Quick Look under the Hood of ABC

A Programmer's Manual

December 25, 2006

Network

ABC is similar to SIS/MVSIS in that it processes the design by applying a sequence of transformations to *the current network*, which is stored in memory during the runtime. Initially, the current network is created by reading the design specification from file. The current network is modified step by step by applying individual synthesis commands and can be written out in the end for future use. (There is an option of making ABC work with several synthesis snapshots of the same design but we are not discussing it this short tutorial.) (There is also an option of compiling ABC as a static library and directly calling individual network transformation routines from the user's software.)

This paper introduces the ABC internal data representation and clarifies its differences compared to those of SIS/MVSIS. An ABC network can be characterized by specifying its type and its functionality representation. The following types of the network are supported: a netlist, a logic network, and an AIG. The functions of nodes in the network can be represented using SOPs, BDDs, two-input AND-gates, and gates from a standard-cell library.

Table 1 summarizes the currently supported combinations of types and functionality of networks in ABC.

Type \ Functionality	SOP	BDD	AND2	Gates
Netlist	x		x	x
Logic network	x	x	x	x
AIG			x	

Table 1. Supported type/functionality combinations.

Netlist

The programmer who intends to use ABC for programming logic synthesis application may skip the description of the netlist and concentrate on using logic networks and AIGs. However, the programmer who intends to build a parser for a new type of input file may need to learn about the netlist representation because the netlist has to be constructed while parsing the input file. (There is an exception to this rule. If the goal is to integrate ABC with another tool, such as SIS, which does not represent nets explicitly, the programmer can bypass construction of the netlist and directly construct a logic network in ABC.)

Netlist is the basic “raw” network representation, which is in one-to-one correspondence with the design specification in the input file. Netlist is composed of nets, logic nodes, latches, and PI/PO terminals. Each net (as well as PI/PO terminal) has a unique name. The nodes and latches are identifiable by the names of the nets they are driving. Only single-output nodes and latches are currently supported. Each PI terminal, node, and latch drives a net. Each latch, PO terminal, and non-constant node (node with one fanin or more) is driven by a net. A net can be simultaneously driving a latch, a node, and a PO terminal but it cannot be driven by more one object. For example, a net cannot be driven by a node and a latch. In a netlist, nets cannot be connected to other nets. Nets should always be connected to other objects (nodes, latches, and PI/PO terminals). The non-net objects cannot be connected to other non-net objects, but only to nets.

The local functions of the nodes is represented using SOPs or AIGs, in the case of a technology independent netlist, or using gates from a standard cell library, in the case of the mapped netlist. The netlist is currently used only for reading/writing designs from the input file while the majority of logic optimizations is performed more efficiently using a representation called a *logic network*. In the current release, the input file is always parsed into a netlist, which is then automatically converted into a logic network. Similarly, the output file is created from a netlist, which is derived on-the-fly from the current network represented using a logic network or an AIG. The procedures applied to the currently network on the command line assume a logic network or an AIG.

A netlist is constructed using APIs of *src/base/abc* package. A new netlist is started by making a call to *Abc_NtkAlloc*. The PI/PO terminals are created by calling *Abc_NtkCreatePi* and *Abc_NtkCreatePo*. Nets are created or retrieved using their name by calling *Abc_NtkFindOrCreateNet*. The nodes and latches are created by *Abc_NtkCreateNode* and *Abc_NtkCreateLatch*, respectively. All components of the network (nodes/nets/latches/terminals) are called objects. The API used to interconnect objects is *Abc_ObjAddFanin*, which takes two objects and establishes the fanin/fanout relationship between them. The objects can be added to the network in any order. The fanin/fanout relationship can be also established in any order. In the end, the procedure *Abc_NtkFinalizeRead* should be called, which finalizes the netlist. The constructed netlist should be checked using *Abc_NtkCheck*, which makes sure that a valid netlist is created and reports the mismatches, if any.

Logic network

A logic network is essentially a netlist, from which the nets have been removed. This is the way the current network is represented in SIS/MVSIS. In ABC, the default representation is an AIG, but a logic network is one of the valid intermediate network representations. Duplication of logic networks in ABC is similar to that in SIS. Objects (nodes, latches, PI/PO terminals) can be added in any order. When duplicating a network or creating a network similar to the already existent one, it is convenient to call procedure *Abc_NtkStartFrom*.

It should be noted that only the PI/PO/latch/latch-input/latch-output names are saved in the logic network data structure, while all the internal node names are discarded. The reason for disposing of the internal node names is that ABC is meant for deep-synthesis using AIGs, and it is hard to preserve AIG node names while they are transformed during AIG manipulations, such as rewriting. Even though the internal names are currently not stored, command *dress* can recover some of them after synthesis. This command is based on equivalence checking of the final and the original network, and transferring the names from the original network to the functionally identical nodes in the final network.

In a logic network, nodes can directly point to other nodes. Iterators over the fanins/fanouts of a node are available: *Abc_ObjForEachFanin* and *Abc_ObjForEachFanout*. The PI/PO terminals are connected directly to the nodes. A PO terminal can connect directly to a PI terminal, if they have the same name and functionality. The PI terminal may have many fanouts but cannot have fanins. A PO terminal has only one fanin and cannot have fanouts. A terminal is not a logic node and it does not have a logic function. The iterator through the nodes *Abc_NtkForEachNode* does not iterate over the terminals. By default, the pointers to the terminals are not collected along with the pointers to the internal nodes by DFS traversal procedures. In a logic network, the pointers to the PI/PO terminals are stored in the corresponding arrays. The iterators *Abc_NtkForEachPi* and *Abc_NtkForEachPo* iterate over these nodes. The iterators *Abc_NtkForEachCi* and *Abc_NtkForEachCo* iterate over PIs and latch outputs, and POs and latch inputs, respectively.

The functionality of nodes in the logic network can be represented using SOPs, BDDs, or AIGs. Only one type of representation can be used for all nodes of the network. Converting between the representation can be done simultaneously for all nodes of the network by calling APIs, such as *Abc_NtkLogicToSop*. The manager used to represent the functionality of the nodes are stored in the data member of the network *pNtk->pManFunc*. Additionally, a logic network can be mapped. In this case, each node is annotated with a matching gate from a standard cell library stored at *pNtk->pManFunc*. The LUTs after FPGA mapping of the current network is represented using logic nodes whose functions are BDDs. The procedure for making the nodes minimum base *Abc_NtkMinimumBase*, which removes duplicated fanins if present, can be called after constructing a new network. This procedure uses BDDs for minimizing the support set of the functions of the nodes.

Adding, removing, and duplicating nodes in a logic network in ABC is similar to how these operations work in SIS. Procedures *Abc_ObjPatchFanin* and *Abc_ObjTransferFanout* are analogous to their counterparts in SIS. Procedures for collapsing several nodes in the logic networks are currently not available. The reason why logic network procedures are not fully developed in ABC is because the old-fashioned SIS-like manipulation of logic networks in ABC has been to some extent replaced by the manipulation of AIGs, which makes these operations unnecessary.

Visualization of small networks (up to 100 nodes) can be performed using command *show,* or by calling the corresponding internal procedure.

AIG

And-Inverter Graph (AIG) is the primary internal representation of the current network in ABC. It is the only network type that is accepted by the technology mappers and the majority of other commands, such as *balance*, *collapse*, *renode, rewrite, refactor, retime*.

AIG is a specialized type of the ABC network, in which each node is a two-input AND gate and each fanin/fanout edge has an optional complemented attribute indicating the inverter on that edge. Because the local function is the same for all nodes, it is not represented in the node data structure (*pNode->pData* is NULL for AIGs without choice nodes). During construction AIG is compacted on-the-fly using one-level structural hashing, which requires that, for each ordered pair of edges (possibly with complemented attributes), there is at most one node having these edges as fanins. The structural hashing ensures that

- for each AND node, there are no other ANDs with the same children
- the constants are propagated (and can appear as fanins of the COs only)
- there is no single-input nodes (inverters/buffers)

In addition to these requirements, several other properties are kept invariant during AIG manipulation in order to speed up fast processing of AIGs:

- there are no dangling nodes (the nodes without fanout)
- the AND nodes are stored in the topological order in the array of objects
- the constant 1 node has always number 0 in the array of objects (nodes/PIs/POs/latches)
- the level of each AND gate reflects the levels of its fanins
- the EXOR-status of all nodes is up-to-date (the status bit is set to 1, if the AND is the root of an EXOR of two other nodes)

Because of the above restrictions on the AIGs, manipulating them directly is trickier than manipulating logic networks. For example, it is not possible to duplicate the node because two nodes with the same fanins are not allowed by structural hashing. It is not possible to collapse the nodes because the only node type allows is a two-input AND. It is not possible to add explicit buffers or inverters. The following operations, which can be performed on an AIG, ensure that the AIGs remains structurally hashed with other invariants preserved:

- building new nodes (*Abc_AigAnd*)
- performing elementary Boolean operations (*Abc_AigOr, Abc_AigXor*, etc)
- replacing one node by another (*Abc_AigReplace*)
- propagating constants (*Abc_AigReplace* called with one of the argument being the constant)

AIGs used in the lossless synthesis procedures may contain choice nodes. This happens, for example, during lossless logic synthesis when several network snapshots are FRAIGed together. This is done to mitigate structural bias present in any particular logic

structure and thereby increase the quality of technology mapping. A choice node is an equivalence class of AIG nodes belonging to the same network and having the same Boolean function up to complementation. The choices are represented by linking together the AIG nodes belonging to the same equivalence class using *pNode->pData* pointer of the node data structure, and setting the phase bit in each node, which can be used to check phase difference between the nodes in the equivalence class.

The transformation from an AIG into a logic network involves a non-trivial step performed by *Abc_NtkLogicMakeSimpleCos*. This procedure ensures that each CO is driven by a unique internal node and that the CO-to-driver edge does not have a complemented attribute. Ensuring this property is important for efficient processing of large designs.

Visualization of small AIGs (up to 100 nodes) can be performed using command *show,* or by calling the corresponding internal procedure.

Nodes

Internal components of networks of all types are represented using one data structure, called *object*. An object has a type field indicating whether it is a net, a logic node, a latch, or a PI/PO terminal. All objects in a network have unique *object IDs*, 32-bit integer identifiers assigned during construction of the network. The actual value of the ID assigned to an object is not important as long as the ID is unique. The object ID can be used to retrieve the pointer to the object.

Nodes can have zero or more fanouts, such as another node, a latch, or a PO terminal. A node may have zero or more fanins. Constant nodes have zero fanins. Inverters and buffer have one fanin. In general, there is no limit on the number of fanins and fanouts of a node. The fanins can be duplicated. Duplicated fanins can be removed using procedure *Abc_NtkSweep*, which can also be invoked on the command line as *sweep*. Programmably, duplicated fanins can be removed by calling *Abc_NtkMinimumBase*.

In an AIG, there is exactly one constant node (this constant 1 object has ID = 0 and is not considered a node) with no fanins and all other nodes have the functionality of two-input AND gate and exactly two fanins. Inverters and buffers are not allowed as separate single-input nodes in an AIG. These rules are enforced by structural hashing, which is always performed when adding new nodes to an AIG, or replacing an existent node.

A node in a logic network always has a logic function assigned to *pNode->pData* field of the node. The node function is a completely specified Boolean function. Incomplete specification and non-determinism are not allowed in the current version of ABC. The function of a node can be represented in a number of ways: as an SOP, as a BDD, and as a gate from a library. The network has a functionality manager, whose type depends on the functionality representation of the nodes.

In an AIG, the logic function of a node is not represented because, by default, any non-constant node has a function of a two-input AND gate. Instead, when the network is an AIG with choices, the data pointer used for storing the logic function of a node, *pData*, is reused for representing choice nodes (it stores the pointer to the next entry in the linked list of functionally equivalent nodes).

When the functionality of a logic network is represented using SOPs, the network representation is similar to the one used in SIS. The difference is that SOP representation in ABC is simpler: it is a zero-terminated C-string specifying the SOP as it would appear in a BLIF file. For example, an SOP of a three-input AND is "111 1\n", while an SOP of a MUX is "11- 1\n0-1 1\n", where '\n' is single new line character with integer code 10, as in the C programming language. This simple SOP representation is adopted because it is works well in most cases and easy to follow when working with the code. For operations, such as factoring, whose performance critically depends on the efficiency of the SOP representation, the SOPs are converted into bit-strings, similar to the ESPRESSO *pset_family* data structure.

When the functionality of a logic network is represented using BDDs, the network is similar to that in BDS. In this case, each node has a pointer to its local BDD expressed using the topmost *k* variables of the local BDD manager, where *k* is the number of fanins of the node. The *i*-th variable of the BDD manager corresponds to the *i*-th fanin. If the local function does not depend on the *i*-th fanin, the *i*-th variable is not used in the local BDDs. The removal of duplicated fanins and inessential variables, which reduces the fanin space and remaps the BDD variables to the top of the local BDD manager to form a contiguous range, is performed by the procedure *Abc_NtkMakeMinBase*. The local BDD manager is never reordered.

Similar rules hold for representing local functions using AIGs.

Some operations performed on the network require a particular functionality representation of the nodes. For example, disjoint-support decomposition requires the BDD representation of the node functions, while extracting common logic using *fast_extract* requires that the nodes have SOPs. The transformation between the functional representations is performed by a procedure applied to all nodes in the network. (This procedure can be called on the command line using commands *sop, bdd,* or *aig*). The functionality changing procedures do not modify the network structure. They replace the functionality manager of the network and update the pointers to the functionality representation of the nodes.

Latches

The latches currently supported are generic D-latches with one input, one output, and no reset signal, which belong to the same clock domain. After power-up each latch is

assumed to be in its initial state, which is 0, 1, or unknown. Support of different latch/flip-flop types and memory elements belonging to multiple clock domains may be added in the future releases.

Each time a latch is created in ABC, two single-input single-output nodes are created. They belong to the types “block input” and “block output”. These nodes are called a latch input and a latch output. The latch input/output is added to the array of CIs/COs. The iterators over CIs/COs iterate through these additional nodes but do not iterate through the latches. This design allow for a clean separation of logic into combinational part (contained between CIs and COs) and latches.

Edges

An edge in a logic network is a connection between two objects (for example, two nodes, or a node and a latch). The nodes connected by an edge are in the fanin/fanout relationship. In the case of a netlist or a logic network represented using SOPs, an edge contains only the ID of a fanin object. In an AIG, the edge contains an optional complemented attribute. In a sequential AIG, an edge contains additional information about the number of latches and their initial values. In all cases, the ID of a node is a 32-bit integer number.

Iterators

Iterators through different types of objects belonging to the same network are provided. The iterator through nodes iterates through the logic nodes, and does not iterate through latches and PI/PO terminals. Iterators through PIs/POs and CIs/COs are also available. A CI/CO iterator goes first through the PIs/POs, then through the LOs/LIs.

Abbreviations and glossary

ABC – The name of a new logic synthesis system
AIG – And-Inverter Graph, a Boolean network composed of two-input ANDs and inverters
BLIF – Berkeley Logic Interchange Format, a format traditionally used by SIS, VIS, and MVSIS to represent logic networks
BDD – Binary Decision Diagram, a canonical graph-based representation of Boolean functions
CI – Primary Input and Latch Outputs

CO – Primary Output and Latch Inputs
FPGA – Field Programmable Gate Array
FRAIG – (a) Functionally Reduced AIG and (b) AIG package with on-the-fly detection of functionally equivalent AIG nodes
FRAIGing – Transforming an AIG into a Functionally Reduced AIG
LI – Latch Input
LO – Latch Output
LUT – Look-Up Table, a programmable logic component that can implement an arbitrary Boolean function up to a fixed number of inputs
PI – Primary Input
PO – Primary Output
SAT – Boolean satisfiability
SOP – Sum-Of-Products, a non-canonical representation of Boolean functions
TFI – Transitive Fanin
TFO – Transitive Fanout