APPROACHES FOR MONITORING VECTORS ON MICROPROCESSOR BUSES

Hector Arteaga and Hussain Al-Asaad
Department of Electrical & Computer Engineering
University of California
Davis, CA, U.S.A.

ABSTRACT

This paper introduces two new methods for observing and recording the vectors that have been asserted on a bus. The first is a software approach that uses a novel data structure similar to binary decision diagrams which allows for a compact representation of stored values. Even though the new data structure presented in this paper can potentially grow to contain just as many nodes as there are possible values, such cases are often rare. The second is a hardware approach that is based on a simple circuit consisting of a small memory and two counters and has the ability to perform at the speed of the microprocessor.

Keywords: Design validation, bus monitoring, microprocessor verification, post-silicon debug.

1 INTRODUCTION

As microprocessor continue to grow in complexity, the visibility to the internal state is adversely affected, making it increasingly difficult to attain critical debugging information. Methods such as JTAG and sample-on-the-fly [1] can be used to alleviate this problem, but continuous observation is often not possible or very time consuming. As a partial solution to this problem, two methods for monitoring vectors asserted on a bus are presented in this paper. The first is a software solution that exploits the binary decision diagrams (BDDs) ability to reduce itself to convey information more concisely. A hardware solution to the same problem that allows at-speed monitoring is also presented.

The paper is organized as follows. Section 2 describes the motivation of using a BDD-like structure to store information on vectors appeared on a bus. The details of our software approach and the obtained simulation results are also presented in this section. Section 3 describes our simple hardware approach and the details of the monitoring circuit and its operation. Finally, we conclude in Section 4.

2 SOFTWARE APPROACH

In this section, we describe our novel software approach and the simulation results that show its feasibility.

2.1 Motivation from bdds

BDDs [2] were originally designed as a way to check for equivalence between two implementations. A reduced and ordered BDD (ROBDD) is a canonical (unique) representation for any given function. Any pair of functions will have different ROBDDs unless the functions themselves are equivalent. Since two implementations are equivalent if and only if any input vector produces the same output in both implementations, their output functions must be the same and equivalence can be determined by comparing their respective ROBDDs.

But how are ROBDDs created? Well, first an order of inputs must be chosen (certain orders having advantages over others). Then beginning with the first input variable in the order chosen, the function is evaluated with the value of this variable set to 0 and 1 and the resulting functions placed to the left and right of the node respectively. Then the two functions just calculated are evaluated with the next variable in the order set to 0 and 1 resulting in four new functions, and so on until all the variables have been covered or

until a definite value can be determined. Alternatively, the ordered BDD (OBDD) can be determined directly from the truth table and then reduced to create the ROBDD. For example, let $f(abcd) = ab + cd$ be the function we wish to determine the ROBDD for using the order $<abcd>$. The truth table for this function, and the corresponding OBDD and ROBDD are shown in Figure 1.

The construction of the OBDD is done as follows. Beginning with node *a* (the root node), the functions *f*(*abcd*) is evaluated with $a = 0$ and $a = 1$, resulting in $f(0bcd) = cd$ and $f(1bcd) = b + cd$. These serve as the input functions to the two *b* nodes, which are then evaluated with $b = 0$ and $b = 1$ resulting in $f(00cd) = f(01cd) = f(10cd) = cd$ and $f(11cd) = 1$. Continuing on with this process for the variables *c* and *d* we end up with the OBDD in Figure 1. Close examination of this OBDD shows that the truth table is present at the leaf nodes of the OBDD. It is also evident from the OBDD that there are several terms that appear a number of times, revealing potential for simplification. Getting rid of these redundant terms by grouping them into single nodes results in the ROBDD shown in Figure 1. Any function with the same truth table as in Figure 1 will have the ROBDD shown in Figure 1.

As is evident by this example, a significant amount of reduction is possible (the BDD was reduced from 15 nodes to only 4, excluding the leaf nodes) and only a few nodes are necessary to specify the function. This observation led us to believe that BDDs could be used to efficiently store the vectors that have been placed on a bus. We next describe the details of our software method.

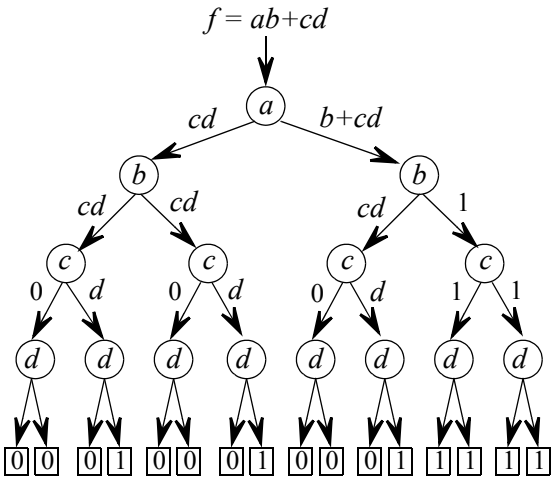
2.2 Novel software method

We introduce a new BDD-like structure called a *bus tree* (BT) that represents the vectors appeared on a bus. The BT starts of as just the root node since initially no vectors have been placed on the bus. As vectors are detected on the bus, the BT adds the paths associated with those vectors, looking for redundancies to reduce the overall BT size. For example, Figure 2 shows the BT with the vectors 0000, 0001, 1011, and 1111 already inserted. The vectors 0000

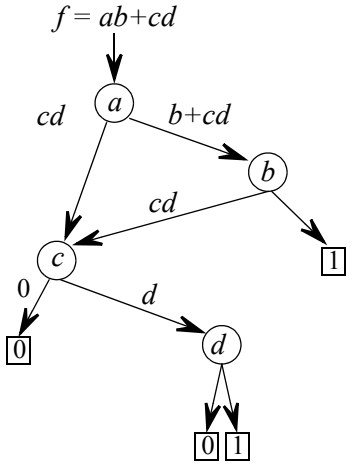
abcd	f
0000	0
0001	0
0010	0
0011	1
0100	0
0101	0
0110	0
0111	1

abcd	f
1000	0
1001	0
1010	0
1011	1
1100	1
1101	1
1110	1
1111	1

(a) Truth table



(b) OBDD



(c) ROBDD

Figure 1 The (a) truth table, (b) OBDD, and (c) ROBDD of the function *f*(*abcd*) = *ab* + *cd*.

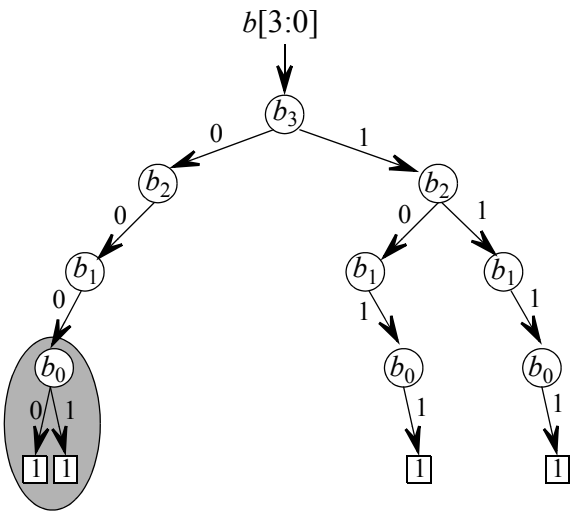


Figure 2 BT with 4 vectors inserted.

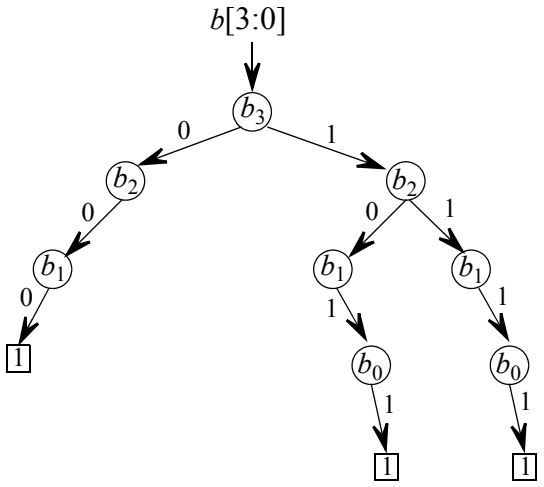


Figure 3 BT with vectors combined.

and 0001 can both be represented with the single vector by replacing the least significant bit with a variable to indicate that both values of that bit are present, i.e. 000x. Thus, the node shaded in Figure 2 is not needed and both vectors can be represented as being in the BT by identifying the left path in the b_1 node above the shaded node as being full, as shown in Figure 3. As more vectors are inserted, it becomes more likely that a reduction such as the one above is possible.

For an n-bit bus, if this method was followed and no reductions made, a full BT would have 2^{n-1} nodes at the b_0 level, 2^{n-2} nodes at the b_1 level, 2^{n-3} nodes at the b_2 level, … 2 nodes at the b_{n-2} level and 1 at the b_{n-1} level (the root node). Thus, in the worst case, the size of the BT without counting the leaf nodes, could grow to be $2^{n-1} + 2^{n-2} + \ldots + 2 + 1 = 2^n - 1$ nodes, which is one less than number of representable values in an n bit number; not a substantial gain. However, this case is highly unlikely and the actual size of the BT is expected to be significantly less. In fact, if the vectors asserted on the bus follow a binary order (0...000, 0...001, 0...010, 0...011,....), then the worst case size of the BT is n. Now, let us discuss the leaf nodes. Since all the leaf nodes are the same, they can all be grouped into a single element (not necessarily an actual node) that identifies a path as being full. By making all full paths point to the same element, all the leaf nodes can be lumped together, thus relieving some of the memory requirements.

2.3 Simulation Results

Figure 4 shows the growth of the BT as random vectors are inserted into it. As expected, the size of the BT grew linearly at first, but quickly tapered off. The growth is approximately 3x for the first 30 - 40 vectors and then begins to slowly die down, reaching a maximum of 193 nodes at the insertion of the 159th random vector (note that there are 256 distinct input vectors). The reason that the growth is not initially 8x is because as more and more vectors are inserted into the BT, the following vectors will have part or all of the vector already present in the BT. For example, if the vector 10010100 is the first vector inserted, the BT will grow from 1 to 8 since 7 new nodes will be necessary (root node already exists). Now, if the next vector to be inserted is 10001001 only 4 new nodes will be created since the 3 most significant bits of the vector, 100, are the same as that of the previously inserted vector. The size will grow to 12 instead of 16. If a vector is the same as a previously inserted vector, the BT remains as it is and only the number of vectors inserted increases. As more and more vectors are inserted into the BT, the likelihood that this will occur increases and fewer nodes are added to the BT. Eventually, the number of nodes needed for a vector to be inserted will be 0 and the size of the BT will reach its maximum. As vectors continue to be inserted, the number of nodes discarded due to the

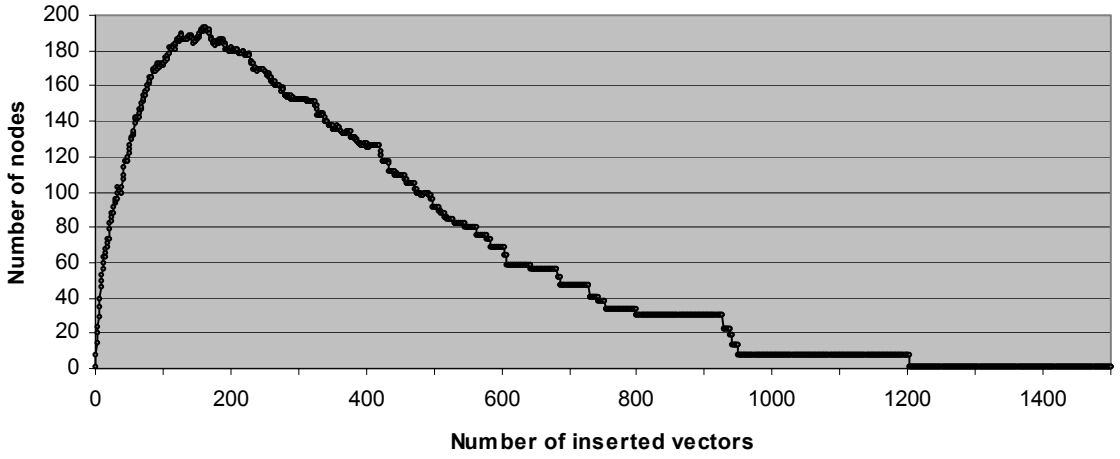


Figure 4 The size of the BT as 1500 random vectors of width 8 are inserted.

reduction of the BT will exceed the nodes added for the vectors and the size will begin to decrease. The BT will continue to decrease in size until all possible vectors have been inserted in which case the BT will return to its original size of 1 node as can be seen in Figure 4. (As a convention, we identify an empty BT as the root node with both its left and right paths empty and a full BT as the root node with both its left and right paths full. Alternatively, flags can be used to identify these two states in which case the initial and full BT size will be 0.)

Table 1 shows the maximum size of the BTs (in nodes) for four different bus lengths over ten random simulations. The table shows that the maximum size of the BT rarely exceeds three fourths of the total representable vectors, 2^n. Of the 40 simulations shown, only 2 resulted in a maximum size that exceeded this quantity. Furthermore, the results indicate that as the bus width increases, it is less likely that the maximum size will surpasses this value.

Table 1 Maximum size of the BT for various bus widths through several random simulations.

	8-bit Bus	10-bit Bus	12-bit Bus	14-bit Bus
1st Simulation	173	750	2982	11769
2nd Simulation	179	753	2987	11856
3rd Simulation	174	757	2965	11857
4th Simulation	180	757	2953	11821
5th Simulation	188	761	2986	11819
6th Simulation	188	760	2930	11808
7th Simulation	192	758	2932	11797
8th Simulation	194	757	2974	11759
9th Simulation	181	789	2947	11968
10th Simulation	192	735	2909	11823
2^n*(3/4)	192	768	3072	12288

3 HARDWARE APPROACH

As an alternative to the software approach, we have also designed a circuit that holds the values placed on a bus in memory as shown in Figure 5. The circuit primarily consists of two counters and a small $2^n \times 1$ bit memory. During normal operation the *output_en* signal will be set to 0 allowing the data

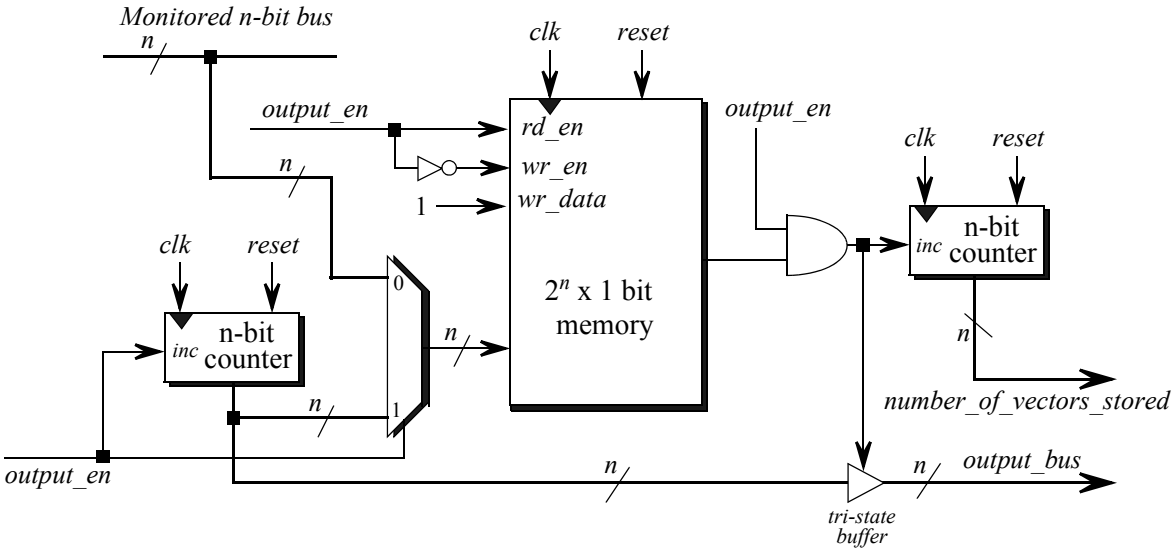


Figure 5 A bus monitoring circuit.

from the bus to reach the memory as an address and prohibiting the writing of any data on the *output_bus*. The single bit in this address will then be set to one since the write enable (*wr_en*) line on the memory will be set to 1 and the write data (*wr_data*) is hard-wired to 1. A value of 1 in a certain memory location indicates that the address corresponding to this location has been observed on the bus.

Once it is desired to view the results, the *output_en* signal is asserted and both counters reset. This allows the memory to obtain its addresses from the first counter and indicates that the memory should now be read from. As the counter executes its sequence of numbers, the corresponding memory location will be read. If a particular memory location contains a value of 1 the second counter will increment its value and the tri-state buffer will be enabled allowing the address to propagate to the *output_bus*. Once 2^n cycles have elapsed, all the memory locations will have been read and all values placed on the *output_bus*. The second counter will contain the total number of vectors placed on the bus, which will simplify the task of determining what percentage of vectors have been asserted. During this time, the processor can be busy performing other operations that do not require the bus to be monitored. To restart the sequence again, the memory is reset and the *output_en* signal is set to 0 once again.

The simplicity of the circuit allows for at-speed operation. However this design's greatest disadvantage is the cost of implementing the memory, which can take up a significant amount of chip area.

4 CONCLUSIONS

Two methods for monitoring bus activity have been presented. The first method is a software solution that uses a BDD-like structure called BT to store the data. This method allows a great degree of flexibility. The size of the BT depends on the order of input vectors appearing on the bus and it is bounded by $[n, 2^n - 1]$. The software approach cannot be run as fast as the hardware due to the time needed to process the BT. The second method is a hardware approach that allows for at speed data acquisition. However, the cost of this approach can be high due to the chip area of the used $2^n \times 1$ memory.

ACKNOWLEDGEMENT

This material is based upon work supported by the National Science Foundation under Grant No. 0092867.

REFERENCES

[1] M. S. Abadir, T. M. Mak, and Li-C. Wang, "Tutorial 15: Validation and verification of high-performance microprocessors: Common challenges and solutions", Proc. *International Test Conference*, 2003.

[2] R. Bryant, "Graph-based algorithms for boolean function manipulation", *IEEE Transactions on Computers*, C-35, pp. 677-691, June 1986.

[3] D. D. Josephson, S. Poehlman, and V. Govan; "Debug methodology for the McKinley processor", Proc. *International Test Conference*, 2001, pp. 451-460.

[4] A. Carbine and D. Feltham, "Pentium Pro processor design for test and debug", Proc. *International Test Conference*, 1997, pp. 294-303.

[5] B. Vermeulen, T. Waayers, and S. K Goel, "Core-based scan architecture for silicon debug", Proc. *International Test Conference*, 2002, pp. 638-647.