean function representation, Boolean satisfiability.

1. Introduction
Traditionally, designers have first specified an ISA to represent the programmability and then implemented a micro-architecture to implement the ISA. However, for statically-scheduled, horizontally-microcoded machines that combine the programmability of a conventional processing element with the performance of an ASIC, the distinction between the ISA and micro-architecture is unclear. In this paper, we particularly explore a methodology in which an experienced ASIC designer can define a data path and then automatically extract the instruction set for such sub-RISC machines.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

If we forego binary compatibility, we can then remove the need to specify an ISA. We instead extract the ISA directly from the micro-architectural implementation. In order to enable this extraction, we have developed a formal model where the micro-architecture is represented as a data path and constraints on the valid uses of the data path. The model is expressed as a single characteristic Boolean function where satisfying solutions to the function represent the valid uses of the data path. We define the programmability of the system as precisely these valid uses. Although this model does implicitly represent the programmability, it is not that useful unless we can factor the potentially immense number of satisfying solutions into a small set of schedulable atomic actions (i.e. an instruction set).

In order to extract instructions, we first developed a new method to represent a Boolean function as a set of on-set minterms called minimal minterms and a special generator relation. From this set of minimal minterms, we can then create the entire set of on-set minterms by applying the generator relation to elements of the power set of the set of minimal minterms. The generator relation first creates a minterm by applying bit-wise OR to a subset of minimal minterms. The resulting minterm is then tested to determine if it is in the on-set. We prove that this technique represents a cover of all the on-set minterms.

Although we can represent a Boolean function with this technique, we are more interested in how to use it to extract an ISA. As we will show, the minimal minterms represent the set of smallest schedulable atomic actions. This set of schedulable actions is precisely the instruction set. Moreover, the parallelism between instructions can be determined by applying the generator relation to subsets of the instructions. Furthermore, the characteristics of our model are such that we can determine if any set of instructions cannot be executed in parallel by simply checking if any pair in the set is in conflict. Thus, we can efficiently represent the programmability with a set of minimal minterms and a conflict table. Finally, from this representation, we can automatically generate the software development tools and a hardware implementation.

The paper is organized as follows. In Section 2, we describe our minimal minterms method, prove its correctness and demonstrate its use with a small example. In Section 3, we show how to apply the minimal minterms method in our methodology to allow us to efficiently represent the programmability of a system. In Section 4, we discuss the results of using this technique in general, and we demonstrate its application with respect to our methodology on a few interesting data path systems. Finally, we conclude the paper in Section 5.

2. Minimal Minterms Method
Our method represents a Boolean function by using a unique set of on-set minterms called minimal minterms and a generator relation. This unique set is found using our findMinimalMinterms algorithm. An application of the generator relation to any subset of the minimal minterms is then used to
create a minterm. A satisfiability test is then applied to determine if the minterm is in the on- or off-set. We will prove that this method allows us to find all the on-set minterms of the function. Before we describe and prove the correctness of this method, we need to first define a few key terms.

2.1 Definitions

We assume that we are given a Boolean function and we are interested in finding the on-set minterms. To do this, we find a set of on-set minterms that are minimal. We then use the max function to create all on-set minterms from the set of minimal minterms where max and minimal are defined as follows.

Definition 1 A max of a set of minterms is the bit-wise OR of the binary representations of these minterms.

From now on when we use minterm, we will implicitly mean an on-set minterm unless otherwise specified.

Definition 2 A minterm is minimal iff it cannot be created by applying max to a subset of the other minterms.

Theorem 1 A Boolean function has a unique set of minimal minterms.

Proof For each minterm $m_{on}$, attempt to create it by applying max to the elements in the power set of the other minterms. If none of these applications of max returns $m_{on}$, then it is minimal; otherwise, it is not. This test uniquely labels the set of minimal minterms.

Definition 3 A differentiating literal is a literal that makes a minterm minimal.

Now that we have the appropriate definitions, we next propose a method for finding minimal minterms.

2.2 Finding Minimal Minterms

We have developed an algorithm called **findMinimalMinterms** (FMM) that returns the set of minimal minterms for a Boolean function. The function must be in conjunctive normal form (cnf) because we utilize a satisfiability (SAT) solver. The algorithm is shown in Figure 1.

Lemma 1 Any minterm found by FMM cannot be created by applying max to a subset of the previously found minterms.

Proof Lines (10) and (11) create a constraint so that at least one differentiating literal must be positive. If this literal is positive, then its corresponding literal must be positive. By line (9), these constraints are only added once after the first minterm is found because the constraints do not change between iterations, and this theorem is vacuously true on the first iteration. Lines (12) and (13) indicate that if the set of positive literals that define a minterm are asserted, then none of the associated differentiating literals can be positive. Lines (10) through (13) together indicate that any subsequent satisfying minterm must have at least one positive literal that is not associated with any previously found minterm that is covered by the satisfying minterm. This literal therefore precludes finding minterms that can be created by applying max to a subset of the previously found minterms.

Lemma 2 A minterm cannot be created by setting a subset of positive literals to negative in a minterm previously found by FMM.

Proof Assume such a minterm can exist. By lines (2) through (7) an earlier attempt to set positive literals to negative would have found this minterm. This is a contradiction.

Theorem 2 FMM finds the unique set of minimal minterms.

Proof FMM only finds minimal minterms: Assume that there is a minterm $m_{on}$ found by FMM that is not minimal. By Lemma 1, a set must then exist that contains at least one minterm that has not been found previously. It must also include some subset of minterms such that $m_{on}$ is created when max is applied to the set. Furthermore, all minterms from this set must have the same negative literals as $m_{on}$ because it is impossible to apply max on this set to get $m_{on}$ (bit-wise OR is monotonically increasing). However, by Lemma 2, any minterm in this set would have been found previously by FMM because you could create the minterm by setting a subset of the positive literals in $m_{on}$ to negative. This contradicts the previous requirement imposed by Lemma 1. Thus, FMM cannot find non-minimal minterms.

FMM finds all the minimal minterms: Assume there is a minterm $m_{on}$ that is minimal that is not found by FMM. By lines (2) and (7), FMM iterates until it cannot find anymore minterms. It has already been proven that non-minimal minterms cannot be found. By Theorem 1, there is a unique set of minimal minterms. Thus, FMM finds the unique set of minimal minterms.

In Figure 2, we graphically illustrate the process of discovering minimal minterms by the FMM algorithm. In a), we show the minimal minterm with no positive literals. If this minterm exists, it will always be found first. In b) and c), we see that the minterm in b) is not minimal because a subset of its positive literals can be set to negative to get the minimal minterm in c). A minimal minterm can contain a disjoint set of positive literals as shown in d). Likewise, as shown in e), a minimal minterm may share positive literals with the existing minterms as long as the minterm cannot be created by applying max to the existing minimal minterms like the one shown in f). Minimal minterms are allowed to cover an existing minterm, but they must always introduce at least one positive differentiating literal in order to be minimal. An example of this is shown in g). Finally, although a minimal minterm cannot cover existing solutions like in h), a
minimal minterm can be composed of subsets of positive literals from different existing minterms as shown in i).

![Figure 2. FMM Algorithm Illustrated](image)

### 2.3 Generating All Minterms

After finding the set of minimal minterms, we then apply max to subsets of the set of all minimal minterms to create minterms. We create a conjunction of a created minterm and the base function to determine if the resulting minterm satisfies the base function. If it does then it is in the on-set; otherwise, it is in the off-set. Given the set of minimal minterms and max, we now have a method to create the entire set of minterms.

Let F be a Boolean function and M be the set of minterms in the on-set of F. Let MM be the set of minimal minterms found by FMM for F.

**Theorem 3** Any non-minimal minterm in the set NMM = \{m_{on} | m_{on} ∈ M and m_{on} ∉ MM\} can be created by applying max to a subset of MM and then determining if the resulting m_{on} satisfies the function F.

**Proof** Assume that there is a minterm m_{on} in NMM that cannot be found by applying max to a subset of MM. FMM continues until there are no minterms that cannot be created by applying max to a subset of previously found minterms. This follows from the proof of Lemma 1 and the fact that FMM continues until there are no more satisfying minterms. Thus, FMM would have found m_{on} ∈ MM. The other possibility is that MM is not complete. By Theorem 2, FMM finds the unique set of minimal minterms. Thus, MM is complete. This is a contradiction because it was assumed that m_{on} ∉ MM.

We now provide a theorem to find all the minterms.

**Theorem 4** The set of all on-set minterms of F is found by finding MM and then applying max to each element of the power set of MM. A satisfiability test is applied to the minterm that results from applying max to determine if it is in the on-set of F.

**Proof** The power set represents all combinations of minimal minterms. By Theorem 3, any non-minimal minterm can be found from MM. Thus, all non-minimal minterms will be found. The union of the non-minimal minterms and minimal minterms represents the entire on-set. Thus, all the minterms are found.

This method may find duplicate minterms in the case where one minimal minterm is the same as a previous except that more literals are positive. Simply remove redundant solutions to get the set of all minterms.

### 2.4 Example

To get a better insight into FMM, we show a small example. In Figure 3, we show the covering of F as a prime cube cover in a) and as the FMM minimal minterms in b). In c), the minimal minterms are labeled, and their associated constraint, as created in FMM, is shown. To cover the minterms not covered by the minimal minterms, we apply max to the power set of the minimal minterms as shown in d). We skip s0 because adding it to the input set of max does not change anything. Each minterm that results by applying max is denoted with its input set, on-set status, and associated minterm. Note that \{s1, s3\}, \{s3, s4\} and \{s1, s2, s4\} cover the uncovered minterms of b).

![Figure 3. FMM Example](image)

### 2.5 Conflict Tables

If our function F has a particular form, then we can perform lookups in a conflict table instead of testing if a minterm satisfies our function. This modification provides a more efficient test if the dimensions of the conflict table are small.

Let m_{off} be any off-set minterm of F that is created by applying max to at least k elements of MM. Let m_k be any off-set minterm of F that is created by applying max to exactly k elements of MM. Let m_{on} be the minterm of F that results by applying max to a size n subset S ⊆ MM. Let m_{nk} be a minterm of F that is created by applying max to some size k subset of S. We assume that the minimal minterm with all negative literals, if it exists in MM, is not included in the count of k because adding it does not change the result of applying max to a set without it.

**Theorem 5** If for all m_{nk} there exists an m_k such that the result of a bit-wise AND of m_{nk} and m_k is m_k, then for n ≥ k, m_{nk} is in the off-set iff there is an m_{nk} that is in the off-set.

**Proof** m_{nk} must contain the same positive literals as m_k because applying the bit-wise AND to m_{nk} and m_k creates m_k. The only way for this to happen under the monotonically increasing bit-wise OR of max is for m_k to be created only from the minimal minterms used to create m_{nk}. Without loss of generality, it can be assumed that m_{nk} is created with n minimal minterms where n ≥ k. Thus, m_{nk} can be relabeled as m_{nk}. Furthermore, because m_k only uses minimal minterms that were used to create m_{nk}, m_k can be relabeled as m_k.

Theorem 5 allows us to represent a function in terms of its minimal minterms and a k-dimensional conflict table. If k is small (i.e. 2 or 3), this is a very compact representation.

### 2.6 Comparison to a Cube Cover

A cube cover of a function only distinguishes minterms as on or off. There is no concept that a minterm is created by applying a relation to other minterms. Thus, actions can be associated with each on-set minterm or cube, but not with a subset of minterms.
as is offered with minimal minterms. Associating atomic actions with minimal minterms is the cornerstone to how we represent programmability.

3. Representing Programmability
Although we have presented and proven a general covering technique, we are most interested in how to use this technique to represent the programmability of a system. The systems that we are focusing on are statically-scheduled, horizontally-microcoded data paths. To do this, we have developed a method to model a system in terms of the logically consistent uses of the resources of the data path. From this model, we use FMM to efficiently find the set of minimal minterms. We prove that the minimal minterms represent the set of smallest schedulable atomic actions supported by the system. We also prove that applications of the max operator discover the inherent parallelism among these actions. Furthermore, from this representation, we can efficiently generate the required software and hardware components of the system.

3.1 Definition of Programmability
Our system is represented as a set of guarded rules where each rule is associated with an atomic action of the data path [1]. If a rule’s guard condition is satisfied, then the associated atomic action is executed. We formally model these conditions and the associated atomic actions as a programmability constraint.

**Definition 4** The programmability constraint of a system is a Boolean function that describes the legal executions of a data path in terms of a set of primary variables that are associated with atomic actions and the “presence” of signals.

Our programmability constraint captures all valid data path executions. It does this by creating restrictions so that on each cycle, no set of atomic actions can use a resource in a conflicting manner (not using the resource is always allowed). Given a valid combination of atomic actions, one can determine the associated data path execution.

Although pipelined execution paths can easily be captured by restricting a read to follow a write on pipeline registers, we weaken this restriction from an AND to an OR. Thus, all cycle-to-cycle data path executions are satisfying solutions as well. These cycle-to-cycle executions, which we call operations, represent the set of smallest schedulable atomic actions of a statically-scheduled, horizontally-microcoded machine. Any valid pipelined or parallel data path execution can be created by composing these operations. To extract these operations, we apply FMM to our programmability constraint. To determine conflicts between operations, we apply max to the operations and then test if the resulting minterm satisfies the programmability constraint.

**Definition 5** An operation is the minimal minterm and associated schedulable atomic action that is extracted from the programmability constraint.

We now argue that these operations represent the instruction set and applications of max to subsets of the operations indicate the inherent parallelism between the operations.

Let S be a system, P be the programmability constraint of S, and O be the set of operations extracted from P.

**Theorem 6** The set of operations O represent the set of smallest schedulable atomic actions that are supported by the system S.

**Proof** An operation is irreducible in terms of the atomic actions it uses. If it were not, then it would not be a minimal minterm. Therefore, because operations are irreducible, they represent the smallest atomic actions with respect to a given atomic action (i.e. minterm) that contains them. From Theorem 4, all schedulable atomic actions (i.e. minterms) can be created from the set of operations O (i.e. minimal minterms). Therefore, the set of operations O are a set of smallest schedulable atomic actions with respect to all minterms of P. Finally, because Theorem 1 states that the set of minimal minterms (i.e. the set of operations O) is unique, they are the only set of smallest schedulable atomic actions for the system S.

We now proceed to show that concurrent schedules are constructed by combining operations using max.

**Theorem 7** Any valid concurrent schedule can be created by applying max to a subset of the set of operations O and then determining if the resulting minterm satisfies the programmability constraint P.

**Proof** Applying max is equivalent to scheduling a subset of the set of operations O concurrently because a bit-wise OR of the literals associated with the resources of the concurrent operations represents running the operations concurrently. If the resulting minterm is in the on-set of the programmability constraint P, then the schedule is valid. By the fact that operations are minimal minterms and Theorem 3, all valid concurrent schedules can be created.

The results of Theorem 6 and Theorem 7 provide us a new method for representing the programmability of the system.

**Theorem 8** The programmability of a system S can be represented by the set of operations O and applications of max to each element of the power set of the set of operations O. If the minterm that results from applying max satisfies the programmability constraint P then the corresponding schedule is valid.

**Proof** This follows directly from Theorem 4 because the set of operations O are the minimal minterms.

Although it may already be efficient to check if a minterm is in the on-set, the programmability constraint that we use has a special form which allows us to use an even more efficient test.

**Theorem 9** A subset of the set of operations O is in conflict iff applying max to some pair of operations in the subset creates a minterm that is in the off-set of P.

**Proof** An operation asserts literals on the primary variables that indicate that it is using a group of resources in a particular way. If another operation asserts that it is using a resource from this group in a different way, then there is a conflict. Moreover, from the structure of the problem and definition of max, no subsequent operation can assert or negate a primary variable such that the conflict goes away. Thus, the theorem follows by invoking Theorem 5 with k = 2.

Theorem 9 allows us to create a simple method to determine conflicts. First, a conflict table is generated that indicates which pairs of operations in O are in conflict. We can use this conflict table to check if a subset of the operations in O is in conflict by performing lookups in the conflict table on all pairs.

With this final result, we have an efficient way to represent the programmability of a system. Next, we will outline how we
have applied this method specifically to create an environment for developing programmable platforms.

3.2 Multiple Views Methodology

In Figure 4, we illustrate our multiple views methodology. The labeled arrows indicate what a view adds or uses from the model. Given that we can create the data path structure (dp), specify the data path constraints (dpc), extract the operations (op) and generate the conflict table (ct), we can automatically generate the required software development tools and hardware implementation in a matter of seconds. This methodology is enabled by the techniques described in the previous section. The data path structure (dp) and constraints (dpc) constitute the programmability constraint. The extracted operations (op) and conflict table (ct) represent the programmability. A more detailed examination of the multiple views, the tools they create and simulation performance results has been presented previously [1].

![Figure 4: Multiple Views](image)

A small example and its implementation is shown in Figure 5.

```plaintext
atom example
    input in1, in2;
    output out1, out2;
    reg mem16;

rule r1 { out1 = in1; out2 = in2; mem16 = 0; }
rule r2 { out1 = in1; out2 = in2; mem16 = 1; }
rule r3 { out1 = in1; out2 = in2; mem16 = 2; }
rule r4 { out1 = mem16; out2 = 0; }
rule r5 { out1 = mem16; out2 = 1; }
rule r6 { out1 = mem16; out2 = 2; }

if (r1 = in1, in2) { out1 = in1; out2 = in2; mem16 = 0; }
```

![Figure 5: Example Data Path](image)

The programmability constraint is shown symbolically in Figure 6. Constraints map rules to the activation of atomic actions and use of I/O (1-3). Users can define their own constraints in terms of inputs, outputs, and rules (4). Each output and state address expression is assigned or explicitly not assigned (nop rules are implicitly created for this) (5,6). Causal constraints are defined to reflect causal relationships between inputs, outputs, and atomic actions that reflect the flow of data and relationships to rules (8-15). Single assignment constraints restrict that each output and unique state address is assigned only one value (16,17). Finally, one rule or no rules that share an enumeration can be activated simultaneously (1,7,18). From our data path and extracted constraints, we can automatically extract the operations and conflict table.

![Figure 6: Programmability Constraint](image)

```plaintext
(b1 = b2) & (c1 = c2) -> (d1 = d2) & (e1 = e2) & (f1 = f2) & (g1 = g2) & (h1 = h2) & (i1 = i2) & (j1 = j2) & (k1 = k2) & (l1 = l2) & (m1 = m2) & (n1 = n2) & (o1 = o2) & (p1 = p2) & (q1 = q2) & (r1 = r2) & (s1 = s2) & (t1 = t2) & (u1 = u2) & (v1 = v2) & (w1 = w2) & (x1 = x2) & (y1 = y2) & (z1 = z2)
```

![Figure 7: Programmability Constraint for Example](image)

From the constraints, the operations and conflict table are automatically extracted and are shown in Figure 8.

```plaintext
op1: out1 = in1, out2 = in2, mem[234] = in2
op2: out1 = in1, out2 = in2, mem[235] = in2
op3: out1 = in1, out2 = in2, mem[236] = in2
```

![Figure 8: Operations and Conflict Table for Example](image)
After finding the operations and conflict table, we can automatically generated all the other tools and implementations shown in the multiple views diagram.

3.3 Related Work
The closest concept to a minimal minterm in set theory is the computation of a minimal. The minimal of a subset of an ordered set is an element such that no other element that is less than the minimal is in the subset. If we think of a minterm as a characteristic function of a subset where a variable being true indicates that the variable is in the set, then in terms of a Boolean function, a minimal would be a minterm that does not contain any other minterms in the function. This is different from a minimal minterm which says that a minterm is minimal if it cannot be created by applying \textit{max} to a subset of minterms.

Although there are several existing architecture description language (ADL) efforts [2], leading research efforts, such as LISA [3], EXPRESSION [4], and nML [5], require the designer to specify the instruction set, and then map it to an architectural implementation. This mixed top-down and bottom-up approach is different than our bottom-up approach because we extract the implementation. This mixed top-down and bottom-up approach to specify the instruction set, and then map it to an architectural LISA [3], EXPRESSION [4], and nML [5], require the designer language (ADL) efforts [2], leading research efforts, such as MIMOLA effort, which says that a minterm is minimal if it cannot be created by applying \textit{max} to a subset of minterms.

In order to determine the effectiveness of our method, we compared the number of minimal minterms versus the number of cubes it takes to represent a function. We generated all functions up to 4 variables and a random sample of functions with 5 to 18 variables. The minimal minterms were extracted using our FMM algorithm, and the prime cube covers were found using ESPRESSO [7]. Results show that minimal minterms are much more efficient. The ratio between cubes and minimal minterms grows exponentially for functions with more than five variables.

![Figure 9. Cubes vs. Minimal Minterms (mm)](image)

In order to determine the efficiency of using FMM to extract operations and a conflict table to represent the programmability of a system, we created five interesting data path systems. The designs are a dlx processor (dlx), a convolution coding processor (cc), a polyphase filter for mp3 (mp3), a packet processor (pp) and a stream processor (stream). Results on a 1.67 GHz Pentium 4 with 1 GB RAM indicate that FMM is highly effective at quickly finding a small number of minimal minterms to represent large functions. In fact, the ratio of the number of minimal minterms to the total number of minterms found by relsat [8] is miniscule. We could not compare minimal minterms to cubes due to the size of the functions. Even if we could, cubes do not represent the operations in our formulation.

<table>
<thead>
<tr>
<th>design name</th>
<th>primary vars</th>
<th>cnf vars</th>
<th>cnf clauses</th>
<th>total minterms</th>
<th>minimal minterms</th>
<th>FMM time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dlx</td>
<td>80</td>
<td>2,861</td>
<td>7,648</td>
<td>45,408</td>
<td>83</td>
<td>1.2 s</td>
</tr>
<tr>
<td>Cc</td>
<td>95</td>
<td>3,369</td>
<td>9,029</td>
<td>435,456</td>
<td>46</td>
<td>0.8 s</td>
</tr>
<tr>
<td>mp3</td>
<td>189</td>
<td>6,234</td>
<td>16,576</td>
<td>611,529,523,200</td>
<td>52</td>
<td>11.8 s</td>
</tr>
<tr>
<td>Pp</td>
<td>207</td>
<td>7,074</td>
<td>19,665</td>
<td>701,582,400</td>
<td>68</td>
<td>18.3 s</td>
</tr>
<tr>
<td>Stream</td>
<td>291</td>
<td>10,365</td>
<td>28,714</td>
<td>&gt;291,404,905,665,527,808</td>
<td>127</td>
<td>23.4 s</td>
</tr>
</tbody>
</table>

Table 1. Extracting Operations

5. Conclusion
We have developed and proven a technique to represent a Boolean function as a set of minimal minterms and a simple generator relation. Results indicate that the technique is quite efficient when compared to prime cube covers. We have primarily focused on how to apply this technique to represent the programmability of a system. If a system is modeled in terms of logically consistent uses of its data path, then the minimal minterms represent the set of smallest atomic actions supported by the system (i.e. instructions), and the generator relation (i.e. compatibility table [the dual of the conflict table]) represents the inherent parallelism among the instructions. From the extracted minimal minterms, conflict table, and data path description, we automatically generate the required software development tools and hardware implementation. Our results indicate that the technique is efficient in terms of the number of minimal minterms and runtimes required. Thus, we believe we have effectively addressed the issue of representing the programmability of a system.

6. References