Power Virus Generation Using Behavioral Models of Circuits

K. Najeeb, K. Vishnu Vardhan Reddy Konda, H. Siva Kumar Sastry, V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai, India-600036
E-mail:veezhi@gmail.com
Phone:91-44-22574368 Fax:91-44-22574352

Vivekanand M Vedula
Validation and Test Solutions
Intel Corporation Austin, TX78746

***Abstract*— The problem of peak power estimation in CMOS circuits is essential for analyzing the reliability and performance of circuits at extreme conditions. The dynamic power dissipated is directly proportional to the switching activity (number of gate outputs that toggles (changes state)) in the circuit. The *Power Virus* problem involves finding input vectors that cause maximum dynamic power dissipation (maximum toggles) in circuits. Techniques that use a gate-level representation of the given circuit to generate a power virus are reported in the literature. As the power virus problem is *NP-complete* the gate-level techniques are less scalable with increasing design size and produce less optimal vectors. In this paper, an approach for power virus generation using *behavioral models* of digital circuits is presented. The proposed technique converts the given behavioral model *automatically* to an integer (word-level) constraint model and employs a integer constraint solver to generate the required power virus vectors. Experimenting the proposed technique on ISCAS'85 behavioral level benchmark circuits and the standard DLX processor model show that the above technique is fast and yields higher-quality results than the known gate-level techniques. Interestingly, the paper attempts to *generate an assembly program that cause the maximum dynamic power dissipation* on the given DLX processor model. To the best of our knowledge the proposed technique is the first reported that considers power virus generation using behavioral level models.**

Keywords– Behavioral Models, Dynamic power dissipation, Power virus, Integer Constraint Solvers, Hardware Description Languages (HDL).

Title Page ends

I. INTRODUCTION

The high transistor density, together with the growing importance of reliability as a design issue, has made early estimation of worst case power dissipation (peak power estimation) in the design cycle of logic circuits an important problem. The peak power consumption corresponds to the highest switching activity generated in the circuit under the test during one clock cycle. The dynamic power dissipated (P_R) in the combinational portion of a sequential circuit can be computed as [1]:

$$P_R = \frac{1}{2} \cdot V_{DD}^2 \cdot f \times \sum_{\forall gates\ g} \Big[N(g) \times C(g) \Big] \qquad (1)$$

where the summation is performed over all gates g, and $N(g)$ is the number of times gate g has switched from 0 to 1 or vice versa within a given clock cycle, $C(g)$ is the output capacitance of gate g, V_{DD} is the supply voltage and f is the frequency of operation. In this work we made the assumption that the output capacitance for each gate is equal to the number of fanouts. Therefore, the total switching activity is the parameter that needs to be maximized for maximum power dissipation. Accurate estimation of maximum power consumption involves finding a pair of input vectors which when applied successively, maximize the value of P_R as given by the equation (1), among all possible input vector pairs. *These two vectors can be applied one after another repeatedly to cause the estimated power dissipation for an indefinite time*. Given that the circuit has n primary inputs, there are 4^n possible two input vector sequences to be considered for an exhaustive search. It is easy to see that the power virus generation problem for sequential circuits reduces to solving multiple instances of the same problem on its underlying combinational part. Let S_i denote the state of a sequential circuit S at the beginning of the clock cycle i and I_i denote the primary inputs at the clock cycle i. The power virus problem on S is to find a k-cycle sequence $(S_1, I_1, S_2, I_2, \cdots, I_k, S_{k+1})$, such that, $S_{k+1} = S_1$, and at every cycle the input I_i is calculated so as to generate maximum toggle activity in the combinational portion of S. Two different power estimation metrics are defined in the context of sequential circuits, namely, the *peak single-cycle power* and the *peak sustainable power* [2]. The former denotes the maximum power dissipated during any one cycle of the k cycles while the latter denotes the average power measured across the k cycles. The values of I_0 and hence S_1, is got by *warming* up the circuit as discussed in [3].

II. PREVIOUS WORK

This section presents the previous work reported in the literature for the power virus generation and constraint modeling based test generation. The section proceeds further to present the salient contributions of this paper.

A. Power Virus Generation for Gate level representations

Several approaches have been proposed to estimate the maximum power consumption for CMOS circuits in [2]–[10]. In [4] Devadas *et al.* reduce the problem to a weighted max-satisfiability problem on a set of multi-output Boolean functions obtained from the circuit logic description. This approach took time exponentially proportional to the number of primary inputs (PIs) and hence applicable only to small circuits. Kriplani *et al.* [6] have presented a pattern independent algorithm to find an upper bound on the maximum instantaneous current through the power supply lines of CMOS circuits. The technique presented in [5] takes time exponential with respect to the number of levels in the circuit and hence is not scalable for circuits which are sufficiently deep. An estimation of average switching in combinational circuit using symbolic simulation is discussed in [8]. The method proposed in [9] propagates signal uncertainty waveforms throughout the circuit to obtain a loose upper bound on the maximum power. n upper bound on the number of simultaneous switching gates using partitioning techniques is presented in [11]. On the sequential end, several techniques are proposed in [2], [3]. Of these, the genetic algorithm presented in [2], addresses both single-cycle power and sustainable power metrics. Four different heuristics are presented in [2] and it assumes a **variable-delay** model.

B. Constraint Modeling based Test Generation

Constraint propagation techniques across different domains, that is, (both arithmetic and boolean domains) have been explored to generate functional tests and high level ATPG vectors on HDL descriptions [12] [13] [14]. The ideas discussed in [15]–[18] use constraint solvers to generate tests for functional verification of higher-level architectural features.

C. CONTRIBUTION OF THIS PAPER

This paper proposes a methodology for power virus generation using a behavioral description of a circuit. Hardware Description Languages (HDL) like Verilog enables one to develop such behavioral models [19]. The salient features of the proposed technique are outlined below:

1) *The input to the proposed technique is a behavioral level HDL model*: The behavioral circuit models are lesser in size and deal with word-level variables when compared with the corresponding gate level circuit models that deal with bit-level variables. This word-level representation in-turn exploits the word-level parallelism that exist in modern processors to speed-up the computations involved in the generation of the power-virus. This enables a larger exploration of the search space within a given time, when compared to the gate-level techniques. In addition, *the word-level variables grow less dramatically than the bit-level variables with increasing design functionality*. This accounts for the better scalability of the proposed technique with increasing design size than the gate-level techniques.
2) *Automatic generation of the constraint model from the behavioral model*: Automatic extraction of constraints from a behavioral or RTL description is a challenging problem [20]. The proposed methodology converts the given behavioral description into an A^2M graph based representation [21] and further converts the same into

a set of integer constraints. The constraints are solved using an integer solver to generate the required power virus vectors. All the above steps are *fully automated*.

3) *Correlation to Gate level representation*: As the proposed technique considers only behavioral level models, it does not capture the gate level implementation details. In other words, the variables of the behavioral model does correspond to wires in the gate level representation, but not all wires in the gate level representation are variables in the behavioral model. For example, an adder is represented by the + sign in the behavioral model. Its actual implementation at the gate level (say, carry-ripple or carry look-ahead circuit) is not taken into account by the proposed technique. The power virus computed by the proposed technique shall maximize the toggles on the inputs and outputs of the adder. The interesting question is that will this maximize the number of toggles in the underlying gate level representation too, as this is the required *real estimate* of the power. The answer to the above question is predominantly positive. This paper discusses the basic intuition behind this positive answer. This is further supported by experimental validation.
4) *Handling sequential designs with dynamic unrolling*: The behavior of a sequential circuit S over k time frames can be modeled as a combinational circuit using the conventional time frame expansion approach, which unrolls the combinational part of S, k times [22]. Given the maximum number of time frames (MAX_TFS) for which the model can be unrolled, the proposed technique dynamically unrolls for k time frames, for different values of k, $1 \leq k \leq MAX\_TFS$, to find an optimal solution.

To the best of our knowledge the proposed technique is the first reported that considers power virus generation using behavioral level models.

III. Constraint Generation

The proposed methodology may be divided into the following three phases, namely,

1) Generation of the A^2M graph from the input behavioral model;
2) Generation of constraints from the A^2M graph representation; and,
3) Solving the constraints using a constraint solver.

The following subsections explain the first two phases in detail.

A. A^2M graph generation

The synthesizable part of the verilog code that describes the behavior of a circuit has three major structures, namely, the **A**`lways block`, the **A**`ssign (continuous) statements` and the **M**`odule instantiations` [19]. Hence, the acronym A^2M. The A^2M Graph essentially has 4 entities, namely, `Module`, `Component`, `Signal` and `ADD Node`. Every *module definition* in the verilog code maps on to a `Module` entity of the A^2M graph. Every *always*, *assign* and *module instantiation* statements in the verilog code map on to a `Component` entity of the A^2M graph.

The `Module` entity contains the various fields necessary to encapsulate a module definition in verilog. They include `Name` of the module, an array of all the `Components` (always, assign and module instantiation statements) inside the module, an array of all the `Signals` (all `reg` and `wire` verilog variables including the `input` and `output` variables) inside a module, an array storing all the indices of entries in the above `Signal` array corresponding to the `input` to the module, and an array storing all the indices of entries in the above `Signal` array corresponding to the `output` to the module.

The `Component` entity contains the various fields required to encapsulate the three types of structures, namely, a module instantiation, an assign statement and an always block. Two additional `Component` types, namely, the `feedback component` and the `stem component` are defined to handle the *loops* and *stems* respectively inside the design. The *stems* are signals that drive multiple components. The various fields inside a `Component` are `name` and `type` of the Component, pointer to the `Parent Module` of the Component (the verilog module inside which the Component is defined), `fan-ins` (input signals to the component), `fan-outs` (output signals of the component), and the `ADDNode` corresponding to the Component. The `ADDNode` captures the functionality of the `Component` and is defined only for the `Components` that corresponds to an `always` block or an `assign` statement. The ADD Framework is an internal representation of the HDL Description and has been shown to be *complete and efficient*. More details on ADD is available in [23]–[26].

B. Constraints Generation

This section deals with different verilog constructs, their corresponding A^2M representations and their equivalent constraint models. This in turn, shall explain the *automatic* generation of the *model constraints* from the given behavioral verilog model. The standard ILOG [27] constraint solver was used for experimentation. Hence, a ILOG-type syntax is used to illustrate the example model constraints in this paper.

1) The `reg` *and* `wire` *variables:* Two types of variables are commonly used in verilog models, namely, the `reg` and `wire` variables [19]. These variables can either be *bits* or integers (*bit-vectors*). The integer domain deals more with *bit-vectors* rather than individual bits. This in turn, increases the modeling complexity.

Any *bit-vector* of size n in the input behavioral model is mapped on to a `Signal` entity in the A^2M graph with $size = n$. This is treated as an integer variable ($IloIntVar$ in ILOG) whose value ranges from 0 to $2^n - 1$. The *bit* variables are also treated as *bit-vector*s of size $n = 1$.

The `reg` and `wire` variables in a verilog expression map on to one of the two types of nodes on the A^2M graph, namely, the *read nodes* and the *write nodes*. As the names suggest, the variables on the right hand side (left hand side) of a verilog expression map on to read (write) nodes. The

read and write nodes in the A^2M graph have two attributes attached to it, namely, the $range$ and the $index$. The need for these attributes arises from the fact that a vector can be referred in the following three ways in a verilog code.

- *Bit Select*: For example, $ir[3]$, that selects the third bit of the bit-vector ir. The respective node in the A^2M graph has $index = 3$ and $range = [-1 : -1]$. Since, the integer solver do not directly deal with bits, the corresponding ILOG representation of $ir[3]$ is
$$2^3 * (IloDiv(ir, 2^3) - 2 * IloDiv(ir, 2^4)),$$
where, `IloDiv` stands for integer division in ILOG.
- *Part select*: For example, $ir[16 : 13]$, that selects the four bits starting from the bit 13 thru 16 of the bit-vector ir. The respective node in A^2M graph has $range = [16 : 13]$ and $index = -1$. The corresponding ILOG representation is
$$2^{13} * IloDiv(ir, 2^{13}) - 2^{17} * IloDiv(ir, 2^{17})$$
- *The entire vector*: For example ir. The respective node in A^2M graph has $index = -1$ and $range = [-1 : -1]$. The ILOG representation models the variable ir as a normal integer ($IloIntVar$).

The variables defined inside a module M carry different values for different instantiations of M. To encapsulate this, every variable in M is declared as an array of size equal to the number of instantiations of M. For example, given that the variable ir is defined in M and that M is instantiated twice, say, instantiation 0 and instantiation 1, $ir[0]$ ($ir[1]$) denotes the value of ir in instantiation 0 (1). On similar lines, the third bit of ir in instantiation 0 is modeled as

$$2^3 * (IloDiv(ir[0], 2^3) - 2 * IloDiv(ir[0], 2^4)).$$

The part select of ir comprising of the four bits (bits 13 to 16) in instantiation 1 is modeled as

$$2^{13} * IloDiv(ir[1], 2^{13}) - 2^{17} * IloDiv(ir[1], 2^{17})$$

2) Modeling sequential circuits: Every sequential circuit can be represented by the conventional Huffman model [22]. The combinational and the sequential parts are clearly distinguished in this representation. To model the circuit in the integer domain the following two basic principles are used:

- Each sequential element is a variable in the integer domain; and,
- Each combinational element produces a constraint on its inputs and outputs in the integer domain.

The behavior of a sequential circuit S over k time frames can be modeled as a combinational circuit using the conventional time frame expansion approach, which unrolls the combinational part of S, k times [22]. The above mentioned approach for modeling in the integer domain is illustrated by using an example of a counter. The following verilog code models a counter

```
reg [4:0] counter;

always @(posedge Clk)
```

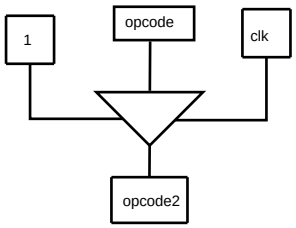

Fig. 1. Always statement with clk

```
counter = counter + 1;
```

Without loss of generality, let there be only one instantiation of the module containing the above verilog code. Therefore, the variable shall be denoted by $counter[0]$ in the corresponding A^2M graph. To unroll a circuit over time frames, another dimension is added to the variables to represent the time frame. Thus, the variable $counter[0][j]$, denotes the variable $counter$ in the j^{th} time-frame. The combinational part of the Huffman Model is the one that sets constraints on the variable $counter[\ ][\ ]$ across time-frames. The verilog code and the underlying A^2M graph imply the following constraint:

$$counter[0][j] == counter[0][j-1] + 1$$

Thus, the clock in verilog is realized as a time frame in the corresponding constraint model.

3) Assign Statement: Given that a circuit is unrolled for MAX_TFS time frames, the assign statement in verilog leads to constraints on the variable for all time frames. For example, the assign statement

$$assign\ sum = input\_1 + input\_2$$

inside the k^{th} instantiation of a module M shall lead to the following set of constraints.

```
for(tf = 0; tf< MAX_TFS;tf++)
{
```
$$(sum[k][tf] == input\_1[k][tf] + input\_2[k][tf]);$$
```
}
```

The other operators that include *bitwise, comparison, concatenation* etc. supported by verilog can be modeled similarly.

4) Always Statement: The constraint model for an always construct in verilog depends on the *event* in its *sensitivity list*. The event can be either *clocked* or *non-clocked* and is represented by an event node in the corresponding A^2M graph. The following code is a clocked always structure:

$$always\ @(posedge\ clk)$$
$$opcode2 <= opcode;$$

The A^2M graph for the above case is shown in figure 1. The model constraint for the above A^2M graph for the k^{th} *instantiation* of the module to which it belongs to will be:

$$(opocde2[k][tf+1] == opocde[k][tf])$$

The following code is a non-clocked always structure.

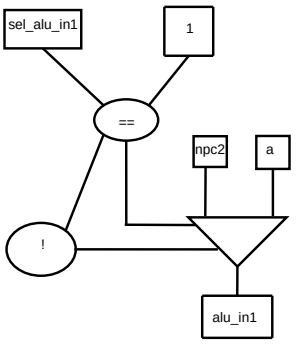

Fig. 2. Always statement without clk

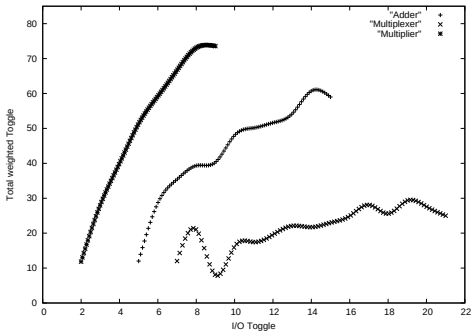

Fig. 3. Behavioral Vs Gate Level Correlation

$$always\ @(sel\_alu\_in1\ or\ npc2\ or\ a)$$
$$if(sel\_alu\_in1 == 1)$$
$$alu\_in1 <= npc2;$$
$$else$$
$$alu\_in1 <= a;$$

The A^2M graph for the above case is shown in figure 2. The model constraint for the above A^2M graph for the k^{th} *instantiation* of the module to which it belongs to will be:

$$(alu\_in1[k][tf] == (sel\_alu\_in1[k][tf] == 1) * npc2[k][tf] + (sel\_alu\_in1[k][tf] == 0) * a[k][tf])$$

5) Module Instantiation: The module instantiations lead to generation of *interface constraints* that establish a connection between the input-output variables of the instantiating module and the instantiated module. For example, consider the module instantiation in Verilog.

```
module alu(out,in1, in2)
```

Let the following be its k^{th} instantiation.

```
alu myalu (o1,i1, i2)
```

The following are the interface constraints assuming that the circuit is unrolled for MAX_TFS time frames. Note that the following ensures the connectivity of the variables in every time frame.

```
for(j = 0; j< MAX_TFS;j++)
{
```

$$(alu\_out[k][j] == myalu\_o1[k][j]);$$
$$(alu\_in1[k][j] == myalu\_i1[k][j]);$$
$$(alu\_in2[k][j] == myalu\_i2[k][j]);$$

```
}
```

6) Power Virus Constraints: As mentioned earlier, the power dissipated is directly proportional to the number of toggles. It is straightforward to see that every variable in the verilog model corresponds to a signal in the corresponding A^2M graph. Every signal in the A^2M graph corresponds to a input or output signal in the underlying gate level netlist. These signals also get modeled as variables in the constraint model. Given this, it is easy to infer that by toggling the variables in the constraint model between successive time-frames shall lead to toggling gate outputs in the underlying gate level netlist. For every variable V in all its k instantiations in the constraint model, the following constraints are added.

```
for(j = 0; j< MAX_TFS-1;j++)
{
```

$$(V[k][j+1] == not(V[k][j]));$$

```
}
```

The `not` operator stands for the standard negation in case V is a bit and for the bitwise negation if V is a word (bit-vector).

IV. Experimental Results

The constraints generated as described in the previous section is input to a constraint solver that *automatically* generates the required vectors. In this section we shall describe details of the experiments conducted for the ISCAS combinational circuits and the 16-bit 5-stage pipelined DLX processor.

A. Correlation between Behavioral and Gate level representations

As mentioned earlier, the proposed approach maximizes the number of toggles on the inputs and outputs of the different behavioral level A^2M structures and does not take into account the gate level representation of the same. The ultimate objective is to generate maximum toggles on the underlying gate level netlist. The graph shown in figure 3 plots for different behavioral level constructs, that includes, an 8-bit adder, an 8-to-1 multiplexer and a 4-bit multiplier, the number of toggles in the input and output signals against the total number of toggles in the corresponding gate level implementation. A similar trend is seen for most of the other A^2M constructs. The Magma Blast Fusion tool using the TSMC 0.13 micron standard cell library was used to synthesize the behavioral representations to the corresponding gate level netlists. The graph shows an increase in the total number of toggles in the circuit with increase in the number of toggles in the corresponding inputs and outputs. This suggests that maximizing the toggles on the inputs and outputs for these structures shall in turn maximize the toggle in the underlying gate level netlist. Experimental results shown in the subsequent sections further strengthens our claim.

B. Power virus for combinational Circuits

As mentioned earlier the power virus generation for combinational circuits involves generating a pair of input vectors which when applied successively shall cause the maximum toggle in the circuit. Given a n-input circuit the number of such pairs shall be $O(4^n)$ resulting in a huge search space. Even for moderately large circuits the above problem becomes difficult to solve. A simpler version of the above is as follows: Given the first of the two input vectors, generate the other vector such that successive application of both produces maximum toggle in the circuit. This reduces the search space to $O(2^n)$. The proposed approach follows the simplified version.

For generating the power virus the constraint model of the combinational circuit is unrolled for two time frames. The constraint variables corresponding to the input of the behavioral model in the first time frame are initialized with the value of the random vector. Solution to the above constraint model that includes the power constraints yields the second vector that maximizes the toggle. Table I presents the results obtained by applying the above technique for the ISCAS'85 [28] (ISCAS High-Level Models) behavioral circuits.

Circuit	No. of Toggles at behavioral level	No. of Toggles at gate level	No. of wires at gate level	No. of Toggles (Random)
74181	103	123	201	77
74182	20	25	70	17
74283	33	51	104	33

TABLE I
COMPARISON OF BEHAVIORAL TOGGLES WITH RANDOM GATE LEVEL TOGGLES

From table I it is seen that the number of toggles generated at the gate level netlist by applying the vectors output by our method (column 3) is much better than what is generated by applying random pairs (column 5) of vectors. In addition, the total toggle count is closer to the number of wires (upper bound on number of possible toggles) in the circuit (column 4). Column 2 gives the number of toggles generated by the behavioral level representations of the corresponding circuits.

C. Power virus for the DLX processor

The DLX processor is a pipelined sequential circuit. Two types of experiments were carried out on the model. The first experiment was to generate instructions that will maximize the toggle when input to the processor given its initial state S. This enables automatic generation of assembly instruction sequences from any given initial state S that can maximize the toggle count and hence cause *peak single-cycle power* across different cycles. For experimental purpose this initial state S was arrived by applying a sequence of random instructions. Table II presents the percentage improvement in toggle count got by application of the instructions generated by the proposed technique in contrast to application of randomly generated instruction sequences of different lengths. It also presents the time and memory requirements.

Length of the Sequence	% improvement in toggles over random	Memory requirement (Our method)	Time requirement (Our method)
2	247.89	21 MB	1.2 sec
3	197.51	34 MB	2.1 sec
4	200.35	46 MB	0.9 sec

TABLE II
GENERATING INSTRUCTION SEQUENCES OF SPECIFIED LENGTH

A sequence of length 4 *automatically output* by the method assuming an initial state S is as follows:

```
NOP
ADD R1, R0 R2
ADD R1, R0 R2
ADD R1, R16, R13
```

The above was generated by unrolling the circuit for four time frames. An interesting point that is revealed during the experimentation is that an introduction of a `NOP` between two non-`NOP` instructions causes large number of toggles in the system. This is something which may be taken note by the compiler developers who tend to generate NOPs in executable code for various reasons, that include data and control hazard management in modern superscalar and VLIW based architectures. The results in table II indicate that our proposed approach is much better than the random one. Note that the memory requirement linearly scales with the length of sequence.

The second experiment is to find a loop of instructions which when applied continuously should maximize the *peak sustained power*. As in the first experiment the processor is driven to an initial state S by applying a random sequence of instructions. From this state S, a loop of instructions were generated that achieves the objective mentioned above. This needs some additional constraints for the following reason. To ensure same sustained power dissipation during every execution of the loop with say, k instructions, the system should be in the same state at the start of every loop. This implies that for every variable v, $v[i][0] = v[i][k]$. This ensures that at the end of each k cycle the system will be in the same state. Table III presents the results of this experiment.

Length of the Loop	Normalized per cycle toggle count	Length of the Loop	Normalized per cycle toggle count
5	50	9	55
6	70	10	49
7	67	11	46
8	62		

TABLE III
LOOP GENERATION FOR MAXIMIZING TOGGLES

From table III it is seen that the *per cycle toggle* count is the maximum for loops of length 6 and decreases with subsequent increase of the loop length. The reason for the same may be

that the DLX has a five stage pipeline. The following is a code of length 6 output by our method that generates the maximum per cycle toggle.

```
NOP
ADD R1, R0, R2
ADD R1, R0, R2
LD R1, [R2]
NOP
NOP
```

Several interesting points are exhibited by the above sequence. The first one is that the NOPs are predominant. The second is that even though the value of register $R1$ is changed by the ADD instructions, the LD (LOAD) instruction initializes the register $R1$ back with a value stored in a memory location whose address and content does not change across loops. Given that the above instructions were generated assuming the initial state of the processor to be S, the state of the processor at the end of every execution of the above six instructions should also be equal to S. This also explains why ST (STORE) instructions that write into data memory are not generated, as they change the state of the processor.

V. Conclusion and Future Work

This paper presented a technique that used behavioral descriptions to generate power virus test vectors. To the best of our knowledge, this is the first technique reported for the behavioral level power virus generation. Experimenting the proposed technique on ISCAS'85 behavioral benchmark circuits and the 16-bit DLX processor yielded encouraging results. Interestingly, the paper addressed the problem of *generating an assembly program that cause the maximum dynamic power dissipation* on the given DLX processor model. The technique assumed zero delay model. Future work shall involve modifying the technique to account for unit-delay and variable-delay models.

Acknowledgement: This work is supported by the IIT Madras - Intel joint research project on Processor verification.

References

[1] M. S. Hsiao, "Peak power estimation using genetic spot optimization for large vlsi circuits," *Design, Automation and Test in Europe Conf.*, pp. 175–179, 1999.

[2] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Peak power estimation of vlsi circuits: New peak power measures," *IEEE Transactions on VLSI Systems*, vol. 8, no. 4, pp. 435–439, Aug 2000.

[3] T. Chou and K. Roy, "Statistical estimation of sequential circuit activity," in *In Proc. ACM IEEE Intl. Conf. on CAD*, 1995, pp. 34–37.

[4] S. Devadas, K. Keutzer, and J. White, "Estimation of power dissipation in CMOS combinational circuits using boolean function manipulation," *IEEE Trans. on Computer-aided Design*, vol. 11, no. 3, pp. 373–383, March 1992.

[5] C.-Y. Wang and K.Roy, "Maximum power estimation for CMOS circuits using deterministic and statistic approaches," *Proc. of 9th Intl. Conf. on VLSI Design*, pp. 364–369, Jan 1995.

[6] H. Kriplani, F. Najm, P. Yang, and I. Hajj, "Pattern independent maximum current estimation in power and ground buses of CMOS vlsi circuits," *IEEE Transactions on CAD*, vol. 14, no. 8, pp. 998–1012, Aug 1995.

[7] C.-Y. Wang and K.Roy, "Maximization of power dissipation in large CMOS circuits considering spurious transitions," *IEEE Transactions on Circuits and Systems*, vol. 47, no. 4, pp. 483–490, Apr 2000.

[8] J. Monteiro, S. Devadas, A.Ghosh, K. Keutzer, and J. White, "Estimation of average switching activity in combinational logic circuits using symbolic simulation," *IEEE Transactions on CAD*, vol. 16, no. 1, pp. 121–127, Jan. 1978.

[9] H. Kriplani, F. Najm, and I. Hajj, "Maximum current estimation in CMOS circuits," *Proc. of Design Automation Conf.*, 1992.

[10] T.-L. Chou, K. Roy, and S. Prasad, "Estimation of circuit activity considering signal correlations and simultaneous switching," *IEEE Intl. Conf. on Computer Aided Design*, vol. C, no. 27, pp. 193–200, March 1978.

[11] K. Zhang, T. Shinogi, H. Takase, and T. Hayashi, "A method for evaluating upper bound of simultaneous switching gates using circuit partition," in *Asia and South Pacific Design Autmation Conf. (ASPDAC)*, 1999, p. 291.

[12] R. Vemuri and R. Kalyanaraman, "Generation of design verification tests from behavioral vhdl programs using path enumeration and constraint programming," in *IEEE Transaction on VLSI Systems*, vol. 3(2), June 1995, pp. 201–214.

[13] A. K. Chandra and V. S. Iyengar, "Constraint solving for test case generation: a technique for high-level design verification," in *Proceedings of International Conference on Computer Design: VLSI in Computers and Processors*, 1992, pp. 245–248.

[14] F. Fallah, S. Devadas, and K. Keutzer, "Functional vector generation for hdl models using linear programming and boolean satisfiability," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20(8), August 2001, pp. 994–1002.

[15] D. Lewin, L. Fournier, and M. Levinger, "Constraint satisfaction for test program generation," pp. 45–48, 1995.

[16] A. Aaharon, D. Goodman, and M. Levinger, "Test program generation for functional verification of powerpc processors in ibm," pp. 279–285, 1995.

[17] A. Adir, E. Almog, L. Fournier, and E. Marcus, "Genesys-pro: Innovations in test program generation for functional processor verification," pp. 84–93, April 1994.

[18] J. Yuan, C. Pixley, A. Aziz, and K. Albin, "A framework for constrained functional verification," in *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*. Washington, DC, USA: IEEE Computer Society, 2003, p. 142.

[19] S. Palnitkar, "Verilog hdl: A guide to digital design and synthesis." Singapore: Pearson Education, 2004.

[20] L. Lingappan, S. Ravi, and N. K. Jha, "Satisfiability-based test generation for nonseparable rtl controller-datapath circuits," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25(3), March 2006, pp. 544–557.

[21] K. U. Bhaskar, M. Prasanth, V. Kamakoti, and K. Maneparambil, "A framework for automatic assembly program generator (a^2pg) for verification and testing of processor cores," in *In Proc. Asian Test Symposium (ATS)*, Dec 2005, pp. 40–45.

[22] M. Abromovici, M. A. Breuer, and A. D. Friedman, "Digital systems testing and testable design." IEEE Press, 2001.

[23] I. Ghosh and M. Fujita, "Automatic test pattern generation for functional register transfer level circuits using assignment decision diagrams," in *IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems*, vol. 20(3), March 2001, pp. 402–415.

[24] I. Ghosh, L. Zhang, and M. Hsiao, "Automatic design validation framework for hdl descriptions via rtl atpg," in *Proceedings of the Asian Test Symposium*, 2003, pp. 148–153.

[25] V. Chaiyakul and D. D. Gajski, "Assignment decision diagram for high-level synthesis," in *Technical Report 92-103 University of California, Irvine, CA*, December 1992.

[26] V. Chaiyakul, D. D. Gajski, and L. Ramachandran, "High-level transformations for minimizing syntatic variances," in *Proceedings of Design Automation Conference*, June 1993, pp. 413–418.

[27] "Ilog technology web site." [Online]. Available: http://www.ilog.com

[28] "Iscas-85 high level models web site." [Online]. Available: http://www.eecs.umich.edu/ jhayes/iscas.restore/benchmark.html