系統晶片驗證 (SoC Verification)

109 學年下學期 電機系電子所選修課程 943 U0250

Homework Assignment #3 [BDD-Based Verification]

(Due: 9:00pm, Thursday, April 15, 2021)

0. Objectives

1. Learning how to implement a BDD-based verification tool using the reachability analysis algorithm.

2. Verifying the "vending machine" design from HW #1.

1. Problem Description

In this program assignment, you are going to implement a BDD-based verification tool to verify the "vending machine" design in HW #1. The same platform, V3, as in HW #1 is provided, where a complete Verilog front-end, synthesizer, and additionally a simple object-oriented BDD package are included. You are required to write your program on top of the reference code. The generated executable has the following usage:

bddv [**-File** <*dofile*>]

where the **bold words** indicate the command name or required entries, square brackets "[ ]" indicate optional arguments, and angle brackets "< >" indicate required arguments. Do not type the square or angle brackets.

In additional to the circuit and simulation related commands supported in HW#1, this BDD-based verification tool has the following functionalities. Please refer to Section 2 for supported commands and their usage:

1. A BDD calculator that can specify the BDD variables, perform various logic operations on BDDs (e.g. AND, OR, INV, etc), conduct existential quantifications, retrieve cofactors, compare two BddNodeVs, and simulate a pattern, etc.

2. Printing BDDs in text or "dot" format.

3. Building the BddNodeVs for the circuit gates with the specified BDD variable ordering on PIs.

4. Setting the initial state of the circuit.

5. Constructing the transition relationship BDD.

6. Performing the reachability analysis with the specified number of timeframes. If the fixed point is reached, this process should stop.

7. Checking whether an assertion property (i.e. AG(p)) is true (safe) by the set of reachable states. If the property is false (violated), print out the counter example.

2. Supported Commands

Other than the commands supported in HW #1, in this homework, we will support these new commands:

`BAND:`	`BDD And`
`BCOFactor:`	`Retrieve BDD cofactor`
`BCOMpare:`	`BDD comparison`
`BCONstruct:`	`Build BDD From Current Design`
`BDRAW:`	`BDD graphic draw`
`BEXist:`	`Perform BDD existential quantification`
`BINV:`	`BDD Inv`
`BNAND:`	`BDD Nand`
`BNOR:`	`BDD Nor`
`BOR:`	`BDD Or`
`BREPort:`	`BDD report node`
`BRESET:`	`BDD reset`
`BSETOrder:`	`Set BDD variable order from circuit`
`BSETVar:`	`BDD set one variable order`
`BSIMulate:`	`BDD simulation`
`BXNOR:`	`BDD Xnor`
`BXOR:`	`BDD Xor`
`PCHECKProperty:`	`check assertion property by BDDs`
`PINITialstate:`	`set Initial state BDD`
`PIMAGe:`	`build the next state images in BDDs`
`PTRansrelation:`	`build the transition relationship in BDDs`

All the command interfaces are included in the reference code (e.g. in “bddCmd.cpp” and “proveCmd.cpp”). You don’t need to work on them. All but the proof-related commands, `PCHECKProperty`, `PINITialstate`, `PTRansrelation`, `PIMAGe`, are ready for execution. You only need to finish the TODOs in the `ntk` and `prove` packages.

Please refer to the documents/tutorials of V3 and HW #1 for the lexicographic notations and the circuit-related commands.

Please note that many commands require the lookup from name or IDs to BddNodeV. In our command convention, "string varName" refers to a valid variable name which must start with [a-zA-Z_] and contains only [a-zA-Z_[]]. On the other hand, "string bddName" can be a valid string name (i.e. varName) or ID (i.e. an integer). In V3, each net is associated with a net ID. You can use the commands "`print ntk`" and "`netinfo`" to browse the netlist and find the corresponding net IDs. Besides, since BDD is constructed from a Boolean netlist. You need to apply the command "`blast ntk`" before constructing BDDs.

2.1 Command "BRESET"

Usage: **BRESET <(size_t nSupports)> <(size_t hashSize)> <(size_t cacheSize)>**

Description: Reset the BDD manager. It cleans up all the BDD nodes and resets the number of BDD supports (variables) as well as the buckets in BDD hash and cache. It should also delete all the "BddNodeVInt*" in the "BddManager::_uniqueTable" (See Section 3) and BddNodeVs for reachability analysis. The initial (default) value for the number of BDD supports (variables) and the numbers of buckets in BDD hash and cache are 64, 8009, and 30011, respectively. Please note that the BDD manager will be reset even if the specified parameters are the same as the previous setting.

2.2 Command "BSETVar"

Usage: **BSETVar <(size_t level)> <(string varName)>**

Description: Define the association of the BDD input variables --- Associates the <varName> to the <level>-th BDD input variable. If "level <= 0" or "level > number of supports in BDD manager", a command error will be issued. Please note that BDD input variable level starts from '1', while "<level> = 0" is for "const 1" BddNodeV, "<level> = 1" is the lowest and "<level> = n" is the highest BDD nodes. It is OK to associate different names to the same <level> of support. However, if the <varName> has been associated with other BddNodeV, an error will be issued, even if the <varName> has been associated with the same input level. Please refer to the command BSETOrder for the BDD input variable association with circuit inputs.

2.3 Command "B{INV|AND|OR|NAND|NOR|XOR|XNOR}"

Usage: **BINV <(string varName)> <(string bddName)>**

BAND <(string varName)> <(string bddName)>...

BOR <(string varName)> <(string bddName)>...

BNAND <(string varName)> <(string bddName)>...

BNOR <(string varName)> <(string bddName)>...

BXOR <(string varName)> <(string bddName)>...

BXNOR <(string varName)> <(string bddName)>...

Description: Build a BddNodeV with the specified function. The first argument after the command will be the name of the newly built BddNodeV. It must be a valid variable name (i.e. varName). If it has been associated with another BddNodeV already, the old BddNodeV will be overwritten *without warning*. So please be careful not to overwrite the BddNodeV for a signal in a circuit. The rest of the argument(s) is(are) the input(s) to this function. They can be valid variable names or valid net IDs in the circuit (i.e. bddName = varName | netId). If the corresponding BddNodeV of any of them does not exist, a command error will be issued. Other than "BINV", which has only one function input, all the other commands can accept more than one function inputs (note: one is OK). An error will be issued if the number of function inputs does not meet the requirement. The resulted BddNodeV with the name association will be recorded in BDD manager.

Example:

```
bdd> band c a b  // build the BDD for c = AND(BDD("a"), BDD("b"))
bdd> bor c d     // as an assignment from d to c
```

2.4 Command "BCOFactor"

Usage: **BCOFactor <-Positive | -Negative>**

<(string varName)> <(string bddName)>

Description: Retrieve the positive or negative cofactor of a BDD node. Please note that the parameter order cannot be altered.

Example:

```
bdd> bcof -p a b  // a = positiveCofactor(b)
```

2.5 Command "BEXist"

Usage: **BEXist <(size_t level)> <(string varName)> <(string bddName)>**

Description: Perform the existential qualification on a BDD node. Please note that the parameter order cannot be altered.

Example:

```
bdd> bex 3 fa f  // fa = ∃a.f, assume a's level = 3
```

2.6 Command "BCOMpare"

Usage: **BCOMpare <(string bddName)> <(string bddName)>**

Description: Compare the functional equivalence of two BddNodeVs. If any of the bddNames does not correspond to a BddNodeV, an error will be issued. Otherwise, the output should be in one of the following format:

```
"nodeA" and "nodeB" are equivalent.
"nodeA" and "nodeB" are inversely equivalent.
"nodeA" and "nodeB" are not equivalent.
```

2.7 Command "BSIMulate"

Usage: **BSIMulate <(string bddName)> <(bit_string inputPattern)>**

Description: Evaluate the input pattern <inputPattern> for the BddNodeV <bddName>. If the BDD node <bddName> does not exist, or if the length of the pattern is shorter than the level of the node <bddName>, an error will be issued. It is OK if the length of the pattern is longer than the level of the node in which the latter bits of the pattern will be ignored. If the pattern contains any non-0/1 character, an error will be issued. Please note that pattern bits 0 (i.e. inputPattern[0]) ~ n-1 (i.e. inputPattern[n-1]) are corresponding to the supports level 1 ~ n, respectively.

Examples:

```
bdd> bsim b 11001              // for BddNodeV with name = "b"
BDD Simulate: 11001 = 1
```

2.8 Command "BREPort"

Usage: **BREPort <(string bddName)> [-ADDRess] [-REFcount]**
[-File <(string fileName)>]

Description: Report the cone of BDDs from the <bddName> (all the way to the const-1 node). If the BddNodeV of <bddName> does not exist, an error will be issued. Otherwise, it should be reported in the plain text format as follows:

```
[3](+) 0x8c67a68 (1)
  [2](+) 0x8c67a18 (3)
    [0](+) 0x8c679e0 (8)
    [1](+) 0x8c67af0 (5)
      [0](+) 0x8c679e0 (8)
      [0](-) 0x8c679e0 (8)
  [1](-) 0x8c67af0 (5)(*)
```

where "[3]" means the BDD input (support) level, "(+)" means it is a PosEdge (no bubble), "0x8c67a68" is the address of the BddNodeVInt, and “(1)” is the reference count. Please note that the children of a BddNodeV are printed in the left (positive cofactor) then right (negative cofactor) order, and are aligned in “2-space indent” from the parent node. If a child node has been printed before (e.g. “[1](-) 0x8c67af0 (5)”), an “(*)” is printed at the end of the line.

The options [-ADDRess] and [-REFcount] control whether the “address” and “reference count” fields are printed or not. The defaults for both of them are OFF. The option [-File <(string filename)>] directs the output to file “fileName”.

2.9 Command “BDRAW”

Usage: **BDRAW <(string bddName)> <(string fileName)>**

Description: Draw the cone of BDDs from the <bddName> into a “DOT” format file “<fileName>”. Please refer to the “Graphviz - Graph Visualization Software : http://www.graphviz.org/Documentation.php” for details.

2.10 Command “BCONstruct”

Usage: **BCONstruct <-Netid (int netId) | -Output (int outputIndex) | -All>**

Description: Build BDD for a specific gate or the entire circuit. If the specified gate does not exist in the circuit, an error will be issued. If the BDD of the gate or the circuit has been built, do nothing. The <netId, BddNodeV> associations will be recorded.

2.11 Command “BSETOrder”

Usage: **BSETOrder < -File | -RFile >**

Description: Associate the BDD input variable order with the circuit inputs. If the design has not yet been read in, an error will be issued. The options “-File | -RFile” defines the input order to be “(-File) the original input order in the design file”, or “(-RFile) the reverse of the original input order in the design file”. Note that PIs will have lower BDD levels than DFFs. Please also note that every PI/PPI is associated with a BDD name that is as its name in the circuit, and every next-state PPI is associated with a BDD name that is as its name in the circuit with the suffix “_ns” (e.g. “state[0]” to “state[0]_ns”).

The BddNodeVs for the PIs and DFFs will be associated with the specified BddNodeVs in “BddMgrV:_supports” (See Section 3 for more details). Please note that the order cannot be re-associated by this command without resetting the BDD manager. An error will be issued if trying to re-associating the input order with different options, or if the number of circuit inputs is greater than the number of supports in BDD manager.

2.11 Command “PINITialstate”

Usage: **PINITialstate [(string varName)]**

Description: Set the initial state BDD. The initial state in our program will be all 0's. If the optional parameter (string varName) is specified, the BddNodeV of the initial state will be associated to the "varName". It can be used in later reporting (by the "BREPort" command). If there has already been a BddNodeV associated with the "varName", an error will be issued.

Error will also be issued for the following situations:

1. Circuit is not yet constructed, flattened, or bit-blasted (by `BLAST NTK` command).
2. The BDD of the circuit has not yet been built
3. The "varName" is not a legal variable name

Examples:

```
bdd> pinit 3        // cannot associate the init state BDD to a number
Error: Illegal option (3)
```

2.12 Command "PTRansrelation"

Usage: **PTRansrelation [<(string triName)> |**
<(string triName) (string trName)>]

Description: Construct the BDD for the transition relationship of the circuit. If one optional parameter is specified, it will be associated with the BddNodeV of the transition relationship with the PI variables (i.e. without existentially quantified). If the second parameter is specified, it will be associated with the BddNodeV of the transition relationship with the PI variables existentially quantified. They can be used for later reporting (by the "BREPort" command). Both names must be legal variable names. If there has already been a BddNodeV associated with the any of the "varName", an error will be issued.

Examples:

```
bdd> ptr tri          // tri = TR(Y, X, I)
bdd> ptr aa bb        // aa = TR(Y, X, I);  bb = TR(Y, X)
```

2.13 Command "PIMAGe"

Usage: **PIMAGe [-Next <(int numTimeframes)>] [(string varName)]**

Description: Perform image computations for "numTimeframes" timeframes. If the option "-Next <(int numTimeframes)>" is not specified, perform 1-step image computation. If the fixed point has been reached, the reachability analysis will stop and print out a notice.

The optional parameter "varName" is to associate with the BddNodeV of the final set of reachable states. It can be used in later reporting. If there has already been a BddNodeV associated with the "varName", it will be overwritten without a warning.

Examples:

```
bdd> pimage kk            // perform 1-step image computation
                          // store the BddNodeV with name "kk"
bdd> pimage -n 10 img     // perform 10-step image computation;
                          // store the BddNodeV with name "img"
Fixed point is reached ( time : 7 )
                // Fixed point is reached; image 8-10 will not be computed
```

2.14 Command PCHECKProperty

Usage: **PCHECKProperty < -Netid <netId> | -Output <outputIndex> >**

Description: Check the monitor for "netId" or "outputIndex". If the corresponding BddNodeV is not yet constructed, an error will be issued. There can be three kinds of proof results:

1. Fixed point is reached and no bug is found. Printing ---
```
Monitor "varName" is safe.
```
2. No bug is found but the fixed point has not yet reached. Printing ---
```
Monitor "varName" is safe up to time 10.
```
3. A bug is found. Printing with a counter-example ---
```
Monitor "varName" is violated.
Counter Example:
0: 0000
1: 0001
2: 1010
```

3. Reference code

The reference code is compressed as "hw3.tgz". Its structure is similar to the one in HW #1. The BDD-related codes are similar to those in HW#1, except for the command interface part.

All your TODO's are, as in suggested order, in the file "ntk/v3NtkBdd.cpp" (for circuit-to-BDD construction) and the package "prove" (for the BDD-based verification algorithm). However, please feel free to modify the BDD, prove, or any packages if needed (e.g. implement a "restrict()" operation).

To install the required packages and compile the codes for the first time, simply type "**./INSTALL**" and the executable "**bddv**" should be created. Please be sure that all the scripts (i.e. INSTALL and engine/*.script, etc) are in executable mode before calling "**./INSTALL**". Otherwise, use "chmod +x" to fix it. To recompile the codes after you write something, simply type “make” and the program will be incrementally recompiled.

3.1 `Class BddMgrV`

The BDD manager (`class BddMgrV`) defined in “bddMgrV.h” has the following data members:

i. `vector<BddNodeV> _supports;`

   Please refer to the definition in the lecture note. It is initialized in BddMgrV’s constructor. Its index is the level of BddNodeV, where level 0 is the constant one, level 1 is the lowest node.

   When building BDDs for a sequential circuit, the BddNodeVs of PIs, current states (DFFs) and next states (DFFs) will be built from the _supports. The PI, current state and next state BDDs have the lower, middle and the higher levels, respectively. Their orders are determined by the command “`BSETOrder`” (see Section 2). For word-level signal, its LSB (i.e. index = 0) and MSB (i.e. index = max) signal will be associated with the lowest or highest level of supports for the “-File” or “-RFile” option, respectively.

ii. `BddHash   _uniqueTable;`

   `BddCache  _computedTable;`

   Please refer to the definition in the lecture note.

iii. `BddArr    _bddArr;`

   A dynamic array of BddNodeVs. Used to associate “netIDs” to their corresponding BddNodeVs. This array should be constructed during the circuit-to-BDD construction.

iv. `BddMap    _bddMap;`

   A map of <string, BddNodeV>. For the BddNodeVs that are associated with names (string), they are stored here.

iv. `bool             _isFixed;`

   `BddNodeV         _initState;`

   `BddNodeV         _tr;`

   `BddNodeV         _tri;`

   `vector<BddNodeV> _reachStates;`

   These data members are for the reachability analysis. “`_isFixed`” denotes whether the current reachability analysis has reached fixed point or not, and “`_initState`” is the BddNodeV of the initial state, set by the “`PInitialstate`” command (See Section 2). The BddNodeVs for the transition relationship, with or without PIs existentially quantified, are recorded in "`_tr`" and "`_tri`", respectively. "`_reachStates`" stores the BddNodeVs for the set of reachable states, accessed by the timeframe index where 0 refers to the initial state. If the fixed point is reached, no image computation will be performed and thus the size of "`_reachStates`" will not increase any more.

3.2 `Class BddNodeVInt / class BddNodeV`

There are two classes about BDD nodes: "`class BddNodeVInt`" and "`class BddNodeV`". The class "`BddNodeVInt`" is an internal BDD node implementation. It acts as the nodes on the (BDD) graph and handles all the detailed BDD node operations. However, it is a private class and can only be accessed by `class BddNodeV`. On the other hand, the class "`BddNodeV`" defines all the interface functions that you will need to use to operate on BDD nodes. Basically it is a wrapper class for `BddNodeVInt`. It contains the "`BddNodeVInt`" pointer address as well as the complemented edge information. We have overloaded most of the operators and you can just use it as an object variable (e.g. `BddNodeV a, b, c; a = b && c`). With the object oriented programming styles, all the reference count and complemented edge issues have been taken care by the implementation in `BddNodeVInt`. As a BDD package user, you don't need to worry about it.

Please refer to the C++ header files for the available member functions of `class BddNodeV`. Here're some important ones in `class BddNodeV`:

i. `const BddNodeV& getLeft() const;`

   `const BddNodeV& getRight() const;`

   Get the left / right child `BddNodeV`.

ii. `BddNodeV getLeftCofactor(unsigned i) const;`

   `BddNodeV getRightCofactor(unsigned i) const;`

   Get the left / right (i.e. positive / negative) cofactor of `BddNodeV`. It will be the inverse of `getLeft() / getRight()` if this `BddNodeV` is complemented.

iii. `BddNodeV exist(unsigned l) const;`

   Perform the existential quantification for the BDD variable of level '`l`'.

iv. `BddNodeV nodeMove(unsigned fromLevel, unsigned toLevel, bool& isMoved) const;`

   In the cone of this `BddNodeV`, move the `BddNodeVs` with levels ≥ fromLevel to levels ≥ toLevel. Note that if (fromLevel > toLevel), the nodes are moved down, and if (fromLevel < toLevel), the nodes are moved up.

If moving up, there should be no node >= toLevel in the beginning. That is, toLevel > level of this node. After the move, there will be no BDD nodes between [fromLevel, toLevel).

If moving down, after the move, there will be no BDD node >= from Level.

In eithr case, before the move, need to make sure:

* (thisLevel - fromLevel) < absolute value of (fromLevel - toLevel)

* There is no node < fromLevel (except for the terminal node).

If any of the above is violated, there will be no action and the parameter isMoved will be set to false. Otherwise, it will be true.

v. `BddNodeV getCube(size_t ith=0) const;`

Get the i^th cube of the BddNodeV. It will traverse from the left (positive) cofactor first. The returned cube will also be stored in a BddNodeV.

vi. `size_t countCube() const;`

Count the number of cubes (i.e. paths to terminal `BddNodeV::_one`) of this `BddNodeV`.

4. What you should do?

You are encouraged to follow the steps below for this homework assignment:

1. Read the specification carefully and make sure you understand the requirements.

2. Play with the BDD calculator. Try to trace the code from BddCmd.cpp. Make sure you understand how to construct BDDs from an expression, how a BddNodeV is associated with a name string, and how a BddNodeV is printed by the “BREPort” command, etc.

3. Run the testcases with the dofiles in the “tests” directory. Understand the command usage. A reference program “bddv-ref” is provided in the “ref” directory (for Linux platform).

4. Understand the BDD test program under “src/bdd/test” (Note: To compile this test program, you need to compile “bddv” to get “libbdd.a” first. Type “make” in the “hw3” directory first). You can use this simple example to understand how to use the BDD package and the basic idea of its OOP design.

5. Study the header files in the “bdd” package and understand the requirements of TODO’s in the files “v3NtkBdd.cpp” and “proveBdd.cpp” in the “prove” package.

6. **[Build BDDs from circuit]** Study the code in the "ntk" package. The TODOs are in the file "v3NtkBdd.cpp", where "void V3Ntk::buildNtkBdd()" is to construct BDDs for the whole circuit, and "void V3Ntk::buildBdd(const V3NetId& netId)" is to build a BDD for a net with netId. You need to understand how the network is traversed and how the fanin cone is collected in an array, how the fanins of a gate can be accessed (e.g. by "V3Ntk:: getInputNetId"), how a BddNodeV is constructed (i.e. by BddNodeV's operators), and how to access and store BddNodeVs to BddMgrV::_bddArr" for netId-to-BddNodeV associations.

7. **[Construct BDDs for initial state and transition relationship]** Implement the functions for `PINITialstate` and `PTRansrelation` commands. Use the provided small testcases and reference program to verify your result.

8. **[Reachability Analysis Algorithm]** Implement the reachability analysis and property checking functions for `PIMAGe` and `PCHECKProperty` commands. Verify your implementation with the reference program.

9. **[Bdd-Based Verification]** Write/Define **at least 3 monitors** for the BUGGY "vending machine" design in HW #1 (you can reuse the monitors from HW#1). Prove them with yours and the reference programs. Compare the results on correctness and runtime, memory usage, etc. If any of the properties is false, simulate with the sequential solver of HW#1 to verify the correctness.

   Please note that the monitors must be some POs in the design. You can use "assign" in Verilog and create extra POs to specify the expression you want to prove. Of course, you can also implement a sequential block to specify the monitored signals.

10. **[Advanced Techniques]** Please note that it is very likely that the BDD engine will abort due to the memory explosion problem. Please refer to the lecture notes to see if the algorithms can be improved by any of the advanced techniques. Besides, you can also try to "abstract" the design in order to reduce the proof complexity. However, please make sure your abstraction of the design can lead to affirmative conclusion to the proofs of the original design.

5. What you should turn in

1. Your modified source code. Please note that we will run plagiarism checking on your source code. Any act of plagiarism will lead to severe deduction of your point and we will report this to the departmental office.

2. Your original and abstracted "vending.v" (rename it properly) and some dofiles you used to test your program and verify your design. Put them in "tests/" directory. We will use them to test your program.

3. A report file named "<yourID>_report.pdf" in PDF format. Please describe: (i) Your implementation (make it brief), (ii) The assertions you add and their

meanings, (iii) Your verification results, (iv) The comparison with the ref program, and (v) the advanced techniques and/or abstraction of the design. Please place the report in the root directory (i.e. the same directory as "Makefile") of the homework.

IMPORTANT: Please type "make clean" and remove the unnecessary files (e.g. files in "engine" directory, core dumps, *.o) before submission. Remember to rename the directory to "<yourID>_hw3" before compressing it. Besides, you should compress it under Linux workstation by the command:

```
tar zcvf <yourID>_hw3.tgz <yourID>_hw3
```

6. Grading

We will test your submitted program, your assertions (in your original and abstracted "vending.v"), and your dofiles, and compare its outputs with those of our reference program. Both correctness and runtime will be evaluated. The report is important. We will also grade your score based on the report.