Multiplication via CRT

Thomas Reps
Department of Computer Science
University of Wisconsin-Madison
1205 University Avenue
Madison, WI 53706
reps@cs.wisc.edu

Abstract

Multiplication circuits are an important class of circuits whose behavior one would like to verify. Decision diagrams remain an important tool for verification, but for many kinds of decision diagrams, bit-vector multiplication cannot be represented by a polynomial-sized decision diagram, so one cannot directly express the specification of the desired operation efficiently.

Past work has used the Chinese Remainder Theorem to break down the problem into smaller-sized decision diagrams (for different moduli) that can be represented efficiently. However, past work had two limitations: (i) it focused on non-overflowing computations, and (ii) it used an imprecise characterization of when overflow occurs. Our work uses representations for the individual-moduli specifications that are exponentially more efficient than used in past work, and represents the full $(w \times w)$-bit multiplication relation: $\{(\vec{x}, \vec{y}, \vec{z}) \in \mathbb{B}^w \times \mathbb{B}^w \times \mathbb{B}^{2w} \mid \vec{x} \times \vec{y} = \vec{z}\}$. **MISSING**: For the specification problem, say how large a w we can handle when the goal is to specify the multiplication relation. For the verification problem, say how large a w we can handle when the goal is to verify a multiplication circuit with respect to a multiplication specification, for various types of circuits (shift-and-add, Karatsuba?).

1 Introduction

The goal is to represent the full $(w \times w)$-bit multiplication relation:

$$\{(\vec{x}, \vec{y}, \vec{z}) \in \mathbb{B}^w \times \mathbb{B}^w \times \mathbb{B}^{2w} \mid \mathit{val}(\vec{x}) \times \mathit{val}(\vec{y}) = \mathit{val}(\vec{z})\},$$

where $\mathit{val}(\vec{x})$ denotes the numeric value of bit-vector $\vec{x}$. As in past work, we represent the multiplication relation in an indirect way via the Chinese Remainder Theorem (CRT)—i.e., using numbers $n_1, \ldots, n_s$ that are (i) relatively prime, (ii) all relatively small, and (iii) $2^{2w} < N = \Pi_{i=1}^{s} n_i$. We use a collection of CFLOBDDs [2], where the i^{th} CFLOBDD, C_i, implements the relation

$$\{(\vec{x}, \vec{y}, \vec{v}) \mid \mathit{val}(\vec{x}) \times \mathit{val}(\vec{y}) =_{\bmod n_i} \mathit{val}(\vec{v})\}.$$[1]

That is, the plies of the CFLOBDDs correspond to the bits of the numbers x and y interleaved in some order,[2] and in C_i each value v in the value tuple has the property that $0 \leq v < n_i$. Because for each $\vec{x}$ and $\vec{y}$, the desired value of $\vec{z}$ can be recovered from the set of s values $\{\mathit{val}(\vec{v_i})_{\bmod n_i} \mid 1 \leq i \leq s \wedge \mathit{val}(\vec{x}) \times \mathit{val}(\vec{y}) =_{\bmod n_i} \mathit{val}(\vec{v_i})\}$, the collection of such CFLOBDDs is a representation of the multiplication relation.

We denote this representation by CRT[CFLOBDD].[3] As we show below, each individual CFLOBDD can be represented efficiently, so overall the CRT[CFLOBDD] representation is an efficient representation of the multiplication relation.

[1] Strictly speaking, instead of relations of the form $\{(\vec{x}, \vec{y}, \vec{v}) \in \mathbb{B}^w \times \mathbb{B}^w \times \mathbb{B}^{2w} \mid \mathit{val}(\vec{x}) \times \mathit{val}(\vec{y}) =_{\bmod n_i} \mathit{val}(\vec{v})\}$, we use multi-terminal CFLOBDDs with $2w$-bit inputs to implement functions of the form $\lambda(x, y) \in \mathbb{B}^w \times \mathbb{B}^w.(\mathit{val}(x) \times \mathit{val}(y))_{\bmod n_s}$

[2] We often use the order "high-order bits to low-order bits as you traverse a matched path in the CFLOBDD."

[3] Each instantiation of CRT[CFLOBDD] is parameterized by a specific set of moduli $\{n_1, \ldots, n_s\}$. Whether the parameters are the generic set $\{n_1, \ldots, n_s\}$ or some specific set, such as $\{3, 5, \ldots, 101, 103\}$, will always be clear from context.

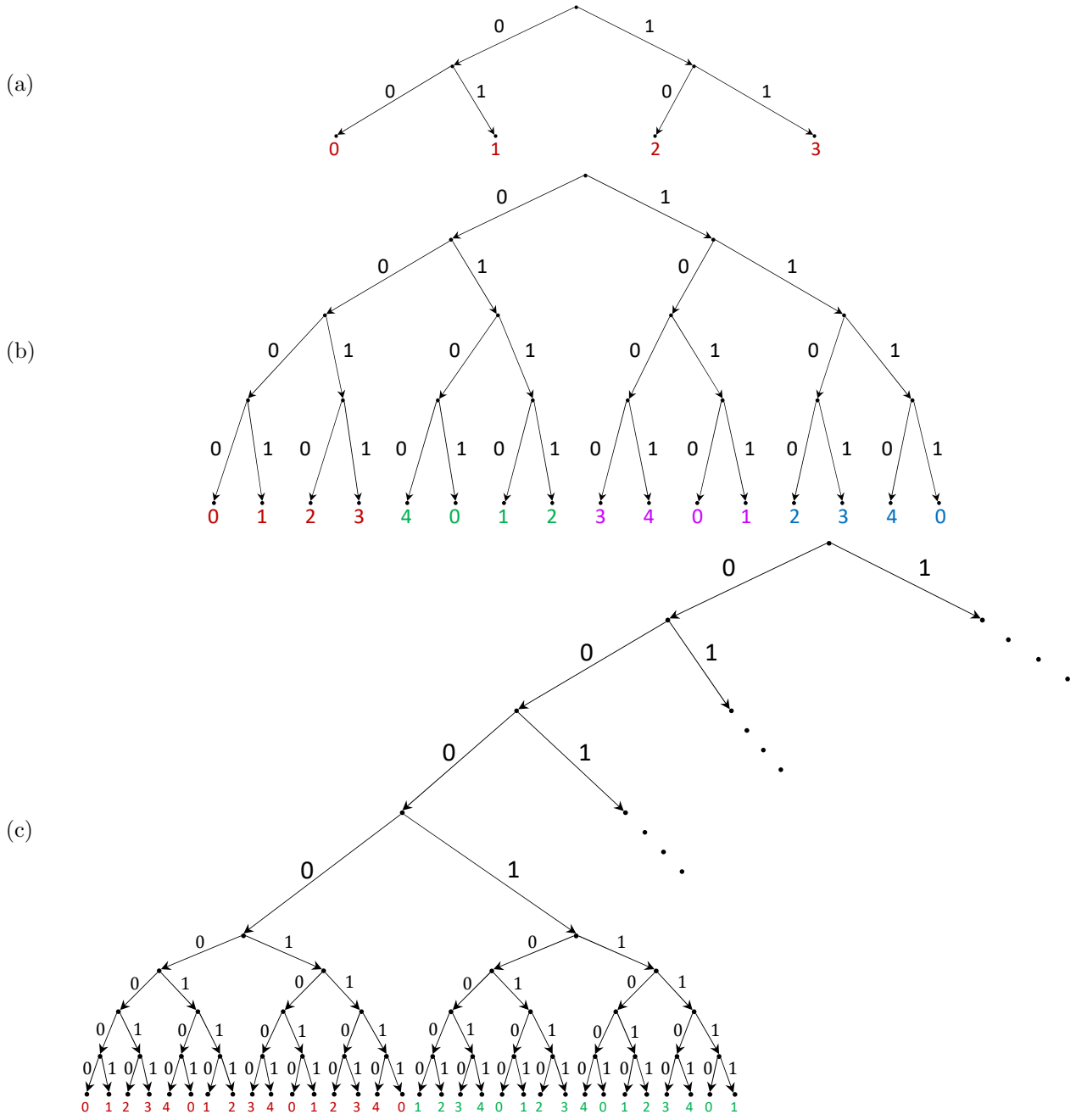

Figure 1: Decision trees of depths 2, 4, and 8 that represent the functions $\lambda x \in \mathbb{B}^2.val_{mod\ 5}(x)$, $\lambda x \in \mathbb{B}^4.val_{mod\ 5}(x)$, and $\lambda x \in \mathbb{B}^8.val_{mod\ 5}(x)$, which that map each 2-bit, 4-bit, and 8-bit number, respectively, to its value mod 5. The bit order is high-order to low-order as one descends in the tree.

Why CFLOBDDs and not MTBDDs? The advantage of CFLOBDDs—i.e., the CRT[CFLOBDD] representation—is that it is exponentially smaller than the alternatives, such as a set of MTBDDs (i.e., a CRT[MTBDD] representation).

We know that the size of a Multi-Terminal BDD (MTBDD)[4] for the multiplication relation is inherently exponential in the number of bits of n_i (regardless of the variable order).[5] We do not know whether CFLOBDDs exhibit the same problem, but we believe that CFLOBDDs prob-

[4]Also known as an Algebraic Decision Diagram (ADD).

[5]Bryant [1] showed that, regardless of what variable order is used, a (read-once) BDD that encodes the middle-bit value of the multiplication function must have size that is exponential in the number of bits in the numbers being multiplied.

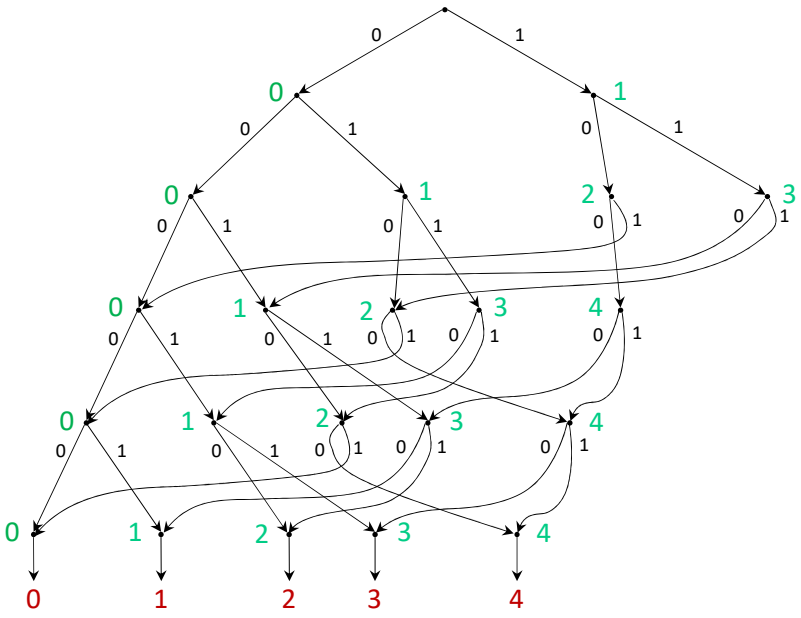

Figure 2: Five-ply MDBDD that represents the function $\lambda x \in \mathbb{B}^5.val_{mod\ 5}(x)$, which maps each 5-bit number to its value mod 5. The bit order is high-order to low-order as one descends in the MTBDD. Note that in an MTBDD that represents the value-mod-5 function for numbers with a greater number of bits, the additional plies would repeat the pattern of plies 4 and 5.

ably do. However, the moduli $\{n_1, \ldots, n_s\}$ are all relatively small (e.g., each expressible with 7 bits), so the size explosion should not be significant. By the Chinese Remainder Theorem, the s CFLOBDDs $\{C_1, \ldots, C_s\}$ = CRT[CFLOBDD] give us a representation of the multiplication relation $\{(\vec{x}, \vec{y}, \vec{z}) \in \mathbb{B}^w \times \mathbb{B}^w \times \mathbb{B}^{2w} \mid val(\vec{x}) \times val(\vec{y}) = val(\vec{z})\}$ whenever $2^{2w} < N = n_1 \times \ldots \times n_s$. For instance, if $\{n_1, \ldots, n_s\}$ consists of all primes from 3 to 103 (inclusive), N is a 134-bit number that is approximately 1.9924E+40. When we use (multi-terminal) CFLOBDDs that support 128-bit inputs,[6] CRT[CFLOBDD] is a representation of 64-bit multiplication.

The advantage of CFLOBDDs comes from their greater ability to share sub-structures. In particular, MTBDDs can only share the "bottom portions of DAGs," whereas CFLOBDDs can share (some) "middle portions of DAGs." We first illustrate the opportunities for such middle-of-a-DAG sharing opportunities with the "value-mod-n_i" function. Consider a decision tree for "value-mod-n_i" that has w plies. The input bits are the bit representation of a w-bit number, and the leaf reached by a specific bit-sequence bs represents the value of bs interpreted as a binary number. (The bit-order we will use is "high-order to low-order as one descends in the tree"). For instance, Figure 1 shows decision trees of depths 2, 4, and 8 that represent the value-mod-5 functions $\lambda x \in \mathbb{B}^2.val_{mod\ 5}(x)$, $\lambda x \in \mathbb{B}^4.val_{mod\ 5}(x)$, and $\lambda x \in \mathbb{B}^8.val_{mod\ 5}(x)$, which map each 2-bit, 4-bit, and 8-bit number, respectively, to its value mod 5.

In each tree in Figure 1, each prefix-tree consisting of the root and all nodes and edges down to some depth d can also be interpreted as a map: the prefix-tree maps each high-order d-bit prefix $\vec{p}$ of a w-bit binary number to the value $val(\vec{p})$ mod 5. That is, the prefix-tree of Figure 1(c) that corresponds to Figure 1(a) gives values to the high-order 2-bit prefixes of Figure 1(c), whereas the prefix-tree of Figure 1(c) that corresponds to Figure 1(b) gives values to the high-order 4-bit prefixes of Figure 1(c). At each ply, as one goes one ply deeper from a node with value v, the left and right children have the values $2v$ mod 5 and $2v+1$ mod 5, respectively. Consequently, for each such prefix-tree, a left-to-right listing of the values at the "leaves" of a prefix-tree always has the form 0, 1, 2, 3, 4, 0, 1, ..., as we see in all three of the trees in Figure 1. The regularity of this pattern allows both the MTBDD and CFLOBDD for this function to have highly compressed representations (although the CFLOBDD representation is exponentially more succinct than the MTBDD representation).

Consider the five-ply MTBDD for the value-mod-5 function $\lambda x \in \mathbb{B}^5.val_{mod\ 5}(x)$, shown in Figure 2.

[6] For 128-bit inputs, the top-level grouping of each CFLOBDD is at level 7.

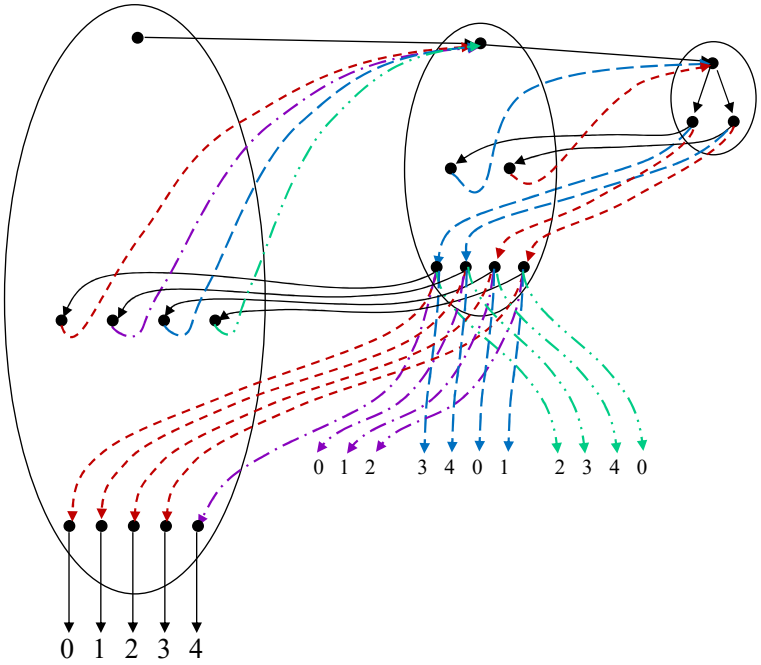

Figure 3: Level-2 CFLOBDD that represents the function $\lambda x \in \mathbb{B}^4.val_{mod\ 5}(x)$, which maps each 4-bit number to its value mod 5. The bit order is high-order to low-order as one passes through the ForkGrouping. To avoid clutter, the targets of three of the dotted-dashed purple return edges, all of the dashed blue return edges, and all of the double-dotted-dashed green edges are shown indirectly. (The number indicates the value-tuple value associated with the level-2 exit vertex that is the target of the return edge.)

Because the bit order is high-order to low-order as one descends in the MTBDD, one can think of the MTBDD as a finite-state automaton whose states represent the value mod 5 of the prefix of a 5-bit number read so far. In the next ply, reading a 0 causes the prefix-value to be multiplied by 2 (mod 5); reading a 1 causes the prefix-value to be multiplied by 2 and added to 1, all carried out in arithmetic mod 5. For each ply, there are at most five target nodes. Consequently, the total size of the MTBDD representation of the function that maps each w-bit number to its value mod n_i is $O(n_i w)$, which means that it is exponentially smaller than the decision tree for the function (whose size is $O(2^w)$).

In contrast, Figure 3 shows the level-2 CFLOBDD for $\lambda x \in \mathbb{B}^4.val_{mod\ 5}(x)$, which maps 4-bit numbers to numbers mod 5. In this CFLOBDD, the level-1 proto-CFLOBDD is shared in the level-2 A-connection and all four level-2 B-connections. As shown in Figure 4, essentially the same pattern seen in Figure 3 continues at higher levels (except that higher-level groupings have five middle vertices): at level m, the A-connection and every B-connection share the same proto-CFLOBDD at level $m-1$. The various sets of B-connection return edges are each directed to a different pattern of exit vertices at level m, resulting in a total number of return edges that is quadratic in the number of middle vertices at level m. Consequently, the total size of the CFLOBDD representation of the function that maps each w-bit number to its value mod n_i is $O({n_i}^2 \log_2 w)$, which means that it is double-exponentially smaller than the decision tree for the function—and exponentially smaller than the function's MTBDD.

Representing the Full $(w \times w)$-Bit Multiplication Relation. Turning to the function for multiplication mod n_i, CFLOBDDs again exhibit double-exponential compression compared to the decision tree. For instance, Figure 5 shows the multiplication-mod-5 function for 4-bit numbers: $\lambda(x, y) \in \mathbb{B}^4 \times \mathbb{B}^4.(val(x) \times val(y))_{\text{mod } 5}$ (where the bits of the two arguments are not interleaved). Figure 6 shows the general pattern for 2^l-bit numbers (i.e., $b = 2^l$). In the top-most grouping, the multiple different patterns of B-connection return edges cause the grouping to have $O(n^2)$ edges. Because the lower-level groupings of the (shared) top-level A-connection and B-connections also have a quadratic number of return edges at each level (see Figures 3 and 4), the size of the CFLOBDD representation of the function for multiplication of w-bit numbers mod n_i is $O({n_i}^2 \log_2 w)$.

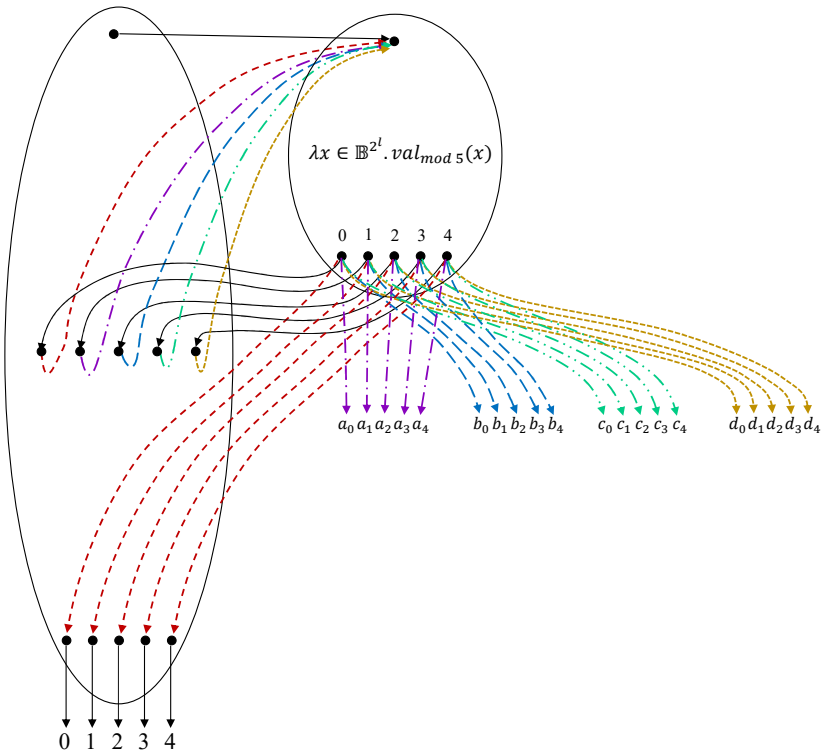

Figure 4: Level-$(l+1)$ CFLOBDD that represents the function $\lambda x \in \mathbb{B}^{2^l}.val_{mod\ 5}(x)$, which maps each 2^l-bit number to its value mod 5. The bit order is high-order to low-order. To avoid clutter, the targets of some of the level-l-to-level-$(l+1)$ B-connection return-edges are shown indirectly: a_0 equals $2^{2^l} \bmod 5$, with $a_1, \ldots, a_4$ being the sequence $(a_0+1) \bmod 5, \ldots, (a_0+4) \bmod 5$. Similarly, $b_0, \ldots, b_4$ is the sequence $(2 \times 2^{2^l}) \bmod 5, \ldots, (b_0+4) \bmod 5$; $c_0, \ldots, c_4$ is the sequence $(3 \times 2^{2^l}) \bmod 5, \ldots, (c_0+4) \bmod 5$; and $d_0, \ldots, d_4$ is the sequence $(4 \times 2^{2^l}) \bmod 5, \ldots, (d_0+4) \bmod 5$. These numbers indicate the value-tuple value associated with the level-2 exit vertex that is the target of each return edge.

With two w-bit inputs, we are working with two values that are each in the range $[0, \ldots, 2^w - 1]$. Thus, we must choose $\{n_1, \ldots, n_s\}$ so that there is no wrap-arround at $N = n_1 \times \ldots \times n_s$. Thus, we must have $(2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1 < N$. (To simplify matters, we can use the simpler constraint $2^{2w} < N$.)

For instance, suppose that $w = 4$, so $2^w = 16$. Suppose that we use the modulus set $\{5, 7, 11\}$, so $N = 385$. Then $(2^w - 1)^2 = 15 \times 15 = 225 < 385 = N$, and no product wraps around N.

In the CRT representation, $15 = \langle 15 \bmod 5, 15 \bmod 7, 15 \bmod 11 \rangle = \langle 0, 1, 4 \rangle$, so

$$\begin{aligned} 15 \times 15 &= \langle 0, 1, 4 \rangle \times \langle 0, 1, 4 \rangle \\ &= \langle (0 \times 0)_{\bmod 5}, (1 \times 1)_{\bmod 7}, (4 \times 4)_{\bmod 11} \rangle \\ &= \langle 0, 1, 5 \rangle. \end{aligned}$$

We can find the value a that $\langle 0, 1, 5 \rangle$ represents as follows:

$$a =_{\bmod N} 0 \cdot c_1 + 1 \cdot c_2 + 5 \cdot c_3,$$

where, in general,

$$m_i = n_1 \cdot n_2 \cdot \ldots n_{i-1} \cdot n_{i+1} \ldots \cdot n_s$$

and

$$c_i = m_i \cdot (m_i^{-1} \bmod n_i).$$

In the case of $\langle 0, 1, 5 \rangle$ and modulus set $\{5, 7, 11\}$, we have $m_1 = 77$, $77^{-1} \bmod 5 = 3$; $m_2 = 55$,

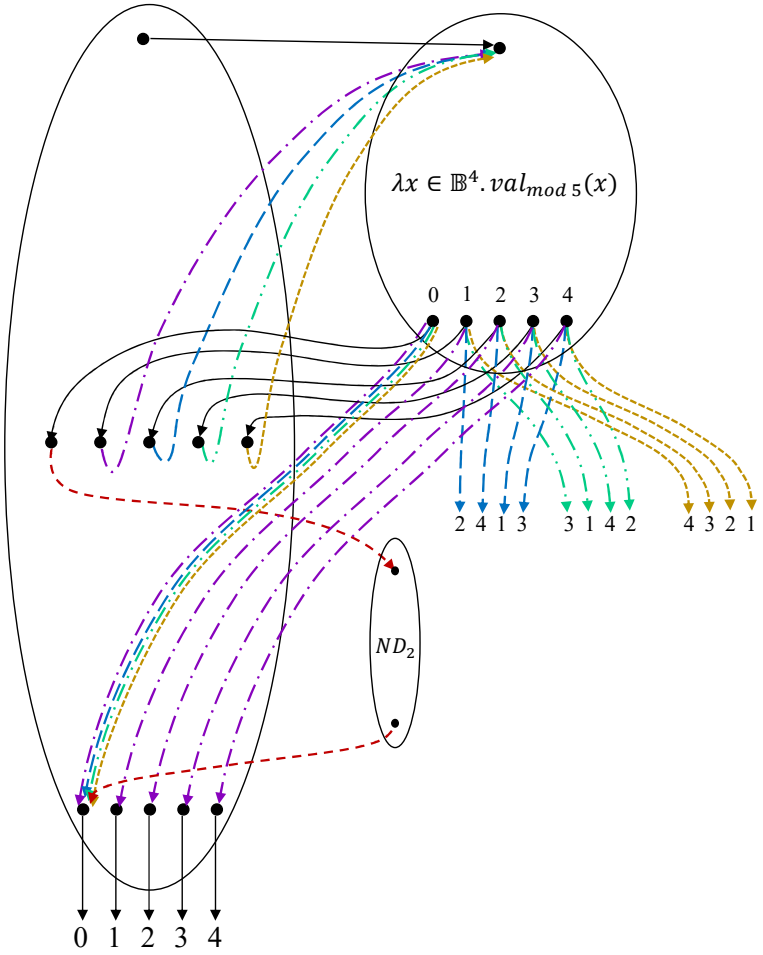

Figure 5: Level-3 CFLOBDD that represents the multiplication-mod-5 function for 4-bit numbers: $\lambda(x, y) \in \mathbb{B}^4 \times \mathbb{B}^4.(val(x) \times val(y))_{\text{mod } 5}$. The bit order is high-order to low-order, and the bits of x and y are concatenated (i.e., not interleaved). To avoid clutter, the targets of some of the level-2-to-level-3 B-connection return-edges are shown indirectly. (The number indicates the value-tuple value associated with the level-3 exit vertex that is the target of the return edge.) "ND_2" denotes the NoDistinctionProtoCFLOBDD for level 2.

$55^{-1} \bmod 7 = 6$; and $m_3 = 35$, $35^{-1} \bmod 11 = 6$, so

$$
\begin{aligned}
c_1 &= 77 \cdot 3 = 231 \\
c_2 &= 55 \cdot 6 = 330 \\
c_3 &= 35 \cdot 6 = 210 \\
a &=_{\text{mod } 385} 0 \cdot 231 + 1 \cdot 330 + 5 \cdot 210 \\
&=_{\text{mod } 385} 0 + 330 + 1050 \\
&=_{\text{mod } 385} 225
\end{aligned}
$$

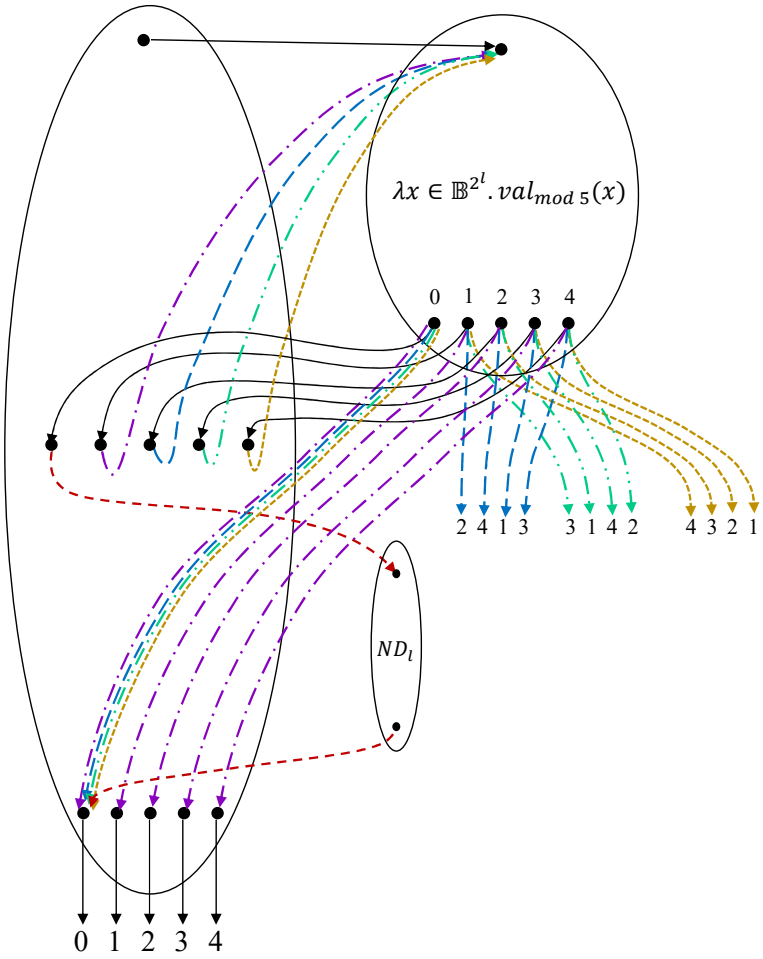

Figure 6: Level-$(l+1)$ CFLOBDD that represents the multiplication-mod-5 function for 2^l-bit numbers: $\lambda(x, y) \in \mathbb{B}^{2^l} \times \mathbb{B}^{2^l}.(val(x) \times val(y))_{\text{mod } 5}$, for $l \geq 2$. The bit order is high-order to low-order, and the bits of x and y are concatenated (i.e., not interleaved). To avoid clutter, the targets of some of the level-l-to-level-$(l+1)$ B-connection return-edges are shown indirectly. (The number indicates the value-tuple value associated with the level-$(l+1)$ exit vertex that is the target of the return edge.) "ND_l" denotes the NoDistinctionProtoCFLOBDD for level l.

```
[1]   abstract class Grouping {
[2]     level: int
[3]     numberOfExits: int
[4]   }

[5]   class InternalGrouping extends Grouping {
[6]     AConnection: Grouping
[7]     AReturnTuple: ReturnTuple
[8]     numberOfBConnections: int
[9]     BConnections: array[1..numberOfBConnections] of Grouping
[10]    BReturnTuples: array[1..numberOfBConnections] of ReturnTuple
[11]  }

[12]  // To indicate the role of a Grouping in a recursive call
[13]  enum Position { TopLevel, A, B }

[14]  class DontCareGrouping extends Grouping {
[15]    level = 0
[16]    numberOfExits = 1
[17]  }

[18]  class ForkGrouping extends Grouping {
[19]    level = 0
[20]    numberOfExits = 2
[21]  }

[22]  // Multi-terminal CFLOBDD with a value at each terminal
[23]  class CFLOBDD {
[24]    grouping: Grouping
[25]    valueTuple: ValueTuple
[26]  }
```

Figure 7: Basic class definitions for CFLOBDDs.

2 The CRT[CFLOBDD] Representation With Non-Interleaved Vocabularies

```
[1]   unsigned logLogOfMaxModulus = 3 // Each of the relatively prime moduli must be <= 256
[2]   unsigned maxModulus = 1 << (1 << logLogOfMaxModulus) // 2**(2**logLogOfMaxModulus)

[3]   CFLOBDD NumsModK(unsigned k, Position p) {
[4]     assert(2 <= k ∧ k <= maxModulus)
[5]     assert(p != TopLevel)
[6]     assert(maxLevel >= logLogOfMaxModulus + 1) // Ensures that there are k exit vertices
[7]     InternalGrouping g = new InternalGrouping(maxLevel)
[8]     if (p == A) {
[9]       g.AConnection = ProtoCFLOBDDNumsModK(maxLevel-1, k)
[10]      g.AReturnTuple = [1, ..., k]
[11]      g.numberOfBConnections = k
[12]      for (i = 1; i <= g.numberOfBConnections; i++) {
[13]        g.BConnections[i] = NoDistinctionProtoCFLOBDD(g.level)
[14]        g.BReturnTuples[i] = [i]
[15]      }
[16]    }
[17]    else if (p == B) {
[18]      g.AConnection = NoDistinctionProtoCFLOBDD(g.level)
[19]      g.AReturnTuple = [1]
[20]      g.numberOfBConnections = 1
[21]      g.BConnections[1] = ProtoCFLOBDDNumsModK(maxLevel-1, k)
[22]      g.BReturnTuples[i] = [1, ..., k]
[23]    }
[24]    g.numberOfExits = k
[25]    return RepresentativeCFLOBDD(g, [0, ..., k-1])
[26]  }
```

Figure 8: Construction of a CFLOBDD that represents numbers mod k (either in the topmost Grouping's A-connection or B-connection, as specified by the Position argument). After following (in the A- or B-connection, as specified) the assignment a for some number q, where the earliest bit in a corresponds to the most-significant bit of q, the exit-vertex position (1-based indexing) is $(q \bmod k) + 1$ in the value-tuple $[0, \ldots, k-1]$.

```
[1]   Grouping ProtoCFLOBDDNumsModK(unsigned lev, unsigned k) {
[2]     if (lev == 0) return RepresentativeForkGrouping
        // Adjust if constructing a low-level Grouping that doesn't have k middle vertices
[3]     unsigned numberOfMidVertices = ((lev-1) >= logLogOfMaxModulus)
                                           ?  k
                                           :  min(k, 1 << (1 << (lev-1)))
        // Adjust if constructing a low-level Grouping that doesn't have k exit vertices
[4]     unsigned numberOfExits = (lev >= logLogOfMaxModulus)
                                    ?  k
                                    :  min(k, 1 << (1 << lev))
[5]     unsigned AConnectionPathsModK = (1 << (1 << (lev-1)) mod k
[6]     InternalGrouping g = new InternalGrouping(lev)
[7]     g.AConnection = ProtoCFLOBDDNumsModK(lev-1, k)
[8]     g.AReturnTuple = [1, ..., numberOfMidVertices]
[9]     g.numberOfBConnections = numberOfMidVertices
[10]    g.BConnections[1] = g.AConnection
[11]    g.BReturnTuples[1] = [1, ..., numberOfMidVertices]
[12]    for (i = 2; i <= g.numberOfBConnections; i++) {
[13]      g.BConnections[i] = g.AConnection
[14]      unsigned nextVal = ((i-1) * AConnectionPathsModK) mod k
          // Continue numbering, starting from nextVal
          // Create BReturnTuples[i] using 1-based indexing
[15]      g.BReturnTuples[i] = [nextVal + 1, ...,
[16]                            ((nextVal + numberOfMidVertices - 1) mod numberOfExits) + 1]
[17]    }
[18]    g.numberOfExits = numberOfExits
[19]    return RepresentativeGrouping(g)
[20]  }
```

Figure 9: Utility procedure for construction of a Grouping that represents numbers mod k.

```
[1]   CFLOBDD MultModK(unsigned k) {
[2]     CFLOBDD cA = NumsModK(k, A)
[3]     CFLOBDD cB = NumsModK(k, B)
[4]     return BinaryApplyAndReduce(cA, cB, λx, y.((x × y) mod k)))
[5]   }
```

Figure 10: Construction of a CFLOBDD that represents the multiplication relation mod k. The bits of the two arguments are concatenated (not interleaved).

```
[1]   class MultRelation {
[2]     unsigned numberOfMultRelations = 26
[3]     Moduli:  array[1..numberOfMultRelations] of unsigned
[4]     ModularMultRelations:  array[1..numberOfMultRelations] of CFLOBDD
[5]     MultRelation() // Constructor declaration
[6]   }
```

Figure 11: Class MultRelation, which represents the multiplication relation via the Chinese Remainder Theorem.

```
[1]   MultRelation::MultRelation() {
[2]     Moduli[1] = 3
[3]     Moduli[2] = 5
[4]     Moduli[3] = 7
[5]     Moduli[4] = 11
[6]     Moduli[5] = 13
[7]     Moduli[6] = 17
[8]     Moduli[7] = 19
[9]     Moduli[8] = 23
[10]    Moduli[9] = 29
[11]    Moduli[10] = 31
[12]    Moduli[11] = 37
[13]    Moduli[12] = 41
[14]    Moduli[13] = 43
[15]    Moduli[14] = 47
[16]    Moduli[15] = 53
[17]    Moduli[16] = 59
[18]    Moduli[17] = 61
[19]    Moduli[18] = 67
[20]    Moduli[19] = 71
[21]    Moduli[20] = 73
[22]    Moduli[21] = 79
[23]    Moduli[22] = 83
[24]    Moduli[23] = 89
[25]    Moduli[24] = 97
[26]    Moduli[25] = 101
[27]    Moduli[26] = 103

[28]    UNSIGNED n = 1 // Use arbitrary-precision arithmetic for n
[29]    for (i = 1; i <= numberOfMultRelations; i++) {
[30]      n = n * Moduli[i]
[31]    }
[32]    assert( n < (1 << (1 << maxLevel-1)) )

[33]    for (i = 1; i <= numberOfMultRelations; i++) {
[34]      ModularMultRelations[i] = MultModK(Moduli[i])
[35]    }
[36]  }
```

Figure 12: Constructor for class MultRelation. With the first 26 odd primes, we can represent 133-bit numbers, which is sufficient for handling multiplication mod 2^{64}. We'll need maxLevel = 7.

```
[1]   CFLOBDD Factor(unsigned v) {
[2]     MultRelation R
[3]     array[1..numberOfMultRelations] of CFLOBDD slices
[4]     for (i = 1; i <= numberOfMultRelations; i++) {
[5]       CFLOBDD P = ConstantCFLOBDD(mod(v, R.Moduli[i]))
[6]       slices[i] = BinaryApplyAndReduce(R.ModularMultRelations[i], P, λx, y.x == y)
[7]     }

        // It would probably be better to do a tree reduction, starting by
        // conjoining slices[1] and slices[numberOfMultRelations],
        // slices[2] and slices[numberOfMultRelations-1], etc.
[8]     CFLOBDD ans = slices[1]
[9]     for (i = 2; i <= numberOfMultRelations; i++) {
[10]      ans = BinaryApplyAndReduce(ans, slices[i], λx, y.x ∧ y)
[11]    }
[12]    return ans
[13]  }
```

Figure 13: Operation to identify all factorizations of a number `v`. If `v` is a prime, the CFLOBDD returned has exactly two assignments that lead to the terminal 1: $[\vec{x} \mapsto 1, \vec{y} \mapsto v]$ and $[\vec{x} \mapsto v, \vec{y} \mapsto 1]$. If `v` is composite, in the CFLOBDD returned, the assignments that lead to the terminal 1 are the ones with the property $[\vec{x} \mapsto a, \vec{y} \mapsto b]$ such that $a \times b = v \bmod n$, where n is the product of the moduli in `MultRelation` R.

References

[1] Bryant, R. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. on Computers 40* (1991), 205–213.

[2] Sistla, M. A., Chaudhuri, S., and Reps, T. W. CFLOBDDs: Context-free-language ordered binary decision diagrams. *ACM Trans. Program. Lang. Syst. 46*, 2 (2024), 7.