# Homework 2 (100 points)
# Due Monday, February 21 @11:59pm
# **Gradescope**

The following homework problems are to be completed and turned in on Gradescope on the due date. **Please follow the Gradescope tutorial on Canvas when submitting homework.**

## 1. BDDs: Static Variable Ordering [10 pts]

We saw that variable order is highly significant for something as simple as a multiplexor. How about something like a *comparator*? A simple comparator takes two 2-bit unsigned binary numbers $a_1a_0$ and $b_1b_0$ and compares their magnitude and sets the output $z = 1$ just if $a_1a_0$ is **less than or equal** to $b_1b_0$. Do this:

(i) [3 pts] **Draw** the BDD for this "bad" variable ordering: $\boldsymbol{a_0a_1b_0b_1}$.

(ii) [4 pts] **Draw** a gate-level netlist using any AND, OR, NOT, EXOR gates you want, to implement this simple comparator circuit.

(iii) [3 pts] Go grab this paper from the Canvas page: *Minato et al "Shared BDDs with Attributed Edges for Efficient Boolean Function Manipulation," Proceedings DAC 1990.* Read Section 4 of this paper (it's just 1/2 page long) and look at Figure 7. This is a very simple ordering heuristic from the early days of BDDs. Apply Minato's ordering heuristic (and where you need to break ties or make any arbitrary ordering decision, just tell us what you did and show the work). **Show** what variable ordering it produces.

## 2. BDDs: Dynamic Variable Ordering [10 pts]

Real BDD software uses something more sophisticated: *dynamic* variable ordering. Read the paper on the Canvas page: R. Rudell, *"Dynamic Variable Ordering for Ordered Binary Decision Diagrams," Proc. ICCAD Conference, 1993.*

The key idea in Rudell's method is to iteratively move all the nodes for one single variable "up" the BDD, then "down" the BDD, until you find the best, smallest BDD. The idea is not to find the best overall order for *all* variables, but just the *best* location for this *single* variable.

Do This: from the BDD you built for the bad ordering in problem 1, **show** how to use the "sifting method" from Rudell's paper to sift variable $b_1$ up the tree to the top. You don't need to sift it back down to the bottom, just show the dynamically reordered BDD after each sifting step, and then **comment** on which location in this set of reorderings is best for variable $b_1$.

## 3. Derived Operators for BDDs [20 pts]

Suppose we have a software package that has data structures representing variables and Boolean functions as BDDs, and that the following operations are available as subroutines in this software. (We will write these in a simplified sort of C language notation):

*bdd* **var2func**(*var* x) Generate the BDD corresponding to a single variable x. Input is a single variable (of type *var*), and the returned output is a BDD.

*bdd* **ITE**(*bdd* I, *bdd* T, *bdd* E)

Compute the if-then-else operation. Inputs are 3 BDDs, called I (*if* part), T (*then* part) E (*else* part), and the returned output is another BDD.

*bool* **iszero**(*bdd* func) Returns integer 1 just if func is the always-zero function. Input is a BDD, output is boolean 1 or 0 (it's *not* a BDD).

bdd **cofactor**(*bdd* func, *var* x , *bool* val)

Computes cofactor of func with respect to *var* x, setting x = val. func is a BDD, var is a variable, val is boolean 0 or 1.

*bdd* **AND**(*bdd* f, *bdd* g)
*bdd* **OR**(*bdd* f, *bdd* g)
*bdd* **EXOR**(*bdd* f, *bdd* g)
*bdd* **NOT**(*bdd* f)

Compute the basic logical (gate type) operations on BDDs AND, OR and EXOR create new BDDs representing the logical AND, OR and exclusive-or of their inputs. NOT creates the BDD for the complement of its input.

*bdd* CONST1, CONST0

You can assume these two BDDs are already defined. These are just the constant 1 function and the constant 0 function. Note **iszero**(CONST0) == (bool) 1.

No other operations are implemented, and there is no way for you to examine the BDD data structure directly.

Describe (in C-like pseudo-code notation–we *don't* need real code here!) how you would implement the following operations:

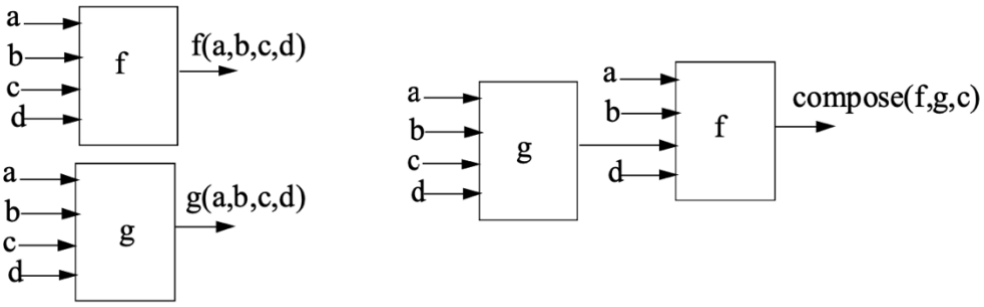
- *bool* **depends**(*bdd* f, *var* x)Determine whether function f depends on the specified variable x. This means if you change x, at least sometime the output of f will also change. f is a BDD, x is a variable, **depends** returns boolean 0 or 1.

- *bdd* **univquant**(*bdd* f, *var* x)
  Compute universal quantification of function f w.r.t. variable x. f is a BDD, x is a variable, **univquant** returns a BDD.

- *bool* **opposite**(*bdd* f, *bdd* g)
  Determine whether two functions are complementary, i.e., if one of them is the complement of the other. f and g are BDDs, **opposite** returns boolean 0 or 1.

- *bdd* **exchange**(*bdd* f, *var* a, *var* b)
  Exchange roles of specified variables in function f. For example, **exchange**(a•b + d, a, d) —> d•b + a. f is a BDD, a and b are variables, **exchange** returns a BDD.

- *bdd* **compose**(*bdd* f, *bdd* g, *var* x)
  Creates a new function (*composition* function) with var x in f set to the output of g. The picture below clarifies what we are computing. f and g are BDDs, x is a variable, and **compose** returns a BDD.




**Hint:** lots of things that look *difficult* are *easier* when you **cofactor** them and look at the cofactors. Play around with ITE of various combinations of the cofactors. The first 3 operators are pretty straightforward, the last 2 are rather tricky. *None* of these things requires some sophisticated

recursive algorithm, just a few lines of calls to the right operators with the right inputs. To emphasize this, here is the answer for the first part, for **depends**(*bdd* f, *var* x):

```
bool depends(bdd f, var x) {
    return( ! iszero( EXOR( cofactor(f, x, (bool)0), cofactor(f, x, (bool)1) ) );
}
```

Notice how this works. If function $f(\,)$ depends on variable $x$, then if I change $x$ from $x = 0$ to $x = 1$, there ought to be at least some pattern for the remaining inputs that makes the output of the function *change*. But this is exactly what the Boolean Difference tries to compute. So, we compute the BDD for a new function $f(\ldots x = 0 \ldots), f(\ldots x = 1 \ldots)$ using calls to cofactor and to EXOR. What does this new function tell us? If the new function is zero always, for all inputs, then you cannot affect the output of function $f$ by changing variable $x$. In other words, $f$ does not depend on variable $x$. So if the **iszero**( ) function returns boolean 1, it means the original function does *not* depend on $x$. To get the true/false return for **depends**() correct, we have to invert this, which is what the "!" does.

## 4. Reverse FSM Reachability Analysis [10 pts]

In the lecture about FSM verification, we introduced the concept of *reachability analysis*, where we represent as a Boolean function the set of all states that our machine can visit from a known start state in 0 clock ticks (we called it $R_0$), 1 tick ($R_1$), 2 ticks ($R_2$), and so on. This is actually a *forward* reachability analysis: you pick a state in which the FSM starts and then you go forward in time to see what states you can reach. It turns out that you can also go *backwards* in time. You pick an "end state" and you go *backwards* to see which sets of *prior* states could have got you into that state. It turns out that this has one very nice technical advantage: you *don't* need to build the BDD for the transition relation:

$\delta$(old state, input, next state) = 0 or 1

In practice it is often true that $\delta$ can make a huge, nasty, difficult BDD.

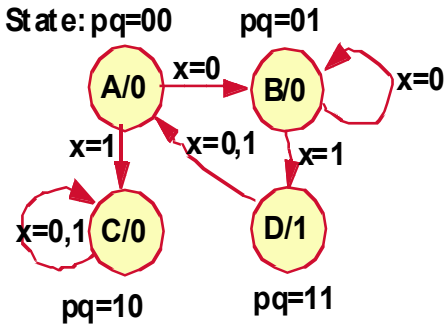
Consider again our simple machine from the lecture.

Let $R_0 = \{B\}$ be our arbitrarily chosen end state. Then we expect the set of states prior to $R_0$— the states we could have been in one clock tick in the *past*, and still reached state $B$ one tick *later* — to be $\{A, B\}$, which we call $R_{-1}$. In this notation, $R_{-k}$ denotes the Boolean function that represents the set of all states that could reach our end state $\{B\}$ in not more than $k$ clock ticks. It turns out that

there is another quantification sort of formula to figure out how to compute these sets. For our particular little FSM, that formula is:

$$R_{-(k+1)}(p,q) = R_{-k}(p,q) + (\exists x) R_{-k}[p^+(p,q,x), q^+(p,q,x)]$$

where $R_{-k}(p,q)$ is the Boolean function for a set of reachable states, and $(p,q)$ is the bit pattern representing a state in our FSM. $p^+(p,q,x)$ and $q^+(p,q,x)$ are the next-state equations from the FSM. If you start in state $(p,q)$ and see input $x$, then you go to state $(p^+, q^+)$.




$R_0$ = end state = { B }
$R_{-1}$ = prior states 1 tick before = {A B}
$R_{-2}$ = prior states 2 ticks before = {A B D}
$R_{-3} = R_{-2}$

For this problem, do the following:

(i) [4 pts] Explain briefly *why* this formula works. In English, what is going on here? The key is to look at each term in the equation, and the quantification operation, and understand why this formula gives us a new Boolean equation that represents the states our FSM could be in one clock tick earlier.

(ii) [6 pts] Assume that $R_0(p,q) = \{\text{state } B\} = \bar{p}q$. Then, show how to use this formula to compute $R_{-1}(p,q)$. Does the result make sense?

## 5. Your Very Own "Baby" BDD Package: YBDD [50 pts]

The best way to really understand how BDDs work is to *write* a real BDD package. Of course, since that's a lot of work, you're not doing that here. In this problem, you get to "complete" the missing code in a skeleton for a toy BDD package. You add a couple hundred lines of code and, as the French say: *voila*, a BDD package.

Your main task in this problem is on the implementation of the ITE operator. Once you know how to do this, everything else is pretty straightforward. You will support shared nodes for multi-rooted DAGs, but no complement arcs inside the BDD. We provide a command line interface to drive the BDD package. We provide C++ version of the code skeleton you need. The provided files include:

- The C++ files, including README
- Scripts to exercise your BDDs are in folder ***tests/***. These are source files that can be sourced through the command line interface.

**The BDD Data Structure in C++**

The BDD structure has been done for you in *bdd_node.h.* Here is a brief description of each field and method in the BDD object:

**Instance Variables:**

- **marked:** Used for traversing the BDD. Marks that you have visited a note, used for recursive algorithms that do things like count the number of nodes. When you create a new BDD node, set this field to FALSE.
- **index:** For non-terminal nodes, index defines the variable associated with a node. Variables are indexed 0 to $n-1$. The index therefore defines the global ordering. For terminal nodes, index should be set to ZERO_INDEX or ONE_INDEX for the constant 0 and 1 nodes, respectively.
- **label:** A String representing the name of the node (redundant information if index is present but useful for the methods described below)
- **low, high:** These are low and high pointers to BDD nodes.

**Instance Methods:**

- **bdd_node(....):** The constructor method, does instance variable initialization.
- **int getIndex()**: Returns the index of that node.
- **bdd_node* getLow():** Returns pointer to the 'low' BDD child node.
- **bdd_node* getHigh():** Returns pointer to the 'high' BDD child node.
- **char* getLabel():** Returns the label of the node.
- **int Size():** Recursively calculates the size of the BDD structure including the current node and its children.
- **void Print(....):** Recursively prints the BDD node structure starting at the current node, and working down to the children.
- **void cleanMarks():** Cleans up after Size or Print have been called.
- **void total():** Counts total number of BDD nodes in your multi-rooted DAG, and prints this number.

We also defined a simple Hash tables in bdd_node.h. The class name is Hash_Table. The key is of type char *, while stored element is of type bdd_node *. The interfaces to it are:

- **Hash_Table(....):** The constructor method, initialize number of elements**.**
- **int getSize():** Returns number of elements in the Hash table.
- **void insert(char *key, bdd_node *element**): Insert an element using a string key.
- **void remove(char *key):** Remove an element using a string key.

For more detailed information on any of these variables or methods, refer to comments in *bdd_node.h*. The command line interface is defined in *bdd.cxx*.

**BDD Basics: Core Manipulation Routines**

We have provided you with a set of stubs, i.e., *empty* routines that need to be filled in. Do **not** change these routine names since they are used in the command line interface. You need to fill in the following stubs in *bdd_functs.h*:

- **bdd_node* Zero(), bdd_node* One():** Routines to create the constant 0 and 1 BDD nodes. They have to be inserted in the unique_table.
- **bdd_node* NewVar(String label):** Create a new BDD variable**.**
- **bdd_node* ITE(I, T, E):** This is the main ITE routine.
- **bdd_node* NOT(f), bdd_node* AND(f, g), bdd_node* OR(f, g), bdd_node* XOR(f, g):** These should be defined in terms of the ITE operator.

These are the main methods you must code in order to create a working BDD package.

**Command Line Interface**

A command line interface is linked in with the program BDD. Executing **./ybdd** will bring up the command line interface. The package will respond with a prompt:

```
YBDD>
```

At this point the user can issue one of the following commands:

- **boolean** *list-of-var-names*
- **eval** *dest expr*
- **source** *filename*
- **bdd** *funct*
- **size** *funct*
- **total**
- **quit**

The following describes what each command does.

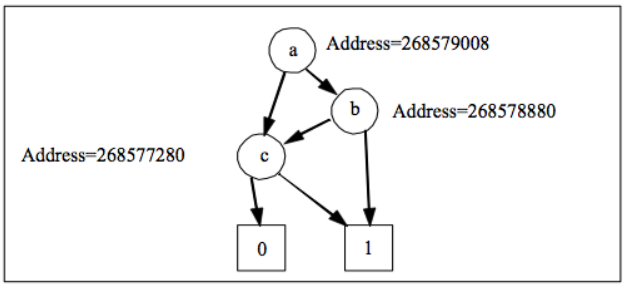
**boolean:** The command boolean defines a list of variables. The order of the variables is the order in which they are defined. Example: boolean a b c d command defines variables a, b, c, d with indices 0, 1, 2, 3 respectively.

**eval:** The command eval evaluates the expression and assigns it to the destination. Expressions can have the following operators: ! (complement), & (AND), + (OR) and ^(XOR). Only **ONE** operator per eval

command can be used. Example: eval f a & b evaluates the Boolean expression (ab) and creates the corresponding BDD with the function name f, i.e., it does f = ab.

**bdd:** The command bdd prints the BDD representation for the function. The BDD representation for a&b + c and its corresponding pictorial representation are shown below:

```
a:268579008
        c:268577280
                [0]
                [1]
        b:268578880
                c:268577280
                [1]
```




**size:** The command size returns the number of BDD nodes in the function. For example size will return 5 for the above function.

**total:** The command total returns the total number of BDD nodes created.

**source:** All the commands can be placed in a file and then the file can be sourced via the command source. An example source file (for the function: **f = a&b + b&c + a&c**) is:

```
boolean a b c
eval t1 a & b
eval t2 b & c
eval t3 c & a
eval t4 t1 + t2
eval f t4 + t3
size f
bdd f
```

**To Test Your Code**

You can look in folder ***tests/*** for a good selection of test scripts. The **sanitycheck** scripts are all small and simple, and should be tried first. Next, try some of the **easy** scripts, which are just a bit bigger. The adder and multipler directories contain, no surprise, adders and multipliers. The “r” and “w” labels on the scripts tell whether the variable ordering is “right” or “wrong”. “Wrong” orders make intentionally *very* big BDDs.

**Coding credit: (35 pts earned on Gradescope coding assignment)**

Specifically we are looking for the following:

- **[5 pts] Submit your source code to finish the skeleton (we only need your** *bdd_functs.h,* don't change file name!**)** – It should be uploaded via Gradescope coding submission interface. We want this source code and it needs to be executable on **Ubuntu 18.04** (i.e., the grading system used by Gradescope). You can run as many times as you want via the submission interface and choose the file you want to finally upload to be graded. You should write good comments inside your *bdd_functs.h* near these added functions and include any info we need to understand your code. This part is maunally graded to test if your submission is executable and passes similarity check.

- **[10 pts] Passing these five sanity test** – check these test scripts in ***tests/sanitycheck/***. This part is automatically graded by the interface.

- **[10 pts] Passing other tests that are randomly chosen within the folder *tests/***. This part is automatically graded by the interface.

- **[10 pts] Ling adder verification.txt file** – check the description of problem 5(ii) below, we only need your *verification.txt*. A template of *verification.txt* has been provided, where you are required to fill in the missing part and then run it using `source verification.txt`. You should not change these exisiting lines inside *verification.txt*. The first command of this file is: `boolean a7 b7 a6 b6 a5 b5 a4 b4 a3 b3 a2 b2 a1 b1 a0 b0`; where we define the variable order of BDD for you. The last two commands of this file are : 1) point out which function you show BDD for and its diagram should explicitly answer the verification question, i.e. `bdd funct`; 2) show the size of this BDD, i.e., `size funct`. Particularly, we will use this unique size number to determine whether your *verification.txt* file is correct. You should write good comments inside your code and include any info we need to understand your code. 7 pts of this part are automatically returned by the interface, the remaining 3 pts are manually given for comments of descriptions on how you design the `funct`.

**Written-up credit**

(i) **[10 pts] Benchmark numbers for adders:** in ***tests/adders***, *addr4.src, addr8.src, addr16.src, addr64.src, addw4.src, addw8.src, addw16.src, addw64.src* are a big set of adders. 4,8,16 and 64 bit adders with two different variable orderings. We want a **table** in your writeup homework that reports the sizes of your BDDs: one for the "r" benchmarks and one for the "w" benchmarks. NOTE: *addw16.src* and *addw64.src* might not be able to finish in any reasonable amount of time on your machines. That's fine. Just report that they did not finish.

(ii) **[5 pts] Ling adder verification:** In snooping around the web, we found this set of equations which defines a famous (and famously difficult) fast adder structure called a *Ling Adder*: for $i = 0,1,2, \ldots 7$

$$g_i = a_i b_i$$
$$p_i = a_i + b_i$$
$$t_i = g_i \oplus p_i \text{ (Ling transfer signal)}$$

$$s_7 = t_7 \oplus c_7$$
$$c_7 = G_1 + P_1 G_0$$
$$G_1 = G_{7:4} = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4$$
$$= p_7(g_7 + g_6 + p_6 g_5 + p_6 p_5 g_4)$$
$$G_0 = G_{3:0} = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$$
$$= p_3(g_3 + g_2 + p_2 g_1 + p_2 p_1 g_0)$$
$$P_1 = P_{7:4} = p_7 p_6 p_5 p_4$$

$$H_1 = g_7 + g_6 + p_6 g_5 + p_6 p_5 g_4$$
$$H_0 = g_3 + g_2 + p_2 g_1 + p_2 p_1 g_0$$
$$P_{6:3} = p_6 p_5 p_4 p_3$$
$$h_8 = H_1 + P_{6:3} H_0 \text{ (Ling Pseudo Carry)}$$
$$c_7 = p_7 h_8$$

A Ling Adder does some fancy magic with the carry signal, in the form of carry lookahead, to make the adder delay very small. In this simple example, the two 8-bit operands are $[a_7 - a_0]$ and $[b_7 - b_0]$, and the result is the sum $[s_7 - s_0]$. There is no carry-in to the low bit (bit 0) in these equations. $c_7$ is the carry signal *out* of the top bit (bit 7) of the adder.

Show that you can use your YBDD code to verify the Ling adder $s_7$ sum bit (just this one bit). Is it correctly the top bit of an 8-bit addition with no carry-in, yes or no? This means you have to use a correct 8-bit reference adder to compare it to (it's not that hard using YBDD commands set). That being said, you only need to compose a source file named *verification.txt* (a sequence of commands) to finish this task. Show some "illustrative output" from running your *verification.txt*, and explain how it worked in your homework writeup.

**Hint:** see a correct 8-bit reference adder at *tests/adders/addr8.src*, which we already included in the verification.txt template. The top bit of this reference adder is called `s7_true`.