High Capacity and Automatic Functional Extraction Tool for Industrial VLSI Circuit Designs

Sasha Novakovsky
Design Technology
Intel Corporation
+972-4-8565810
nsasha@intel.com

Shy Shyman
Design Technology
Intel Corporation
+972-4-8565037
sshyman@intel.com

Ziyad Hanna
Design Technology
Intel Corporation
+972-4-8565303
zhanna@intel.com

Abstract

In this paper we present an advanced functional extraction tool for automatic generation of high-level RTL from switch-level circuit netlist representation. The tool is called FEV-Extract and is part of a comprehensive Formal Equivalence Verification (FEV) system developed at Intel to verify modern microprocessor designs. FEV-Extract employs a powerful hierarchical analysis procedure, and advanced and generic algorithms for automatic recognition of logical primitives, to cope with variety of circuit design styles and their complexity. Logic equations are then extracted to generate a behavioral RTL model described in industrial standard HDL languages, to be used in the formal equivalence verification, logic simulation, synthesis and testability flows.

Categories and Subject Descriptors
D.3 [**VERIFICATION, MODELING AND SIMULATION**]: Formal verification techniques. Switch, logic and high-level simulation, design validation, HW/SW co-simulation, combinational and sequential equivalence checking. Model checking. Theorem proving.

General Terms
Algorithms for design verification.

Keywords
Switch Level Analysis, Functional Abstraction, Formal Equivalence Verification (FEV), Design For Testability (DFT), Synthesis, Logic simulation, Binary Decision Diagrams (BDDs), Satisfiability procedures, Hardware Description Languages (HDL).

1. Introduction

Advances in VLSI technology have made possible the implementation of large and increasingly complex systems in a single integrated chip. Rapid verification of the VLSI implementation and finding logical bugs are amongst the most challenging and pressing problems in the modern circuit design projects. Static verification or formal verification methods are very promising in proving the correctness of circuit implementation compared to its RTL hardware description. The first step in the verification is to analyze the circuit implementation and extract its logic behavior. During the extraction process, implementation issues and "unsafe" structures can be detected at early stage of the verification process.

This paper describes how the functional extraction flow is efficiently performed in FEV-Extract to address the challenging design and verification needs.

In the next section we provide a review of the related work and the advantages in FEV-Extract compared to other tools.

2. Related Work

Functional abstraction or extraction methods have been under research for many years. Each method tries to address certain aspects of the problem. However, to our knowledge, there is no method that is generic and powerful enough to address the challenging needs of modern microprocessor designs.

In [2] and [3], the authors proposed a BDD based method to extract the functional behavior from a transistor circuit by building a transition relation of the "micro-latch" pull-up and pull-down functions computed for each of the storage nodes in the circuit. Though this method is considered generic for extracting the finite state machine from a circuit representation, it is limited in capacity because it builds a BDD for the entire transition relation, which naturally may blow up.

Pattern matching methods [1], are based on graph isomorphism algorithms to identify circuit configurations based on a pre-defined set of circuit patterns such as Domino, latches, CMOS gates etc. This method is limited because it does not cover all the possible configurations in the circuit and therefore, the user is requested to either enrich the pattern set, or to assist the extractor by adding additional hints and attributes to help identifying the unresolved circuit configurations. The advantage of this method is its simplicity, however it is inefficient and not "safe" because it usually requires user intervention to help understanding the circuit logic behavior. The authors of [4] combined the algorithmic and pattern matching approaches by employing algorithms for analyzing combinational circuit

configurations, while leaving sequential circuits to be identified using the classical pattern matching methods.

R.E Bryant (CMU) established a strong mathematical basis and sound algorithms [5] to analyze MOS circuits using graph algorithms. Bryant's method extracts the functional behavior of channel connected sub networks – CCSN, using Gaussian elimination procedure. However, this method focuses on unit delay analysis and is not directly applicable to analyzing and extracting RTL models. In a later work [6], Bryant tried to address the problem of sneak paths and false glitches by using simplification and quaternary logic but the method is still targeted to unit delay modeling and massive intervention is required from the user.

Very few of the methods offered in the literature provide hierarchical analysis. Therefore, the majority of the methods suffer from capacity limitations, which are usually resolved by partitioning the circuit, thereby introducing productivity and correctness issues.

FEV-Extract provides a significant leap in the functional extraction domain compared to the other methods listed above. It is fully automated, compared to [4], and provides high capacity for the modern VLSI design. The tool's main objective is to analyze VLSI circuit automatically, in a fast and accurate manner, and to extract its RTL zero delay representation. Because the tool is mainly targeted for formal equivalence verification tasks, with a constant need to analyze complex and large circuits, overcoming capacity limitations of the above approaches is a must. Other flows that utilize FEV-Extract capabilities are static timing analysis, power estimation, and fault grading. In this paper we will focus on zero delay modeling for formal verification and simulation needs.

In section 4, we describe FEV-Extract flow. In sections 5 and 6, we elaborate on the novel techniques employed in FEV-Extract, focusing on symbolic analysis methods for extracting the functional behavior of combinational networks, on loop analysis algorithms and the automatic identification of logical primitives. Hierarchical analysis described in section 7 provides the boost in extraction performance and capacity. In section 8, we present some experimental results, which clearly demonstrate a significant advantage of FEV-Extract over other methods. We conclude in section 9 with recommendations for future work. The conclusions appear in section 10.

3. Preliminaries

In this section we define some concepts that will be used in the following sections.

Template model is an in-memory representation of the sub hierarchies of a model without instantiations.

DAG - Directed Acyclic Graph – is a data structure for representing Boolean formulas.

Switch level model consists of a set of nodes and a set of transistors. Each transistor is assumed bi-directional. Each node may be classified as *regular*, *input*, or *storage node*. The term *input node* refers to a signal coming from the environment of the circuit. The term *storage node* denotes a node that can store its value when not driven, and can share charge with other storage nodes.

A storage node is assigned with *size* from the set {1,2…k}, which denotes the node's ability to store charge compared to other storage nodes. An input node is assigned a size w > k. A regular node is assigned a size = 0.

Each transistor is assigned with a *strength* s from set {k+1,k+2,…,w-1}, i.e. transistors are always stronger then storage nodes. This attribute represents the transistor's conductance relative to other transistors. A path *p* is a directed path originating at Root(*p*) and terminating at Dest(*p*), and consists of a set of transistors Trans(*p*). The strength *s* of a path *p*, denoted |*p*|, is defined as:

$$|p| = \min\left(Size\left[Root(p)\right], \min_{t \in Trans(p)} Strength(t) \right).$$

A path is termed *definite* if no transistor in Trans(*p*) is in state X. A path *p* is termed *unblocked* if there are no prefix path *p'* of *p* and a definite path *q* such that Dest(*p'*) = Dest(*q*) and |*q*| > |*p'*|.

Switch level models can be partitioned to sub networks called *CCSN-s (Channel Connected Sub Network)*

The *state* of node *n* is represented as two formulas *n.h* and *n.l*. *n.h* represents the condition under which there is pull- up path to the node. *n.l* represents the conditions under which there is a pull-down path to the node. Then we can define the state of node *n* using the following table:

Value	n.h	n.l
Z	0	0
0	0	1
1	1	0
X	1	1

Table 1. Dual Rail state encoding

4. FEV-Extract Flow

The FEV-extract flow can be summarized as follows:

- Model build and analysis - the hierarchical template model is read and built in memory.
- Then we apply a decision procedure to determine which instances have to be smashed and which can be analyzed separately on the template basis.

For each analyzed template we apply the following steps:

- Partition the model into CCSN-s.
- Symbolically analyze each CCSN and derive dual rail formulas for each storage node (mainly CCSN outputs).
- Perform zero delay single rail transformation.
- Simplify the equations based on user given and internally derived relations.

- Find loops in the model and analyze them to derive state elements and dynamic logics.
- Generate zero-delay output model.

5. Symbolic Analysis of CMOS Circuits

In the following sections we describe the main innovations of our work with respect to [5] and [6].

5.1 Partial Strength Order

During the analysis of each CCSN, two systems of equations are generated for each strength *s*. The first system, called "clear", represents conditions under which no node is the destination of a definite path of strength *s*. The second system, termed "state", denotes the combined effects of all unblocked paths with strength greater than or equal to *s*. You may recall that a path *p1* 'wins' over path *p2* if $|p1| > |p2|$. The method is referred to as *Total Strength Ordering*.

Using total strength order requires user intervention to tune transistor strengths in order to solve contentions. In today's VLSI design style, when a large portion of a circuit is implemented using asynchronous design and ratio logic (rather than complementary logic), this is a tedious task, and can lead to inaccurate modeling. FEV-Extract overcomes this issue by using *Partial Strength Order* based on the transistor's physical strength. The strength for each transistor is calculated based on its physical sizes (width and length), with proper consideration for fabrication process parameters (such as P/N ratio). A transistor *t* is considered stronger than transistor *r* iff Strength(*t*)>SR*Strength(*r*). Here SR stands for strength ratio. Using partial strength order enables solving contention without user intervention.

5.2 Logic Function Representation

As in ANAMOS [5] and TRANALYZE [6], FEV-Extract uses DAG to represent Boolean equations. The original DAG used by ANAMOS and TRANALYZE assigns terminals to circuit nodes. FEV-Extract introduces powerful sharing of the DAG sub-formulas using formula templates. The DAG terminals are indexes rather than real circuit nodes, and the indexes are mapped at every instance to the appropriate circuit nodes. This technique allows FEV-Extract to store the entire model logic in the memory, unlike ANAMOS, which stores the logic in files.

5.3 Logic Simplification

FEV-Extract simplifies the logic equations using signal relations that either are provided by the user for top cell input signals or are derived from the extracted formulas. This notion of simplification was also introduced in [6], where the main motivation to simplify the output logic was to reduce false contentions. FEV-Extract uses the simplification technique mainly for reducing the following verification efforts by the proper state identification. We also use BDDs for more powerful simplification of small equations only.

6. State Identification

One of the key challenges in automatic functional extraction is the resolution of circuit loops and extraction (out of them) of logical elements. Loops can be resolved into two types of elements: *Combinational elements* used for dynamic implementation of combinational functions (e.g. domino), and *Sequential elements*, used for the implementation of storage elements such as latches or BUS keepers.

The device (combinational or sequential) identification flow has three main steps: 1) Loop finding, 2) Functional loop analysis and inference, and 3) Extracting the appropriate logical element.

6.1 Loop Finding

Inherently VLSI models have loop structures for combinational and sequential logical elements. A *combinatorial loop* may either form a single sequential state element, like latch, or be a part of a dynamic implementation of the pure combinatorial RTL logic, while a *sequential loop* spans over one or several sequential state elements.

The goal of FEV-Extract is to identify and resolve combinational loops only. Sequential loops are left in the extracted finite state machine of the circuit. In order to distinguish between the combinational and sequential loops, we make an assumption that most of sequential loops have longer paths than the combinational ones (every sequential loop passes through at least one smaller combinational loop that forms its state element). Based on this assumption we employ a common loop identification algorithm, such as DFS graph traversal, to find structural loops in several steps, starting from the shortest loops (3 to 4 transistors or 2 CCSN-s in a loop) that, for sure, can't be sequential. Each loop is analyzed and logical elements are identified and marked. Once we identify a sequential element, the algorithm recognizes it, marks appropriately, and ignores it in longer loops that contain it as a sub-loop.

The loop identification algorithm looks first for the SCC-s (Strongly Connected Components) using a simple DFS linear traversal algorithm. It then traverses through all the possible loop paths of every found SCC, looking for the loops with limited stack depth. The second stage of the algorithm is exponential. However, since most of the structural loops are small and even bounded, the run time is very reasonable.

FEV-Extract identifies structural loop logic on circuit nodes with accumulative capacitance storage capabilities (single CCSN in a loop) and extracts appropriate latch logic.

6.2 Functional Loop Analysis and Inference

Given a combinational loop, we want to identify its corresponding logical element. In order to analyze the loop correctly, there is a need to analyze it together with additional logic surrounding it. For doing this, let us introduce a concept of *stage*, which is a set of CCSN-s that form the combinational loop and correspond to a functional element (combinational or sequential). The stage includes a *driving logic*, a combinational logic that introduces new values into the loop during the circuit execution, a *feedback path* that is a logic function used to store

the current value in case the driving logic is disabled, and a *collateral logic* that is usually used to simplify the overall stage functionality.

Each stage is analyzed separately in three steps, which are: 1) Identification of the stage output(s) on which the loop logic is to be solved; 2) Generation of the overall zero-delay collapsed stage functionality on the stage output(s); and 3) Stage inference, which is identification of the functional parts that form the stage, like asynchronous set/reset, clock/enable, driving data, feedback type and control.

Let's now assume that the overall stage logic *S* is collapsed into the form:

S = *f*(i1, i2 , …, S'),

where *f* is a dual rail function of *S* in quaternary format, and *S'* is the previous value of *S* itself.

The following stage functional parts are identified using the dual rail logic manipulations on the function *f*, using DAG and BDD representations:

- The driving logic - part of stage formula that doesn't depend on the stage previous value (quantify out variable *S'* from the equation *f*):

  D = S(S'=1) && S(S'=0)

- The driving control - either high or low rail of the driving logic holds:

  DC = D.1 || D.h

- The stage feedback logic - assuming that contention on the stag output is checked using CCSN extracted logic we may and weaken the feedback:

  FB = S && !DC

- The feedback control:

  FBC = FB.l || FB.h

- The stage feedback type - checking how 0 and 1 values on the stage output are propagated by the feedback logic.

  The feedback has High Retain type if:

  FB.h(S'=1) != 0 && FB.h(S'=0) == 0

  The feedback has Low Retain type if:

  FB.l(S'=0) != 0 && FB.l(S'=1) == 0

  The feedback has Full Retain logic if it follows both High and Low Retain rules.

  Self-reset feedback types are identified appropriately when

  FB.h(S'=0) != 0 || FB.l(S'=1) != 0

  In case the feedback logic doesn't propagate either 1 nor 0 values:

  FB.h(S'=1) == 0 && FB.l(S'=0) == 0

  the loop is considered structural but not functional one.

- Asynchronous Set and Reset logics are identified as independent parts of *DC* that leads *D* to constant 1 or 0 appropriately.

6.3 Extracting the appropriate logical elements

Given the stage parts inferred in the previous phase, FEV-Extract solves stages as one of the following logical structures: BUS retainer, domino (or Precharge logic), latch, or self-reset loop logic. The identification is based on the logical behavior of the inferred stage. For example, domino elements are recognized if the driving logic is fully separated into *set* and *reset* paths, either one of the paths is controlled by clock signal (the precharge logic), and the feedback loop forms full or half keeper that matches the precharge logic polarity. Latch elements are identified in a slightly different manner. Two latches in a tour, with opposite control logic and same set/reset can be combined into one Flip Flop sequential element.

7. Hierarchical Extraction

Hierarchical extraction is performed template by template, generating a hierarchical output model.

Hierarchical extraction in FEV-Extract is attempted for each sub circuit that contains more than a predefined number of primitives, as the overhead of analyzing small cells separately may result in inefficiency.

Hierarchical extraction outperforms flat extraction in computing time and memory space. In flat extraction, every portion of the model is analyzed, causing the identical blocks to be analyzed repeatedly. Hierarchical extraction significantly reduces the tool run time. For example, flat memory array extraction that may take ~15 hours can be extracted hierarchically just in one minute.

The hierarchical extraction in FEV-Extract has three levels of analysis:

- ***Native hierarchy***: Each model has a hierarchy inherent to it. Every template is extracted only once and instances are appropriately mapped to the template logic.
- ***Recognized hierarchy***: Like [5], FEV-Extract partitions the model into CCSN-s and identifies the CCSN templates by recognizing the similarity in transistor structures. Every CCSN template is extracted only once. Unlike [5], partial strength ordering technique requires the transistor strengths to be normalized, in order to achieve better results.
- ***Partially flattened model***: CCSN-s that collide with the netlist native hierarchy and cross hierarchical boundaries (except of BUS CCSN-s) are smashed and flat extraction is locally performed.

7.1 Partial flattening

Each instance that collides with CCSN boundaries is smashed. The smashing is performed on the instance itself, maintaining its internal hierarchical structure. In order to identify the CCSN-s' boundaries and mark smashing instances, the native model hierarchy is traversed in DFS manner, performing the following instance interface analysis:

- Instance pins that are connected directly to the transistor source or drain are considered to be outputs.

- Instance pins that are connected to any sub-cell output are considered to be outputs.
- Instance pins that are connected through the hierarchy to the top cell input pins are considered to be inputs
- Black box interface is taken according to its definition in the netlist.
- An instance with at least one output pin (except for BUS outputs) connected to an output pin of another instance of the same hierarchical level is considered to be smashed.

7.2 BUS Extraction

From the hierarchical point of view, we may classify all the model signals as either BUS or regular signals. Most of the RTL formats allow only one assignment to be specified for a regular signal. Complying with this rule requires that the entire logic of the signal will be analyzed as one indivisible entity, thus preventing hierarchical or step-by-step extraction. BUS signals can be multiply driven, and have several assignments assuming a strong or relaxed MUTEX relation between the drivers' control signals.

FEV-Extract performs the following steps to generate BUS logic:

- BUS-es are identified as multiply driven nodes that have drivers with several hierarchical levels.
- Each BUS node can be driven by a single local CCSN and an arbitrary number of hierarchical bus drivers.
- The BUS' multiply driven logic is generated according to the output format.

7.3 Memory Array Extraction

Circuit memory configuration usually uses a huge signal called *memory bit line* that either passes data to the memory states (write operation), passes the stored memory values to the output logic (read operation), or does both. The memory cells are usually instances of the same logic template. In most cases, the entire memory is one single CCSN consisting of two memory bit lines, which are inputs to the sense-amplifier circuitry. Such CCSN-s tend to be very big and cause major performance issues in extraction, verification and debugging. However, in FEV-Extract we developed a special method, similar to the BUS analysis method described above, to avoid such an explosion.

8. Experimental Results

The algorithms described in sections 6 and 7 considerably boosted the capacity, performance, and productivity of FEV-extraction flow. This section presents experimental evidence that supports our claims. All tests are taking from the current Intel design.

8.1 Hierarchical extraction

Table 2 presents the memory and CPU run time data for hierarchical versus flat extraction on 15 test cases. We also included the number of transistors in each test case for reference. Tests 1-11 are control logics and data paths. Tests 12-15 are memory arrays. All measurements were conducted on Linux dual CPU P4 machine, 1.8 GHz (one CPU is used).

Test case	# trans	Flat(s)	Hier(s)	Flat(MB)	Hier(MB)
1	1346	5	1.6	29	18
2	5829	14	3	60	24
3	9857	10	4	37	18
4	11260	15	8	51	26
5	19800	25	6	60	20
6	24912	78	7	63	30
7	25207	20	10	53	27
8	43813	68	33	108	31
9	50926	68	30	118	31
10	93376	156	65	155	40
11	98306	553	25	181	27
12	108388	692	7	231	24
13	198254	1430	28	300	30
14	225108	464	50	260	54
15	396728	6711	60	598	61

Table 2. Comparison between flat and hierarchical approaches

The leap achieved by the hierarchical approach is evident – the hierarchical approach is ~**25x** faster! The gain in memory usage is also easily noticed. Note how the flat approach suffers greatly from the increase in the number of transistors while the hierarchical approach is immune to that.

8.2 Productivity

While it is trivial to measure the run time and memory usage, it is difficult to measure the impact of the automation provided in FEV-Extract. Since lack of such a capability entails manual work (also called *tuning* of the circuit), combined with several iterations, it is important to supply measurements criteria, in order to estimate the benefit from the automatic state/domino identification. You may note that the problem worsens in cases where iterations of the extraction tool may take hours and even days. In this section we suggest measurement criteria to estimate the time needed to get the circuit model abstracted and ready for verification. We do not claim that the formula we suggest is 100% accurate since subjective influences cannot be ignored, but we view it as good approximation. The formula that we suggest incorporates two guidelines: 1) Learning is logarithmical – first attributes are hard to find and tune; 2) Tuning effort for iteration is measured for an attribute category rather than one attribute. For example, when the tool exits with an error because some complex domino structure is not tuned, it is more likely that the circuit designers will tune the most, if not all of the instances of this complex structure.

First we introduce a few definitions:

T_e = Average time to analyze extraction failure and tune the circuit.

T_a = Turn-around time for the extraction tool

L = Learning factor. Denotes improvement in designer's ability to tune the circuit.

K = Number of extraction iterations.

Based on these definitions, the formula to estimate the elapsed time to get the circuit model ready for verification is:

$$E_t = T_a * K + \sum_{i=0}^{i=k-2} \frac{T_e}{L^i}$$

where the first product stands for the run time of the extraction tool, while the second one represents the iterative manual work needed to tune the circuit. Note how the formula is sensitive to the learning factor – as the designer is more experienced, the elapsed time will decrease exponentially.

In table 3, we compare another in-house tool ('Manual' in this table) that required tuning to perform analysis to Fev-Extract ('Auto'). We used a learning factor of 2 and for T_e we used 30 minutes (those numbers are taken just from our experience).

Test case	# trans	T_a		K		E_t	
		Manual	Auto	Manual	Auto	Manual	Auto
13	198254	1430	28	7	2	~4.1h	~0.5h
12	108388	692	7	5	1	~3h	~1m
15	396728	6711	60	5	2	~10h	~1h

Table 3. Impact of automatic identification

Note how the impact of automatic identification strengthens as the run time of the tool increases.

9. Future Work

Looking ahead, innovation in circuit design is always encouraged to achieve the maximum speed and density. This puts the challenge ahead on FEV-Extract to be accurate and automatically handle self-timed logic, pulse circuits, and other fancy circuit implementations. It is very important to improve FEV-Extract to be efficient in handling delay dependent circuits as it handles today zero delay circuits. For doing this we are planning to develop new algorithms for unit delay extraction, improve extraction accuracy by considering actual circuit delays and thus combining functional and static timing analysis capabilities to cope with future >10 GHZ circuit designs. For bridging the gap between high level RTL model and detailed circuit implementation, we are planning to identify high level circuit structures such as memory arrays (i.e. extract memory two-dimensional constructs rather than bit-wise latch-s), pipelines, arithmetic operators and additional RTL constructs.

10. Conclusions

In this paper we presented FEV-Extract, which was developed as part of Intel's Formal Equivalence Verification CAD system [8,9]. We explained its working flow, its main algorithms that enable automatic identification of logical elements, its hierarchical analysis flow, and a few other innovative algorithms that overall make FEV-Extract a step function compared to other published methods in academic or in the EDA industrial world. With the advent of complex VLSI circuits, FEV-Extract is indispensable for generating an accurate functional representation of custom circuit designs, and thus becomes a major component in the design and verification flow.

The algorithms presented were implemented in FEV-Extract and are successfully used in Intel chip design projects. In addition, we have started a patent process for key algorithms and methods employed in FEV-Extract.

Acknowledgments

Thanks to Intel DT Strategic Cad Lab researchers Jeremy Casas and Carl Seger for participating in the development of the core technologies in FEV-Extract. Additional thanks go to the CAD infra structure group in Design Technology for the co-development of numerous algorithms and software modules needed in FEV-Extract implementation.

References

[1] Daniel Fischer, Yossi Levhari, Gadi Singer: NETHDL: Abstraction of Schematic to High Level HDL. Design Technology, Intel Israel (74) Ltd. ICCAD 1990.

[2] Timothy Kam, P.A. Subrahanyam: Comparing Layouts with HDL Models: A Formal Verification Technique.

[3] Timothy Kam, P.A. Subrahanyam: State Machine Abstraction from Circuit Layouts using BDD's: Applications in Verification and Synthesis.

[4] A. Lester, P. Bazargan-Sabet, A. Greiner: LIP6/ASIM Laboratory, University Pierre et Marie Curie – Paris: Yagle, a second generation functional abstractor for CMOS VLSI circuits.

[5] R.E Bryant, "Boolean analysis of MOS circuits" IEEE Transaction CAD/IC, 1987, 634-649.

[6] R.E Bryant, "Extraction of gate level models from transistor circuits by four valued symbolic analysis", In international Conference On Computer Aided Design, pages 350-353, 1991.

[7] R.E Bryant, "Algorithmic aspects of symbolic switch network analysis" IEEE Transaction, CAD/IC 1987,618-633.

[8] John Moondanos, Carl Seger, Daher Kaiss, Ziyad Hanna. "CLEVER: Divide and Conquer Combinational Logic Equivalence Verification with False Negative Elimination". CAV 2001.

[9] Zurab Khasidashvili, John Moondanos, Daher Kaiss, Ziyad Hanna, "An Enhanced Cut-point algorithm in formal equivalence verification". HLDVT 2001.