

# A CUDD Tutorial

Ethan L. Schreiber  
The University of California, Los Angeles  
ethan@cs.ucla.edu

December 21, 2008

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Acquiring CUDD</b>	<b>2</b>
<b>3</b>	<b>Making CUDD</b>	<b>2</b>
3.1	For All Platforms: . . . . .	2
3.2	For Linux: . . . . .	2
3.3	For Cygwin and OS X: . . . . .	2
<b>4</b>	<b>Linking the CUDD libraries</b>	<b>3</b>
<b>5</b>	<b>Basic Architecture</b>	<b>3</b>
5.1	Garbage Collection . . . . .	3
5.2	The Unique Table . . . . .	3
5.3	Complements . . . . .	3
5.4	Data Structures . . . . .	4
5.4.1	DdManager . . . . .	4
5.4.2	Useful DdManager functions: . . . . .	4
5.4.3	DdNode . . . . .	4
<b>6</b>	<b>Sample Program - Half-Adder</b>	<b>6</b>
6.1	Creating the BDD . . . . .	6
6.1.1	Description of createHalfAdderBDD(DdManager*) . . . . .	7
6.2	Reordering the BDD . . . . .	7
6.2.1	Manual Reordering . . . . .	7
6.2.2	Automatic Reordering . . . . .	8
6.2.3	Other useful reordering functions . . . . .	8
6.3	Restricting the BDD . . . . .	9
6.4	Printing the BDD . . . . .	10
<b>7</b>	<b>Further Resources</b>	<b>10</b>

# 1 Introduction

CUDD is the Colorado University Decision Diagram Package. It is a *c/c++* library for creating binary decision diagrams (BDD) as well as zero-suppressed BDDs (ZDD) and algebraic decision diagrams (ADD.) This document will only discuss the BDD functionality of CUDD. We will highlight and discuss

## 2 Acquiring CUDD

You can download CUDD by FTP with anonymous login from *vlsci.colorado.edu*. The latest version of CUDD is located in *pub*. As of the date of this publication, the current version is *cudd-2.4.1.tar.gz* .

## 3 Making CUDD

CUDD Is easiest to compile under linux but is also doable under Cygwin and OS X. The Makefile is well documented.

### 3.1 For All Platforms:

You need to modify the following line in the Makefile:

```
XCFLAGS = -mcpu=pentiumpro -malign-double -DHAVE_IEEE_754 -DBSD
```

The *-mcpu* flag is deprecated and unnecessary for compilation. You should simply remove it so that line becomes:

```
XCFLAGS = -malign-double -DHAVE_IEEE_754 -DBSD
```

### 3.2 For Linux:

If you are using a 32 bit OS, you should now be ready to compile. If you are using a 64 bit install, you should use the following line for XCFLAGS instead of the above:

```
XCFLAGS = -ansi -DBSD -DHAVE_IEEE_754 -DSIZEOF_VOID_P=8 -DSIZEOF_LONG=8
```

Now, simply type *make* and enter and the command line and you are set.

### 3.3 For Cygwin and OS X:

The function *void util\_print\_cpu\_stats(FILE \*fp)* inside the file *\$CUDD\_ROOT/util/cpu\_stats.c* also causes trouble. The fix is to disable this function. (If someone has a better solution, please email me about it.) To disable the function, you need to change two lines. First, find the code snippet towards the top of the file that reads as follows:

```
#if defined(_IBMR2)
#define etext _etext
#define edata _edata
#define end _end
#endif
```

And replace the first line (*#if defined(\_IBMR2)*) with:

```
#if 0
```

Next, replace the first line of *void util\_print\_cpu\_stats(FILE \*fp)* as follows. Replace:

```
#ifdef BSD
```

With

```
#if 0
```

You are now ready to compile. Now, simply type *make* and enter and the command line and you are set.

## 4 Linking the CUDD libraries

When your build compiles successfully, an include directory will be created in `$CUDD_ROOT` with symbolic links to all the external header files. Note that the following header file (as well as some others) should now exist as we will use it later:

```
$CUDD_ROOT/include/cuddObj.hh
```

There also should be 6 new c archive files created:

```
$CUDD_ROOT/cudd/libcudd.a
$CUDD_ROOT/util/libutil.a
$CUDD_ROOT/epd/libepd.a
$CUDD_ROOT/mtr/libmtr.a
$CUDD_ROOT/st/libst.a
$CUDD_ROOT/obj/libobj.a
```

Given this information, the following is a simple Makefile that will compile a c++ program utilizing CUDD:

Listing 1: Sample Cudd Makefile

```
CC = g++
CUDD_INCLUDE=cudd/cudd/libcudd.a cudd/util/libutil.a cudd/epd/libepd.a\
             cudd/mtr/libmtr.a cudd/st/libst.a cudd/obj/libobj.a

cudd_test: cudd_example.cpp
    $(CC) $(CUDD_INCLUDE) cudd_example.cpp -o cudd_example
```

## 5 Basic Architecture

### 5.1 Garbage Collection

CUDD has a built in garbage collection system. When we build BDDs, we tend to do so in a bottom up manner. In this process, we build many small BDDs that are subsumed as we traverse up the tree. When one of these small BDDs is subsumed, its memory can be reclaimed. In order to facilitate the garbage collector, we need to "reference" and "dereference" each node in our BDD. To reference a node, we use the function `Cudd_Ref(DdNode*)` and to dereference a node (and all of its descendants), we use `Cudd_RecursiveDeref(DdNode*)`. We will discuss this further later on in this document.

### 5.2 The Unique Table

While it is not necessary to work directly with the unique table, it is important to know that it exists, especially if it becomes necessary to inspect the CUDD source. This table is used to guarantee that a specific node is unique, that is to say that if two nodes contain the same children and represent the same variable, there should be merged into one node.

### 5.3 Complements

Each `DdNode` (see 5.4.3 for details) in our BDD either has two children or is a leaf with a constant value. The two children of a `DdNode` are referred to as the "then" child and the "else" child which are followed when we assign the value of this `DdNode` to true or false respectively. If we continue this process of following "then" or "else" children until we reach a leaf, the value of our assignment is the constant value of the leaf we reach. However,

there is one caveat with CUDD. "else" children can be complemented. If the else child is complemented, then when we reach a leaf node, we would take the complement of the value of the leaf. i.e., if the value of the leaf is 1 and we have traversed through an odd number of complement arcs, the value of our assignment is 0. In 5.4.3, we will discuss how to deal with complement arcs.

## 5.4 Data Structures

The two most important data structures within CUDD are the DdManager and the DdNode. We will now briefly discuss each.

### 5.4.1 DdManager

The DdManager is the central struct of CUDD. Creating this struct is the first thing you do when writing a CUDD program and it needs to be passed to almost every CUDD api function. It is never necessary to manipulate or inspect this struct directly, instead we will use the CUDD api. In order to initialize the DdManager, we use the following function:

Listing 2: The function to initialize the DdManager:

```
DdManager * Cudd_Init(
    unsigned int numVars,      // initial number of BDD variables (i.e., subtables)
    unsigned int numVarsZ,    // initial number of ZDD variables (i.e., subtables)
    unsigned int numSlots,    // initial size of the unique tables
    unsigned int cacheSize,   // initial size of the cache
    unsigned long maxMemory // target maximum memory occupation.(0 means unlimited)
);
```

Listing 3: For our purposes, we can call Cudd\_Init like this:

```
Cudd_Init(0,0,CUDD.UNIQUE_SLOTS, CUDD.CACHE_SLOTS, 0);
```

### 5.4.2 Useful DdManager functions:

- **int Cudd\_ReadSize(DdManager \* dd):** Returns the number of variables stored in the manager.
- **int Cudd\_ReadNodeCount(DdManager \* dd):** Returns the number of nodes stored in the manager. (i.e., many nodes can represent the same variable)

### 5.4.3 DdNode

The DdNode is the core building block of BDDs. It is defined as follows:

Listing 4: The decision diagram node:

```
struct DdNode {
    DdHalfWord index;      // Index of the variable represented by this node
    DdHalfWord ref;        // reference count
    DdNode *next;          // next pointer for unique table
    union {
        CUDD_VALUE_TYPE value; // for constant nodes
        DdChildren kids;       // for internal nodes
    } type;
};
```

The **index** is a unique index for the variable represented by this node. The nodes are numbered in order of creation starting with 0. These indices are permanent: note this means that if we reorder the BDD using an ordering heuristic (which will be discussed later), the nodes will change order but the indices will stay the same. To get the position in the order of a variable, we can use *Cudd\_ReadPerm (DdManager \*dd, int i)*; and to get the variable index of the variable at a position, we can use *Cudd\_ReadInvPerm (DdManager \*dd, int i)*;

**ref** stores a reference count for this variable. Every time *Cudd\_Ref* is called on this *DdNode*, the reference count is incremented by 1. Every time *Cudd\_Recursive\_Deref* is called on this node or an ancestor, this count is decremented by 1. When the reference count is 0, CUDD knows this node can be garbage collected.

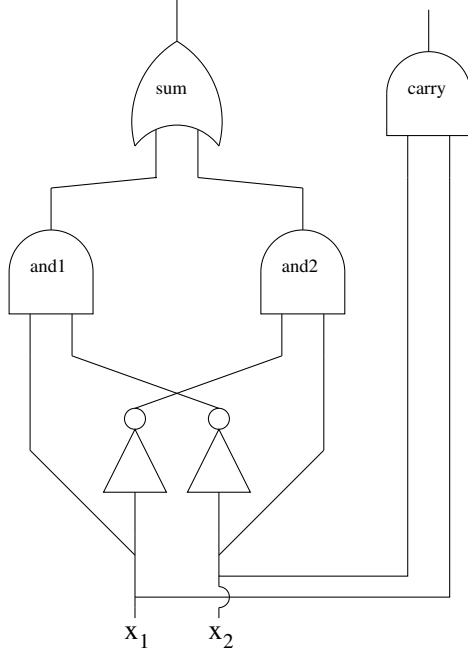
All Cudd nodes representing the same variable are linked together for the unique table. **next** contains a pointer to the next *DdNode* that represents the same variable as this *DdNode*.

Each *DdNode* contains either a value if it is a leaf node or pointers to two child *DdNode* structs. These values are stored in the **type** field. The following macros will help you work with this field.

- **Cudd\_IsConstant(DdNode\* node)** - Returns 1 if node is constant (meaning a leaf), and 0 otherwise.
- **Cudd\_T(DdNode\* node)** - Returns a pointer to the "then" child of a *DdNode*. This value is never complemented.
- **Cudd\_E(DdNode\* node)** - Returns a pointer to the "else" child of a *DdNode*. Remember that the value returned could be complemented. To find out if it is, we use:
- **Cudd\_IsComplement(DdNode\* node)** - Returns 1 if node is complemented, 0 otherwise. If a node is complemented, we can use:
- **Cudd\_Regular(DdNode\* node)** - Returns the regular version of a complemented node.
- **Cudd\_V(DdNode\* node)** - If a node is constant, this returns the value of the node.

## 6 Sample Program - Half-Adder

### 6.1 Creating the BDD



This is a half adder circuit that we will compile into an OBDD. It has the following truth table:

$x_1$	$x_2$	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Listing 5: C++ code to generate the Half-Adder circuit above as an OBDD in Cudd.

```
1 DdNode** createHalfAdderBDD (DdManager *manager)
2 {
3     DdNode *x0 = Cudd_bddIthVar (manager, 0);
4     DdNode *x1 = Cudd_bddIthVar (manager, 1);
5
6     DdNode *and1 = Cudd_bddAnd (manager, x0, Cudd_Not (x1));
7     Cudd_Ref (and1);
8
9     DdNode *and2 = Cudd_bddAnd (manager, Cudd_Not (x0), x1);
10    Cudd_Ref (and2);
11
12    DdNode *sum = Cudd_bddOr (manager, and1, and2);
13    Cudd_Ref (sum);
14
15    Cudd_RecursiveDeref (manager, and1);
16    Cudd_RecursiveDeref (manager, and2);
17
18    DdNode *carry = Cudd_bddAnd (manager, x0, x1);
19    Cudd_Ref (carry);
20
21    // There are two BDD roots so we return both of them.
22    DdNode **outputs = new DdNode*[2];
23    outputs[0] = sum;
24    outputs[1] = carry;
25
26    return outputs;
27 }
```

### 6.1.1 Description of createHalfAdderBDD(DdManager\*)

Listing 5 contains source code to create a BDD of a half adder circuit. This circuit takes two bits as inputs and returns the sum and carry of these two inputs. The construction of the BDD is done from the bottom up. We start by creating two Cudd variables, one for each input. We then combine these variables with or/and operations until we have our BDD. Lets go through this in some detail:

- **Cudd\_bddIthVar(manager,i)** - **Lines 3 and 4**: To declare our input variables, we use Cudd\_bddIthVar. This function will create a variable with index  $i$  if it does not exist or return a pointer to the existing variable if it does. Note that we do *not* have to call Cudd\_Ref on DdNode\* which are returned from Cudd\_bddIthVar.
- **Cudd\_bddAnd(manager,node1,node2)** (also **Cudd\_bddOr**) - **Lines 6,9,18**: We use these functions to combine DdNode pointers using conjoin and disjoin operations. If we have one BDD representing formula  $x$  and another representing formula  $y$ , these functions will return a new BDD representing  $x \wedge y$  or  $x \vee y$  respectively.
- **Cudd\_Ref(manager,node)** - **Lines 7,10,13,19**: Note that we must increase the reference count of all DdNode structs returned by Cudd\_bddAnd and Cudd\_bddOr.
- **Cudd\_RecursiveDeref(manager,node)** - **Lines 15,16**: Note that we had to decrease the reference count of and1 as well as and2 after they were subsumed by the new BDD pointed to by sum.

## 6.2 Reordering the BDD

The order we traverse variables down each path of a BDD of course can have a tremendous effect on the number of nodes needed to construct it. CUDD provides a rich set of tools for reordering BDDs. The ordering of variables is controlled by a heuristic function. CUDD provides a number of heuristic functions which are described in the CUDD documentation. [2]

Reordering can either be invoked manually or automatically.

### 6.2.1 Manual Reordering

To invoke ordering manually, you must call the following function:

Listing 6: Function to force reordering of the BDD.

```
int Cudd_ReduceHeap(  
    DdManager * manager,           // DD manager  
    Cudd_ReorderingType method,    // method used for reordering  
    int minsize                    // bound below which no reordering occurs  
);
```

We pass the manager pointer as always. The second parameter allows us to specify which reordering heuristic to use. The following is the list of constant ints representing the heuristics provided by CUDD.

CUDD_REORDER_NONE	CUDD_REORDER_SAME
CUDD_REORDER_RANDOM_PIVOT	CUDD_REORDER_EXACT
CUDD_REORDER_RANDOM	CUDD_REORDER_SIFT
CUDD_REORDER_WINDOW2	CUDD_REORDER_SIFT_CONVERGE
CUDD_REORDER_WINDOW2_CONV	CUDD_REORDER_SYMM_SIFT_CONV
CUDD_REORDER_WINDOW3	CUDD_REORDER_GROUP_SIFT
CUDD_REORDER_WINDOW3_CONV	CUDD_REORDER_GENETIC
CUDD_REORDER_WINDOW4	CUDD_REORDER_ANNEALING
CUDD_REORDER_WINDOW4_CONV	CUDD_REORDER_GROUP_SIFT_CONV
CUDD_REORDER_SYMM_SIFT	

The final parameter is the minimum number of nodes that must be in the BDD in order to reorder. This prevents the cost of reordering small enough BDDs.

### 6.2.2 Automatic Reordering

Alternatively, ordering can be triggered automatically when the number of nodes in the BDD passes a certain threshold. The following is the functions used for dynamic reordering (it is tuned off by default):

Listing 7: Function to turn on automatic reordering of variables.

```
Cudd_AutodynEnable(
    DdManager * manager,          // DD manager
    Cudd_ReorderingType method, // method used for reordering
)
```

The parameters passed are the same as for Cudd\_ReduceHeap.

Listing 8: Function to turn on automatic reordering of variables.

```
Cudd_AutodynEnable(
    DdManager * unique,          // DD manager
    Cudd_ReorderingType method, // method used for reordering
)
```

### 6.2.3 Other useful reordering functions

Listing 9: A function to order variables according to a specified order as opposed to a heuristic.

```
int Cudd_ShuffleHeap(
    DdManager * manager, // DD manager
    int * permutation    // required variable permutation
)
```

The permutation is an array of positions in the order. The value of the  $i^{th}$  slot in the array represents the position of the variable with index  $i$ .

Listing 10: A function to return the position in the order of the  $i^{th}$  variable.

```
int Cudd_ReadPerm(
    DdManager * manager, // DD manager
    int i              // The variable to get the position of
)
```

Listing 11: A function to return the variable index of the variable currently at position pos.

```
int Cudd_ReadInvPerm(
    DdManager * manager, // DD manager
    int pos             // The position of the variable index to get
)
```



### 6.3 Restricting the BDD

Listing 12: This function will restrict BDD to the BDD represented by restrictBy

```
DdNode * Cudd_bddRestrict(  
    DdManager * manager,    // DD manager  
    DdNode * BDD,           // The BDD to restrict  
    DdNode * restrictBy)   // The BDD to restrict by.
```

The following is code to restrict a BDD to a set of assignments to its inputs. It takes a node to restrict and a map of assignments to inputs. The key of the map is the index of the variable to assign and the value is whether to assign it to true or to false. The function returns the original BDD restricted to the assignment.

Listing 13: This function uses restrict to test the BDDs created in listing 5

```
void test(DdManager* manager, DdNode **node)  
{  
    DdNode *x0 = Cudd_bddIthVar(manager,0);  
    DdNode *x1 = Cudd_bddIthVar(manager,1);  
  
    const int SIZE=4;  
    DdNode* restrictBy[SIZE];  
    DdNode* testSum[SIZE];  
    DdNode* testCarry[SIZE];  
  
    // Restrict by the following assignments  
    restrictBy[0] = Cudd_bddAnd(manager,Cudd_Not(x0),Cudd_Not(x1)); // x1=0 and x2=0  
    restrictBy[1] = Cudd_bddAnd(manager,Cudd_Not(x0),x1);         // x1=0 and x2=1  
    restrictBy[2] = Cudd_bddAnd(manager,x0,Cudd_Not(x1));         // x1=1 and x2=0  
    restrictBy[3] = Cudd_bddAnd(manager,x0,x1);                   // x1=1 and x2=1  
  
    for (int i=0;i<SIZE;i++) {  
        Cudd_Ref(restrictBy[i]); // Reference restrictBy  
  
        // Now restrict by the new functions  
        testSum[i] = Cudd_bddRestrict(manager,node[0],restrictBy[i]);  
        testCarry[i] = Cudd_bddRestrict(manager,node[1],restrictBy[i]);  
  
        Cudd_RecursiveDeref(manager,restrictBy[i]); // clean up restrictBy  
    }  
  
    cerr << "(x1=0, x2=0): sum = " << 1-Cudd_IsComplement(testSum[0])  
        << " Carry = " << 1-Cudd_IsComplement(testCarry[0]) << endl  
        << "(x1=0, x2=1): sum = " << 1-Cudd_IsComplement(testSum[1])  
        << " Carry = " << 1-Cudd_IsComplement(testCarry[1]) << endl  
        << "(x1=1, x2=0): sum = " << 1-Cudd_IsComplement(testSum[2])  
        << " Carry = " << 1-Cudd_IsComplement(testCarry[2]) << endl  
        << "(x1=1, x2=1): sum = " << 1-Cudd_IsComplement(testSum[3])  
        << " Carry = " << 1-Cudd_IsComplement(testCarry[3]) << endl;  
  
    for (int i=0;i<SIZE;i++) {  
        Cudd_RecursiveDeref(manager,testSum[i]);  
        Cudd_RecursiveDeref(manager,testCarry[i]);  
    }  
}
```

## 6.4 Printing the BDD

Cudd provides a nice function for dumping a BDD to graphviz format. The following listing prints the circuit to a dot file.

Listing 14: This function writes the BDDs created in listing 5 to a dot file

```
/**
 * This takes an array of 2 output nodes and prints them to a dot file
 */
void toDot(DdManager *manager, DdNode **outputs)
{
    char **inputNames = new char*[2];    // Label the two input nodes
    inputNames[0] = new char[3];
    inputNames[1] = new char[3];
    inputNames[0] = "x1";
    inputNames[1] = "x2";

    char **outputNames = new char*[2];   // Label the two output nodes
    outputNames[0] = new char[4];
    outputNames[1] = new char[6];
    strcpy(outputNames[0], "sum");
    strcpy(outputNames[1], "carry");

    FILE *f = fopen("./half_adder.dot", "w");

    // manager: The cudd manager
    // 2       : The number of outputs
    // outputs: An array of outputs (DdNode*)
    // inames  : Maps input nodes to their names
    // onames  : Maps output nodes to their names
    // f       : The file to write to

    Cudd_DumpDot(manager, 2, outputs, inputNames, outputNames, f);
}
```

## 7 Further Resources

Both websites in the references section are highly recommended. Furthermore, the half-adder program used in this tutorial as well as a Makefile to build it are available on the web at:

[http://www.cs.ucla.edu/~ethan/code/half\\_adder\\_cudd.tar.gz](http://www.cs.ucla.edu/~ethan/code/half_adder_cudd.tar.gz)

## References

- [1] Jacqueline E. Rice. Local cudd tutorial, November 2004. <http://www.cs.uleth.ca/~rice/cudd.html>.
- [2] Fabio Somenzi. Cudd: Cu decision diagram package: Release 2.4.1, May 2005. <http://vlsi.colorado.edu/~fabio/CUDD/>.