2000

Design and Evaluation of a Specialized Computer Architecture for Manipulating Binary Decision Diagrams

Robert K. Hatt
Portland State University

Recommended Citation

Hatt, Robert K., "Design and Evaluation of a Specialized Computer Architecture for Manipulating Binary Decision Diagrams" (2000). *Dissertations and Theses.* Paper 6491.
https://doi.org/10.15760/etd.3627

THESIS APPROVAL

The abstract and thesis of Robert K. Hatt for the Master of Science in Electrical Engineering were presented October 9, 2000, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

[redacted]

Marek Perkowski, Chair

[redacted]

Michael Driscoll

[redacted]

Sarah Mocas
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL:

[redacted]

Douglas Hall, Chair
Department of Electrical Engineering

ABSTRACT

An abstract of the thesis of Robert K. Hatt for the Master of Science in Electrical Engineering presented October 9, 2000.

Title: Design and Evaluation of a Specialized Computer Architecture for Manipulating Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are an extremely important data structure used in many logic design, synthesis and verification applications. Symbolic problem representations make BDDs a feasible data structure for use on many problems that have discrete representations. Efficient implementations of BDD algorithms on general purpose computers has made manipulating large binary decision diagrams possible. Much research has gone into making BDD algorithms more efficient on general purpose computers. Despite amazing increases in performance and capacity of such computers over the last decade, they may not be the best way to solve large, specialized problems. A computer architecture designed specifically to execute algorithms on binary decision diagrams has been created here to evaluate the possible performance improvements in BDD manipulation. This specialized computer will be described and its implementation discussed with respect to the important aspects of efficient BDD manipulations. This thesis will demonstrate that significant performance increases are possible using a specialized computer architecture for manipulating binary decision diagrams.

DESIGN AND EVALUATION OF A SPECIALIZED COMPUTER ARCHITECTURE FOR MANIPULATING BINARY DECISION DIAGRAMS

by

ROBERT K. HATT

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL ENGINEERING

Portland State University
2000

DEDICATION

This thesis is dedicated to my wife Carol McConnell for encouraging me to do something I had only talked about for years, and for helping me with this opportunity to pursue it to completion. Even more thanks to Carol for putting up with missed evenings and weekends while I was doing all of my school work. Also to my daughter Maggie for being born in time to make my life more challenging and special than ever before.

Table of Contents

List of Figures

List of Tables

Design and Evaluation of a Specialized Computer Architecture for Manipulating Binary Decision Diagrams

Chapter 1
Introduction

Modern integrated circuit (IC) technology has enabled designers to build circuits with millions of transistors (logic gates). In 1984 a pair of designers might have sat in front of a computer terminal to capture and simulate a netlist or schematic representation of an application specific integrated circuit (ASIC) containing seven thousand gates. Today small teams of engineers use logic synthesis and simulation to design circuits with hundreds of thousands or even millions of gates. The problem of creating and verifying these large circuits has pushed current computer technology to its limits. Logic simulation alone can no longer be used for complete verification of these ICs, other efficient representations of these circuits are necessary to complete the required design, analysis and verification. Binary Decision Diagrams are one such critical representation.

Reduced Ordered Binary Decision Diagrams (ROBDDs or just BDDs) are an efficient data structure for the representation of many (large) boolean functions. As such, they have become a very important tool for use in design and verification of logic. In 1986 Bryant [Bryant86] described algorithms for efficient manipulation of BDDs. The implementation of these algorithms has led to the use of BDDs for problems ranging from logic decomposition [Bertacco97] [Chang96], logic synthesis [CYang98], formal verification[Bryant95], test generation, and graph manipulation

[Cortadella99] [Sekine97]. They are used in all modern commercial synthesis and logic verification tools.

The performance of Binary Decision Diagram algorithms on a general purpose computer has improved since the first libraries of functions were written, and they have been used to solve many difficult problems. Algorithm performance varies depending on the BDD library package and the type of problem being solved. Different types of applications of BDDs may show different efficiency with different packages because the package implementation varies. Overall the BDD can be considered one of the successes of design automation research. Yet, despite the efficiency of BDDs for representing many types of circuits, there are many types of circuits where the BDD representation is exponential in size. This problem cannot be overcome easily as it is an inherent limitation of the ROBDD data structure. Consequently other types of decision diagrams, and representations, have been developed in attempts to address these types of circuits [Becker97][Narayan98][Minato96].

Current general purpose engineering workstations are typically limited by the fact that they are organized as 32-bit word architectures and operating systems are restricted by this word length constraint. Application programs run on these general purpose computers cannot exceed the limits imposed by the operating system. Because of the cost and general efficiency one computer is used for a large variety of problems, but for special applications this generality may limit the ability to solve large problems in the most efficient manner.

Specialized computer architectures have been created to solve many problems, often in response to the capacity and performance limitations of general purpose computers. Vector processing machines (often called super computers) from companies like Cray Inc., and computers to execute LISP programs from companies like Texas Instruments and Symbolics Corporation are some of the most common examples. A specialized computer for manipulating Binary Decision Diagrams should be designed and evaluated in an attempt to address performance and capacity when compared to implementations on general purpose computers. This may also give additional insight into the behavior of BDD algorithms and how they could be improved.

The following chapter will give an introduction to Binary Decision Diagrams. It will discuss some of the primary issues involved in creating and manipulating BDDs. Chapter 3 will discuss performance issues involving BDDs and their different types of applications. Several freely available BDD packages are described and the BuDDy package, which is used as an example of a typical BDD package, is described using an example N-queens problem. Chapters 4, and 5 will describe the specialized computer design, simulation models and implementation issues involved with the design of the specialized architecture. Finally, chapter 6 will discuss the results of the simulations and describe the performance in comparison to a general purpose computer. This will allow some conclusions to be drawn and future design considerations to be described.

Chapter 2
Binary Decision Diagram Background

2.1 Introduction to ROBDDs

A Reduced Ordered Binary Decision Diagram (ROBDD or just BDD) is a directed a-cyclic graph which represents a boolean function. The graph begins with a single root node. Every path starting from the root node will end at a terminal node which represents either 0 or 1. Each non-terminal node is labelled with one of the variables in the function and has two outgoing edges. The edges point to nodes which represent the negative and positive co-factors of the function respectively. Shannon (and Boole) found that a boolean function can be described in terms of two sub-functions (called cofactors) when a specified variable is either zero (negative cofactor) or one (positive cofactor). The Shannon expansion is often described by the following formula

Figure 1. Shannon Expansion

$$f = x \cdot f|_{x=1} + \bar{x} \cdot f|_{x=0}$$

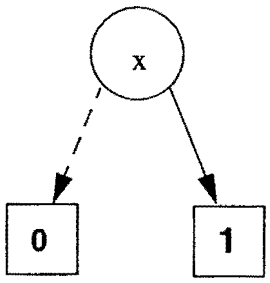
Figure 2. Shannon Co-factors of a Boolean function

For the boolean function

$$f = a \cdot b + \bar{a} \cdot c$$
$$f|_{a=1} = b$$
$$f|_{a=0} = c$$

The BDD for the function of one variable x is shown in Figure 3 on page 7. For variable x, this function is 0 when x is zero (the negative cofactor) and 1 when x is one (the positive cofactor). Just expanding a function with respect to the co-factors of a function will give a decision tree as shown in Figure 4 on page 8. To make a Reduced BDD duplicate nodes are not allowed. Duplicate nodes are defined as nodes which are labelled with the same variable and whose outgoing edges point to the same nodes respectively (both lo edges point to the same node, also the corresponding hi edges also point to a common node). In Figure 4 it can be seen that nodes labelled 4 and 5 meet this criteria. Also all duplicated terminal nodes should be combined so that there are only two terminal nodes; one representing the constant 0 the other representing constant 1. Therefore nodes 8,10,12,13 are combined and nodes 9,11,14,15 are combined. This results in the graph in Figure 5 on page 8.   Finally any node whose edges both point to the same node are removed from the graph and the incoming edges are redirected to the destination of the outgoing edges which were just removed. All of the steps assume the ordering of the input variables (used by the function) do not change in the BDD. This results in the creation of a Reduced Ordered BDD.

Figure 3. BDD for a single variable



Reduced and Ordered are the most important structural features of BDDs because this makes the ROBDD a canonical representation. This canonicity property means that given two ROBDDs for the same function (using the same variable ordering) the graphs are isomorphic and functional equivalence can be easily tested.

It is not feasible to first build a decision tree and then reduce it because it would require a number of nodes exponential in the number variables ($O(2^N)$ where N is the number of variables in the function) to create the original decision tree. Only after Bryant [Bryant86] described recursive methods of performing operations on BDDs did they become a useful data structure for representing and manipulating boolean functions.

Only unique nodes are stored in the tree. Thus if two nodes are to represent the same function (variable and co-factors) they will be represented by the same node. There is no redundancy in the tree, all nodes are unique

Figure 4. Decision Tree for a function

Table 1: Truth Table for F=a*b+a*c

a	b	c	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

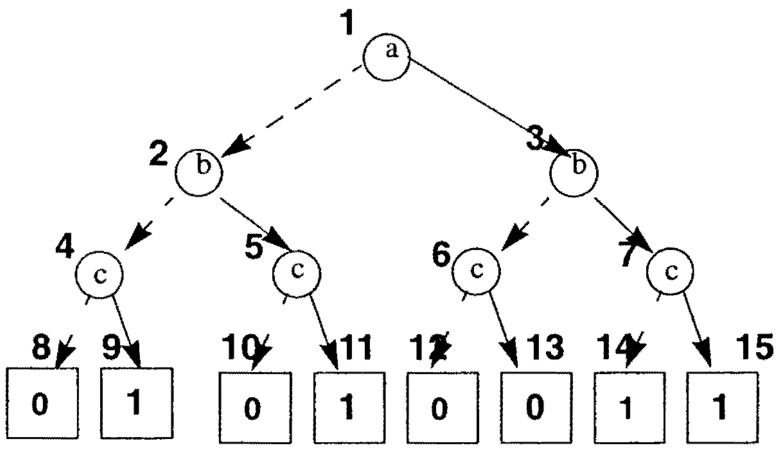
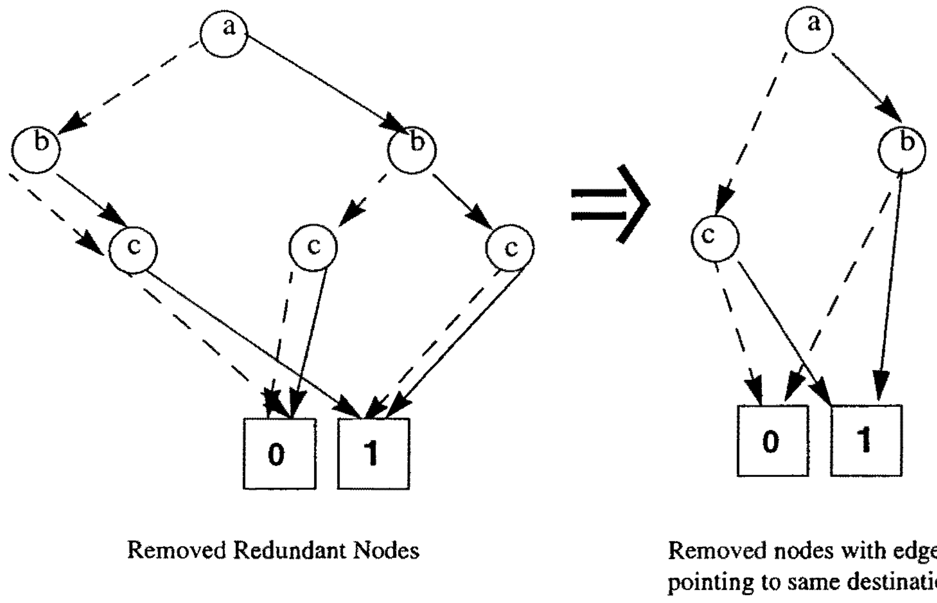


Figure 5. Reduced Ordered BDD

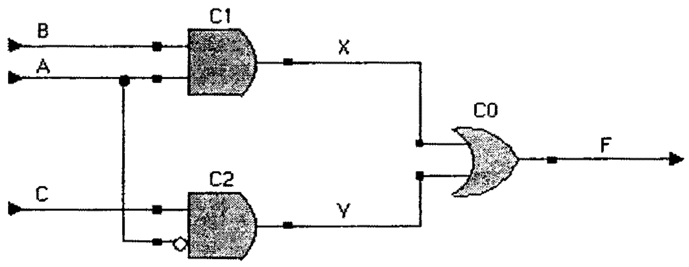


Bryant's algorithms for building BDDs are based on recursive operations over the BDD data structure. The core operator is called the If-Then-Else operator

$\text{ite}(F,G,H) = F*G + \overline{F}*H$

It can be used to build all two variable boolean functions. For example to create a BDD to represent a function f = a*b + $\bar{a}$*c would only require one call to the ITE operator f = ite(a,b,c). But if this same function were given as a gate level schematic, the function must be derived by walking the netlist and building the function iteratively.

Figure 6. Schematic Representation of the function f = a*b + $\bar{a}$*c



To create a BDD for the function f = a*b + $\bar{a}$*c would require creation of a single node BDD for variable (function) a, b, and c. Then intermediate functions shown in Figure 7 must be created to achieve the desired function f.

Figure 7. Netlist Representation of the function f = a*b + $\bar{a}$*c.

```
Input a,b,c;
Output f;
f2 = a ANDb
f3 = NOT a
f5 = f3 AND c
f = f2 OR f5
```

2.2 The importance of variable ordering.

The size of the BDD is highly dependent upon the ordering in which the variables are represented in the tree. Minimum size representations can always be found using known algorithms [Drechsler98], but the computation time to find the minimum representation is often unacceptable because the worst case representation of a boolean function using a BDD is exponential in size. As shown in Figure 8 on page 10. the BDD representation for the boolean function $f = a * b + \overline{a} * c$ (which might be recognized as a multiplexor) is represented differently depending on the chosen variable ordering. The dotted lines indicate the low edge (negative cofactor where the variable = 0). The ordering a<b<c in Figure 8a requires only 5 nodes, the ordering c<b<a in Figure 8b. requires 7 nodes.

Figure 8. BDD Variable Ordering

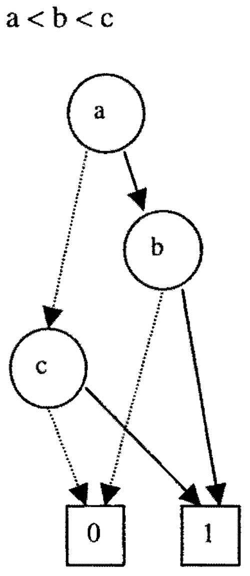


Figure 8a.

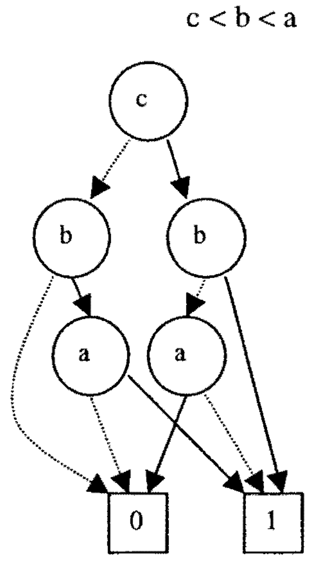


Figure 8b.

A second example of the size effects of variable ordering shows the number of nodes used to represent the different outputs of an eight bit adder. The number of nodes for each adder output bit are given for a good variable ordering and a bad variable ordering in Table 2 on page 11. It can be seen that for the most significant bit of the eight bit adder (16 input variables, 8 output variables) a good variable ordering gives a BDD of only 24 nodes and a bad variable ordering gives a size of $O(2^{(N/2)+1})$ where N is the number of outputs in the add function.

Table 2: Nodes in an 8+8 adder

Output Bit	**Good Order**	**Bad Order**
0	3	3
1	6	7
2	9	15
3	12	31
4	15	63
5	18	127
6	21	255
7	24	511

It is not always possible to know before creating the BDD what will be a good variable ordering. For example when reading a function from a circuit netlist, it may be very difficult to determine a good ordering because the function is not known in advance. It is possible to choose an ordering and create the BDD, then try a different ordering and create another BDD and determine which is smaller, but this would mean that N! (N factorial) orderings must be tried to find the best ordering (with respect to the size of the BDD)

Rudell [Rudell93] found that variables adjacent in the ordering can be swapped without affecting the other levels of the BDD. This idea that variable swapping is a local operation led to the variable reordering algorithm called sifting. Sifting can be done dynamically while the BDD is being created to try to keep the BDD size manageable. Sifting may require exponential run time to find a minimum BDD but it is often not necessary to try all possible orderings because a minimal BDD is not required, the BDD size must be kept small enough to be managed efficiently for the available computing resources and problem.

2.3 Garbage Collection

As BDD algorithms execute, while keeping the data structure canonical, they call many operators whose results are used only temporarily. The nodes generated during these operations which are no longer an active part of the data structure and are not referenced (visible) by any root nodes are called garbage nodes. Garbage collection is used to retrieve these nodes and put them back on to the list of available (free) nodes. Garbage collection is necessary when the number of free nodes nears exhaustion, and can consume significant CPU time. Because of this performance penalty it is important to perform garbage collection only when necessary.

ROBDDs can be efficient data structures for manipulating boolean functions. Canonicity must be maintained through all operations. Efficiently building and maintaining BDDs to meet those considerations is an important factor in the use of BDDs in many applications. The following chapter describes the main aspects of algorithm implementation for BDDs and their performance considerations.

Chapter 3
BDD Algorithm Implementation and Performance

3.1 Background

In 1990 Brace, Rudell and Bryant [Brace90] described an efficient implementation of a BDD package which has been the basis of most subsequent BDD packages. The main features of an efficient BDD package as described by Brace et. al. are:

- A unique table for the efficient creation of new BDD nodes and making sure all created nodes are unique.
- A computed cache to store intermediate results of operations for use during recursion using the ITE and other operators.
- Efficient garbage collection to recover nodes that are no longer referenced
- Good dynamic variable reordering heuristics to be used while BDDs are being manipulated

Because BDDs are an efficient representation of boolean functions in many cases and exponential in others, much work has been performed on refining BDD algorithms for fast execution and minimum memory use on general purpose processors. It has been intensely studied by various implementers of BDD library packages. The algorithms described by Bryant were based on efficient traversal of the BDD recursively in a depth first recursive manner. Many variations of these methods have been implemented in attempts to create more efficient packages. Most packages give different performance depending on the actual size and the problem being solved. No single approach has been proven best for all problems.

Many different aspects of BDDs have been studied in attempts to find more efficient ways to manipulate them on general purpose processors. Specifically, memory reference locality [Manne97], cache effects and memory paging [Klarlund96][Long97] have been the target of improved BDD packages. Breadth first manipulation has shown improvements on some problems [Ranjan96a]. Chen [Chen97] implemented a hybrid approach combining breadth-first and depth-first BDD manipulation and showed performance improvements over both depth-first only and breadth-first only implementations. Parallel BDD packages that can run on networks of workstations have been created [Milvang98][Stornetta96]. All of these approaches incorporate design trade-offs in an attempt to create packages which can manipulate large BDDs faster.

Comparisons of various packages and their efficiency for various types of problems have also been made; [Sentovich96]. Yang et. al.[BYang98] studied various aspects of BDD performance as applied to symbolic model checking algorithms. The study included computed cache replacement policies, garbage collection frequency as well as variable ordering. They have shown that a larger computed cache size can have a much greater effect on model checking computations than for building BDDs for combinational circuits. Also, because garbage collection of un-referenced nodes can be time consuming, it was shown that model checking computations have a large rebirth rate (i.e. nodes that are un-referenced will become references again later in the computation). This led them to conclude that garbage collection should occur less frequently. Additionally they suggested that the combined breadth-first and depth-first approach might lead to additional efficiencies.

All of this research has applied to improving algorithms on general purpose processors. Because the speed of these processors and the availability of large memories has become more affordable during the past decade continually larger problems can be addressed using BDDs. There are limitations to 32bit computer architectures and operating systems that prevent them from solving extremely large problems. As 64 bit processors and operating systems with large main memories become more available, larger problems will be solvable by a general purpose architecture.

Generally a special computer architecture is used for two reasons, capacity and performance. With the advent of 64-bit word architectures and operating systems, capacity may no longer be an advantage of a specialized computer. This only leaves performance as a realistic improvement provided by a special purpose BDD computer. Attempting to build a special purpose computer may give insight into the behavior of BDD algorithms and provide a platform for the analysis of architectural trade-offs for different architectures and BDD algorithms, much as an instruction set simulator might for a general purpose computer. No published research has been found on the study of any specialized hardware implementation of BDD algorithms. The remainder of this thesis will be devoted to the description of a special purpose computer architecture for manipulating binary decision diagrams.

3.2 General Goals

A specialized computer for evaluating BDDs would enable the efficient solution of BDD problems than can be accomplished by a general purpose computer. To make the effort to use a specialized computer desirable it must be able to solve larger problems faster than existing BDD packages at a reasonable cost.

The use of the specialized computer should be transparent to the user. Compatibility of the procedural interface package with an existing BDD package will allow easy porting of existing code. The cost of the specialized hardware must be commensurate with the size of the problems it can solve. In other words, it should be inexpensive compared to not being able to solve the problem in a reasonable time. Achieving these high level goals would allow easy adoption of the specialized hardware. The useful life of the specialized computer must be such that the investment can be justified vs. next years general purpose computer. These goals may be unobtainable, but that is the purpose of this research, to find out if it is reasonable to expect to achieve the necessary performance to make a specialized computer worthwhile.

A high level look at the overall architecture will give a picture of the components that are required for such a system.

3.3 Choosing a BDD Package

It is not the goal of the paper to compare many BDD packages to find the best performing package. Nor is it the purpose of this project to create an entirely new package for use on general purpose computers. All work will be done based on an existing package and procedural interface to create a subset of the package functions that can be used to verify the performance of the proposed BDD computer architecture. Several packages available from universities have become widely used because they are robust and efficient. They all use various implementations to try and achieve improved performance and memory usage. Below, three publicly available BDD packages, CUDD, CAL and BuDDY will be briefly presented.

3.4 CUDD

A large, comprehensive, robust and efficient library developed by Fabio Somenzi at the University of Colorado at Boulder. It uses complement edges in the internal representation and depth first recursive ITE algorithms. It is very smart about compacting the BDD node into 16 bytes and making sure all nodes are 16 byte aligned. This helps when fetching things from memory on most 32 bit processors which often have 32 byte cache lines. Support for many heuristics for dynamic variable ordering and automatically adjusting cache sizes are built in. It also supports other kinds of functional decision diagrams not discussed here.(ZBDDs, FDDs)[CUDD98]

3.5 CAL

Similar to CUDD in its use of 16 byte (aligned) nodes, this package uses breadth first recursion during most algorithms. It has slightly more complicated access to internal data structures because of the breadth first recursion, but by storing all nodes of a single variable contiguously in memory, the breadth first search has good memory locality access (fewer cache misses), [CAL97]. It has been the basis for several BDD packages designed to run in parallel on multiple workstations including [Milvang98].

3.6 BuDDy

A general BDD package with all of the required features of an efficient BDD package including garbage collection and several dynamic variable ordering heuristics. Includes vector operations for word level operations on BDDs, [BuDDy99].

The BuDDy library of BDD functions was chosen as the BDD library to base the performance comparisons and implementations for this project. Though probably not the fastest BDD package, or the one that consumes the least memory, it is claimed by the author to be as fast as David Long's (CMU) original package (CUDD and CAL claim to be faster than that package as well) and the code is very well documented and readable. It is an excellent tool to study and learn about BDD algorithms. The code is clear, concise and more consistent than the other two packages that were looked at. It is for these reasons that the BuDDY package was chosen for the analysis and as a basis for the BDD algorithms to be modeled.

3.7 Computing Environment

The computing environment used as the basis for all data and statistics gathered in this paper is a dual Intel Pentium II Xeon with 1MB full speed 2nd level cache, 128MB RAM and 80Mb/sec. SCSI disk drive. The operating system is Red Hat Linux

6.1 (SMP kernel). The compilation environment is the GNU C/C++ compiler, gcc version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release) provided with the Red Hat Linux installation. All benchmarks are single threaded and were run on a single processor with no other non-operating system processes running.

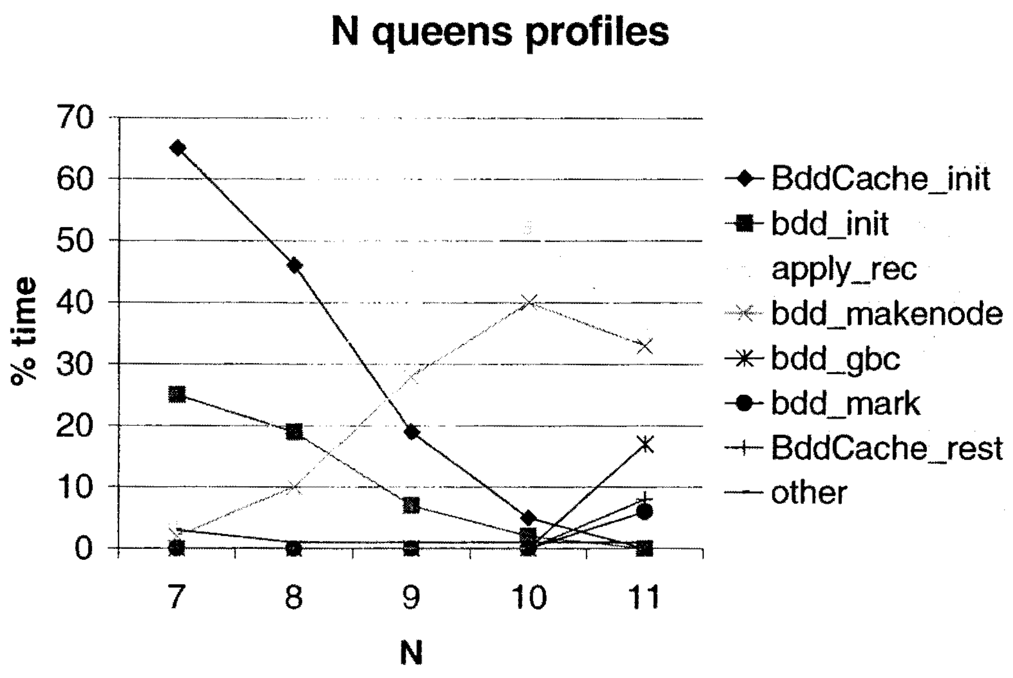
3.8 Performance Measures

Code profiling is a method for finding the percent of total CPU time spent in each function of a program. It is a feature of the compiler used on the general purpose computer and is easily turned on with a single compiler switch. Approximate CPU time and exact number of calls is collected for each function that is profiled. It should be run with a variety of test cases to gain insight into where a program may need to be optimized.

Several sample (simple) programs were run and profiled. None of these programs is large, in the sense that they require large amounts of memory or computation time. They are intended to be a few test cases representative of average BDD computations.

- N-queens - The classic constraint problem written using BDDs as the data structure. Place N queens on an NxN chess board such that no queen can capture another.
- Reachability - Generate random state machines and perform reachability analysis This is a typical model checking type verification task to make sure design constraints are met in possible states.
- State Minimization - Read a description of a state machine, find equivalent states and combine them. Write the results out to a kiss file.

Figure 9. N Queens Profiles



The N queens program was run for several different values of N ranging from seven to eleven. These are the runs to completion in main memory. The profile was examined and the percentage of time spent in the BDD library functions was recorded. The top seven contributors to CPU time usage are shown in Figure 9 on page 21, all other functions are lumped into the others category. For all runs the number of nodes initially allocated was 1,000,003 and the computed cache size was set to 500,009. These numbers are prime numbers and the hash function used by BuDDy is modulus based which works best with prime numbers [Aho86]. The function BddCache_init is actually a constant time when the initial number of nodes is constant. Thus it is a much larger percentage of execution time in small benchmarks and a negligible percentage for the larger values of N. For most realistically sized problems, it is assumed the BddCache_init time will be a very small percentage of the problem. As the problem size grows, so does the amount of time spent in apply_rec, bdd_makenode. Only if the initial number of allocated nodes is exceeded is garbage collection invoked. This occurred when N was equal to eleven. It can be seen that the garbage collection related functions bdd_gbc, bdd_mark, BDDCache_reset were negligible before N was eleven, but were large users of CPU time for N equal eleven.

The results of profiling these programs is given in Table 3 on page 22. Only functions that contributed more than 5% of the CPU time to at least one of the test case runs are included by name, the remaining functions are accumulated in the category labeled others. The functions bdd_gbc, BddCache_reset, and bdd_mark could be combined under a heading titled garbage collection.

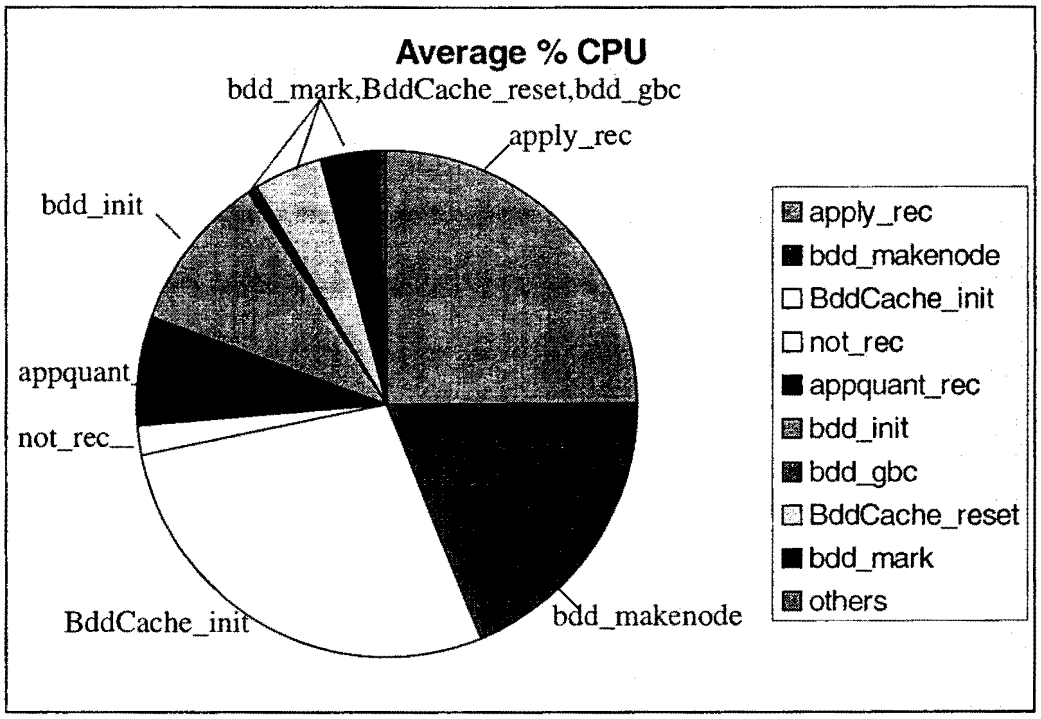
Table 3: Sample Program Profiles

	average %CPU for 5 runs			
Function Name	**N queens**	**State Minimization**	**Reachability Analysis**	**Average**
apply_rec	32.2	7.6	31.2	23.6
bdd_makenode	22.6	4.2	26.8	17.8
BddCache_init	27	42.2	10	26.4
not_rec	5			1.6
appquant_rec		20.2		6.7
bdd_init	10.6	15.8	2.2	9.5
bdd_gbc	3.4		9.4	4.3
BddCache_reset	1.6		8.8	3.5
bdd_mark	1.2			0.4
others	1.4	8	10.6	6.6

The chart in Figure 10 on page 24 gives a good view of where time is spent. The functions apply_rec, bdd_makenode, and garbage collection constitute slightly over 50% of the CPU time used by these programs. The reachability analysis tests spend most of the CPU time using apply_rec to create the internal state machine representations. Also they show the use of garbage collection because the number of nodes created during execution exceeded the number of nodes initially allocated. The appquant_rec, which is the existential and universal quantification operations is lumped in with the "other" category, though for reachability this is also one of the primary operations. It is important to note, that the numbers will vary depending on the arguments for the programs, the number of nodes in the initial node table and the number of slots in the computed cache. In general, if too few nodes are used initially, garbage collection and marking operations will become large portions of the execution time. This is an important point, because garbage collection will become extremely important for operations on large BDDs. The term large is relative, for example if the BDD computer has a large memory then garbage collection may not be as important until the problem starts to fill the node memory.

Also important here is that all of these problems have been programmed using a known good variable ordering appropriate for the problem. Many real world problems may not have a known good variable ordering and depend on heuristics and dynamic variable ordering to keep the BDD sizes manageable. Because of the known good ordering for these problems the dynamic variable ordering was not used in any of the test cases.

Figure 10. BDD function execution time



The functions BddCache_init, and bdd_init show up as large percentages because they dominate the small test case runs. In larger runs, as was seen with the N queens problem, they are actually negligible compared with the other functions.

This Chapter has given background information about BDD algorithm performance research. Several BDD packages were described and the choice of one package as a basis for further research was given. The computing environment used for gathering the performance information was described and the performance of several BDD algorithms was evaluated. It was shown that on a general purpose processor most CPU time is spent in the recursive manipulation (apply_rec) and creation (bdd_makenode) of BDDs. Garbage collection time is also significant as the problems become large with respect to available memory. This performance information is important to the remainder of the thesis because it helped guide the design choices described in subsequent chapters. The remainder of the thesis will describe how the major parts of the BDD algorithms were modeled. Why the different design choices were made and describe the results of the simulations of the BDD Computer architecture.

Chapter 4
BDD Computer Macro Architecture

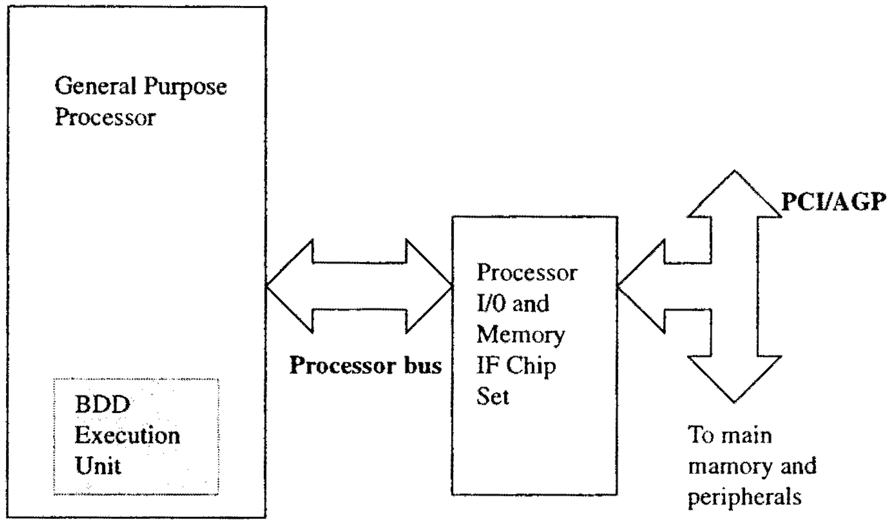
4.1 Introduction

Within the context of accelerating execution of BDDs using special purpose hardware, the operation of the BDD processor on the BDD itself can be designed independently of a general purpose computer. Yet the chosen architecture will interface to a general purpose computer for all non-BDD related operations. The level of integration and how the BDD processor interfaces to the general purpose computer varies and the different interface architectures have unique performance characteristics. This chapter describes the different macro-architectures and the issues associated with the design of each.

4.2 Integrated BDD Processor

The integrated architecture shown in Figure 11 on page 27 assumes that a BDD execution unit could be integrated directly into the general purpose processor. The BDD execution unit would share resources with the processor as well as using the same memory hierarchy. A new memory system and interface does not have to be designed. BDD operations can be implemented with special instructions that are an extension of the processor instruction set. The on chip integration means the BDD instructions will execute with the same performance as the processor. Therefore as the processor semiconductor technology improves and performance increases so will the BDD instructions see similar improvement.

Figure 11. Integrated BDD Execution Unit


General Purpose Processor
BDD Execution Unit
Processor bus
Processor I/0 and Memory IF Chip Set
PCI/AGP
To main mamory and peripherals

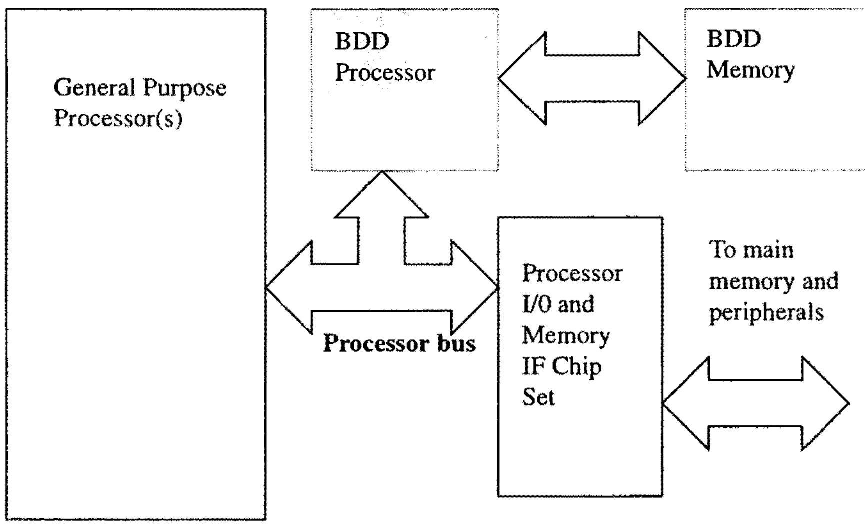
There are several disadvantages of this architecture. The first is the integration with the general purpose processor. This is a difficult design issue and it requires knowledge of the processor micro architecture and IC technology being used. The compiler technology must be crafted to take advantage of the BDD instructions and programs must be recompiled to take advantage of these instructions. Alternately a special library of hand crafted routines could be provided that could be called by applications programs that need to access BDD instructions.

As was described in Section 3, "BDD Algorithm Implementation and Performance," on page 14 the algorithms for BDDs are memory access intensive manipulation of nodes and edges in the BDD. Thus sharing the processor main memory offers no capacity benefits, and probably no memory access benefits. Finally this design is not feasible for anyone but a large microprocessor design company and, because it is not a large volume application, is not worth the engineering effort required for implementation.

4.3 BDD Coprocessor

The second architecture is shown in Figure 12 on page 29. The BDD Coprocessor design features the BDD execution unit as a device attached to the processor bus. Because the unit is attached to the processor bus, there is a larger latency to execute a BDD instruction than with the integrated execution unit. This design is more complex but is more feasible. Increases in complexity arise because the BDD coprocessor does not share resources with the processor, must track the bus transactions, and has its own memory which is separate from the processor's main memory. Some of these complexities may be necessary to achieve high performance BDD manipulation. The memory structures can be customized to specifically improve performance for BDDs. BDD execution can take place in parallel with other programs running on the general purpose processor. The main disadvantage of this architecture is the increased design complexity and the added latency through the processor bus to execute a BDD instruction. If implemented as an ASIC with a general purpose processor core and additional BDD functionality the BDD computer could be on the same chip with the GP processor. This architecture might then be constrained due to I/O limitations of the ASIC package because of large data and address busses for both the general purpose processor and the BDD processor.

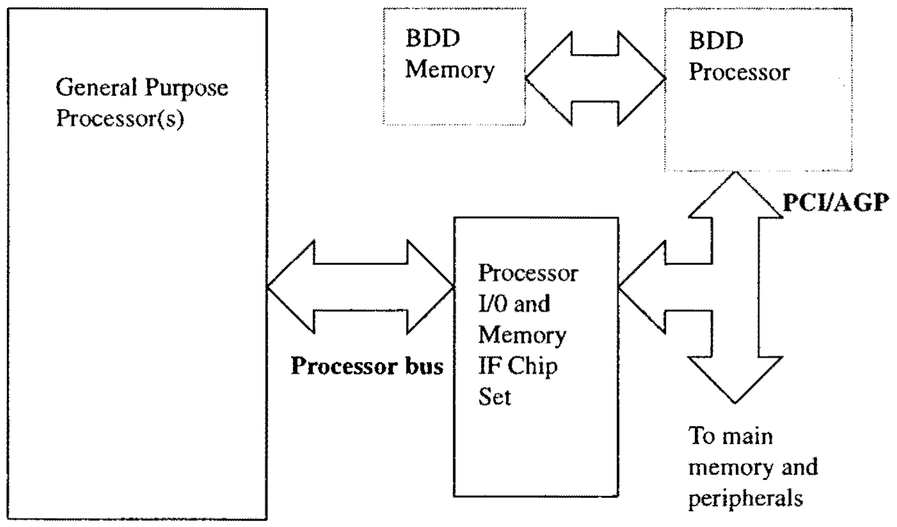
Figure 12. BDD Coprocessor


General Purpose Processor(s)
BDD Processor
BDD Memory
Processor I/0 and Memory IF Chip Set
To main memory and peripherals
Processor bus

4.4 BDD Peripheral processor

The architecture shown in Figure 13 on page 30 is very similar to the BDD coprocessor, but since the BDD computer resides on the PCI (or AGP) bus instead of the processor bus, the latency for the processor to issue a BDD instruction is much higher. It is also less predictable due to many more possible peripherals on the expansion bus. This architecture has all of the execution benefits and most of the design complexity of the coprocessor design, though the PCI/AGP interface is a slightly easier interface design.

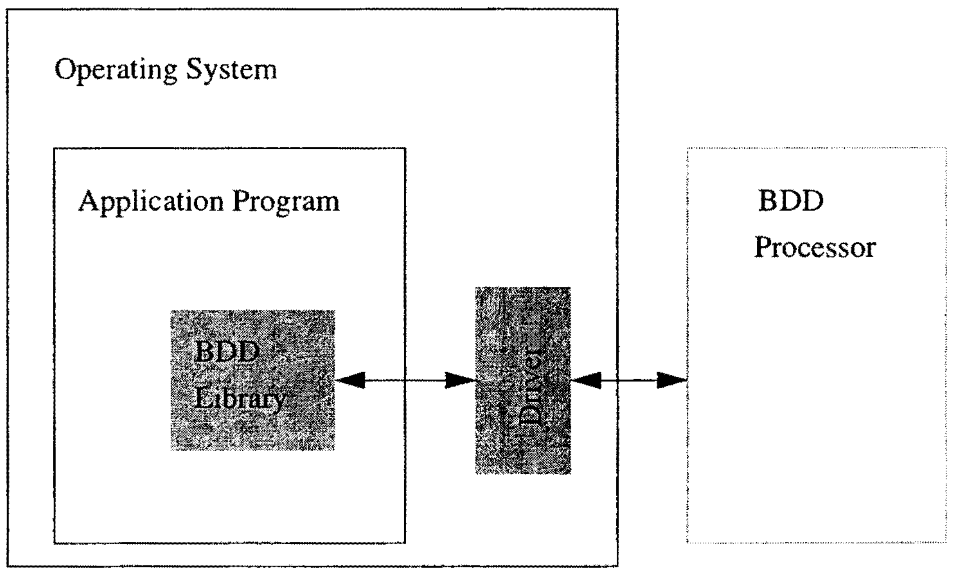
Figure 13. BDD Peripheral Processor


General Purpose Processor(s)
BDD Memory
BDD Processor
PCI/AGP
Processor bus
Processor I/0 and Memory IF Chip Set
To main memory and peripherals

4.5 The Software Architecture

In addition to the hardware architecture the BDD processor must be easily accessible from software. In order to make the BDD functions easily integrated into existing programs, a library of functions will be provided to the programmer. The BDD library will look like a normal software library to the user, but will interface to the BDD processor hardware. See Figure 14 on page 31. The underlying structure of the hardware can change without change to the software library interface. This software architecture allows changes in the underlying software/hardware structure without affecting the user application program.

Figure 14. Software Architecture



Each of the different macro architectures will have different access latency to transfer data between the general purpose processor and the BDD processor. Generally the further the BDD processor interface sits from the processor, the more software overhead will be involved in accessing it. This is discussed along with the performance results in Section 6, “Performance Evaluation,”

4.6 Memory Subsystem

The memory subsystem of the BDD processor is one of the main issues to consider during design. As was shown in Section 3, “BDD Algorithm Implementation and Performance,” the algorithms used to manipulate large decision diagrams have memory access patterns that are difficult to predict. Memory references often cause cache misses in a general purpose processor which means access to main memory is required. Also, the implementation of the unique and computed tables are amenable to different types of memory structures. Because commercially available memories are designed with specific applications in mind, a number of types of memory were considered for the different memory subsystems in the BDD processor architecture.

4.6.1 SRAM, SSRAM

Static RAM (SRAM) offers the highest performance in off the shelf memory components. SRAM with access times of 7.5 ns and below are available. Synchronous SRAM (SSRAM) uses a clock signal to latch the address and data signals thus making the interface to synchronous systems easier. Capacity is an issue for very large memories as SRAMs typically only store about 8Mb per chip.

4.6.2 DRAM, SDRAM, RDRAM

Dynamic RAM (DRAM) offers high capacity and is the least cost per bit memory available. There are a number of different variations that offer high performance and synchronous operation. Synchronous DRAM (SDRAM) has registers to hold output data and input signals to allow synchronous interface to clocked systems. The burst mode in the SDRAMs has been designed to effectively interface with cache memory systems that load data from several sequential addresses on successive clock cycles. Direct Rambus DRAM (RDRAM) is designed for very high speed synchronous burst access up to 800 MHz. The high speed of the RDRAM makes for a difficult design. Because DRAMS can store 128Mb per chip, typical general purpose server computers can be configured with gigabytes of DRAM. The latency for typical

DRAM is 60ns, much larger than static RAM, but once a burst is begun DRAM offers performance nearly the same as the faster SRAM.

4.6.3 CAM

Several commercial options are available for content addressable memory. Several companies (NetLogic Microsystems and Lara Technologies are two) offer CAM devices targeted at network switches which might be usable for other applications. The capacity of these devices is modest compared to SRAM and DRAM.

NetLogic SyncCAM-2 is available in 32k x 144 bit organization with speeds up to 100 MHz. Lara Technology offers similar features. The depth can be increased beyond 32k, but access must be pipelined thus increasing the average latency to find a match. UTMC offers a UTCAM-Engine product which is an IC designed to turn SSRAM or SDRAM into content addressable memory. The performance is lower than for the dedicated CAM, but also offers the opportunity for designing larger CAMs using less expensive memory components.

4.6.4 Memory Performance Summary

The cost per bit of the different types of memory varies significantly. Dynamic RAM with the least expensive cost per bit, but with the worst latency. CAM is the most expensive memory but the latency is size dependent. Static RAM cost is in the middle and has the shortest latency of the three types of memory proposed. New variations of DRAM such as RDRAM and Double Data Rate SDRAM offer higher clock rates and bandwidth than conventional SDRAM memory, but latency is generally not improved. These effects of specific types of DRAM have not been studied in this thesis, but cost must be taken into account when designing a large memory system.

4.7 Architecture and Memory Issues

Each of the architectures described in this chapter offers different complexities and design issues. They have been summarized in this chapter and specific performance numbers will be given in Chapter 6, "Performance Evaluation". Table 4, "Macro Architecture Design Trade-offs," on page 34 shows the different architectures and the relative design complexity of each.

Table 4: Macro Architecture Design Trade-offs

Architecture	**Interface Design Complexity**	**Instruction Latency**	**Capacity Effect**	**Estimated Design Cost**
General Purpose Processor	NA	NA	Limited by OS	NA
Integrated BDD Execution Unit	Very High	Very Small	None	Very High
BDD Coprocessor	High	High (~= GP main memory)	UnLimited	High
BDD Peripheral processor	Average	High (2-3x GP main memory)	UnLimited	Average

This chapter has served to introduce some of the general cost and complexity issues that are involved with the design of the BDD processor. The following chapters will describe the design specific choices that were made in this thesis.

Chapter 5
BDD Micro Architecture and Hardware Models

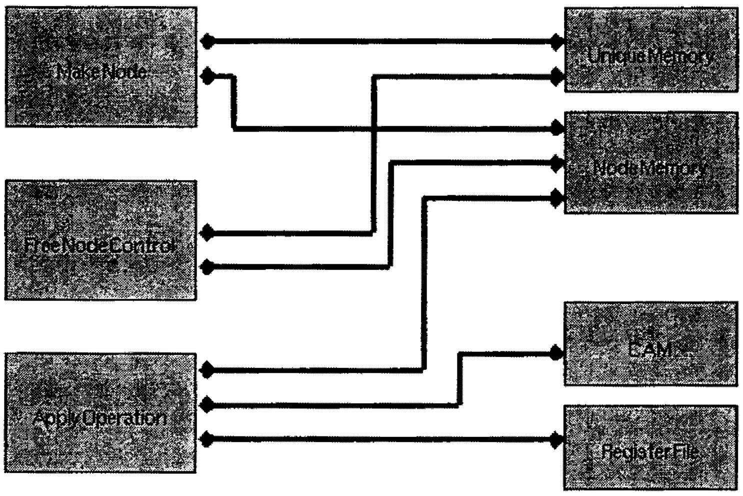
5.1 Simulation and Modeling Environment

To create and evaluate the model of the BDD Processor a combined 'C' and VHDL simulation environment will be used. This environment will allow for the interaction of applications programs with the simulated BDD Processor architecture to determine performance on actual application programs that manipulate BDDs. This can then be compared to the performance obtained by running the application on a general purpose computer. The BDD processor will only accelerate the actual manipulation of the BDD library calls. The performance of the BDD processor is measured for specific time based on the number of BDD clock cycles and clock period for the portions of the program that will be accelerated. These times will be substitutes for the measured percentage of the program that is run on the general purpose processor. Then a comparison of the application run solely on the general purpose architecture with the application run using the BDD processor will be given.

5.2 Processor Model

The model has been designed to implement the BDD algorithms based on the BuDDy package. It is an unsophisticated implementation that attempts to use as few clock cycles and to make as many memory accesses concurrently as possible. The models are being used as an analysis tool to estimate the performance of a simple hardware implementation and may not represent the best implementation.

Figure 15. BDD Processor Functional Units



5.3 Memory Models

Several memory models were designed so that the different types of memory could be tested with a variety of latencies. The models have an abstracted interface so that the latency can be easily changed. The node table, unique table and CAM were designed with an asynchronous interface. A request is received on the input and when the memory has completed the request, an acknowledge signal is sent back to the requesting unit. As can be seen in Figure 15 on page 36, the memories also have multiple ports. The memory models use a simple round robin arbitration scheme to allow access to the memory. This method guarantees that each functional unit will get access to the memory in the order the request was received. If multiple requests are received simultaneously they are serviced in a priority order with the most recently serviced port being the lowest priority.

5.3.1 Node Memory

The node memory is the main memory of the BDD processor. It needs to be very large. This is the most critical capacity/performance trade-off in the system. Fast access to the node memory is required to achieve high performance. The goal of any BDD system is efficient node access, which means keeping the memory busy fetching nodes so the processing can proceed as quickly as possible. The node structure shown in Figure 16 on page 38 shows the intended bit widths of the different fields in the node structure and the VHDL record used to represent it. The field widths are chosen based on expected capacity requirements and memory availability; DRAM modules are typically 64 bits wide. The 32 bit hi-edge and lo-edge fields will allow four billion nodes if enough memory can be installed in the machine. This far exceeds the approximate 256 million node capacity of a software package which uses 28 bit node address fields because nodes are typically 16 bytes in size. Unused bits will be used to expand (if necessary) existing fields and for future enhancements.

Figure 16. Node structure

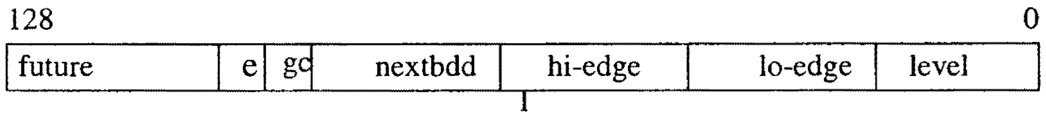


Table 5: Node Memory Fields

Field	Size
level	20
lo-edge	32
hi-edge	32
nextbdd	32
gc	2
e(xternal)	1
future	7

```
type bdd_t is record
  level : bddvar;
  lo  : bddhandle;
  hi  : bddhandle;
  nextbdd : bddhandle;
  gc : gc_t;
end record;
```

5.3.2 CAM (Computed Cache) memory

The computed table can be implemented in several ways. It was implemented as if it were a true content addressable memory. Access is by content, with the result value returned not as an address, but as the result value stored at the found address. When the CAM is full it begins a FIFO overwrite, so that the first cell that was written to the memory is the first one to be overwritten. It can also be thought of as a circular buffer that once it is full starts writing at the beginning again. The CAM must be wide enough to hold the arguments to the BDD apply function, and must return a result value the size of a node handle (32 bits).

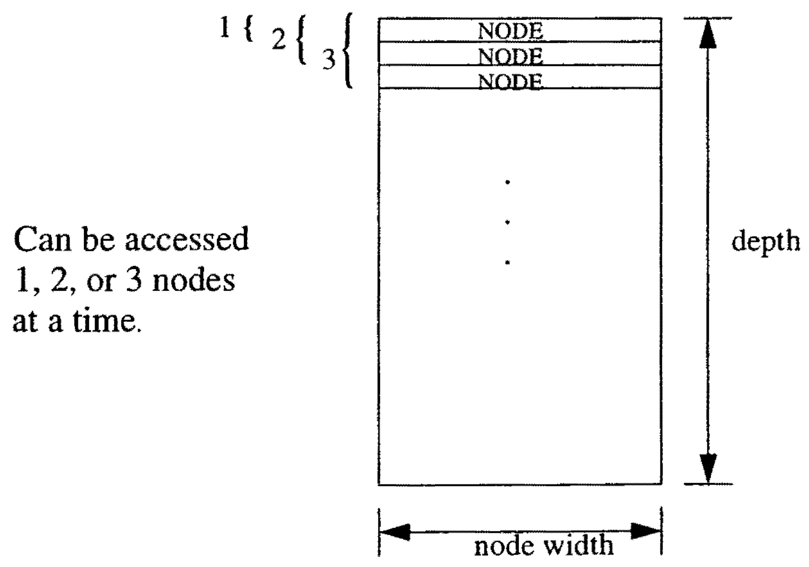
5.3.3 Unique Table

The unique table is used to hold references to individual nodes in the node memory. Because each node in the node memory must be unique this table is implemented as lookup table which holds node addresses. The node to be looked up must be hashed into an address in the unique table memory. The unique memory returns a node address which must then be looked up in node memory. It is possible that several nodes will hash into the same unique table location, thus requiring chaining (linking) of the nodes in the node table to find the correct node. Often this is combined with the node memory, but in this implementation it has been chosen as a separate physical memory. The unique table must be large to minimize hash collisions and wide enough to hold node handles (32 bits).

5.3.4 Register file

There are two register files used in the BDD computer. Both are used during recursive BDD operations to hold temporary node values, addresses and return codes.

Figure 17. Register File



They have the same width as a node but can be accessed as a moving window which can be moved up/down by 1, 2, or 3 registers. This requires three separate busses from the register file to the functional unit performing recursive operations. Because the number of variables used by a function indicates the depth of the BDD graph, the number of recursive calls to reach the bottom of the graph is at most equal to the number variables used in the BDD. Therefore, the depth of the register file determines the maximum number of variables that can be used in a BDD. The current implementation never accesses more than two registers at a time. There may be additional BDD algorithms that have not been implemented here that require more temporary registers during execution so the memory model was designed with some flexibility in mind.

5.3.5 MakeNode block description

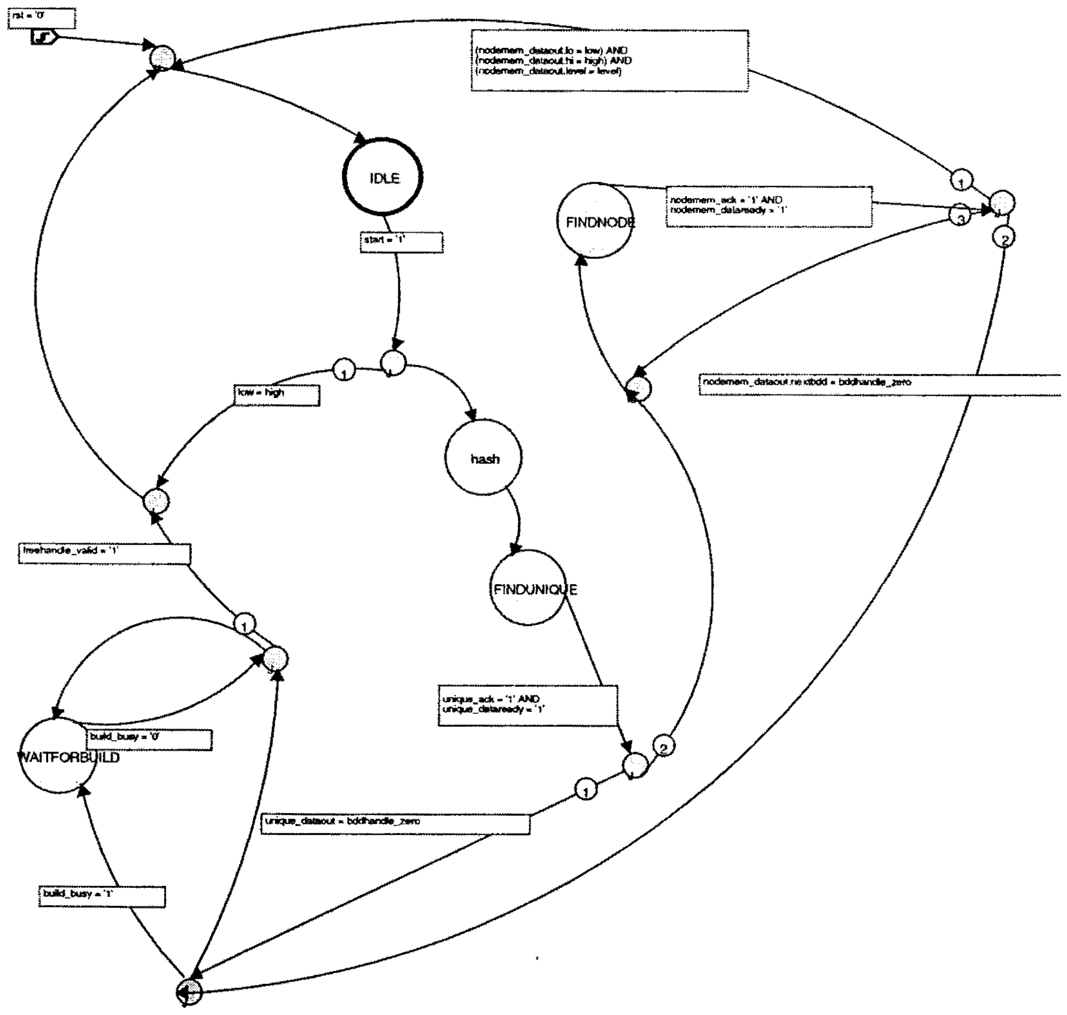
The makenode function of the BDD computer has access to the unique table and node table. It is responsible for finding nodes in the node table and for creation of all new nodes.

Figure 18. MakeNode Pseudo-Code

```
1: make_node(node)
{
  // do not allow both edges to point to same node
2:    if node.lo == node.hi then return low;
  // look up the node in the unique table
3: hash(node);
4: look up node in node table
   {
      walk the chain of nextbdd links
      until the node is found/not found
   }
   if found then return the found node address
  // not found, so build a new node.
5: build new node
   {
C     get next free node from free node list
      write the input node into the free node list address
      return the free node list address
      advance to next free node
    }
```

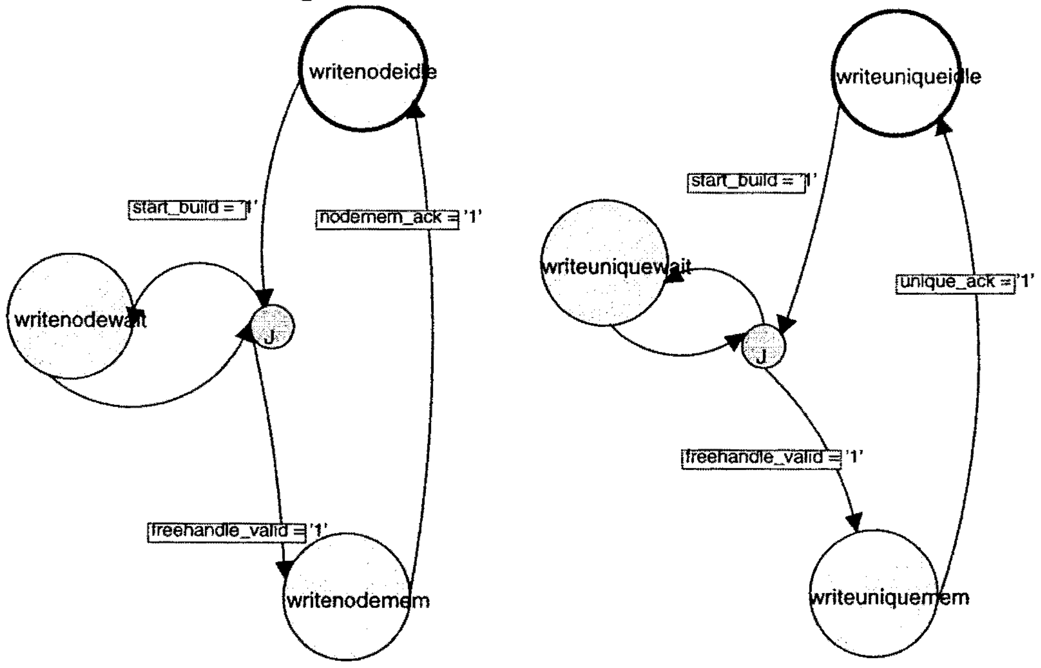
It makes sure that only unique nodes are created so there are no duplicates in the node table. Pseudo-code for make_node function is shown in Figure 18 on page 41. MakeNode was designed as a state machine. Different points in the algorithm were defined as states based on the function and expected effect on hardware implementation size and performance. These points are shown with bold numbers before the beginning of the line in Figure 18 on page 41.

Figure 19. MakeNode FSM



Each of the lines noted with a bold number is one of the points in the code which was broken down for a hardware implementation. Point 1 and 2 correspond to the IDLE state (see Figure 19 on page 42). Point 3 in the code corresponds to the hash state. Point 4 requires two states, findunique for the initial lookup in the unique table and findnode if a hash collision occurs and the nodes are chained. Point 5 and beyond is executed concurrently with the return into state IDLE. The state waitforbuild must wait until the previous build (write to memory) is finished before sending the state machine back into the IDLE state. In short, if the code after point 5 is still executing from the previous call to MakeNode, the fsm will stall in waitforbuild. The state machines that perform the writes to node memory and unique memory are concurrent with the main MakeNode FSM and are shown in Figure 20 on page 43. Both are simple three state machines which are IDLE, or waiting for the memory acknowledge signal on one clock in state writenodemem (writeuniquemem) or several clocks in state writenodewait (*writeuniquemem*). These state machines operate concurrently with each other and the main MakeNode FSM because the node table and unique table are separate memory structures.

Figure 20. MakeNode Build FSM



5.3.6 Apply block description

Apply is the algorithm that performs algorithmic manipulations of the BDD node structure to produce a result node. It is used for all 2 input boolean functions.

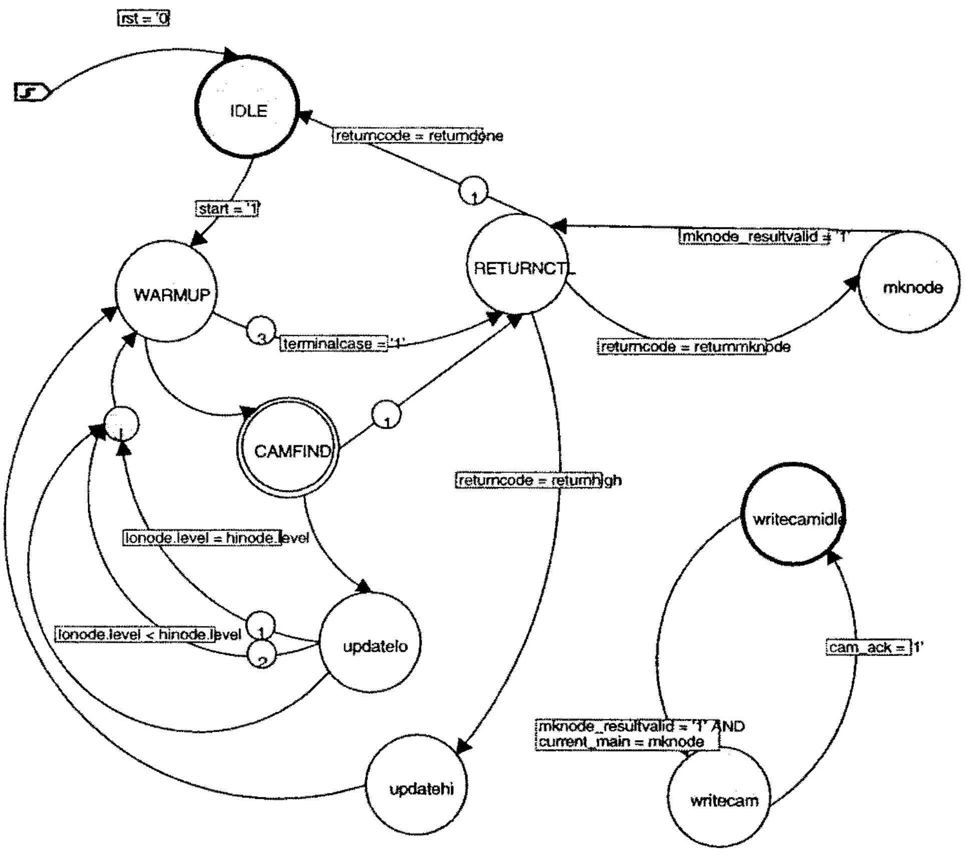
Figure 21. Pseudo Code for Apply (recursive - depth first)

```
set operation
apply(left, right)
{
C / 1: check for terminal cases
  < 2: check cache (terminal case)
  \    fetch left and right nodes, compute arguments for recursive calls
    3: res1 = apply(leftarg1,rightarg1)
    4: res2 = apply(leftarg2,rightarg2)
    5: result = make_node(level,res1,res2)
C / 6: put result in cache
  \ 7: return result of make node
}
```

Note C: available concurrency. (in a breadth first algorithm the two calls to apply can be considered concurrent).

The operation to be performed is static for a given traversal of the BDD, therefore, it can be stored in a register and is not required to be passed as an argument to apply and is set before calling apply. Terminal case check, computed cache check (CAM access) and fetch of left and right nodes from node memory can begin concurrently. In reality the terminal case check at point 1 in Figure 21 on page 44 is performed during state <u>warmup</u> (See Figure 22 on page 46). The CAM is used as a cache for intermediate computed results during the apply operation. If the arguments to apply are not a terminal case, the CAM and node memory accesses are started concurrently in state <u>CAMFIND</u>. Though it would be possible to start the fetch of nodes and check the cache concurrently with testing for the terminal cases, in all terminal cases this would cause requests to node memory and CAM that would have to be aborted. Also, while terminal cases are being checked the nodes that need to be fetched are being computed and as such this calculation is complex enough that it might affect the performance to delay the memory access until the following clock cycle, so the accesses were moved to the CAMFIND state. Points 3 and 4 correspond to states updatelo and updatehi respectively. Entering these states the register file window must be moved so that temporary values can be held until the return point is reached. Point 5 corresponds to state mknode, which will wait until the mknode operation is complete. When exiting state mknode, the CAM write (states writecamidle and writecam) will begin concurrently while the main FSM moves into state RETURNCTL. The main FSM does not have to wait for the CAM write to complete before continuing.

Figure 22. FSM for apply



All of the apply algorithms in general purpose processor code use handles (addresses) as arguments. It might make sense to have entire nodes passed as arguments. There are several reasons why this was not chosen. First, fetches of the children nodes are still required to make subsequent recursive calls. Building of new nodes with make_node is based on the level, lo and hi handles. If the entire node is passed without a handle, the structure of the unique table must be redesigned. These changes to the algorithms were considered to be beyond the scope of what is needed to be accomplished in this thesis.

Most of the recursive algorithms require the variable level of the children to make a determination of the arguments for the next recursive call. It might be an improvement to include the level of the children in the node structure. But since the children node must be fetched anyway for the arguments to the subsequent calls to apply, it does not offer any significant performance gains. Also, this change would complicate the reordering algorithm and might make reordering a non-local operation. This was not seen as a significant enough benefit to make these changes to the node structure.

5.3.7 FreeNodeControl

This block controls the free node list. It performs initialization of the node memory by correctly creating the constant nodes 0 and 1 and creating the free list of nodes. The current implementation also stores the handle of the next available node so that MakeNode block does not have to wait to return its value. It also performs garbage collection. This is why it has access to the node memory and unique memory (as well as sending control signals to the CAM not shown on the diagram)

5.3.8 Garbage Collection

In most software BDD package implementations garbage collection interrupts the execution of BDD algorithms. This is called serial garbage collection. One of the advantages of creating custom hardware is that different techniques can be used from the ones used in pure software implementation. An alternative to serial garbage collection is parallel garbage collection. The garbage collection algorithm to be used is based on the parallel algorithm described by Lamport [Lamport76] and Dijkstra [Dijkstra78]. This algorithm is designed to work with multiple processes operating on the data structure at the same time. Thus, the BDD algorithm can continue to run while the garbage collection algorithm operates in parallel. The goal is to improve performance on large problems by reducing the interruptions caused by a serial garbage collection algorithm. The garbage collector can use memory cycles that are not used by the BDD algorithms.

There are some complications introduced in garbage collection of BDD nodes. Nodes that are garbage collected must be removed from the unique table and added to the list of free nodes. The computed cache may contain references to nodes which are to be garbage collected. Therefore the computed cache entry must be invalidated or the entire cache must be invalidated (and cache operations halted) before the sweep can take place. The hardware clearing of the cache is much faster than in software.

Although garbage collection can consume large amounts of processing time, because the cases used in this thesis are relatively small, i.e. garbage collection is not required, it has not been implemented.

5.3.9 Dynamic Variable Ordering

Dynamic variable ordering is very important in any BDD implementation. Even in specialized hardware where performance exceeds that of a general purpose computer it is required to keep BDD sizes from becoming unreasonable and reducing performance. As this would require much more additional research and development, this topic has not been addressed in this thesis.

Chapter 6
Performance Evaluation

6.1 N Queens performance

The N Queens problem is the simplest of the previously profiled BDD programs. It also represents a cross between manipulations on a combinational logic circuit and a constraint problem, and thus, is a good choice as an problem for evaluation.

Obtaining execution time values for general purpose programs is not a straightforward task. The complexity of modern multi-tasking operating systems, and the processors on which they run, make it difficult to obtain highly accurate execution time values. There is operating system overhead involved in managing the memory space for each running process in the system. Cache misses during execution can cause page faults which must be handled by the OS. Dynamic memory allocation requires management by the OS. These effects are not easily quantified, thus a simple value for the execution time of a program is not always achievable. In a multiprocessor system the OS controls how different tasks (processes) are spread across the processors. This will affect the operating system execution and application programs that are multithreaded. Also, the ability of the OS to collect execution time information is limited by the precision of the hardware clocks available to the OS.

For the experiments performed here, all of the sample BDD applications used are single threaded and were run on a single processor. No other application programs were running on the system when the execution time numbers were collected. Initial

CPU execution times were collected using the UNIX system call clock(). This call returns a number (of clock tics in microseconds). It is called before and after the area of interest in the algorithm that is being simulated (see Figure 23) and the difference is assumed to be the number of microseconds used in that part of the program. The initialization time spent in bdd_init() and bdd_setvarnum() is not included in this measurement because it is not part of the time gathered from the BDD computer simulations.

Figure 23. Algorithm for N queens

```
   /* Initialize with 100000 nodes, 10000 cache entries and NxN variables */
  bdd_init(100000, 10000);
  bdd_setvarnum(N*N)
;// get the current clock number for execution time calculation
  cputime = clock( );
  queen = bddtrue;
   /* Build variable array */
  X = new bdd*[N];
  for (n=0 ; n<N ; n++)
    X[n] = new bdd[N];
  for (i=0 ; i<N ; i++)
    for (j=0 ; j<N ; j++)
      X[i][j] = bdd_ithvar(i*N+j);
    /* Build requirements for each variable(field) */
  for (i=0 ; i<N ; i++)
    for (j=0 ; j<N ; j++)
      build(i,j);
    /* Place a queen in each row */
  for (i=0 ; i<N ; i++){
    bdd e;
    for (j=0 ; j<N ; j++)
      e |= X[i][j];
    queen &= e;
  }
// get the execution time, then calculate the difference from the time collected above
  cputime = clock( ) - cputime;
```

The problem with this method is that it is not known what part of this number is system time and what part is actual user program time. In an attempt to clarify this, the unix utility time was used to gather user, system, and elapsed (wall) time for the same set of N queens executions. Because the runs are so short and the precision of the measurements is only accurate to one microsecond, five runs for each value of N were performed and the numbers averaged and rounded to two significant digits.

These times are shown in Table 6, "CPU Time (sec.)," on page 52

Table 6: CPU Time (sec.)

N	User	System	Elapsed	clock() time
4	.02	.07	.09	.005
5	.03	.07	.09	.01
6	.03	.08	.10	.02
7	.06	.08	.14	.05
8	.18	.09	.27	.17
9	.96	.08	1.04	.91
10*	4.86	.33	5.18	4.78
11*+	45.09	.45	45.50	45.10

* larger initial node allocation
+ Includes garbage collection time

The execution time also depends on how many nodes are initially allocated in the unique and node table. All times were collected with 219983 nodes in the initial table. This number is used because it is a prime number slightly larger than the number of unique nodes generated for a value of N equal nine. Keeping the initial number of nodes constant means that the system overhead due to initialization should be consistent for all of the runs. This also means that for values of N less than nine, the general purpose program could run faster than is shown here because many unused nodes are being initialized. In a most real world problems it is hard to have a good estimate of the number of required nodes, so a large initial allocation can help prevent large amount of time spent in garbage collection.

The original clock() execution times are similar to the user time numbers, though they are slightly less. This is good since they should be less than the user time because the initialization time is missing from the clock time values. It can be seen that the system time is larger than user time for values of N less than eight. This gives less confidence in the quality of the execution time numbers. Eight queens is the first time where user time is the majority of the elapsed time, and for larger values of N the system time is less than ten percent of the elapsed time. These longer runs should have a smaller error in the time measurements and make a better comparison to the execution time for the BDD processor.

Some of the execution time that exists in the general purpose implementation will still exist when using the BDD processor. The for loops and function calls shown in the application will still need to be executed to communicate with the BDD processor and obtain the results. The access to the BDD computer might also cause the OS to suspend the application program until the result is returned. This will create additional system overhead that might not occur in the general purpose implementation of the program. This overhead was not accounted for in any of the times measured here. A more detailed implementation of the software interface as a PCI /AGP driver program could be used in the future to perform this evaluation. The clock() time CPU numbers were used in all the performance comparisons because they measure the part of the algorithm that is also measured in the simulations of the BDD processor. The clock() functions were strategically placed to only capture the CPU time that is the same part of the program that was actually measured from the VHDL simulations, i.e. all initialization and functions that are not accelerated are not included in the clock() times that were collected. Only the functions that were accelerated are included in these times. This gives the best available comparison of the part of the algorithm that is accelerated by the BDD computer.

The only BDD specific functions that were not simulated and thus not included in the measured time are bdd_init(), bdd_setvarnum(), and bdd_done(). Specific implementation of initialization and termination routines was not determined, and as was shown in Figure 9 on page 21, these functions use only a small percent of execution time as the problem sizes increase so they were not simulated (and thus not measured).

Other C/C++ program overhead of checking arguments etc. is not included in the comparison times. This overhead will also exist with the BDD computer and will not be accelerated. Application programs that read and write netlists from files spend a large amount of time with this file I/O. The read and write time might be a larger part of the program than actually manipulating the BDD. In the future additional applications should be profiled to find how much time is spent manipulating BDDs compared with other parts of the program. If the time spent manipulating the BDD is small, little overall performance improvement will be seen. Only the part of the application program that calls BDD functions will be accelerated by the BDD processor. The comparisons presented here show the performance increase of only the BDD portion of the program. For the N queens problem the accelerated portion is over 90% of the execution time for values of N greater than 7. Thus, for programs that are highly dependent on BDD manipulation, the BDD processor will show significant performance improvement.

The N queens problem uses calls to only four different BDD library functions. For the purpose of comparing the time for algorithm completion initialization (function bdd_init and bdd_done) of the BDD package is not included. This is overhead that does not really contribute to the actual time to perform the operations on the BDD data structure and is a linear time operation based on the number of nodes requested during initialization. It is not included in the execution time analysis. Simulations of the N queens problem on the BDD processor were run for values of N from four to nine. Statistics for memory accesses were gathered and the number of simulated clocks were counted. Simulations of larger values of N were impractical because of memory limitations on the host computer. The results are described below.

The simulations were run assuming all memory accesses are a single 10ns clock cycle. This is unrealistic for very large memory implementations, but gives an upper bound on the performance based on the number of node memory accesses.

The macro-architecture determines the distance of the BDD processor from the GP processor and affects the latency to get all operations started. All operations in the Co-processor model involve the CPU bus and the peripheral processor involves the largest latency because it is on a peripheral bus. Both of these models must use the GP processor to

- write the instruction and arguments to BDD processor
- read of result from BDD processor

Because the general purpose computer used is a Pentium II processor (as described earlier in "Computing Environment" on page 19) it is also used as the model for the GP with which the BDD processor is attached. On the Pentium Pro processor bus, it takes a minimum of seven clock cycles to perform a complete write transaction involving a 64 bit transfer. Also, a minimum of seven clock cycles are required for the read transaction to get the result back into the processor. Because the BDD operations may take considerable time, the use of a deferred read cycle would be necessary, so an additional bus transaction to complete the deferred read is necessary. A minimum total of 21 processor clocks are necessary to complete the transaction with the BDD processor.

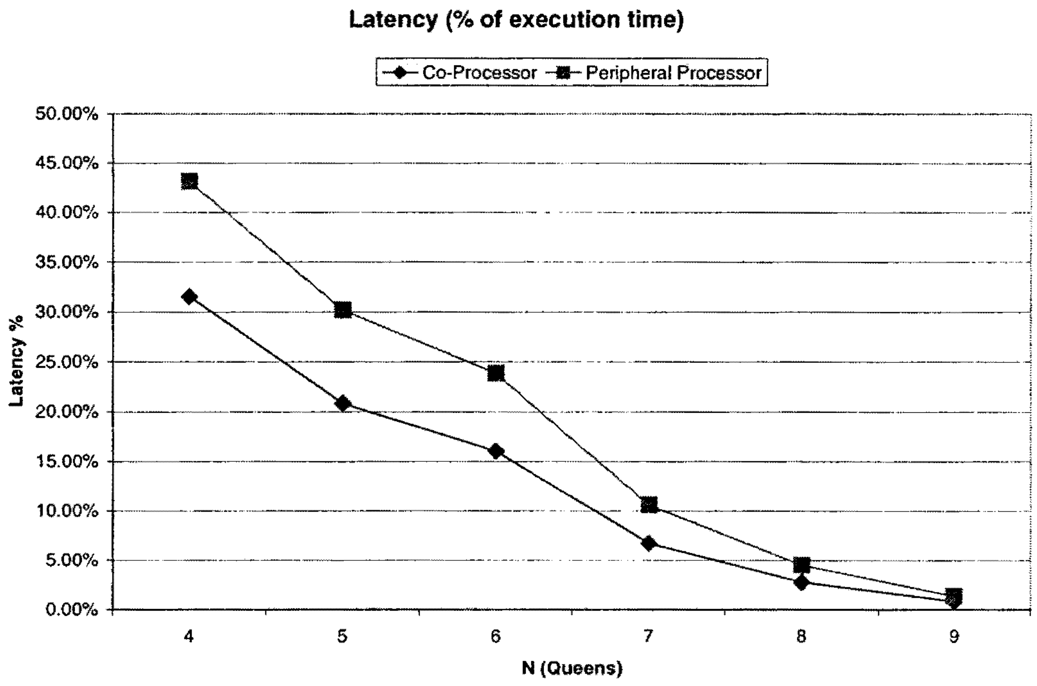
The peripheral processor configuration will have the latency of the processor bus, plus additional time for the chip set (3 CPU bus clks) to negotiate the peripheral (PCI) bus, and the PCI bus transaction is a three PCI clk minimum. Table 7, "BDD Processor Access Latency," on page 57 gives example latencies for a Pentium Pro Processor (includes Pentium II and Pentium III) and the PCI peripheral bus. The numbers for the latency are strictly the hardware numbers and do not involve the software overhead of the device driver that services the BDD processor. The same function call overhead exists whether the general purpose function is being called or the driver function is being called, so it has not been included in these numbers.

Table 7: BDD Processor Access Latency

Architecture	**Pentium Pro-Bus**			**PCI Bus**			**Slow Total**	**Fast Total**
	clks write +read	**Chip Set clks**	**100Mhz Bus (ns)**	**clks write +read**	**33Mhz (ns)**	**66Mhz (ns)**	**PCI 33**	**PCI66**
Integrated	0			0			0	0
Co-Processor	21		210	0			210	210
Peripheral Processor	21	6	270	6	90	45	360	305

This latency is large compared to the 450MHz processor, but there are a relatively small number of calls to the BDD processor (See Table 9 on page 59). Most of the work is done on the BDD processor and, it turns out, that for the larger sizes of N the latency is only a small fraction of the actual processing time. This is shown in Figure 24 on page 58

Figure 24. Latency Effects



All memory access is single cycle, so it mimics a cache hit in a GP processor. The CAM is also single cycle access for both read and write. There is no garbage collection. Because the garbage collection algorithms are implemented differently, the sizes of the memories were chosen to avoid garbage collection to make a fair comparison of the general purpose algorithm and to produce estimate of the best possible performance. Table 8 on page 59 shows the memory latency and size characteristics used for the simulation. The BDD processor execution times with the latencies for the different macro architectures are given in Table 9 on page 59. The BDD processor is

Table 8: Memory Characteristics

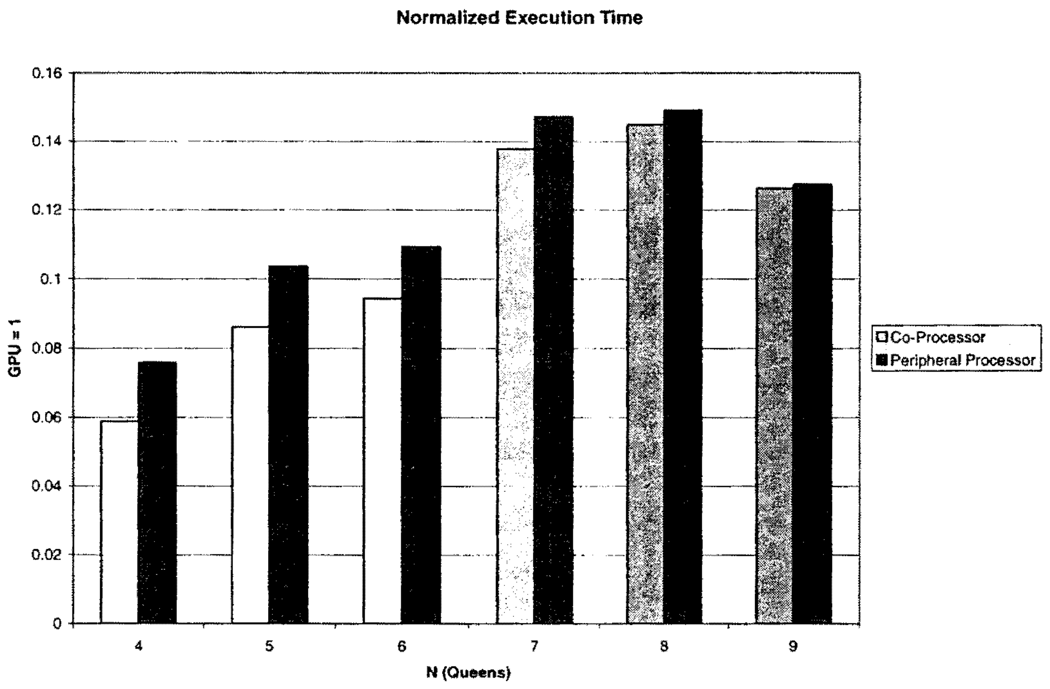
	Latency	Size
Unique	1	219983
CAM	1	32768
Node Memory	1	220,000

assumed to operate at 100 MHz which facilitates access to SRAM which can be accessed in less than 10 ns. Figure 25 on page 60 shows the execution times from Table 9 relative to the GPU execution time. All values have been normalized so the GPU execution time is one. This shows that the implementation, described here using SRAM as the memory of choice, can achieve greater than 7x performance increase over the GP processor for all tested values of N.

Table 9: N Queens BDD Execution Times

N	BDD Calls		BDD processor clks	Execution time (ns)			Approx. GPU execution
	(n)ith var	apply	100MHZ	Integrated	Co-Processor	Peripheral Processor	
4	32	540	17358	173580	293700	379500	NA
5	50	1090	61733	617330	860930	1034930	10000000
6	72	1926	146709	1467090	1886670	2186370	20000000
7	98	3104	621311	6213110	6885530	7365830	50000000
8	128	4696	2362891	23628910	24641950	25365550	170000000
9	162	6752	11337491	113374910	114826850	115863950	910000000

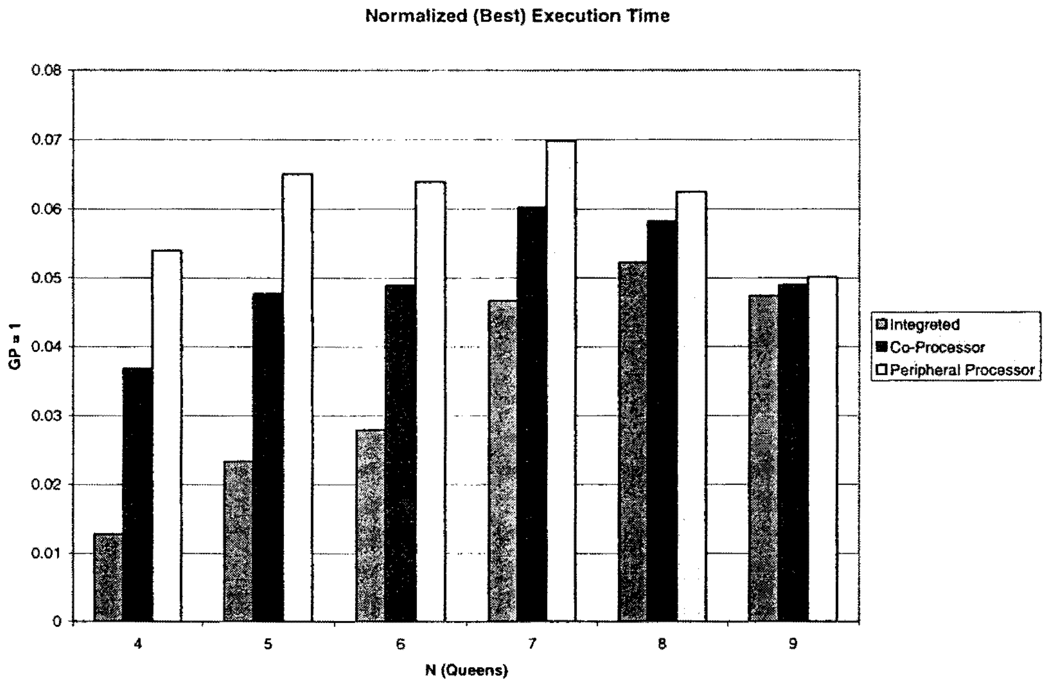
Figure 25. BDD Execution Time (SRAM)



Because there is single cycle memory access, the node memory is busy less than 31% of the time for all values of N. The unique memory and CAM are each busy less than 10% of the time. If single cycle access memory is possible, this architecture is very memory inefficient. CAM accesses can be concurrent with other accesses so are not a limiting factor. Unique reads occur sequentially before the node access so must be combined with node memory access to calculate the performance limit of this memory architecture. Figure 26 on page 61 shows the execution time using these assumptions. It can be seen that performance can be almost 20 times faster (for N = 9) than the same algorithm run on a general purpose processor which has a clock rate 4.5

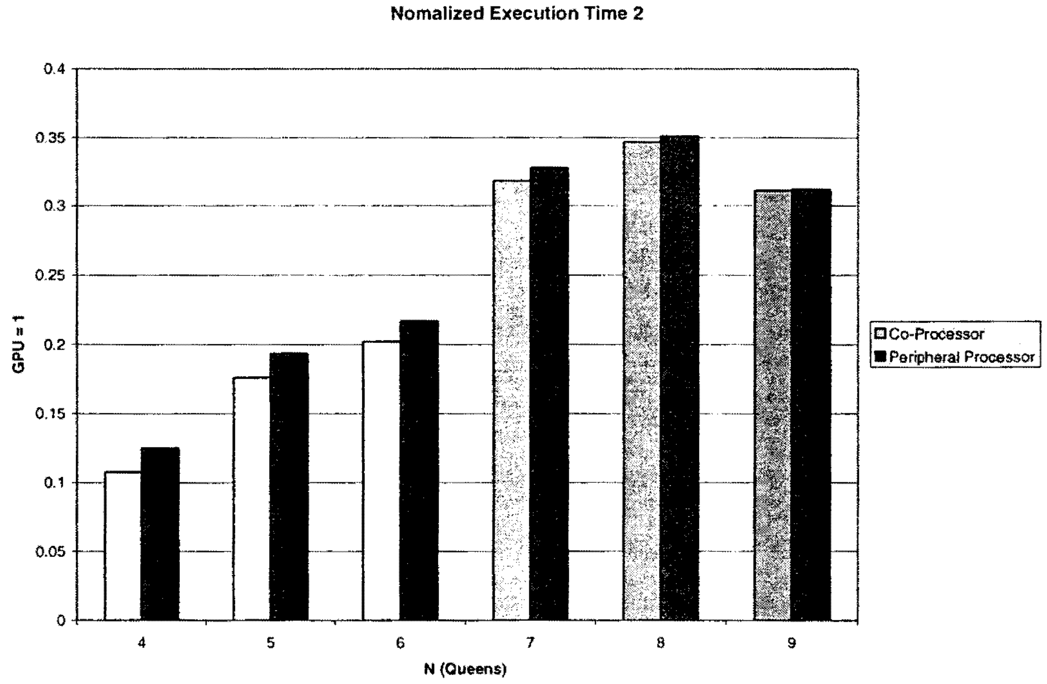
times higher than the BDD processor. This is the best performance possible using single cycle memory access with the sizes specified inTable 8 on page 59.

Figure 26. Best BDD Execution Time



By changing the node memory to 60 ns DRAM and assuming a realistic cache access of 30 ns, the execution performance drops noticeably but is still almost 3x faster than the GPU for all values of N. The execution times are approximate based on the assumptions described here. Since all memory writes occur in parallel with other execution, the current design mask 30ns of all writes to node memory. CAM hits will mask reads to node memory. For every CAM hit in apply, it masks 2 of the node memory reads. For every CAM hit in applynot, it masks 1 node memory read. Adding additional 2 clks/node write, 5 clocks for non-CAM-masked read, 3 clocks for CAM masked reads give new normalized execution times shown in Figure 27 on page 62.

Figure 27. Estimated BDD Execution Time (DRAM Memory)



6.2 Conclusions

One of the major considerations used during the work on this project was cost containment. The possibility of building the design economically using current FPGA and memory technology drove the decision to use a 100MHz clock frequency for the BDD processor. The second reason for that choice was the interface to current Pentium Pro processor technology which has a 100 MHz bus frequency. As has been shown, the latency of the interface to the processor is small in proportion to the actual time spent executing the algorithm, even when slower interface methods such as PCI bus are considered. Therefore the host processor bus speed should not be an overriding consideration when choosing the clock rate of the BDD processor.

All of the comparisons in this paper are based on a general purpose processor clock rate of 450MHz. There are now 1 GHz processors available which should give roughly double the performance of the 450MHz processor used in this paper. As general purpose processors gain performance and 64-bit operating systems become prevalent on engineering workstations within the next several years, they could readily outpace the performance obtainable on a dedicated BDD processor. To make the BDD processor a viable alternative to a general purpose processor, even higher performance of the BDD processor than has been shown here is needed . Additionally, the memory capacity of the BDD processor must be large, this might necessitate the use of some form of DRAM as a size and cost savings measure. This could severely impact the performance making the specialized architecture undesirable. There are many obstacles to overcome to make a specialized BDD processor architecture a viable addition to an engineering workstation users environment.

6.3 Improvements and future work

This thesis is not the end, but describes a possible starting point for exploration of specialized hardware for fast execution of BDD algorithms. The design used in this thesis is very simple. It is nearly a direct translation of the software algorithms and only begins to scratch the surface of an efficient hardware implementation. It could serve as the basis for further explorations into architectural issues specific to BDD algorithms. There is much work that could be done to improve performance, including exploration of different micro-architectures, implementation of breadth-first and parallel BDD algorithms, different pipeline and memory structures and, because ASIC technology is capable of very high performance, higher clock rates must be considered.

In order to make the design specialized hardware more flexible, a programmable micro-architecture should be investigated. Using memory to store micro-programs and redesigning the BDD processor to execute these programs has two immediate benefits, upgrade ability and expend ability. For patches and upgrades to the BDD algorithms, problems in the micro-code can be easily fixed by loading it into the microprogram storage. Additionally, new and experimental algorithms could be implemented and tested without having to redesign the hardware.

Other BDD algorithm implementations such as BDDs with complement edges and breadth first execution of BDD algorithms should also be investigated for additional performance improvement. These algorithms have shown improvement on general purpose processors and should also show significant performance improvement in hardware. Parallel BDD algorithms are also a leading candidate for hardware implementation because specialized hardware can implement parallelization much more effectively than a network of general purpose computers.

Several key aspects of BDD manipulation have not been investigated in this thesis, two are garbage collection and dynamic variable ordering. Both of these issues must be investigated in detail. Concurrent garbage collection offers an opportunity to significantly improve the performance on large BDDs by utilizing unused memory bandwidth rather than interrupting algorithm execution. Dynamic variable reordering must be implemented for a complete and usable BDD system.

The memory models used for the experiments in this BDD processor are quite simple. The commercial CAM technology used for the computed cache is currently limited in size and performance. Other CAM and non-CAM implementations need to be considered. Interleaving of the node memory for higher performance (possibly based on the position of a node in the order of variable) may also give improved performance. Combining the node and unique memory will save I/O and might not cause significant performance penalties depending on the node memory speed. This will also allow a much larger unique table, thus reducing collisions during node lookup and improving performance. Using a memory to cache nodes could also be investigated, but because of the studies showing the unpredictable nature of BDD node access the cache might have to be larger than is practice.

The use of a micro-architecture simulation environment would allow studies of the hardware performance. This was done with VHDL in this thesis but other implementations in C++ or Verilog HDL might be better. Instrumenting existing general purpose BDD packages to do performance analysis might also give additional insight into what kind hardware structures are required for best performance. There is still much work remaining to be done to study and design an efficient specialized BDD computer architecture.

References

[Aho86]. A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison Wesley Publishing, 1986.

[BuDDy99]. J. Lind-Nielsen. BuDDy: Binary Decision Diagram package, Release 1.7, Documenation. Department of Information Technology, Technical University of Denmark, June 1999.

[Becker97]. B.Becker, R. Drechsler. "Decision Diagrams Synthesis -Algorithms, Applications and Extensions -," *Proceedings of VLSI Design Conference*, pp. 46-50, IEEE, 1997.

[Bertacco97]. B. Bertacco, M. Damiani. "The Disjunctive Decomposition of Logic Functions," *Digest of Papers of ICCAD,* pp. 78-82, IEEE, 1997.

[Brace90]. K.S. Brace, R.L.Rudell, R.E.Bryant. "Efficient Implemenation of a BDD Package," *Proceedings of Design Automation Conference*, pp. 40-45, ACM/IEEE, 1990.

[Bryant86]. R.E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C-35, pp. 788-701, Aug. 1986.

[Bryant95]. R.E. Bryant. "Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification," *Digest of Papers of ICCAD*, pp. 326-243, IEEE, 1995.

[CAL97]. R. K. Ranjan, J. V. Sanghavi. The Cal package, Documentation. University of California at Berkeley, 1997.

[Chang96]. S. Chang, M. Marek-Sadowska, T. Hwang. "Technology Mapping for TLU FPGA's Based on Decomposition of Binary Decision Diagrams," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems,* Vol. 15, No. 10, pp. 1226-1236, IEEE, October 1996.

[Chen97]. Y. Chen, B. Yang, R.E.Bryant. "Breadth-First with Depth-First BDD Construction: A Hybrid Approach," Carnegie Mellon Univerisity CMU-CS-97-120, 1997.

[Cho94]. H. Cho, G. D. Hachtel, F. Somenzi. "Redundancy Identification/Removal and Test Generation for Sequential Circuits Using Implicit State Enumeration," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, Vol. 12, No. 7, pp. 935-945, IEEE, July 1993.

[Cortadella99]. J. Cortadella, G. Valiente. *A Relational View of Subgraph Isomorphism*. Research Report LSI-99-33-R, October 1999, Technical University of Catalonia, Barcelona, Spain.

[CUDD98]. F. Somenzi. CUDD: CU Decision Diagram Package Release 2.3 documentation, Dept. of Electrical and Computer Engineering, University of Colorado at Boulder, 1998.

[Dijkstra78]. E.W. Dijkstra, L Lamport, et. al. "On-theFly Garbage Collection: An Exercise in Cooperation," *Communications of the ACM*, Vol. 21 No. 11, pp. 966-975, November 1978.

[Drechsler98]. R. Drechsler, N. Drechsler, W. Gunther. "Fast Exact Minimization of BDDs," *Proceedings of Design Automation Conference*, pp. 200-205, ACM/IEEE, 1998.

[Huang98]. Shi-Yu Huang, Kwang-Ting(Tim) Cheng. *Formal Equivalence Checking and Design Debugging*, Kluwer Academic Publishers, 1998.

[Hachtel98]. Gary D. Hachtel, Fabio Somenzi. *Logic Synthesis and Verification Algorithms*, Chapter 6. Kluwer Academic Publishers, 1998.

[Intel96]. Intel I440BX chip set specification. Intel Inc. 1996.

[Lai94]. Y. Lai, M. Pedram, S. B. K. Vrudhula. "EVBDD-Based Algorithms for Integer Linear Programming, Spectral Transformation , and Function Decomposition," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems.* Vol. 13, No. 8, pp.959-975, IEEE, August 1994.

[Lamport76]. L. Lamport. "Garbage Collection with Multiple Processes: An Exercise in Parallelism," *International Conference on Parallel Processing*, pp. 50-53, IEEE, 1976.

[Long97]. D. E. Long. "The Designs of a Cache-Friendly BDD Library," *Digest of Papers of ICCAD*, pp. 639-645, IEEE/ACM, Nov. 1998.

[Klarlund96]. N. Klarlund, T. Rauhe. "BDD Algorithms and Cache Misses," *Basic Research in Computer Science* Report Series RS-96-26, pp. 1-15, The Danish National Research Foundation, July 1996.

[Manne97]. S. Manne, D. Grunwald, F Somenzi. "Remembrance of Things Past: Locality and Memory in BDDs," *Proceedings of the Design Automation Conference*, pp. 196-201, IEEE/ACM 1997

[Milvang98]. K. Milvang-Jensen, A. J. Hu. "BDDNOW: A Parallel BDD Package," F*ormal Methods in CAD*, 1998. Lecture Notes in Computer Science, No. 1522. pp. 501-507, Springer, 1998.

[Minato96]. S. Minato. *Binary Decision Diagrams and Applications for VLSI CAD*, Kluwer Academic Publishers, 1996.

[Narayan98]. A. Narayan. "Recent Advances in BDD Based Represenations for Boolean Functions: A Survey," *Proc. of 12th International VLSI Design Conference*, pp. 46-50, IEEE, 1999.

[Panda95]. S. Panda, F. Somenzi. "Who Are the Variables in Your Neighborhood," *Digest of Papers of ICCAD*, pp. 74-77, IEEE/ACM, 1995.

[Ranjan96a]. R. Ranjan, J.V. Sanghavi, R.K.Brayton, A. Sangiovanni-Vincentelli. "Binary Decision Diagrams on a Network of Workstations," *Proceedings of IEEE/ ACM International Conference on Computer Design*, pp. 358-364, ACM/IEEE, Oct. 1996.

[Ranjan96b]. R. Ranjan, J.V. Sanghavi, R.K.Brayton, A. Sangiovanni-Vincentelli. "High Performance BDD Package Based on Exploiting Memory Hierarchy," *Proceedings of 33rd Design Automation Conference*, pp. 635-640, ACM/IEEE, 1996.

[Rudell93]. R. Rudell. "Dynamic variable ordering for ordered binary decision diagrams," *Digest of Papers of ICCAD*, pp. 42-47, Nov. 1993.

[Sekine97]. K. Sekine, H. Imai. "Counting the Number of Paths in a Graph via BDDs," *IEICE Trans. Fundamentals*, Vol. E80-A, No. 4. pp. 682-688, April 1997.

[Sentovich96]. E. M. Sentovich. "A Brief Study of BDD Package Performance." *Formal Methods in CAD 1996*, Lecture Notes in Computer Science 1166, pp. 389-403. Springer, 1996.

[Shanley95]. T. Shanley, D. Anderson. PCI System Architecture. Mindshare Inc. Addison-Wesley, 1995.

[Shanley97]. T. Shanley. Pentium Pro and Pentium II System Architecture. Mindshare Inc. Addison-Wesley, 1997.

[Stornetta95]. A. L. Stornetta. "Implementation of an Efficient Parallel BDD Package", Masters Thesis, University of California, Santa Barbara, Dec. 1995.

[Stornetta96]. T Stornetta, F. Brewer. "Implementation of and Efficient Parallel BDD Package," *Proceedings of 33rd Design Automation Conference*, pp. 641-644 ACM/ IEEE, 1996.

[Villa97]. T.Villa, T.Kam, R.Brayton, A. Sangiovanni-Vincentelli. "Implicit Formulation of Unate Covering," *Synthesiss of Finite State Machines: Logic Optimization,*. Chapter 10 pp. 301-321, Kluwer, 1997.

[BYang98]. B. Yang, R.E.Bryant et.al. "A Performance Study of BDD-Based Model Checking," *Formal Methods in CAD 1998*, Lecture Notes in Computer Science 1522, pp. 255-290, Springer, 1998.

[CYang98]. C.Yang, V. Singhal. M. Ciesielski. "BDD Decomposition for Efficient Logic Synthesis." *Proceedings of International Workshop on Logic Synthesis*, pp. 2-4 1999.

Appendix A

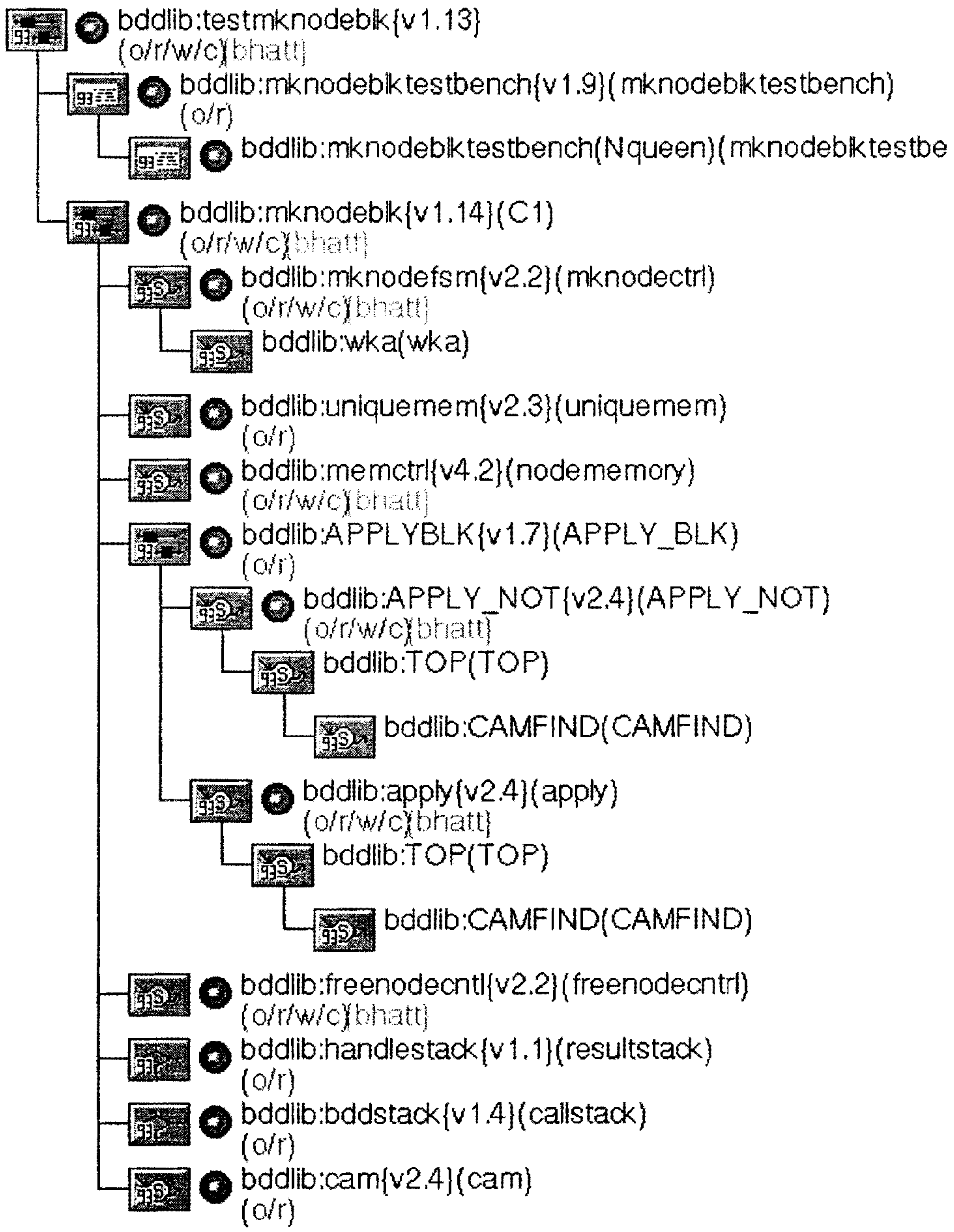
Visual HDL Diagrams and VHDL Source Code

This appendix shows the entire design that was created using Visual HDL from Summit Design Inc. It includes a graphical representation of the design hierarchy used for the tests in the thesis as well as all of the VHDL source code generated using Visual HDL. Because Visual HDL is a graphical tool, much of the design context is lost when looking at only the machine generated code. The source graphics contain design information and comments that are not included directly in the machine generated code. This information which is not visible on the diagram, may/may not appear in the generated source, but the generate code is complete and could be used in any VHDL simulation system.

The top level testbench used for simulation is the entity testmknodeblk. This instantiates the design and the testbench N-queens algorithm. The entire hierarchy is shown graphically below followed by the graphical diagrams for each block. Finally the source code is given. The modules (entities/architectures) in the source code are listed in a bottom up order as would be needed by any VHDL compiler to resolve the dependences.

List of Figures

Figure 1. Design Hierarchy



```
bddlib:testmknodeblk{v1.13}
(o/r/w/c){bhatt}
  bddlib:mknodeblktestbench{v1.9}(mknodeblktestbench)
  (o/r)
    bddlib:mknodeblktestbench(Nqueen)(mknodeblktestbe
  bddlib:mknodeblk{v1.14}(C1)
  (o/r/w/c){bhatt}
    bddlib:mknodefsm{v2.2}(mknodectrl)
    (o/r/w/c){bhatt}
      bddlib:wka(wka)
    bddlib:uniquemem{v2.3}(uniquemem)
    (o/r)
    bddlib:memctrl{v4.2}(nodememory)
    (o/r/w/c){bhatt}
    bddlib:APPLYBLK{v1.7}(APPLY_BLK)
    (o/r)
      bddlib:APPLY_NOT{v2.4}(APPLY_NOT)
      (o/r/w/c){bhatt}
        bddlib:TOP(TOP)
          bddlib:CAMFIND(CAMFIND)
      bddlib:apply{v2.4}(apply)
      (o/r/w/c){bhatt}
        bddlib:TOP(TOP)
          bddlib:CAMFIND(CAMFIND)
    bddlib:freenodecntl{v2.2}(freenodecntrl)
    (o/r/w/c){bhatt}
    bddlib:handlestack{v1.1}(resultstack)
    (o/r)
    bddlib:bddstack{v1.4}(callstack)
    (o/r)
    bddlib:cam{v2.4}(cam)
    (o/r)
```

Figure 2. TESTMKNODEBLK

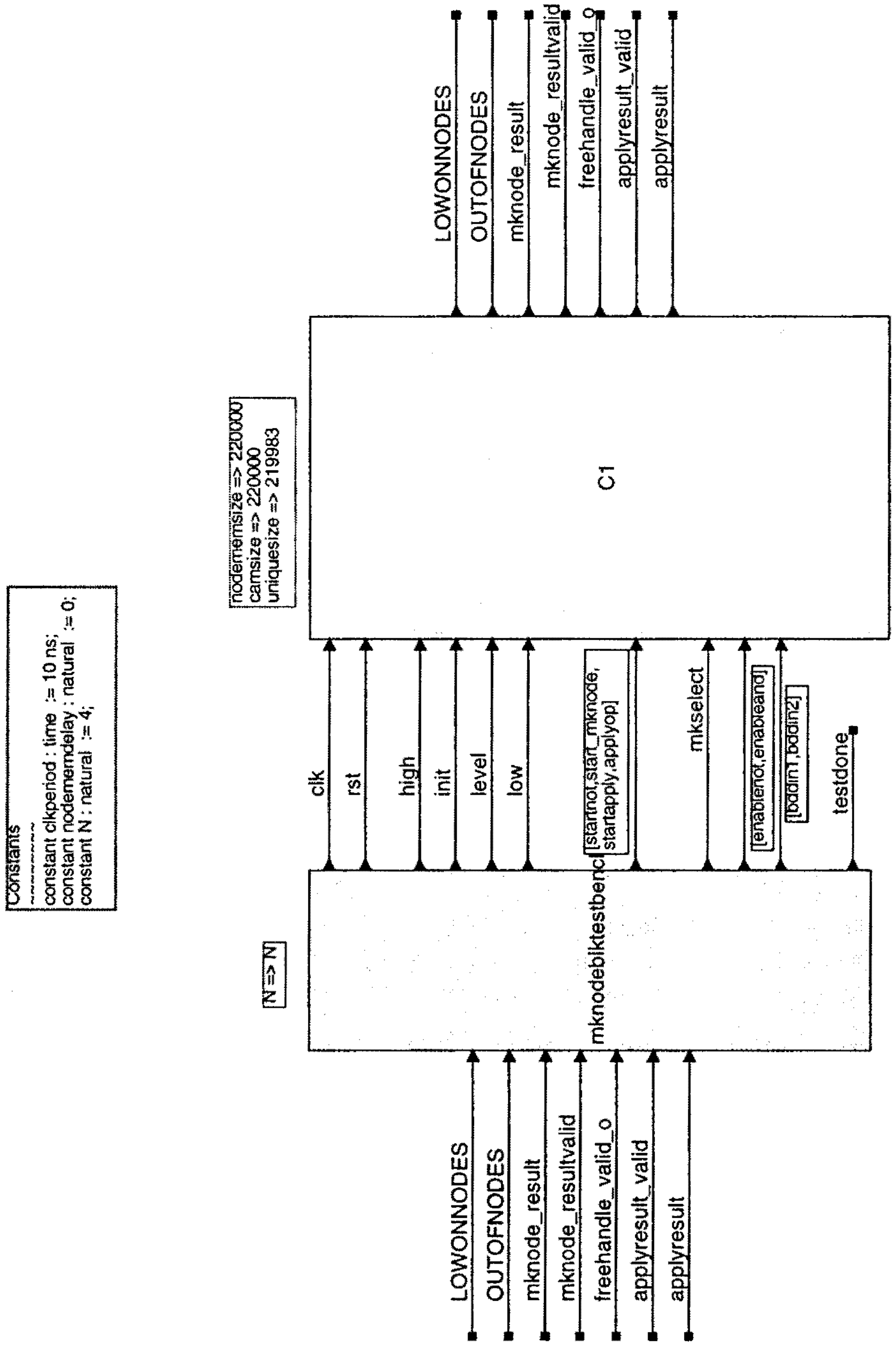


Figure 3. MKNODEBLK

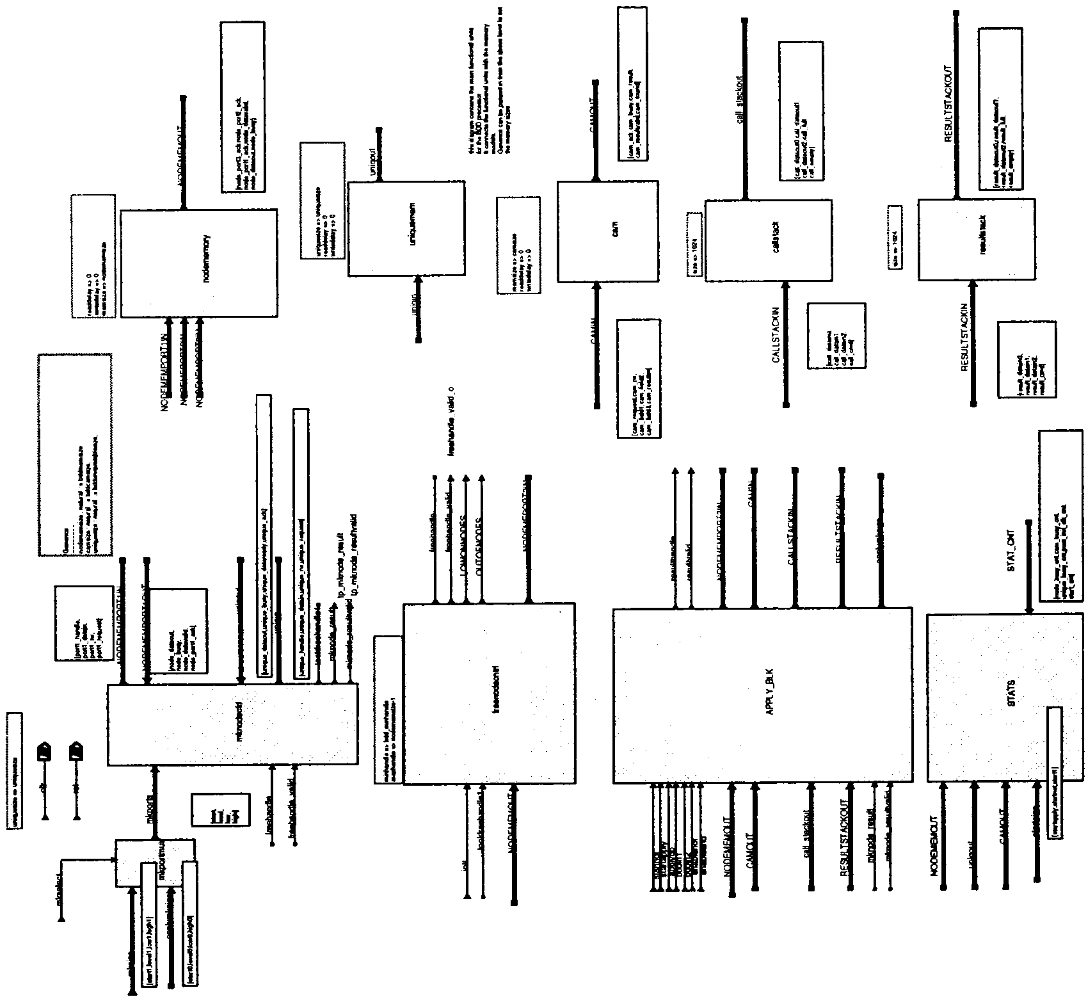


Figure 4. APPLYBLK

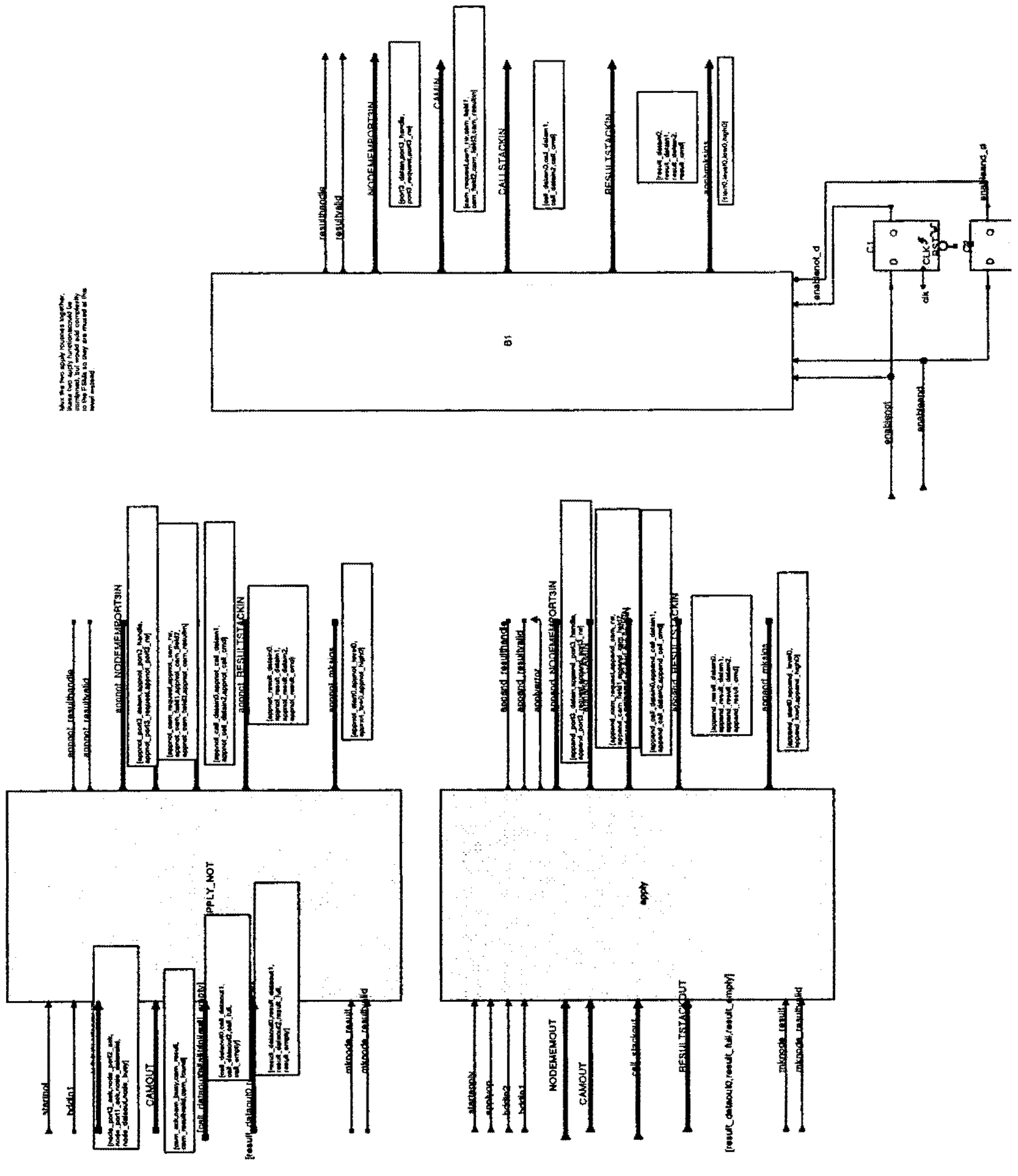
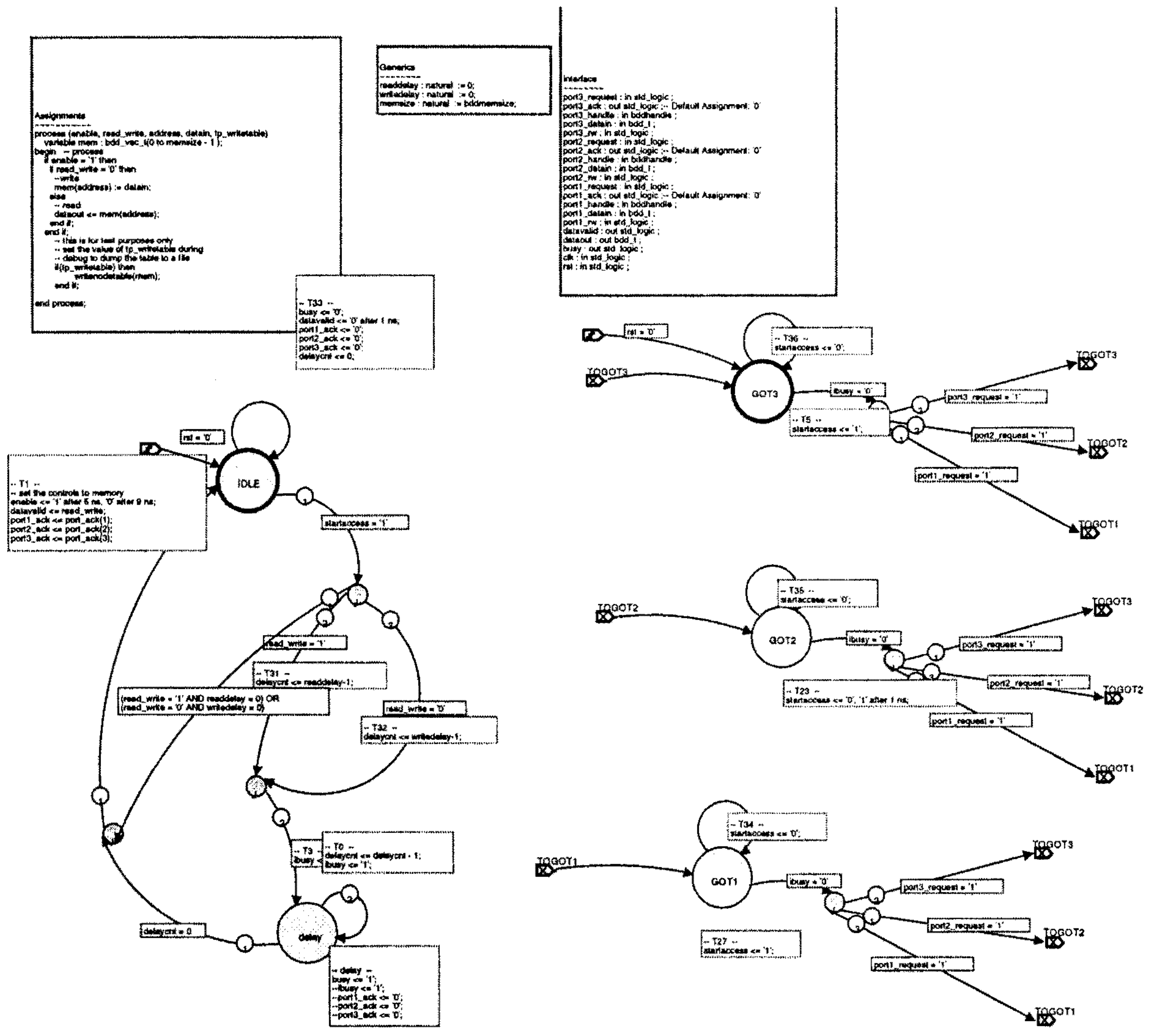


Figure 5. MEMCTRL



```
Assignments
------------
process (enable, read_write, address, datain, tp_writetable)
   variable mem : bdd_vec_t(0 to memsize - 1 );
begin  -- process
   if enable = '1' then
     if read_write = '0' then
       --write
       mem(address) := datain;
     else
       -- read
       dataout <= mem(address);
     end if;
   end if;
       -- this is for test purposes only
       -- set the value of tp_writetable during
       -- debug to dump the table to a file
       if(tp_writetable) then
              writenodetable(mem);
       end if;

end process;
```

```
Generics
---------
readdelay : natural  := 0;
writedelay : natural  := 0;
memsize : natural  := bddmemsize;
```

```
Interface
---------
port3_request : in std_logic ;
port3_ack : out std_logic ;-- Default Assignment: '0'
port3_handle : in bddhandle ;
port3_datain : in bdd_t ;
port3_rw : in std_logic ;
port2_request : in std_logic ;
port2_ack : out std_logic ;-- Default Assignment: '0'
port2_handle : in bddhandle ;
port2_datain : in bdd_t ;
port2_rw : in std_logic ;
port1_request : in std_logic ;
port1_ack : out std_logic ;-- Default Assignment: '0'
port1_handle : in bddhandle ;
port1_datain : in bdd_t ;
port1_rw : in std_logic ;
datavalid : out std_logic ;
dataout : out bdd_t ;
busy : out std_logic ;
clk : in std_logic ;
rst : in std_logic ;
```

```
-- T33 --
busy <= '0';
datavalid <= '0' after 1 ns;
port1_ack <= '0';
port2_ack <= '0';
port3_ack <= '0';
delaycnt <= 0;
```

```
-- T1 --
-- set the controls to memory
enable <= '1' after 5 ns, '0' after 9 ns;
datavalid <= read_write;
port1_ack <= port_ack(1);
port2_ack <= port_ack(2);
port3_ack <= port_ack(3);
```

```
-- T31 --
delaycnt <= readdelay-1;
```

```
-- T32 --
delaycnt <= writedelay-1;
```

```
-- T0 --
delaycnt <= delaycnt - 1;
ibusy <= '1';
```

```
-- delay --
busy <= '1';
--ibusy <= '1';
--port1_ack <= '0';
--port2_ack <= '0';
--port3_ack <= '0';
```

```
-- T36 --
startaccess <= '0';
```

```
-- T5 --
startaccess <= '1';
```

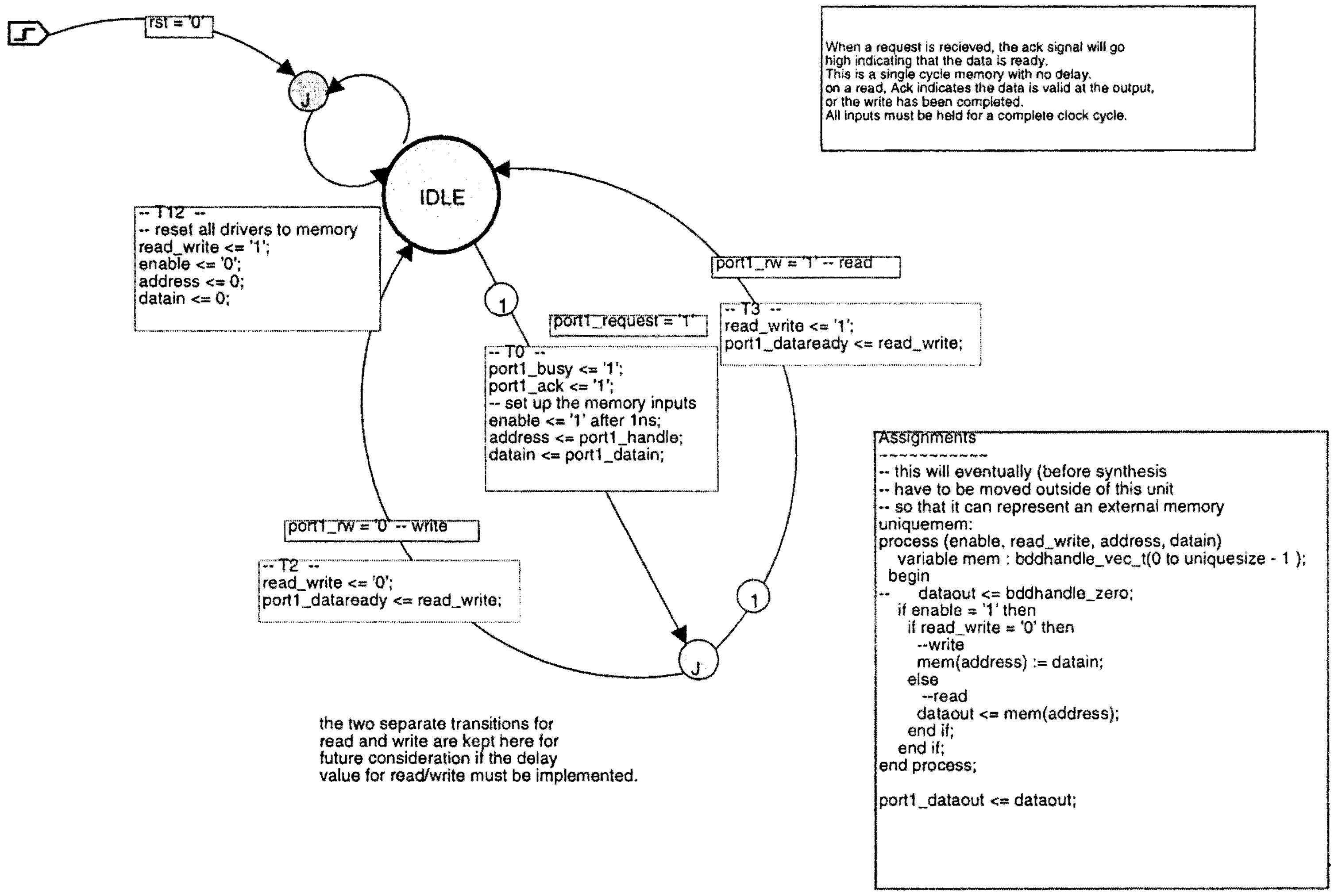
```
-- T35 --
startaccess <= '0';
```

```
-- T23 --
startaccess <= '0', '1' after 1 ns;
```

```
-- T34 --
startaccess <= '0';
```

```
-- T27 --
startaccess <= '1';
```

Figure 6. UNIQUEMEM



```
-- T12 --
-- reset all drivers to memory
read_write <= '1';
enable <= '0';
address <= 0;
datain <= 0;
```

```
-- T0 --
port1_busy <= '1';
port1_ack <= '1';
-- set up the memory inputs
enable <= '1' after 1ns;
address <= port1_handle;
datain <= port1_datain;
```

```
-- T3 --
read_write <= '1';
port1_dataready <= read_write;
```

```
-- T2 --
read_write <= '0';
port1_dataready <= read_write;
```

When a request is recieved, the ack signal will go
high indicating that the data is ready.
This is a single cycle memory with no delay.
on a read, Ack indicates the data is valid at the output,
or the write has been completed.
All inputs must be held for a complete clock cycle.

the two separate transitions for
read and write are kept here for
future consideration if the delay
value for read/write must be implemented.

```
Assignments
~~~~~~~~~~~
-- this will eventually (before synthesis
-- have to be moved outside of this unit
-- so that it can represent an external memory
uniquemem:
process (enable, read_write, address, datain)
   variable mem : bddhandle_vec_t(0 to uniquesize - 1 );
  begin
--      dataout <= bddhandle_zero;
    if enable = '1' then
      if read_write = '0' then
        --write
        mem(address) := datain;
      else
        --read
        dataout <= mem(address);
      end if;
    end if;
end process;

port1_dataout <= dataout;
```

Figure 7. MKNODEFSM

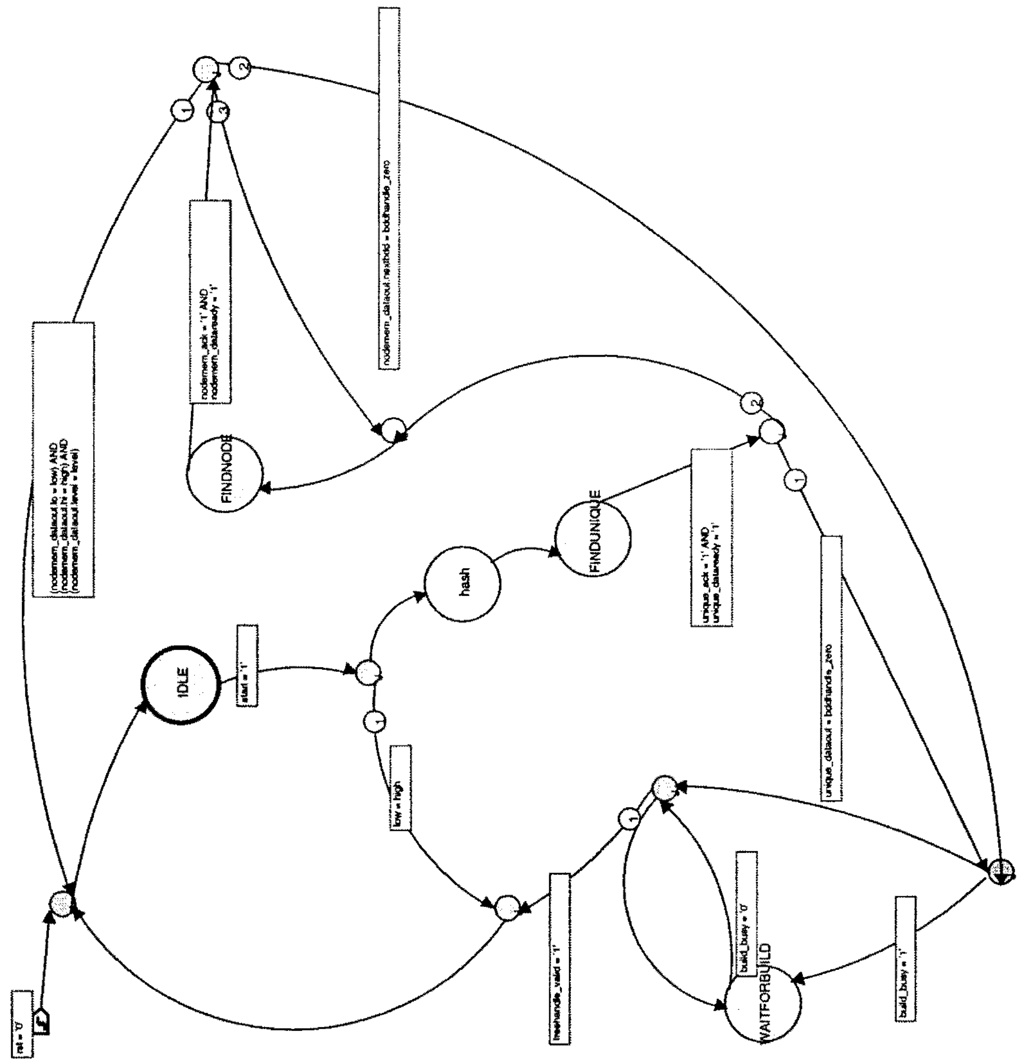


Figure 8. APPLY_NOT.TOP

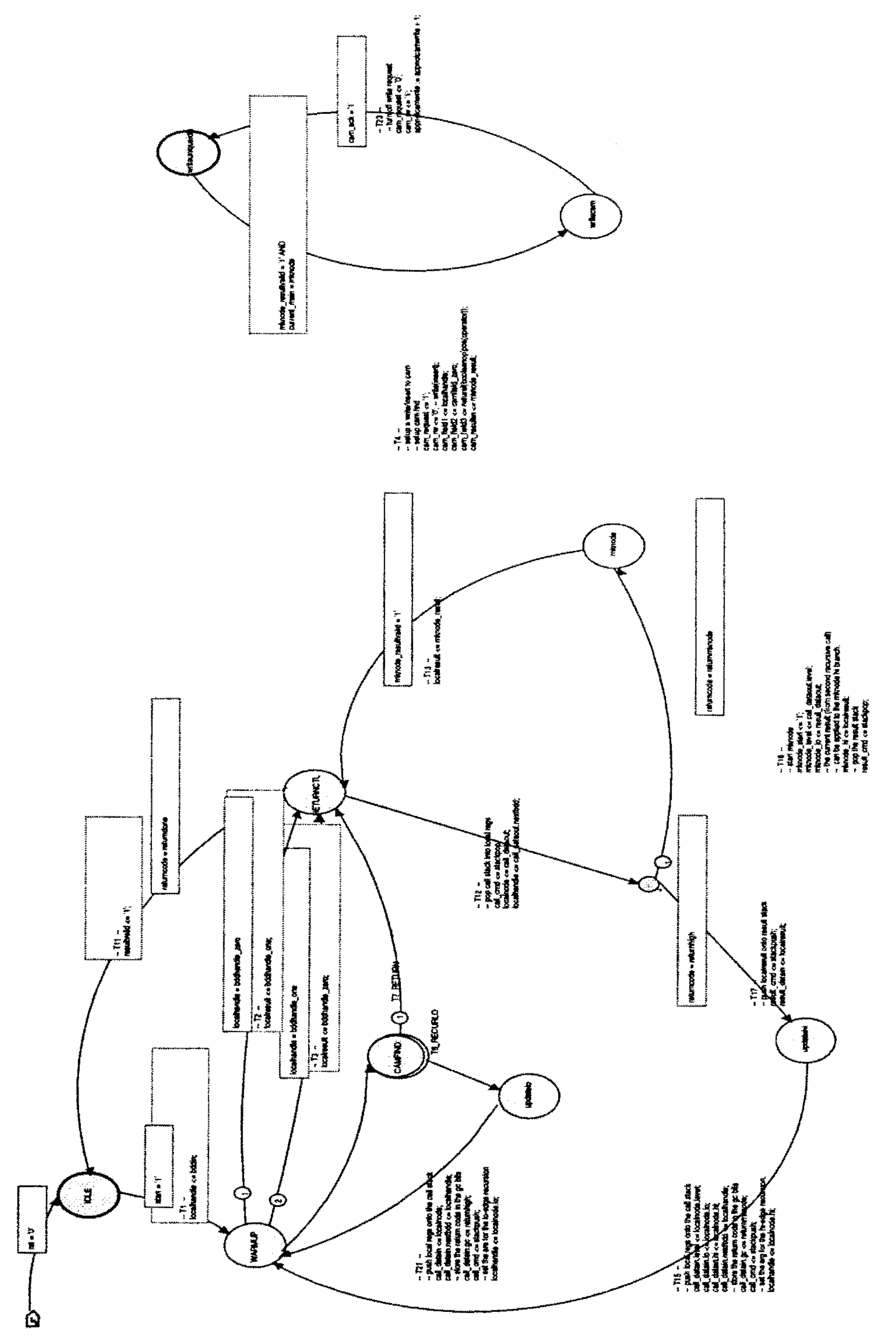
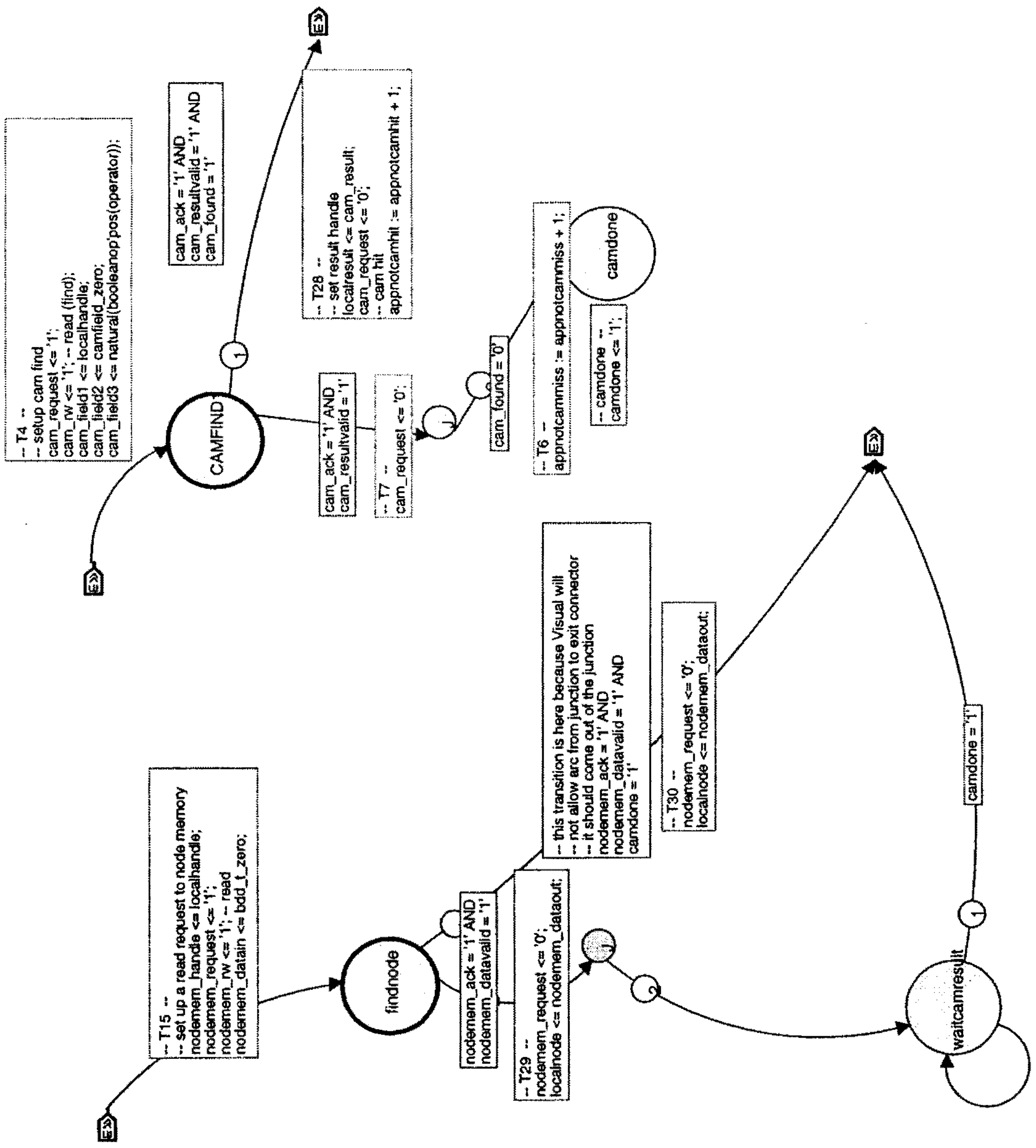


Figure 9. APPLY_NOT.CAMFIND



```
-- T4 --
-- setup cam find
cam_request <= '1';
cam_rw <= '1'; -- read (find);
cam_field1 <= localhandle;
cam_field2 <= camfield_zero;
cam_field3 <= natural(booleanop'pos(operator));

cam_ack = '1' AND
cam_resultvalid = '1' AND
cam_found = '1'

-- T28 --
-- set result handle
localresult <= cam_result;
cam_request <= '0';
-- cam hit
appnotcamhit := appnotcamhit + 1;

cam_ack = '1' AND
cam_resultvalid = '1'

-- T7 --
cam_request <= '0';

cam_found = '0'

-- T6 --
appnotcammiss := appnotcammiss + 1;

-- camdone --
camdone <= '1';

-- T15 --
-- set up a read request to node memory
nodemem_handle <= localhandle;
nodemem_request <= '1';
nodemem_rw <= '1'; -- read
nodemem_datain <= bdd_t_zero;

nodemem_ack = '1' AND
nodemem_datavalid = '1'

-- T29 --
nodemem_request <= '0';
localnode <= nodemem_dataout;

-- this transition is here because Visual will
-- not allow arc from junction to exit connector
-- it should come out of the junction
nodemem_ack = '1' AND
nodemem_datavalid = '1' AND
camdone = '1'

-- T30 --
nodemem_request <= '0';
localnode <= nodemem_dataout;

camdone = '1'
```

Figure 10. APPLY.TOP

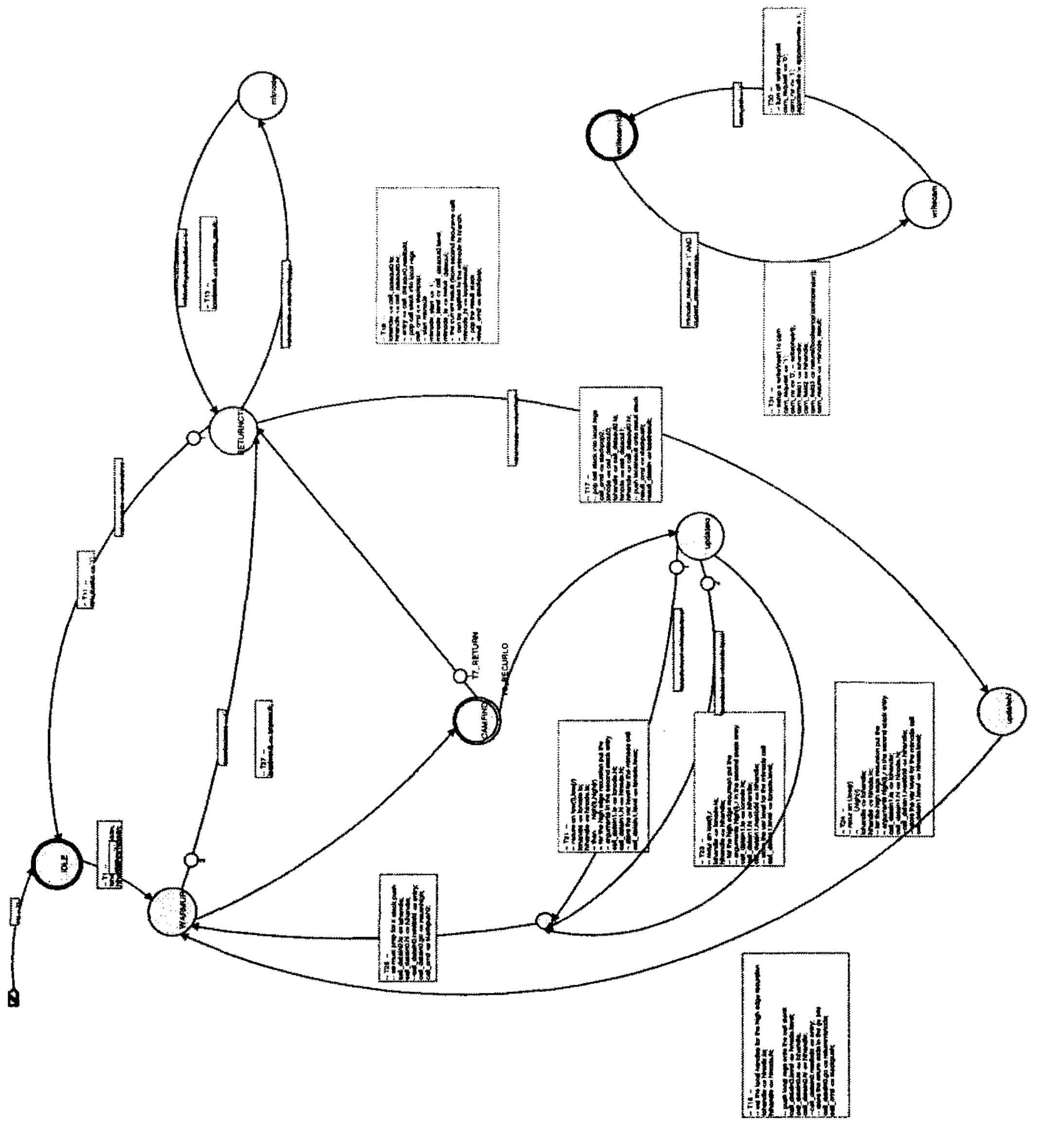
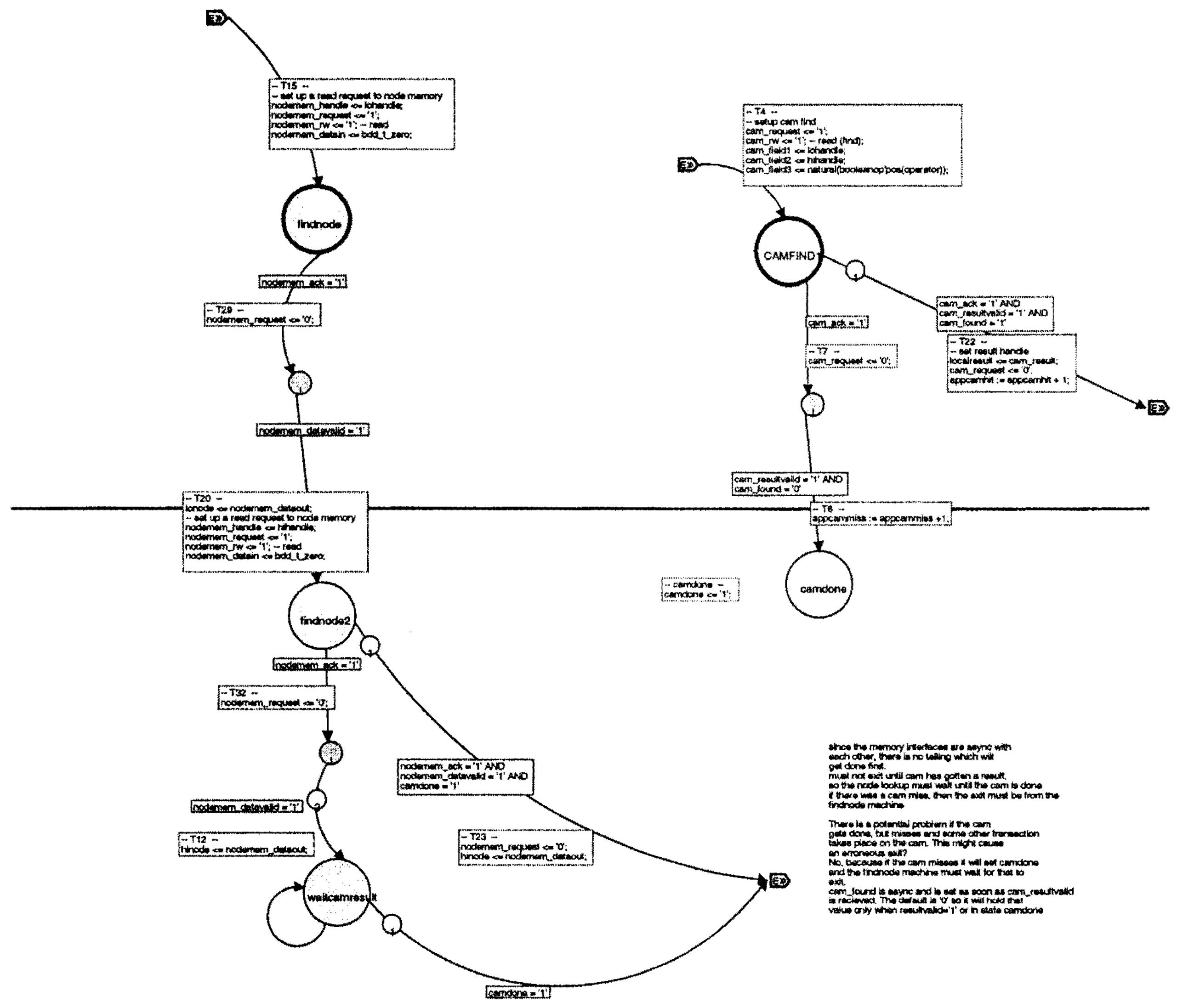


Figure 11. APPLY.CAMFIND



```
-- T15 --
-- set up a read request to node memory
nodemem_handle <= lohandle;
nodemem_request <= '1';
nodemem_rw <= '1'; -- read
nodemem_datain <= bdd_t_zero;
```

```
-- T29 --
nodemem_request <= '0';
```

```
-- T20 --
lonode <= nodemem_dataout;
-- set up a read request to node memory
nodemem_handle <= hihandle;
nodemem_request <= '1';
nodemem_rw <= '1'; -- read
nodemem_datain <= bdd_t_zero;
```

```
-- T32 --
nodemem_request <= '0';
```

```
-- T12 --
hinode <= nodemem_dataout;
```

```
-- T23 --
nodemem_request <= '0';
hinode <= nodemem_dataout;
```

```
-- T4 --
-- setup cam find
cam_request <= '1';
cam_rw <= '1'; -- read (find);
cam_field1 <= lohandle;
cam_field2 <= hihandle;
cam_field3 <= natural(booleanop'pos(operator));
```

```
-- T7 --
cam_request <= '0';
```

```
-- T22 --
-- set result handle
localresult <= cam_result;
cam_request <= '0';
appcamhit := appcamhit + 1;
```

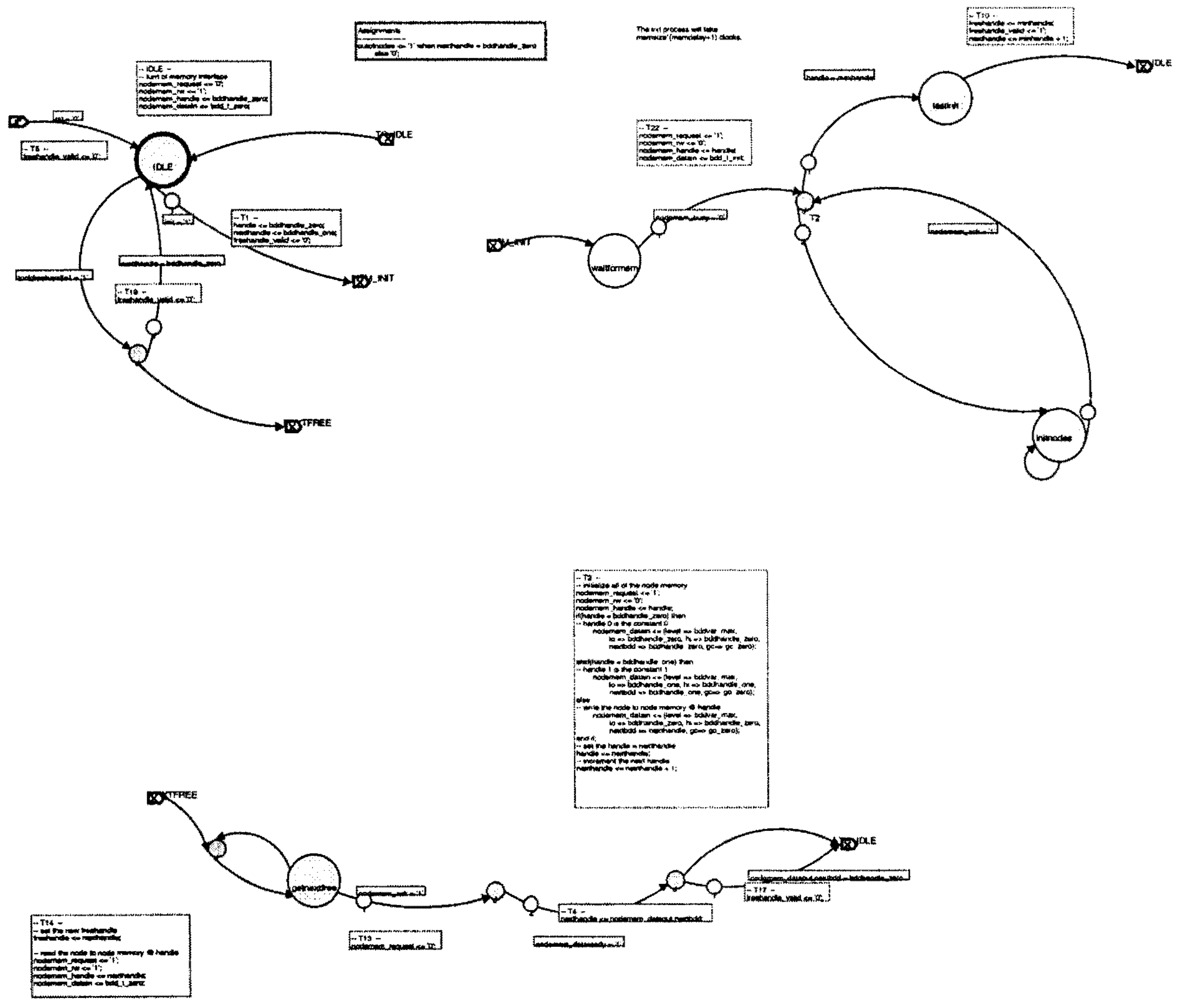
```
-- T6 --
appcammiss := appcammiss +1;
```

```
-- camdone --
camdone <= '1';
```

since the memory interfaces are async with
each other, there is no telling which will
get done first.
must not exit until cam has gotten a result.
so the node lookup must wait until the cam is done
if there was a cam miss, then the exit must be from the
findnode machine

There is a potential problem if the cam
gets done, but misses and some other transaction
takes place on the cam. This might cause
an erroneous exit?
No, because if the cam misses it will set camdone
and the findnode machine must wait for that to
exit.
cam_found is async and is set as soon as cam_resultvalid
is recieved. The default is '0' so it will hold that
value only when resultvalid='1' or in state camdone

Figure 12. FREENODECNTL



```
Assignments
-----------
outofnodes <= '1' when nexthandle = bddhandle_zero
    else '0';
```

```
-- IDLE --
-- turn of memory interface
nodemem_request <= '0';
nodemem_rw <= '1';
nodemem_handle <= bddhandle_zero;
nodemem_datain <= bdd_t_zero;
```

```
-- T5 --
freehandle_valid <= '0';
```

```
-- T1 --
handle <= bddhandle_zero;
nexthandle <= bddhandle_one;
freehandle_valid <= '0';
```

```
-- T19 --
freehandle_valid <= '0';
```

The init process will take
memsize*(memdelay+1) clocks.

```
-- T22 --
nodemem_request <= '1';
nodemem_rw <= '0';
nodemem_handle <= handle;
nodemem_datain <= bdd_t_init;
```

```
-- T10 --
freehandle <= nexthandle;
freehandle_valid <= '1';
nexthandle <= nexthandle + 1;
```

```
-- T3 --
-- initialize all of the node memory
nodemem_request <= '1';
nodemem_rw <= '0';
nodemem_handle <= handle;
if(handle = bddhandle_zero) then
-- handle 0 is the constant 0
      nodemem_datain <= (level => bddvar_max,
            lo => bddhandle_zero, hi => bddhandle_zero,
            nextbdd => bddhandle_zero, gc=> gc_zero);

elsif(handle = bddhandle_one) then
-- handle 1 is the constant 1
      nodemem_datain <= (level => bddvar_max,
            lo => bddhandle_one, hi => bddhandle_one,
            nextbdd => bddhandle_one, gc=> gc_zero);
else
-- write the node to node memory @ handle
      nodemem_datain <= (level => bddvar_max,
            lo => bddhandle_zero, hi => bddhandle_zero,
            nextbdd => nexthandle, gc=> gc_zero);
end if;
-- set the handle = nexthandle
handle <= nexthandle;
-- increment the next handle
nexthandle <= nexthandle + 1;
```

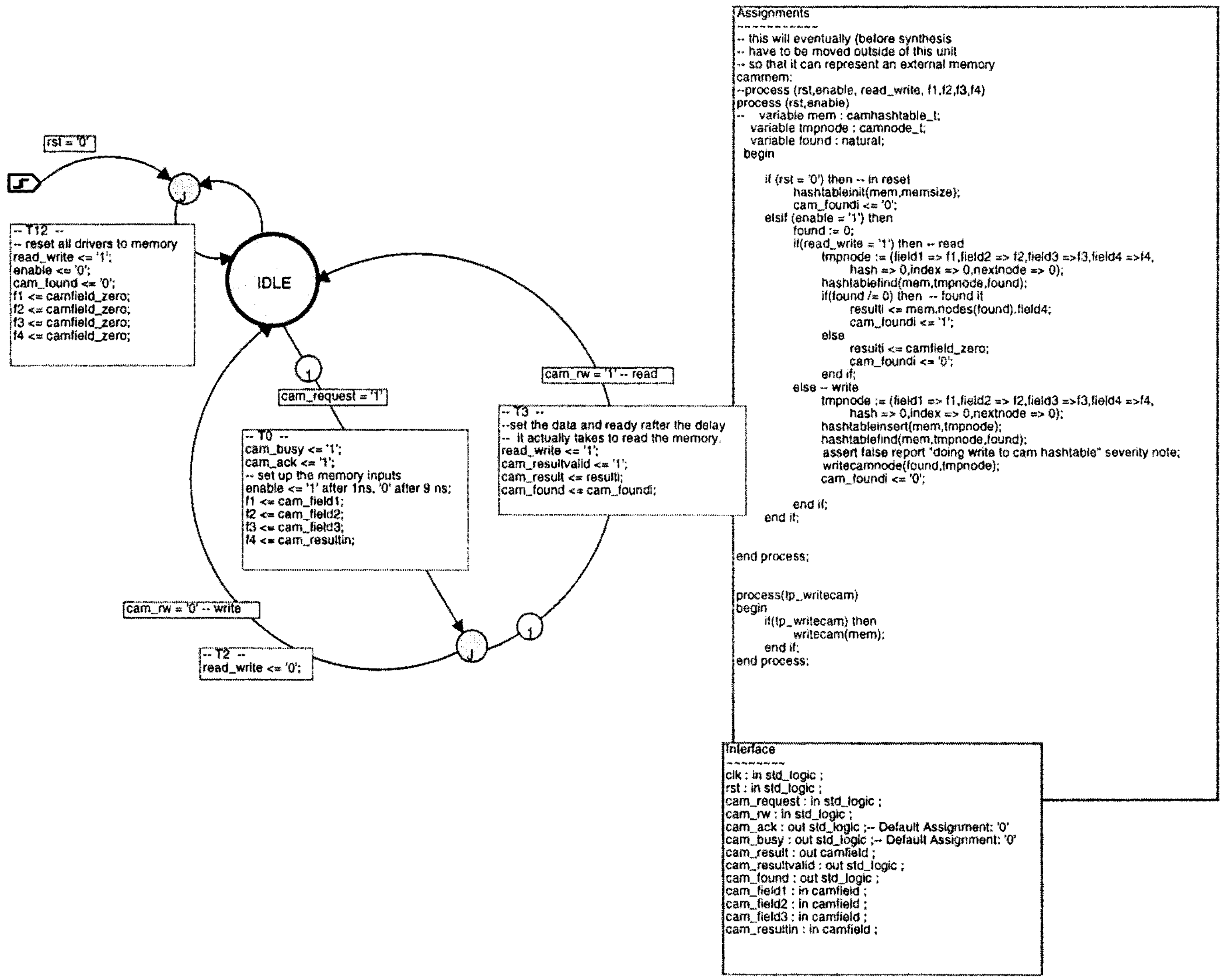
```
-- T14 --
-- set the new freehandle
freehandle <= nexthandle;

-- read the node to node memory @ handle
nodemem_request <= '1';
nodemem_rw <= '1';
nodemem_handle <= nexthandle;
nodemem_datain <= bdd_t_zero;
```

```
-- T13 --
nodemem_request <= '0';
```

```
-- T4 --
nexthandle <= nodemem_dataout.nextbdd;
```

```
-- T17 --
freehandle_valid <= '0';
```

Figure 13. CAM



```
-- T12 --
-- reset all drivers to memory
read_write <= '1';
enable <= '0';
cam_found <= '0';
f1 <= camfield_zero;
f2 <= camfield_zero;
f3 <= camfield_zero;
f4 <= camfield_zero;
```

```
-- T0 --
cam_busy <= '1';
cam_ack <= '1';
-- set up the memory inputs
enable <= '1' after 1ns, '0' after 9 ns;
f1 <= cam_field1;
f2 <= cam_field2;
f3 <= cam_field3;
f4 <= cam_resultin;
```

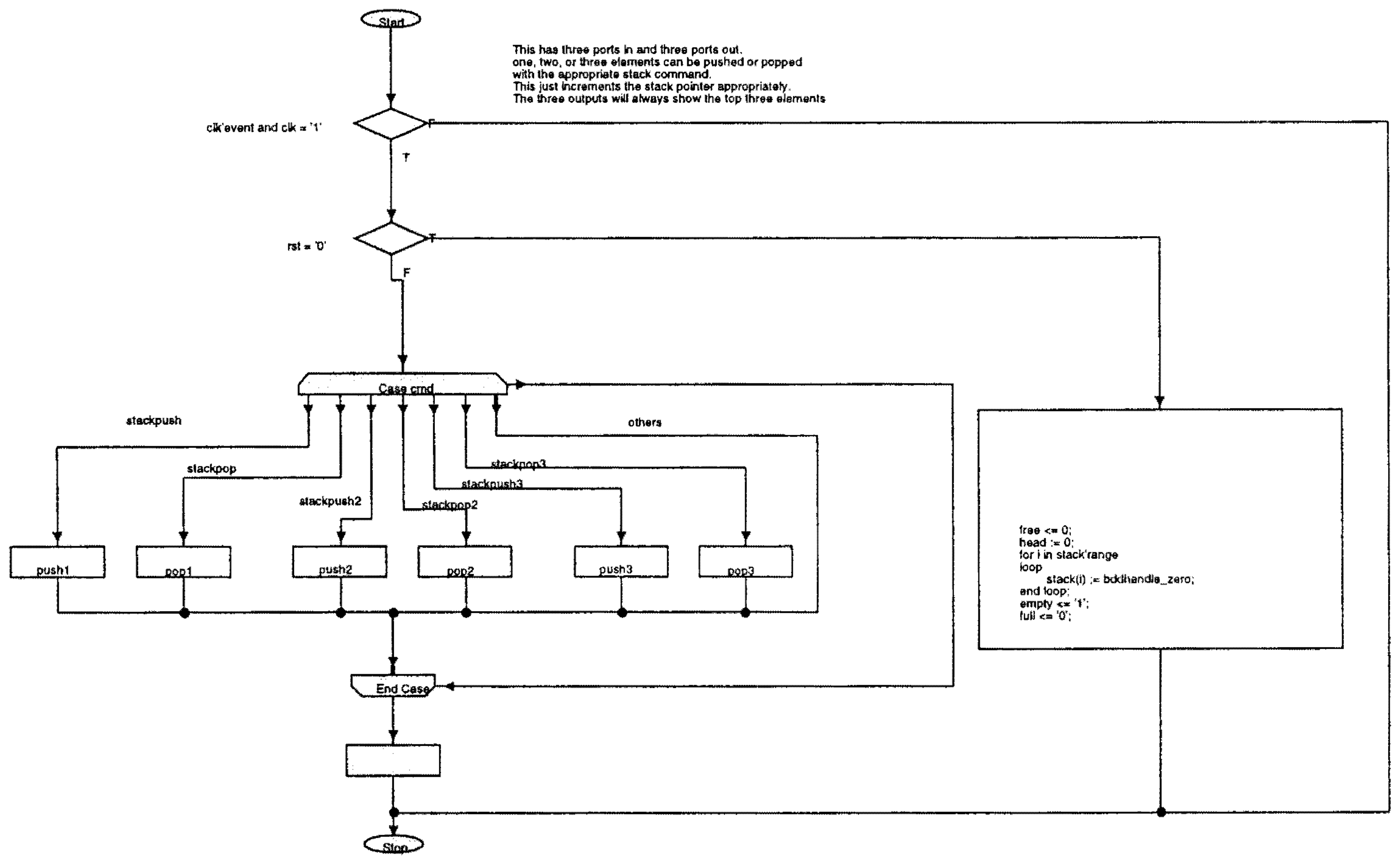
```
-- T3 --
--set the data and ready rafter the delay
--  it actually takes to read the memory.
read_write <= '1';
cam_resultvalid <= '1';
cam_result <= resulti;
cam_found <= cam_foundi;
```

```
-- T2 --
read_write <= '0';
```

```
Assignments
~~~~~~~~~~~
-- this will eventually (before synthesis
-- have to be moved outside of this unit
-- so that it can represent an external memory
cammem:
--process (rst,enable, read_write, f1,f2,f3,f4)
process (rst,enable)
--    variable mem : camhashtable_t;
   variable tmpnode : camnode_t;
   variable found : natural;
 begin

      if (rst = '0') then -- in reset
            hashtableinit(mem,memsize);
            cam_foundi <= '0';
      elsif (enable = '1') then
            found := 0;
            if(read_write = '1') then -- read
                  tmpnode := (field1 => f1,field2 => f2,field3 =>f3,field4 =>f4,
                        hash => 0,index => 0,nextnode => 0);
                  hashtablefind(mem,tmpnode,found);
                  if(found /= 0) then  -- found it
                        resulti <= mem.nodes(found).field4;
                        cam_foundi <= '1';
                  else
                        resulti <= camfield_zero;
                        cam_foundi <= '0';
                  end if;
            else -- write
                  tmpnode := (field1 => f1,field2 => f2,field3 =>f3,field4 =>f4,
                        hash => 0,index => 0,nextnode => 0);
                  hashtableinsert(mem,tmpnode);
                  hashtablefind(mem,tmpnode,found);
                   assert false report "doing write to cam hashtable" severity note;
                   writecamnode(found,tmpnode);
                  cam_foundi <= '0';

            end if;
      end if;


end process;


process(tp_writecam)
begin
      if(tp_writecam) then
            writecam(mem);
      end if;
end process;
```

```
Interface
~~~~~~~~~
clk : in std_logic ;
rst : in std_logic ;
cam_request : in std_logic ;
cam_rw : in std_logic ;
cam_ack : out std_logic ;-- Default Assignment: '0'
cam_busy : out std_logic ;-- Default Assignment: '0'
cam_result : out camfield ;
cam_resultvalid : out std_logic ;
cam_found : out std_logic ;
cam_field1 : in camfield ;
cam_field2 : in camfield ;
cam_field3 : in camfield ;
cam_resultin : in camfield ;
```

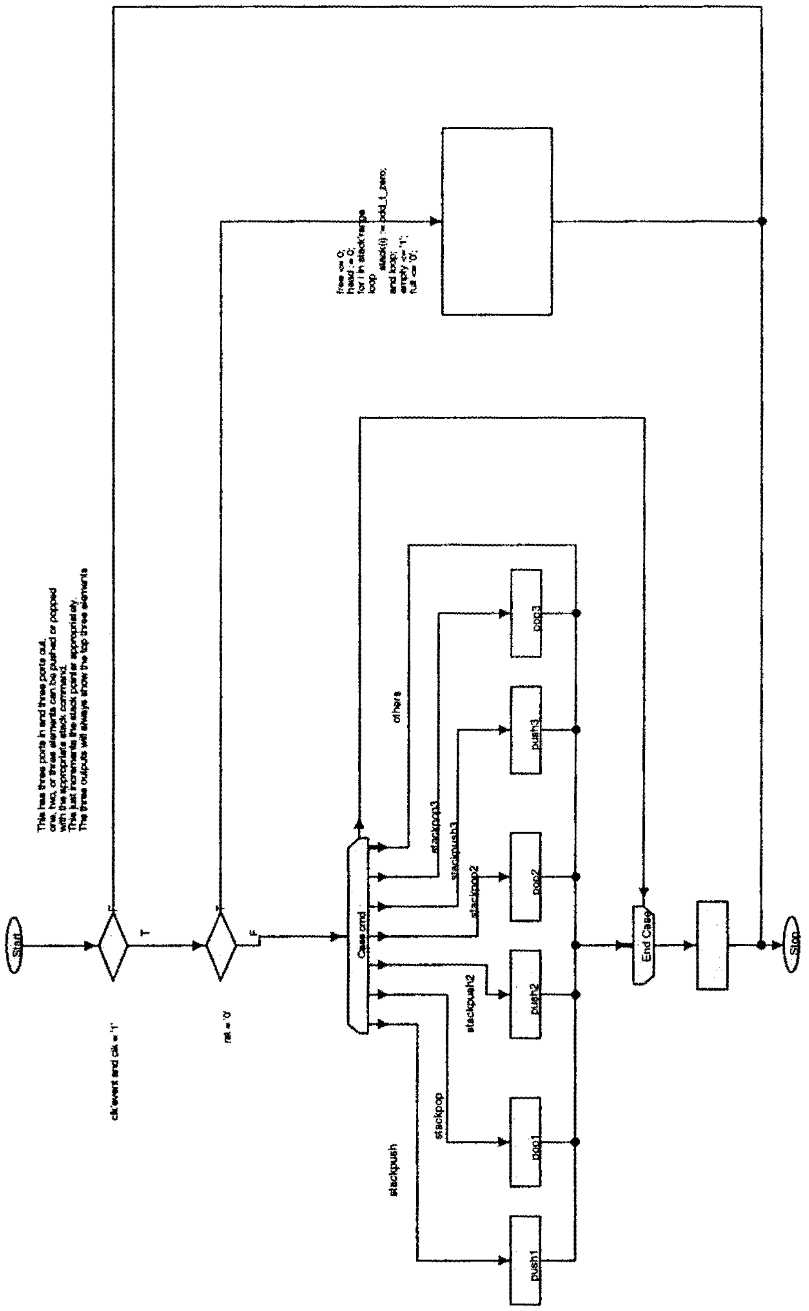
Figure 14. HANDLESTACK



This has three ports in and three ports out.
one, two, or three elements can be pushed or popped
with the appropriate stack command.
This just increments the stack pointer appropriately.
The three outputs will always show the top three elements

```
free <= 0;
head := 0;
for i in stack'range
loop
        stack(i) := bddhandle_zero;
end loop;
empty <= '1';
full <= '0';
```

Figure 15. BDDSTACK



```
--VHDL code written by Robert Hatt for
--Masters Thesis at Portland State University
------------------------------------------
------------------------------------------
-- Date      : Thu Apr  6 13:01:13 2000
--
-- Author    : Bob Hatt
--
-- Company   : PSU
--
-- Description : Package of data types and
--functions to be used for bdd manipulation
--
------------------------------------------
------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
-- use ieee.numeric_std.all;

package  kernel  is

-- subtype bddhandle is UNSIGNED(31 downto 0);
-- subtype hashkey is UNSIGNED(31 downto 0);
-- subtype vartype is UNSIGNED(31 downto 0);
-- constant uniquesize_c : UNSIGNED(31 downto 0);
constant bddmemsize : natural :=  100000;  --1024*1024;
constant bddvarsize : natural := 1000;
constant bdduniquetablesize : natural := 50003 ;  --500997; --311;
subtype bddhandle is natural;  -- range 0 to bddmemsize - 1;
subtype bddvar is natural;  -- range 0 to bddmaxvarnum - 1;
subtype hashkey is natural;  --?? range 0 to bdduniqhashsize - 1;
subtype gc_t is natural; -- range 0 to 3 ???
constant hashkey_zero : hashkey := 0;
constant bddhandle_zero : bddhandle := 0;
constant bddhandle_one : bddhandle := 1;
constant bddvar_zero : bddvar := 0;
constant bddvar_max : bddvar := bddvar'right;  -- this will need to
change for vectors
constant bdd_minhandle : bddhandle := 2;
constant bdd_maxhandle : bddhandle := bddmemsize-1;
constant gc_zero : gc_t := 0;

--type bdd_t;
type bdd_t is record
level : bddvar;
lo  : bddhandle;
hi  : bddhandle;
nextbdd : bddhandle;
gc : gc_t;
      end record;
constant bdd_t_zero : bdd_t :=
(bddvar_zero,bddhandle_zero,bddhandle_zero,bddhandle_zero,gc_zero)
;
constant bdd_t_init : bdd_t :=
(bddvar_max,bddhandle_zero,bddhandle_zero,bddhandle_zero,gc_zero)
;


type bdd_vec_t is array (natural range <>) of bdd_t;

function hash2(a,b : bddhandle) return hashkey;
function hash3(a,b,c : bddhandle) return hashkey;
function bdd_hash(b : bdd_t ; prime : natural) return hashkey;

type bddhandle_vec_t is array (natural range <>) of bddhandle;
type bddhandle_vec_a is access bddhandle_vec_t;


-- return codes for recursive calls
constant returndone : gc_t := 0;
constant returnhigh : gc_t := 1;
constant returnmknode : gc_t := 3;

-- stack cmds
-- subtype stackcmd is std_logic_vector(1 downto 0);
-- constant stackpush1 : std_logic_vector := "01";
-- constant stackpop : std_logic_vector := "10";
type stackcmd is (stacknop,stackpush,stackpush2,stackpush3,stack-
pop,stackpop2,stackpop3);
type booleanop is
(booleanop_zero,booleanop_and,booleanop_greater,booleanop_three,bo
oleanop_less,booleanop_five,

booleanop_xor,booleanop_or,booleanop_nor,booleanop_biimp,booleano
p_ten,booleanop_revimp,booleanop_not,
booleanop_imp,booleanop_nand,booleanop_fifteen);
-- subtype booleanop is UNSIGNED(3 downto 0);
-- constant booleanop_zero : UNSIGNED :=  "0000";
-- constant booleanop_and : UNSIGNED :=   "0001";
-- constant booleanop_nand : UNSIGNED :=  "1110";
-- constant booleanop_or : UNSIGNED :=    "0111";
-- constant booleanop_nor : UNSIGNED :=   "1000";
-- constant booleanop_xor : UNSIGNED :=   "0110";
-- constant booleanop_imp : UNSIGNED :=   "1101";
-- constant booleanop_biimp : UNSIGNED :=  "1001";
-- constant booleanop_revimp : UNSIGNED := "1011";
-- constant booleanop_greater : UNSIGNED:= "0010";
-- constant booleanop_less : UNSIGNED :=  "0100";
-- constant booleanop_not : UNSIGNED :=   "1100";

constant bddcamsize : natural := 65536;  -- 64k x 136 bits wide
subtype  camfield is natural;   -- same size as handle?
constant camfield_zero : camfield := 0;
-- type camnode_t is record
-- field1 : natural;
-- field2 : natural;
-- field3 : natural;
-- field4 : natural;
-- result : natural;
-- previndex : natural;
-- nextindex : natural;
-- index : natural;
-- end record;
-- type camnode_vec_t is array (natural range <>) of camnode_t;
-- constant camnode_t_zero : camnode_t := (0,0,0,0,0,0,0,0);
end;



------------------------------------------
------------------------------------------
-- Date      : Thu Apr  6 13:26:57 2000
--
-- Author    :
--
-- Company   :
--
-- Description :
--
------------------------------------------
------------------------------------------
-- library ieee;
-- use ieee.std_logic_1164.all;
-- use ieee.numeric_std.all;

package body  kernel  is

-- constant uniquesize_c : UNSIGNED(31 downto 0) := X"0000001f";


function hash2(a,b : bddhandle) return hashkey is
begin
return ((((a+b) * (a+b+1))/2) + a);
end hash2;
function hash3(a,b,c : bddhandle) return hashkey is
begin
return(hash2(hash2(a,b),c));
end hash3;
```

```
function bdd_hash(b : bdd_t; prime : natural) return hashkey is
begin
return(hash3(b.level,b.lo,b.hi) MOD prime);
end bdd_hash;



end;



-----------------------------------------
-----------------------------------------
-- Date      : Mon May 15 11:13:50 2000
--
-- Author    : Bob Hatt
--
-- Company   : Portland State University
--
-- Description :
--
-----------------------------------------
-----------------------------------------
library work;
use work.kernel.all;

package campkg is
-- cam size is specified in kernel
-- constant bddcamsize : natural := 251;


type camnode_t is record
field1 : natural;
field2 : natural;
field3 : natural;
field4 : natural;
index : natural;
hash : natural;
nextnode : natural;
end record;
type camnode_vec_t is array (natural range <>) of camnode_t;
type camnode_vec_p is access camnode_vec_t;
type camhashtable_t is record
size : positive;
freeindex : natural;
nodes :  camnode_vec_p;
full : boolean;
init : boolean;
end record;
constant camnode_t_zero : camnode_t := (0,0,0,0,0,0,0);
-- procedure hashtableinit(table : inout camhashtable_t);
procedure hashtableinit(table : inout camhashtable_t; size : in positive);
procedure hashtablefind(table : inout camhashtable_t;node: in
camnode_t;found:out natural);
procedure hashtableremove(table : inout camhashtable_t;node: in
camnode_t;found : out natural);
procedure hashtableinsert(table : inout camhashtable_t;node: in
camnode_t);
procedure writecamnode(index : integer;node : camnode_t);
procedure writecam(table : inout camhashtable_t);
end;


-----------------------------------------
-----------------------------------------
-- Date      : Mon May 15 11:14:01 2000
--
-- Author    : Bob Hatt
--
-- Company   : Portland State University
--
-- Description :
--
-----------------------------------------
-----------------------------------------
use std.textio.all;

package body  campkg  is


procedure hashtableinit(table : inout camhashtable_t;size : in positive) is

begin
if(NOT table.init) then
table.size := size;
table.nodes := new camnode_vec_t(1 to size);
for i in table.nodes'range
loop
table.nodes(i) := camnode_t_zero;
end loop;
table.freeindex := table.nodes'low;
table.init := true;
table.full := false;
end if;
end hashtableinit;

procedure hashtabledelete(table : inout camhashtable_t) is
begin
deallocate(table.nodes);

end;


procedure hashtablefind(table : inout camhashtable_t;node: in
camnode_t;found:out natural) is
variable hash,index,nextfree,previndex : natural;
begin
hash := (hash3(node.field1,node.field2,node.field3) MOD
table.nodes'high)+1;
index := table.nodes(hash).index;

-- walk the list until you find one that matches the node,

found := 0;
while( index /= 0)
loop
if(table.nodes(index).field1 = node.field1 AND
  table.nodes(index).field2 = node.field2 AND
  table.nodes(index).field3 = node.field3)  then
-- this is the one to remove
found := index;
return;
-- exit the loop
end if;
previndex := index;
index:= table.nodes(index).nextnode;
end loop;
return;

end hashtablefind;

procedure hashtableremove(table : inout camhashtable_t;node: in
camnode_t;found : out natural) is
variable hash,index,nextfree,previndex : natural;
begin
hash := (hash3(node.field1,node.field2,node.field3) MOD
table.nodes'high)+1;
index := table.nodes(hash).index;

-- walk the list until you find one that matches the node,

found := 0;
previndex := 0;
while( index /= 0)
loop
if(table.nodes(index).field1 = node.field1 AND
table.nodes(index).field2 = node.field2 AND
table.nodes(index).field3 = node.field3) then
-- this is the one to remove
found := index;

if(previndex = 0) then
-- node is at head of chain.
table.nodes(hash).index := table.nodes(index).nextnode;

else
-- there is an prevous node in the chain.
table.nodes(previndex).nextnode := table.nodes(index).nextnode;
end if;
table.nodes(index) := (field1 => 0,field2 => 0,field3 => 0, field4=> 0,
index => table.nodes(index).index,hash => 0,nextnode =>0);


-- exit the procedure
return;
end if;
previndex := index;
index:= table.nodes(index).nextnode;
end loop;
return;

end hashtableremove;


procedure hashtableinsert(table : inout camhashtable_t;node: in
camnode_t) is
variable hash,index,nextfree : natural;
variable tmpnode : camnode_t;
variable found: natural;
begin
hash := (hash3(node.field1,node.field2,node.field3) MOD
table.nodes'high)+1;
index := table.nodes(hash).index;

if(table.full) then
-- remove the current freehandle from the table
-- then do the normal insert on the free handle
tmpnode := (field1 => table.nodes(table.freeindex).field1,
field2 => table.nodes(table.freeindex).field2,
field3 => table.nodes(table.freeindex).field3,
field4 => table.nodes(table.freeindex).field4,
index => 0,hash => 0,nextnode =>0);
hashtableremove(table,tmpnode,found);
if(found /= table.freeindex) then
assert false report "INSERT: found /= freeindex" severity error;
end if;
if(found = 0) then
assert false report "INSERT: found = 0" severity error;
end if;


end if;


hashtablefind(table,node,found);
if(found /= 0) then
assert false report "node alread in table!!!" severity error;
end if;
table.nodes(table.freeindex).field1 := node.field1;
table.nodes(table.freeindex).field2 := node.field2;
table.nodes(table.freeindex).field3 := node.field3;
table.nodes(table.freeindex).field4 := node.field4;
table.nodes(table.freeindex).nextnode := table.nodes(hash).index;
table.nodes(table.freeindex).hash := hash;
table.nodes(hash).index := table.freeindex;

if(table.freeindex = table.size) then
table.full := true;
table.freeindex := 1;
else
table.freeindex := table.freeindex + 1;
end if;

return;
end hashtableinsert;

procedure writecamnode(index : integer;node : camnode_t) is
variable tmpline : line;
begin
write(tmpline,index);
write(tmpline,':');
write(tmpline,node.field1);
write(tmpline,',');
write(tmpline,node.field2);
write(tmpline,',');
write(tmpline,node.field3);
write(tmpline,',');
write(tmpline,node.field4);
-- writeline(outfile,tmpline);
 writeline(output,tmpline);


end writecamnode;

procedure writecam(table : inout camhashtable_t) is
begin
for i in 1 to table.size
loop
writecamnode(i,table.nodes(i));
end loop;

end writecam;

end;


-----------------------------------------
-----------------------------------------
-- Date        : Mon May  8 19:04:29 2000
--
-- Author      : Bob Hatt
--
-- Company     : Portland State University
--
-- Description :
--
-----------------------------------------
-----------------------------------------
library work; use work.kernel.all;
library std; use std.textio.all;

package bdddebug is

procedure writenode(handle: bddhandle;node : bdd_t);
procedure writenode_rec(mem: in bdd_vec_t; handle:bddhandle);
procedure writenodetable(mem : in bdd_vec_t);
procedure printset(r : bddhandle;mem : in bdd_vec_t);
end;


-----------------------------------------
-----------------------------------------
-- Date        : Mon May  8 19:04:38 2000
--
-- Author      : Bob Hatt
--
-- Company     : Portland State University
--
-- Description :
--
-----------------------------------------
-----------------------------------------
package body bdddebug is

procedure writenodetable(mem : in bdd_vec_t) is
-- file outfile : text;
```

```
variable tmpline : line;
begin
-- open the file

-- file_open(outfile,"nodetable.txt",WRITE_MODE);
for i in mem'range loop
-- write the element in a textual format
-- to the output file
writenode(i,mem(i));

end loop;

-- close the file
-- file_close(outfile);

end writenodetable;


procedure writenode(handle: bddhandle;node : bdd_t) is
variable tmpline : line;
begin
write(tmpline,handle);
write(tmpline,': ');
write(tmpline,node.level);
write(tmpline,',');
write(tmpline,node.lo);
write(tmpline,',');
write(tmpline,node.hi);
write(tmpline,',');
write(tmpline,node.nextbdd);
write(tmpline,',');
write(tmpline,node.gc);
-- writeline(outfile,tmpline);
 writeline(output,tmpline);


end writenode;

procedure writenode_rec(mem: in bdd_vec_t; handle:bddhandle) is
variable node:bdd_t;
begin
node := mem(handle);
if((node.lo /= bddhandle_zero) AND
(node.lo /= bddhandle_one)) then
writenode_rec(mem,node.lo);
end if;
if((node.hi /= bddhandle_zero) AND
(node.hi /= bddhandle_one)) then
writenode_rec(mem,node.hi);
end if;
writenode(handle,node);

end writenode_rec;



procedure printset(r : bddhandle;mem : in bdd_vec_t) is

variable set : bddhandle_vec_t(0 to bddvarsize);
variable tmpline : line;

procedure printset_rec(r : bddhandle) is
variable first : integer;
begin
if(r = bddhandle_zero) then
return;
elsif(r = bddhandle_one) then
write(tmpline,'<');
writeline(output,tmpline);
first := 1;
for i in 0 to bddvarsize
loop
if(set(i) > 0) then
if(first = 0) then
write(tmpline,',');
writeline(output,tmpline);
end if;
first := 0;
write(tmpline,mem(r).level);
writeline(output,tmpline);
if(set(i) = 2) then
write(tmpline,1);
else
write(tmpline,0);
end if;
writeline(output,tmpline);
end if;
write(tmpline,'>');
writeline(output,tmpline);

end loop;
else
set(mem(r).level) := 1;
printset_rec(mem(r).lo);
set(mem(r).level) := 2;
printset_rec(mem(r).hi);
set(mem(r).level) := 0;


end if;

return;
end printset_rec;
begin
printset_rec(r);
end printset;

end;


library ieee;
use ieee.STD_LOGIC_1164.all;
library work;
use work.kernel.all;
use work.campkg.all;
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.ALL;


entity cam is
 generic (
        memsize   : NATURAL          := bddcamsize;
        readdelay  : NATURAL;
        writedelay : NATURAL
        );
 port (
      clk           : in std_logic;
      rst           : in std_logic;
      cam_request   : in std_logic;
      cam_rw        : in std_logic;
      cam_ack       : out std_logic;
      cam_busy      : out std_logic;
      cam_result    : out camfield;
      cam_resultvalid : out std_logic;
      cam_found     : out std_logic;
      cam_field1    : in camfield;
      cam_field2    : in camfield;
      cam_field3    : in camfield;
      cam_resultin  : in camfield
      );

end cam;


architecture cam of cam is

 constant camnode_t_zero  : camnode_t     := (field1 => 0, field2 => 0,
                             field3 => 0, field4 => 0, index
                              => 0, hash => 0, nextnode => 0)
  ;
 constant maxdelay       : NATURAL        := readdelay;
 shared variable mem     : camhashtable_t;
 signal  enable          : std_logic;
 signal  read_write      : std_logic;
 signal  delaycnt        : NATURAL range 0 to maxdelay;
 signal  f1              : camfield;
 signal  f2              : camfield;
 signal  f3              : camfield;
 signal  f4              : camfield;
 signal  resulti         : camfield;
 signal  cam_foundi      : std_logic;
 signal  tp_writecam     : BOOLEAN        := false;

 type visual_IDLE_states is (IDLE);

 signal visual_IDLE_current, visual_IDLE_next : visual_IDLE_states;
 attribute STATE_VECTOR of cam :
       architecture is "visual_IDLE_current";


begin



 -- Combinational process
 cam_IDLE_comb:
 process (rst, cam_request, cam_field1, cam_field2, cam_field3,
cam_resultin,
       cam_rw, resulti, cam_foundi, visual_IDLE_current)
 begin
  cam_ack <= '0';
  cam_busy <= '0';
  f1 <= camfield_zero;
  f2 <= camfield_zero;
  f3 <= camfield_zero;
  f4 <= camfield_zero;


  if (rst = '0') then
   -- reset all drivers to memory
   read_write <= '1';
   enable <= '0';
   cam_found <= '0';
   f1 <= camfield_zero;
   f2 <= camfield_zero;
   f3 <= camfield_zero;
   f4 <= camfield_zero;
   visual_IDLE_next <= IDLE;
  else

   case visual_IDLE_current is
    when IDLE =>
     if ((cam_request = '1') and (cam_rw = '1')) then
      cam_busy <= '1';
      cam_ack <= '1';
      -- set up the memory inputs
      enable <= '1' after 1ns, '0' after 9 ns;
      f1 <= cam_field1;
      f2 <= cam_field2;
      f3 <= cam_field3;
      f4 <= cam_resultin;
      --set the data and ready rafter the delay
      --  it actually takes to read the memory.
      read_write <= '1';
      cam_resultvalid <= '1';
      cam_result <= resulti;
      cam_found <= cam_foundi;
      visual_IDLE_next <= IDLE;
     elsif ((cam_request = '1') and (cam_rw = '0')) then
      cam_busy <= '1';
      cam_ack <= '1';
      -- set up the memory inputs
      enable <= '1' after 1ns, '0' after 9 ns;
      f1 <= cam_field1;
      f2 <= cam_field2;
      f3 <= cam_field3;
      f4 <= cam_resultin;
      read_write <= '0';
      visual_IDLE_next <= IDLE;
     else
      -- reset all drivers to memory
      read_write <= '1';
      enable <= '0';
      cam_found <= '0';
      f1 <= camfield_zero;
      f2 <= camfield_zero;
      f3 <= camfield_zero;
      f4 <= camfield_zero;
      visual_IDLE_next <= IDLE;
     end if;

    when others =>

     visual_IDLE_next <= IDLE;
   end case;
  end if;
end process;

cam_IDLE:
process (clk)
begin

 if (clk'event and clk = '1') then
  if (rst = '0') then
   visual_IDLE_current <= IDLE;
  else
   visual_IDLE_current <= visual_IDLE_next;
  end if;
 end if;
end process;


-- this will eventually (before synthesis
-- have to be moved outside of this unit
-- so that it can represent an external memory
cammem:
--process (rst,enable, read_write, f1,f2,f3,f4)
process (rst,enable)
--   variable mem : camhashtable_t;
  variable tmpnode : camnode_t;
  variable found : natural;
 begin

if (rst = '0') then -- in reset
hashtableinit(mem,memsize);
cam_foundi <= '0';
elsif (enable = '1') then
found := 0;
if(read_write = '1') then -- read
tmpnode := (field1 => f1,field2 => f2,field3 =>f3,field4 =>f4,
hash => 0,index => 0,nextnode => 0);
hashtablefind(mem,tmpnode,found);
if(found /= 0) then  -- found it
resulti <= mem.nodes(found).field4;
cam_foundi <= '1';
else
resulti <= camfield_zero;
cam_foundi <= '0';
end if;
else -- write
tmpnode := (field1 => f1,field2 => f2,field3 =>f3,field4 =>f4,
hash => 0,index => 0,nextnode => 0);
hashtableinsert(mem,tmpnode);
hashtablefind(mem,tmpnode,found);
--assert false report "doing write to cam hashtable" severity note;
--writecamnode(found,tmpnode);
cam_foundi <= '0';

end if;
end if;
```

```
 end process;


 process(tp_writecam)
 begin
 if(tp_writecam) then
 writecam(mem);
 end if;
 end process;
end cam;

-- This has three ports in and three ports out.
-- one, two, or three elements can be pushed or popped
-- with the appropriate stack command.
-- This just increments the stack pointer appropriately.
-- The three outputs will always show the top three elements

library ieee;
use ieee.STD_LOGIC_1164.all;
library work;
use work.kernel.all;
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.ALL;


entity bddstack is
 generic (
      size : NATURAL           := 6
      );
 port (
      clk    : in std_logic;
      rst    : in std_logic;
      cmd    : in stackcmd;
      datain0 : in bdd_t;
      datain1 : in bdd_t;
      datain2 : in bdd_t;
      head0   : out bdd_t;
      head1   : out bdd_t;
      head2   : out bdd_t;
      full    : out std_logic;
      empty   : out std_logic
      );

end bddstack;


architecture bddstack of bddstack is

 signal tp_head  : NATURAL;
 signal free     : NATURAL;


begin
 Start:process (clk, rst, datain0, datain1, datain2, cmd)
  variable stack : bdd_vec_t(0 to size + 4 );
  variable head  : NATURAL;
 begin  -- process
  if clk'event and clk = '1' then
   if rst = '0' then
    free <= 0;
    head := 0;
    for i in stack'range
    loop
    stack(i) := bdd_t_zero;
    end loop;
    empty <= '1';
    full <= '0';
   else
    -- "01" ==> push
    -- "10" ==> pop
    -- others ==> do nothing
    case cmd is
     when stackpush =>
      stack(head) := datain0;
      if(head < (size)) then
      head := (head + 1);
      end if;

     when stackpop =>
      if(head > 0) then
      head := head -1;
      end if;

     when stackpush2 =>
      stack(head+1) := datain0;
      stack(head) := datain1;
      if(head < (size)) then
      head := (head + 2);
      end if;

     when stackpop2 =>
      if(head > 1) then
      head := head -2;
      else
      head := 0;
      end if;

     when stackpush3 =>
      stack(head+2) := datain0;
      stack(head+1) := datain1;
      stack(head) := datain2;
      if(head < (size)) then
      head := (head + 3);
      end if;

     when stackpop3 =>
      if(head > 2) then
      head := head -3;
      else
      head := 0;
      end if;

     when others  =>
      null;
    end case;
    if(head = 0) then
    empty <= '1';
    else
    empty <= '0';
    end if;
    if(head >= size ) then
    full <= '1';
    else
    full <= '0';
    end if;


    head0 <= bdd_t_zero;
    head1 <= bdd_t_zero;
    head2 <= bdd_t_zero;
    if(head = 1) then
    head0 <= stack(head - 1);
    end if;
    if(head = 2) then
    head0 <= stack(head - 1);
    head1 <= stack(head - 2);
    elsif(head >2) then
    head0 <= stack(head-1);
    head1 <= stack(head-2);
    head2 <= stack(head-3);
    end if;

    tp_head <= head;
   end if;
  end if;
 end process;
end bddstack;

-- This has three ports in and three ports out.
-- one, two, or three elements can be pushed or popped
-- with the appropriate stack command.
-- This just increments the stack pointer appropriately.
-- The three outputs will always show the top three elements

library ieee;
use ieee.STD_LOGIC_1164.all;
library work;
use work.kernel.all;
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.ALL;


entity handlestack is
 generic (
      size : NATURAL          := 6
      );
 port (
     clk    : in std_logic;
     rst    : in std_logic;
     cmd    : in stackcmd;
     datain0 : in bddhandle;
     datain1 : in bddhandle;
     datain2 : in bddhandle;
     head0  : out bddhandle;
     head1  : out bddhandle;
     head2  : out bddhandle;
     full   : out std_logic;
     empty  : out std_logic
     );

end handlestack;


architecture handlestack of handlestack is

 signal tp_head  : NATURAL;
 signal free     : NATURAL;


begin
 Start:process (clk, rst, datain0, datain1, datain2, cmd)
  variable stack : bddhandle_vec_t(0 to size + 4 );
  variable head  : NATURAL;
 begin  -- process
  if clk'event and clk = '1' then
   if rst = '0' then
     free <= 0;
     head := 0;
     for i in stack'range
     loop
     stack(i) := bddhandle_zero;
     end loop;
     empty <= '1';
     full <= '0';
   else
     -- "01" ==> push
     -- "10" ==> pop
     -- others ==> do nothing
     case cmd is

      when stackpush =>
       stack(head) := datain0;
       if(head < (size)) then
       head := (head + 1);
       end if;

      when stackpop =>
       if(head > 0) then
       head := head -1;
       end if;

      when stackpush2 =>
       stack(head+1) := datain0;
       stack(head) := datain1;
       if(head < (size)) then
       head := (head + 2);
       end if;

      when stackpop2 =>
       if(head > 1) then
       head := head -2;
       else
       head := 0;
       end if;

      when stackpush3 =>
       stack(head+2) := datain0;
       stack(head+1) := datain1;
       stack(head) := datain2;
       if(head < (size)) then
       head := (head + 3);
       end if;

      when stackpop3 =>
       if(head > 2) then
       head := head -3;
       else
       head := 0;
       end if;

      when others =>
       null;
     end case;
     if(head = 0) then
     empty <= '1';
     else
     empty <= '0';
     end if;
     if(head >= size ) then
     full <= '1';
     else
     full <= '0';
     end if;


     head0 <= bddhandle_zero;
     head1 <= bddhandle_zero;
     head2 <= bddhandle_zero;
     if(head = 1) then
     head0 <= stack(head - 1);
     end if;
     if(head = 2) then
     head0 <= stack(head - 1);
     head1 <= stack(head - 2);
     elsif(head >2) then
     head0 <= stack(head-1);
     head1 <= stack(head-2);
     head2 <= stack(head-3);
     end if;

     tp_head <= head;
   end if;
  end if;
 end process;

end handlestack;

library ieee;
use ieee.STD_LOGIC_1164.all;
library bddlib;
use bddlib.kernel.all;
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.ALL;


entity freenodecntl is
```

```
 generic (
       minhandle : bddhandle          := bdd_minhandle;
       maxhandle : bddhandle          := bdd_maxhandle
       );
 port (
     clk           : in std_logic;
     rst           : in std_logic;
     init          : in std_logic;
     tookfreehandle1  : in std_logic;
     freehandle      : out bddhandle;
     freehandle_valid  : out std_logic;
     LOWONNODES      : out std_logic;
     OUTOFNODES      : out std_logic;
     nodemem_busy     : in std_logic;
     nodemem_ack      : in std_logic;
     nodemem_dataready : in std_logic;
     nodemem_dataout  : in bdd_t;
     nodemem_request  : out std_logic;
     nodemem_handle   : out bddhandle;
     nodemem_datain   : out bdd_t;
     nodemem_rw       : out std_logic
     );

end freenodecntl;

-- The init process will take
-- memsize*(memdelay+1) clocks.


architecture freenodecntl of freenodecntl is

 signal handle     : bddhandle;
 signal nexthandle  : bddhandle;

 type visual_currentstate_states is (IDLE, getnextfree, initnodes, lastinit,
                          waitformem);

 signal currentstate : visual_currentstate_states;
 attribute STATE_VECTOR of freenodecntl :
        architecture is "currentstate";


begin



 -- Synchronous process
 freenodecntl_IDLE:
 process (clk)
 begin

  if (clk'event and clk = '1') then
   if (rst = '0') then
     freehandle_valid <= '0';
     -- turn of memory interface
     nodemem_request <= '0';
     nodemem_rw <= '1';
     nodemem_handle <= bddhandle_zero;
     nodemem_datain <= bdd_t_zero;
     currentstate <= IDLE;
   else

     case currentstate is
      when IDLE =>
       if (init = '1') then
        handle <= bddhandle_zero;
        nexthandle <= bddhandle_one;
        freehandle_valid <= '0';
        currentstate <= waitformem;
       elsif (tookfreehandle1 = '1') then
        if (nexthandle = bddhandle_zero) then
          freehandle_valid <= '0';
          currentstate <= IDLE;
        else
          -- set the new freehandle
          freehandle <= nexthandle;

          -- read the node to node memory @ handle
          nodemem_request <= '1';
          nodemem_rw <= '1';
          nodemem_handle <= nexthandle;
          nodemem_datain <= bdd_t_zero;
          currentstate <= getnextfree;
        end if;
       else
        currentstate <= IDLE;
       end if;

      when getnextfree =>
       if ((nodemem_ack = '1') and (nodemem_dataready = '1')) then
        nodemem_request <= '0';
        nexthandle <= nodemem_dataout.nextbdd;
        if (nodemem_dataout.nextbdd = bddhandle_zero) then
          freehandle_valid <= '0';
          -- turn of memory interface
          nodemem_request <= '0';
          nodemem_rw <= '1';
          nodemem_handle <= bddhandle_zero;
          nodemem_datain <= bdd_t_zero;
          currentstate <= IDLE;
        else
          -- turn of memory interface
          nodemem_request <= '0';
          nodemem_rw <= '1';
          nodemem_handle <= bddhandle_zero;
          nodemem_datain <= bdd_t_zero;
          currentstate <= IDLE;
        end if;
       else
        -- set the new freehandle
        freehandle <= nexthandle;

        -- read the node to node memory @ handle
        nodemem_request <= '1';
        nodemem_rw <= '1';
        nodemem_handle <= nexthandle;
        nodemem_datain <= bdd_t_zero;
        currentstate <= getnextfree;
       end if;

      when initnodes =>
       if (nodemem_ack = '1') then
        if (handle = maxhandle) then
          nodemem_request <= '1';
          nodemem_rw <= '0';
          nodemem_handle <= handle;
          nodemem_datain <= bdd_t_init;
          currentstate <= lastinit;
        else
          -- initialize all of the node memory
          nodemem_request <= '1';
          nodemem_rw <= '0';
          nodemem_handle <= handle;
          if(handle = bddhandle_zero) then
          -- handle 0 is the constant 0
          nodemem_datain <= (level => bddvar_max,
          lo => bddhandle_zero, hi => bddhandle_zero,
          nextbdd => bddhandle_zero, gc=> gc_zero);

          elsif(handle = bddhandle_one) then
          -- handle 1 is the constant 1
          nodemem_datain <= (level => bddvar_max,
          lo => bddhandle_one, hi => bddhandle_one,
          nextbdd => bddhandle_one, gc=> gc_zero);
          else
          -- write the node to node memory @ handle
          nodemem_datain <= (level => bddvar_max,
          lo => bddhandle_zero, hi => bddhandle_zero,
          nextbdd => nexthandle, gc=> gc_zero);
          end if;
          -- set the handle = nexthandle
          handle <= nexthandle;
          -- increment the next handle
          nexthandle <= nexthandle + 1;
          currentstate <= initnodes;
        end if;
      else
        currentstate <= initnodes;
      end if;

    when lastinit =>
      freehandle <= minhandle;
      freehandle_valid <= '1';
      nexthandle <= minhandle + 1;
      -- turn of memory interface
      nodemem_request <= '0';
      nodemem_rw <= '1';
      nodemem_handle <= bddhandle_zero;
      nodemem_datain <= bdd_t_zero;
      currentstate <= IDLE;

    when waitformem =>
      if (nodemem_busy = '0') then
        if (handle = maxhandle) then
          nodemem_request <= '1';
          nodemem_rw <= '0';
          nodemem_handle <= handle;
          nodemem_datain <= bdd_t_init;
          currentstate <= lastinit;
        else
          -- initialize all of the node memory
          nodemem_request <= '1';
          nodemem_rw <= '0';
          nodemem_handle <= handle;
          if(handle = bddhandle_zero) then
          -- handle 0 is the constant 0
          nodemem_datain <= (level => bddvar_max,
          lo => bddhandle_zero, hi => bddhandle_zero,
          nextbdd => bddhandle_zero, gc=> gc_zero);

          elsif(handle = bddhandle_one) then
          -- handle 1 is the constant 1
          nodemem_datain <= (level => bddvar_max,
          lo => bddhandle_one, hi => bddhandle_one,
          nextbdd => bddhandle_one, gc=> gc_zero);
          else
          -- write the node to node memory @ handle
          nodemem_datain <= (level => bddvar_max,
          lo => bddhandle_zero, hi => bddhandle_zero,
          nextbdd => nexthandle, gc=> gc_zero);
          end if;
          -- set the handle = nexthandle
          handle <= nexthandle;
          -- increment the next handle
          nexthandle <= nexthandle + 1;
          currentstate <= initnodes;
        end if;
      else
        currentstate <= waitformem;
      end if;

    when others =>

      -- turn of memory interface
      nodemem_request <= '0';
      nodemem_rw <= '1';
      nodemem_handle <= bddhandle_zero;
      nodemem_datain <= bdd_t_zero;
      currentstate <= IDLE;
    end case;
  end if;
 end if;
end process;
```

```
 outofnodes <= '1' when nexthandle = bddhandle_zero
 else '0';
end freenodecntl;

library ieee;
use ieee.STD_LOGIC_1164.all;
library bddlib;
use bddlib.kernel.all;
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.ALL;


entity apply is
 port (
    clk              : in std_logic;
    rst              : in std_logic;
    lobddin          : in bddhandle;
    hibddin          : in bddhandle;
    resulthandle     : out bddhandle;
    resultvalid      : out std_logic;
    nodemem_dataout  : in bdd_t;
    nodemem_busy     : in std_logic;
    nodemem_datavalid : in std_logic;
    nodemem_ack      : in std_logic;
    nodemem_handle   : out bddhandle;
    nodemem_datain   : out bdd_t;
    nodemem_rw       : out std_logic;
    nodemem_request  : out std_logic;
    cam_ack          : in std_logic;
    cam_busy         : in std_logic;
    cam_result       : in camfield;
    cam_resultvalid  : in std_logic;
    cam_found        : in std_logic;
    cam_request      : out std_logic;
    cam_rw           : out std_logic;
    cam_field1       : out camfield;
    cam_field2       : out camfield;
    cam_field3       : out camfield;
    cam_resultin     : out camfield;
    call_datain0     : out bdd_t;
    call_datain1     : out bdd_t;
    call_dataout0    : in bdd_t;
    call_dataout1    : in bdd_t;
    call_full        : in std_logic;
    call_empty       : in std_logic;
    call_cmd         : out stackcmd;
    result_datain    : out bddhandle;
    result_dataout   : in bddhandle;
    result_full      : in std_logic;
    result_empty     : in std_logic;
    result_cmd       : out stackcmd;
    start            : in std_logic;
    mknode_start     : out std_logic;
    mknode_result    : in bddhandle;
    mknode_resultvalid : in std_logic;
    mknode_level     : out bddvar;
    mknode_lo        : out bddhandle;
    mknode_hi        : out bddhandle;
    operator         : in booleanop;
    operror          : out std_logic
    );

end apply;


architecture apply of apply is

 signal lohandle     : bddhandle;
 signal hihandle     : bddhandle;
 signal camdone      : std_logic;
 signal terminalcase : std_logic;
 signal returncode   : gc_t;
 signal lonode       : bdd_t;
 signal hinode       : bdd_t;
 signal localresult  : bddhandle;
 signal tmpresult    : bddhandle;

 type visual_current_top_states is (TOP);
 constant current_top : visual_current_top_states := TOP;

 type visual_current_main_states is (IDLE, RETURNCTL, WARMUP,
mknode, updatehi,
                         updatelo, CAMFIND);

 signal current_main : visual_current_main_states;

 type visual_TOP_TOP_CAMFIND_CAMFIND1_states is
(CAMFIND1,
                              TOP_TOP_CAMFIND_camdone);

 signal visual_TOP_TOP_CAMFIND_CAMFIND1_current :
   visual_TOP_TOP_CAMFIND_CAMFIND1_states;

 type visual_TOP_TOP_CAMFIND_findnode_states is (findnode,
findnode2,
                              waitcamresult);

 signal visual_TOP_TOP_CAMFIND_findnode_current :
   visual_TOP_TOP_CAMFIND_findnode_states;

 type visual_current_writecam_states is (writecamidle, writecam);

 signal current_writecam : visual_current_writecam_states;


-- since the memory interfaces are async with
-- each other, there is no telling which will
-- get done first.
-- must not exit until cam has gotten a result.
-- so the node lookup must wait until the cam is done
-- if there was a cam miss, then the exit must be from the
-- findnode machine
--
-- There is a potential problem if the cam
-- gets done, but misses and some other transaction
-- takes place on the cam. This might cause
-- an erroneous exit?
-- No, because if the cam misses it will set camdone
-- and the findnode machine must wait for that to
-- exit.
-- cam_found is async and is set as soon as cam_resultvalid
-- is recieved. The default is '0' so it will hold that
-- value only when resultvalid='1' or in state camdone
begin



 -- Synchronous process
 apply_TOP:
 process (clk)
   variable appcamhit   : NATURAL          := 0;
   variable appcamwrite : NATURAL          := 0;
   variable appcammiss  : NATURAL          := 0;
 begin

   if (clk'event and clk = '1') then
     resultvalid <= '0';
     call_cmd <= stacknop;
     result_cmd <= stacknop;
     mknode_start <= '0';
     case current_main is
       when IDLE =>
         if (rst = '0') then
           current_main <= IDLE;
         elsif (start = '1') then
           lohandle <= lobddin;
           hihandle <= hibddin;
           current_main <= WARMUP;
         else
           current_main <= IDLE;
         end if;

       when RETURNCTL =>
        if (rst = '0') then
          current_main <= IDLE;
        elsif (returncode = returndone) then
          -- teh cam is accessed and we know it
          -- will not be busy at this point, then
          -- cam_ack will come back right away,
          -- so there is no reason to wait.
          --for cam_ack = '1'
          --also since we know single cycle access
          --is in place we can just turn off the request.
          --This will have to change if it is not single
          --cycle access.
          cam_request <= '0';
          resultvalid <= '1';
          current_main <= IDLE;
        elsif (returncode = returnhigh) then
          -- teh cam is accessed and we know it
          -- will not be busy at this point, then
          -- cam_ack will come back right away,
          -- so there is no reason to wait.
          --for cam_ack = '1'
          --also since we know single cycle access
          --is in place we can just turn off the request.
          --This will have to change if it is not single
          --cycle access.
          cam_request <= '0';
          -- pop call stack into local regs
          call_cmd <= stackpop2;
          lonode <= call_dataout0;
          lohandle <= call_dataout0.lo;
          hinode <= call_dataout1;
          hihandle <= call_dataout0.hi;
          -- push localresult onto result stack
          result_cmd <= stackpush;
          result_datain <= localresult;
          current_main <= updatehi;
        elsif (returncode = returnmknode) then
          -- teh cam is accessed and we know it
          -- will not be busy at this point, then
          -- cam_ack will come back right away,
          -- so there is no reason to wait.
          --for cam_ack = '1'
          --also since we know single cycle access
          --is in place we can just turn off the request.
          --This will have to change if it is not single
          --cycle access.
          cam_request <= '0';
          lohandle <= call_dataout0.lo;
          hihandle <= call_dataout0.hi;
          -- entry <= call_dataout0.nextbdd;
          -- pop call stack into local regs
          call_cmd <= stackpop;
          -- start mknode
          mknode_start <= '1';
          mknode_level <= call_dataout0.level;
          mknode_lo <= result_dataout;
          -- the current result (from second recursive call)
          -- can be applied to the mknode hi branch.
          mknode_hi <= localresult;
          -- pop the result stack
          result_cmd <= stackpop;
          current_main <= mknode;
        else
          current_main <= RETURNCTL;
        end if;

       when WARMUP =>
        if (rst = '0') then
          current_main <= IDLE;
        elsif (terminalcase = '1') then
          localresult <= tmpresult;
          current_main <= RETURNCTL;
        else
          -- setup cam find
          cam_request <= '1';
          cam_rw <= '1'; -- read (find);
          cam_field1 <= lohandle;
          cam_field2 <= hihandle;
          cam_field3 <= natural(booleanop'pos(operator));
          visual_TOP_TOP_CAMFIND_CAMFIND1_current <=
CAMFIND1;
          -- set up a read request to node memory
          nodemem_handle <= lohandle;
          nodemem_request <= '1';
          nodemem_rw <= '1'; -- read
          nodemem_datain <= bdd_t_zero;
          visual_TOP_TOP_CAMFIND_findnode_current <= findnode;
          current_main <= CAMFIND;
        end if;

      when mknode =>
        if (rst = '0') then
          current_main <= IDLE;
        elsif (mknode_resultvalid = '1') then
          localresult <= mknode_result;
          current_main <= RETURNCTL;
        else
          current_main <= mknode;
        end if;

      when updatehi =>
        if (rst = '0') then
          current_main <= IDLE;
        else
          -- set the local handles for the high edge recursion
          lohandle <= hinode.lo;
          hihandle <= hinode.hi;

          -- push local regs onto the call stack
          call_datain0.level <= hinode.level;
          call_datain0.lo <= lohandle;
          call_datain0.hi <= hihandle;
          --call_datain0.nextbdd <= entry;
          -- store the return code in the gc bits
          call_datain0.gc <= returnmknode;
          call_cmd <= stackpush;
          current_main <= WARMUP;
        end if;

      when updatelo =>
        if (rst = '0') then
          current_main <= IDLE;
        -- recure on low(l),low(r)
        --    set the lo and hi handles
        -- also push args for high edge recursion
        --       high(l),high(r)

        elsif (lonode.level = hinode.level) then
          -- recure on low(l),low(r)
          lohandle <= lonode.lo;
          hihandle <= hinode.lo;
          -- then     high(l),high(r)
          -- for the high edge recursion put the
          -- arguments in the second stack entry
          call_datain1.lo <= lonode.hi;
          call_datain1.hi <= hinode.hi;
          -- store the var level for the mknode call
          call_datain1.level <= lonode.level;
          -- so must prep for s stack push
          call_datain0.lo <= lohandle;
          call_datain0.hi <= hihandle;
          --call_datain0.nextbdd <= entry;
          call_datain0.gc <= returnhigh;
          call_cmd <= stackpush2;
          current_main <= WARMUP;
        -- -- recur on low(l),r
        -- -- then    high(l),r
        -- -- so must prep for s stack push
        -- -- for the high edge recursion put the
        -- -- arguments high(l),r in the second stack entry

        elsif (lonode.level < hinode.level) then
          -- recur on low(l),r
          lohandle <= lonode.lo;
          hihandle <= hihandle;
          -- for the high edge recursion put the
          -- arguments high(l),r in the second stack entry
          call_datain1.lo <= lonode.hi;
          call_datain1.hi <= hihandle;
          call_datain1.nextbdd <= hihandle;
          -- store the var level for the mknode call
          call_datain1.level <= lonode.level;
          -- so must prep for s stack push
          call_datain0.lo <= lohandle;
          call_datain0.hi <= hihandle;
          --call_datain0.nextbdd <= entry;
          call_datain0.gc <= returnhigh;
          call_cmd <= stackpush2;
          current_main <= WARMUP;
        -- recur on l,low(r)
        --         l,high(r)
        -- for the high edge recursion put the
        -- arguments high(l),r in the second stack entry

        else
          -- recur on l,low(r)
          --         l,high(r)
          lohandle <= lohandle;
          hihandle <= hinode.lo;
          -- for the high edge recursion put the
          -- arguments high(l),r in the second stack entry
          call_datain1.lo <= lohandle;
          call_datain1.hi <= hinode.hi;
          --call_datain1.nextbdd <= hihandle;
          -- store the var level for the mknode call
          call_datain1.level <= hinode.level;
          -- so must prep for s stack push
          call_datain0.lo <= lohandle;
          call_datain0.hi <= hihandle;
          --call_datain0.nextbdd <= entry;
          call_datain0.gc <= returnhigh;
          call_cmd <= stackpush2;
          current_main <= WARMUP;
        end if;

      when CAMFIND =>
        case visual_TOP_TOP_CAMFIND_CAMFIND1_current is
          when CAMFIND1 =>
            if (cam_ack = '1' and cam_resultvalid = '1' and cam_found = '1')
              then
              -- set result handle
              localresult <= cam_result;
              cam_request <= '0';
              appcamhit := appcamhit + 1;
              current_main <= RETURNCTL;
            elsif ((cam_ack = '1') and (cam_resultvalid = '1' and cam_found
=
                '0')) then
              if (rst = '0') then
                current_main <= IDLE;
              else
                cam_request <= '0';
                appcammiss := appcammiss +1;
                visual_TOP_TOP_CAMFIND_CAMFIND1_current <=
                  TOP_TOP_CAMFIND_camdone;
              end if;
            elsif (rst = '0') then
              current_main <= IDLE;
            else
              visual_TOP_TOP_CAMFIND_CAMFIND1_current <=
CAMFIND1;
            end if;
        when TOP_TOP_CAMFIND_camdone =>
         if (rst = '0') then
          current_main <= IDLE;
         else
          visual_TOP_TOP_CAMFIND_CAMFIND1_current <=
            TOP_TOP_CAMFIND_camdone;
         end if;

        when others =>

         current_main <= IDLE;
       end case;
       case visual_TOP_TOP_CAMFIND_findnode_current is
        -- this state will lookup the nodehandle (found in the
        -- unique table) in node memory

        when findnode =>
         if (rst = '0') then
          current_main <= IDLE;
         elsif ((nodemem_ack = '1') and (nodemem_datavalid = '1')) then
          nodemem_request <= '0';
          lonode <= nodemem_dataout;
          -- set up a read request to node memory
          nodemem_handle <= hihandle;
          nodemem_request <= '1';
          nodemem_rw <= '1'; -- read
          nodemem_datain <= bdd_t_zero;
          visual_TOP_TOP_CAMFIND_findnode_current <=
findnode2;
         else
          visual_TOP_TOP_CAMFIND_findnode_current <= findnode;
         end if;

        when findnode2 =>
         if (nodemem_ack = '1' and nodemem_datavalid = '1' and cam-
done =
            '1') then
          nodemem_request <= '0';
          hinode <= nodemem_dataout;
          current_main <= updatelo;
         elsif ((nodemem_ack = '1') and (nodemem_datavalid = '1')) then
          if (rst = '0') then
           current_main <= IDLE;
          else
           nodemem_request <= '0';
           hinode <= nodemem_dataout;
           visual_TOP_TOP_CAMFIND_findnode_current <= waitcam-
result;
          end if;
         elsif (rst = '0') then
          current_main <= IDLE;
         else
          visual_TOP_TOP_CAMFIND_findnode_current <=
findnode2;
         end if;

        when waitcamresult =>
         if (camdone = '1') then
          current_main <= updatelo;
         else
          if (rst = '0') then
           current_main <= IDLE;
          else
           visual_TOP_TOP_CAMFIND_findnode_current <= waitcam-
result;
          end if;
         end if;

        when others =>

         current_main <= IDLE;
       end case;
      when others =>
         current_main <= IDLE;
       end case;
       case current_writecam is
        when writecamidle =>
         if (mknode_resultvalid = '1' and current_main = mknode) then
          -- setup a write/insert to cam
          cam_request <= '1';
          cam_rw <= '0'; -- write(insert);
          cam_field1 <= lohandle;
          cam_field2 <= hihandle;
          cam_field3 <= natural(booleanop'pos(operator));
          cam_resultin <= mknode_result;
          current_writecam <= writecam;
         else
          current_writecam <= writecamidle;
         end if;

        when writecam =>
         if (cam_ack = '1') then
          -- turn off write request
          cam_request <= '0';
          cam_rw <= '1';
          appcamwrite := appcamwrite + 1;
          current_writecam <= writecamidle;
         else
          current_writecam <= writecam;
         end if;

        when others =>

         current_writecam <= writecamidle;
       end case;
      end if;
     end process;

     -- Combinational process
     apply_TOP_comb:
     process (current_main,
    visual_TOP_TOP_CAMFIND_CAMFIND1_current,
      visual_TOP_TOP_CAMFIND_findnode_current, current_writecam)
     begin  -- Combinational process
      camdone <= '0';

      case current_main is
       when CAMFIND =>
        case visual_TOP_TOP_CAMFIND_CAMFIND1_current is
         when TOP_TOP_CAMFIND_camdone =>
          camdone <= '1';

         when others =>
          null;
        end case;
       when others =>
        null;
      end case;
     end process;


     resulthandle <= localresult;
     returncode <= call_dataout0.gc;


     process(lohandle,hihandle)
     variable terminaltest : std_logic_vector(3 downto 0);
     begin
     -- do some defaults so we dont get latches
     operror <= '0';
     terminalcase <= '0';
     terminaltest := "0000";

     -- set up the terminal test values so that the terminal test
     -- case statement can work effectively
     -- terminal value 00 means handle = 0
     -- termanl value 11 means handle = 1
     -- terminal value 01 means handle is neither of the two constants
```

```
if(lohandle = bddhandle_one) then
terminaltest(1 downto 0) := "11";
elsif(lohandle = bddhandle_zero) then
terminaltest(1 downto 0) := "00";
else
terminaltest(1 downto 0) := "01";
end if;

if(hihandle = bddhandle_one) then
terminaltest(3 downto 2) := "11";
elsif(hihandle = bddhandle_zero) then
terminaltest(3 downto 2) := "00";
else
terminaltest(3 downto 2) := "01";
end if;


case operator is
when booleanop_and =>
-- tests for the and operator
if(lohandle = hihandle) then
tmpresult <= lohandle;
terminalcase <= '1';
else
case terminaltest is
when "0000" | "0001" | "0011" | "1100" | "0100"=>
tmpresult <= bddhandle_zero;
terminalcase <= '1';
when "1111" =>
tmpresult <= bddhandle_one;
terminalcase <= '1';
when "1101" =>
tmpresult <= lohandle;
terminalcase <= '1';
when "0111" =>
tmpresult <= hihandle;
terminalcase <= '1';
when others =>
-- do not assign a true value to terminalcase because
-- this is the case where you are not in a teminal case and muse
-- recursivly evaluate the bdd
-- neither edge is constant, not a terminal case
tmpresult <= bddhandle_zero;
terminalcase <= '0';
end case;
end if;
-- end tests for the and operator

-- or operator
when booleanop_or =>

if(lohandle = hihandle) then
tmpresult <= lohandle;
terminalcase <= '1';
else
case terminaltest is
when "1111" | "1100" | "0011" | "1101" | "0111"=>
tmpresult <= bddhandle_one;
terminalcase <= '1';
when "0100" =>
tmpresult <= hihandle;
terminalcase <= '1';
when "0001" =>
tmpresult <= lohandle;
terminalcase <= '1';
when "0000" =>
tmpresult <= bddhandle_zero;
terminalcase <= '1';
when others =>
-- neither edge is constant, not a terminal case
tmpresult <= bddhandle_zero;
terminalcase <= '0';
end case;
end if;
-- end test for the or operator


when booleanop_xor =>

if(lohandle = hihandle) then
tmpresult <= bddhandle_zero;
terminalcase <= '1';
else
case terminaltest is
when "0100" | "1100" =>
tmpresult <= hihandle;
terminalcase <= '1';
when "0011" | "0001" =>
tmpresult <= lohandle;
terminalcase <= '1';
when "0000" | "1111"  =>
tmpresult <= bddhandle_zero;
terminalcase <= '1';
when others =>
-- neither edge is constant, not a terminal case
tmpresult <= bddhandle_zero;
terminalcase <= '0';
end case;
end if;
-- end test for the xor operator


-- nand operator

when booleanop_nand =>

case terminaltest is

when "0000" | "1100" | "0011" | "0100" | "0001" =>
-- either input is 0
tmpresult <= bddhandle_one;
terminalcase <= '1';
when "1111" =>
-- both inputs are one
tmpresult <= bddhandle_zero;
terminalcase <= '1';
when others =>
-- neither edge is constant, not a terminal case
tmpresult <= bddhandle_zero;
terminalcase <= '0';
end case;

-- end test for the nand operator

when booleanop_nor =>

case terminaltest is
when "1111" | "1100" | "0011" | "1101" | "0111"=>
-- either input is 1
tmpresult <= bddhandle_zero;
terminalcase <= '1';
when "0000" =>
tmpresult <= bddhandle_one;
terminalcase <= '1';
when others =>
-- neither edge is constant, not a terminal case
tmpresult <= bddhandle_zero;
terminalcase <= '0';
end case;
-- end test for the or operator

when booleanop_imp =>
case terminaltest is
when "1100" | "0100" | "0000" |  "1101" | "1111" =>
-- low edge is zero or  high edge is one
tmpresult <= bddhandle_one;
terminalcase <= '1';
when  "0011" | "0111" =>
-- lo edge is one
```

```
 tmpresult <= hihandle;
 terminalcase <= '1';
 when others =>
 -- neither edge is constant, not a terminal case
 tmpresult <= bddhandle_zero;
 terminalcase <= '0';
 end case;
 if(lohandle = bddhandle_one) then
 tmpresult <= hihandle;
 terminalcase <= '1';
 end if;

 when booleanop_biimp =>
 case terminaltest is
 when "0000" | "1111" =>
 -- low edge is zero or  high edge is one
 tmpresult <= bddhandle_one;
 terminalcase <= '1';
 when "0011" | "1100" =>
 -- lo edge is one high edge is zero
 tmpresult <= bddhandle_zero;
 terminalcase <= '1';
 when others =>
 -- neither edge is constant , not a terminal case
 tmpresult <= bddhandle_zero;
 terminalcase <= '0';
 end case;
 -- end test for the implication operator

 -- difference (greater than)
 when booleanop_greater =>
 case terminaltest is
 when "0000" | "1111" | "1100" =>
 tmpresult <= bddhandle_zero;
 terminalcase <= '1';
 when "0011" =>
 -- lo edge is one high edge is zero
 tmpresult <= bddhandle_one;
 terminalcase <= '1';
 when others =>
 -- neither edge is constant , not a terminal case
 tmpresult <= bddhandle_zero;
 terminalcase <= '0';
 end case;
 -- end test for the difference operator

 -- less
 when booleanop_less =>
 case terminaltest is
 when "0000" | "1111" | "0011" =>
 tmpresult <= bddhandle_zero;
 terminalcase <= '1';
 when "1100" =>
 -- hi edge is one low edge is zero
 tmpresult <= bddhandle_one;
 terminalcase <= '1';
 when others =>
 -- neither edge is constant , not a terminal case
 tmpresult <= bddhandle_zero;
 terminalcase <= '0';
 end case;
 -- end terminal test for less operator

 -- others
 when others =>
 -- some kind of operator error, make it terminal and return zero
 operror <= '1';
 terminalcase <= '1';
 tmpresult <= bddhandle_zero;
 end case;
 end process;
end apply;

library ieee;
use ieee.STD_LOGIC_1164.all;
library bddlib;
use bddlib.kernel.all;
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.ALL;


entity APPLY_NOT is
 port (
     clk              : in std_logic;
     rst              : in std_logic;
     bddin            : in bddhandle;
     resulthandle     : out bddhandle;
     resultvalid      : out std_logic;
     nodemem_dataout  : in bdd_t;
     nodemem_busy     : in std_logic;
     nodemem_datavalid : in std_logic;
     nodemem_ack      : in std_logic;
     nodemem_handle   : out bddhandle;
     nodemem_datain   : out bdd_t;
     nodemem_rw       : out std_logic;
     nodemem_request  : out std_logic;
     cam_ack          : in std_logic;
     cam_busy         : in std_logic;
     cam_result       : in camfield;
     cam_resultvalid  : in std_logic;
     cam_found        : in std_logic;
     cam_request      : out std_logic;
     cam_rw           : out std_logic;
     cam_field1       : out camfield;
     cam_field2       : out camfield;
     cam_field3       : out camfield;
     cam_resultin     : out camfield;
     call_datain      : out bdd_t;
     call_dataout     : in bdd_t;
     call_full        : in std_logic;
     call_empty       : in std_logic;
     call_cmd         : out stackcmd;
     result_datain    : out bddhandle;
     result_dataout   : in bddhandle;
     result_full      : in std_logic;
     result_empty     : in std_logic;
     result_cmd       : out stackcmd;
     start            : in std_logic;
     mknode_start     : out std_logic;
     mknode_result    : in bddhandle;
     mknode_resultvalid : in std_logic;
     mknode_level     : out bddvar;
     mknode_lo        : out bddhandle;
     mknode_hi        : out bddhandle
     );

end APPLY_NOT;


architecture APPLY_NOT of APPLY_NOT is

 constant operator  : booleanop          := booleanop_not;
 signal  localhandle : bddhandle;
 signal  camdone   : std_logic;
 signal  returncode : gc_t;
 signal  localnode  : bdd_t;
 signal  localresult : bddhandle;

 type visual_current_top_states is (TOP);
 constant current_top : visual_current_top_states := TOP;

 type visual_current_main_states is (IDLE, RETURNCTL, WARMUP,
mknode, updatehi,
                    updatelo, CAMFIND);

 signal current_main : visual_current_main_states;

 type visual_TOP_TOP_CAMFIND_CAMFIND1_states is
(CAMFIND1,
                        TOP_TOP_CAMFIND_camdone);
 signal visual_TOP_TOP_CAMFIND_CAMFIND1_current :
  visual_TOP_TOP_CAMFIND_CAMFIND1_states;

 type visual_TOP_TOP_CAMFIND_findnode_states is (findnode, wait-
camresult);

 signal visual_TOP_TOP_CAMFIND_findnode_current :
  visual_TOP_TOP_CAMFIND_findnode_states;

 type visual_current_writecam_states is (writeuniqueidle, writecam);

 signal current_writecam : visual_current_writecam_states;


begin



 -- Synchronous process
 APPLY_NOT_TOP:
 process (clk)
  variable appnotcamhit   : NATURAL          := 0;
  variable appnotcamwrite : NATURAL          := 0;
  variable appnotcammiss  : NATURAL          := 0;
 begin

  if (clk'event and clk = '1') then
   resultvalid <= '0';
   call_cmd <= stacknop;
   result_cmd <= stacknop;
   mknode_start <= '0';
   case current_main is
    when IDLE =>
     if (rst = '0') then
      current_main <= IDLE;
     elsif (start = '1') then
      localhandle <= bddin;
      current_main <= WARMUP;
     else
      current_main <= IDLE;
     end if;

    when RETURNCTL =>
     if (rst = '0') then
      current_main <= IDLE;
     elsif (returncode = returndone) then
      -- since this is the only place
      -- teh cam is accessed and we know it
      -- will not be busy at this point, then
      -- cam_ack will come back right away,
      -- so there is no reason to wait.
      --for cam_ack = '1'
      --also since we know single cycle access
      --is in place we can just turn off the request.
      --This will have to change if it is not single
      --cycle access.
      cam_request <= '0';
      resultvalid <= '1';
      current_main <= IDLE;
     elsif (returncode = returnmknode) then
      -- since this is the only place
      -- teh cam is accessed and we know it
      -- will not be busy at this point, then
      -- cam_ack will come back right away,
      -- so there is no reason to wait.
      --for cam_ack = '1'
      --also since we know single cycle access
      --is in place we can just turn off the request.
      --This will have to change if it is not single
      --cycle access.
      cam_request <= '0';
      -- pop call stack into local regs
      call_cmd <= stackpop;
      localnode <= call_dataout;
      localhandle <= call_dataout.nextbdd;
      -- start mknode
      mknode_start <= '1';
      mknode_level <= call_dataout.level;
      mknode_lo <= result_dataout;
      -- the current result (from second recursive call)
      -- can be applied to the mknode hi branch.
      mknode_hi <= localresult;
      -- pop the result stack
      result_cmd <= stackpop;
      current_main <= mknode;
     elsif (returncode = returnhigh) then
      -- since this is the only place
      -- teh cam is accessed and we know it
      -- will not be busy at this point, then
      -- cam_ack will come back right away,
      -- so there is no reason to wait.
      --for cam_ack = '1'
      --also since we know single cycle access
      --is in place we can just turn off the request.
      --This will have to change if it is not single
      --cycle access.
      cam_request <= '0';
      -- pop call stack into local regs
      call_cmd <= stackpop;
      localnode <= call_dataout;
      localhandle <= call_dataout.nextbdd;
      -- push localresult onto result stack
      result_cmd <= stackpush;
      result_datain <= localresult;
      current_main <= updatehi;
     else
      current_main <= RETURNCTL;
     end if;

    when WARMUP =>
     if (rst = '0') then
      current_main <= IDLE;
     elsif (localhandle = bddhandle_zero) then
      localresult <= bddhandle_one;
      current_main <= RETURNCTL;
     elsif (localhandle = bddhandle_one) then
      localresult <= bddhandle_zero;
      current_main <= RETURNCTL;
     else
      -- setup cam find
      cam_request <= '1';
      cam_rw <= '1'; -- read (find);
      cam_field1 <= localhandle;
      cam_field2 <= camfield_zero;
      cam_field3 <= natural(booleanop'pos(operator));
      visual_TOP_TOP_CAMFIND_CAMFIND1_current <=
CAMFIND1;
      -- set up a read request to node memory
      nodemem_handle <= localhandle;
      nodemem_request <= '1';
      nodemem_rw <= '1'; -- read
      nodemem_datain <= bdd_t_zero;
      visual_TOP_TOP_CAMFIND_findnode_current <= findnode;
      current_main <= CAMFIND;
     end if;

    when mknode =>
     if (rst = '0') then
      current_main <= IDLE;
     elsif (mknode_resultvalid = '1') then
      localresult <= mknode_result;
      current_main <= RETURNCTL;
     else
      current_main <= mknode;
     end if;

    when updatehi =>
     if (rst = '0') then
      current_main <= IDLE;
```

```
        else
          -- push local regs onto the call stack
          call_datain.level <= localnode.level;
          call_datain.lo <= localnode.lo;
          call_datain.hi <= localnode.hi;
          call_datain.nextbdd <= localhandle;
          -- store the return code in the gc bits
          call_datain.gc <= returnmknode;
          call_cmd <= stackpush;
          -- set the arg for the hi-edge recursion.
          localhandle <= localnode.hi;
          current_main <= WARMUP;
        end if;

      when updatelo =>
        if (rst = '0') then
          current_main <= IDLE;
        else
          -- push local regs onto the call stack
          call_datain <= localnode;
          call_datain.nextbdd <= localhandle;
          -- store the return code in the gc bits
          call_datain.gc <= returnhigh;
          call_cmd <= stackpush;
          -- set the are for the lo-edge recursion
          localhandle <= localnode.lo;
          current_main <= WARMUP;
        end if;

      when CAMFIND =>
        case visual_TOP_TOP_CAMFIND_CAMFIND1_current is
          when CAMFIND1 =>
            if (cam_ack = '1' and cam_resultvalid = '1' and cam_found = '1')
              then
              -- set result handle
              localresult <= cam_result;
              cam_request <= '0';
              -- cam hit
              appnotcamhit := appnotcamhit + 1;
              current_main <= RETURNCTL;
            elsif ((cam_ack = '1' and cam_resultvalid = '1') and (cam_found
=
                '0')) then
              if (rst = '0') then
                current_main <= IDLE;
              else
                cam_request <= '0';
                appnotcammiss := appnotcammiss + 1;
                visual_TOP_TOP_CAMFIND_CAMFIND1_current <=
                  TOP_TOP_CAMFIND_camdone;
              end if;
            elsif (rst = '0') then
              current_main <= IDLE;
            else
              visual_TOP_TOP_CAMFIND_CAMFIND1_current <=
CAMFIND1;
            end if;

          when TOP_TOP_CAMFIND_camdone =>
            if (rst = '0') then
              current_main <= IDLE;
            else
              visual_TOP_TOP_CAMFIND_CAMFIND1_current <=
                TOP_TOP_CAMFIND_camdone;
            end if;

          when others =>

            current_main <= IDLE;
        end case;
        case visual_TOP_TOP_CAMFIND_findnode_current is
          -- this state will lookup the nodehandle (found in the
          -- unique table) in node memory

          when findnode =>
            if (nodemem_ack = '1' and nodemem_datavalid = '1' and cam-
done =
              '1') then
              nodemem_request <= '0';
              localnode <= nodemem_dataout;
              current_main <= updatelo;
            elsif (nodemem_ack = '1' and nodemem_datavalid = '1') then
              if (rst = '0') then
                current_main <= IDLE;
              else
                nodemem_request <= '0';
                localnode <= nodemem_dataout;
                visual_TOP_TOP_CAMFIND_findnode_current <= waitcam-
result;
              end if;
            elsif (rst = '0') then
              current_main <= IDLE;
            else
              visual_TOP_TOP_CAMFIND_findnode_current <= findnode;
            end if;

          when waitcamresult =>
            if (camdone = '1') then
              current_main <= updatelo;
            else
              if (rst = '0') then
                current_main <= IDLE;
              else
                visual_TOP_TOP_CAMFIND_findnode_current <= waitcam-
result;
              end if;
            end if;

          when others =>

            current_main <= IDLE;
        end case;
      when others =>

        current_main <= IDLE;
    end case;
    case current_writecam is
      when writeuniqueidle =>
        if (mknode_resultvalid = '1' and current_main = mknode) then
          -- setup a write/insert to cam
          -- setup cam find
          cam_request <= '1';
          cam_rw <= '0'; -- write(insert);
          cam_field1 <= localhandle;
          cam_field2 <= camfield_zero;
          cam_field3 <= natural(booleanop'pos(operator));
          cam_resultin <= mknode_result;
          current_writecam <= writecam;
        else
          current_writecam <= writeuniqueidle;
        end if;

      when writecam =>
        if (cam_ack = '1') then
          -- turn off write request
          cam_request <= '0';
          cam_rw <= '1';
          appnotcamwrite := appnotcamwrite + 1;
          current_writecam <= writeuniqueidle;
        else
          current_writecam <= writecam;
        end if;

      when others =>

        current_writecam <= writeuniqueidle;
    end case;
  end if;
end process;
```

```
 -- Combinational process
 APPLY_NOT_TOP_comb:
 process (current_main,
visual_TOP_TOP_CAMFIND_CAMFIND1_current,
  visual_TOP_TOP_CAMFIND_findnode_current, current_writecam)
 begin  -- Combinational process
  camdone <= '0';

  case current_main is
   when CAMFIND =>
    case visual_TOP_TOP_CAMFIND_CAMFIND1_current is
     when TOP_TOP_CAMFIND_camdone =>
      camdone <= '1';

     when others =>
      null;
    end case;
   when others =>
    null;
  end case;
 end process;


 resulthandle <= localresult;
 returncode <= call_dataout.gc;
end APPLY_NOT;

-- Mux the two apply routines together.
-- these two apply functionscould be
-- combined, but would add complexity
-- to the FSMs so they are muxed at this
-- level instead.

library ieee;
use ieee.STD_LOGIC_1164.all;
library work;
use work.kernel.all;
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.ALL;


entity APPLYBLK is
 port (
    applyerror      : out std_logic;
    start0          : out std_logic;
    level0          : out bddvar;
    low0            : out bddhandle;
    high0           : out bddhandle;
    applyop         : in booleanop;
    bddin1          : in bddhandle;
    bddin2          : in bddhandle;
    call_dataout0   : in bdd_t;
    call_dataout1   : in bdd_t;
    call_dataout2   : in bdd_t;
    call_full       : in std_logic;
    call_empty      : in std_logic;
    call_datain0    : out bdd_t;
    call_datain1    : out bdd_t;
    call_datain2    : out bdd_t;
    call_cmd        : out stackcmd;
    cam_request     : out std_logic;
    cam_rw          : out std_logic;
    cam_field1      : out camfield;
    cam_field2      : out camfield;
    cam_field3      : out camfield;
    cam_resultin    : out camfield;
    cam_ack         : in std_logic;
    cam_busy        : in std_logic;
    cam_result      : in camfield;
    cam_resultvalid : in std_logic;
    cam_found       : in std_logic;
    clk             : in std_logic;
    enableand       : in std_logic;
    enablenot       : in std_logic;
    mknode_result   : in bddhandle;
    mknode_resultvalid : in std_logic;
    node_port3_ack  : in std_logic;
    node_port2_ack  : in std_logic;
    node_port1_ack  : in std_logic;
    node_datavalid  : in std_logic;
    node_dataout    : in bdd_t;
    node_busy       : in std_logic;
    port3_datain    : out bdd_t;
    port3_handle    : out bddhandle;
    port3_request   : out std_logic;
    port3_rw        : out std_logic;
    resulthandle    : out bddhandle;
    result_datain0  : out bddhandle;
    result_datain1  : out bddhandle;
    result_datain2  : out bddhandle;
    result_cmd      : out stackcmd;
    result_dataout0 : in bddhandle;
    result_dataout1 : in bddhandle;
    result_dataout2 : in bddhandle;
    result_full     : in std_logic;
    result_empty    : in std_logic;
    resultvalid     : out std_logic;
    rst             : in std_logic;
    startapply      : in std_logic;
    startnot        : in std_logic
    );


end APPLYBLK;


library bddlib;
architecture APPLYBLK of APPLYBLK is

 signal appand_call_cmd         : stackcmd;
 signal appand_call_datain0     : bdd_t;
 signal appand_call_datain1     : bdd_t;
 signal appand_call_datain2     : bdd_t;
 signal appand_cam_field1       : camfield;
 signal appand_cam_field2       : camfield;
 signal appand_cam_field3       : camfield;
 signal appand_cam_request      : std_logic;
 signal appand_cam_resultin     : camfield;
 signal appand_cam_rw           : std_logic;
 signal appand_high0            : bddhandle;
 signal appand_level0           : bddvar;
 signal appand_low0             : bddhandle;
 signal appand_port3_datain     : bdd_t;
 signal appand_port3_handle     : bddhandle;
 signal appand_port3_request    : std_logic;
 signal appand_port3_rw         : std_logic;
 signal appand_result_cmd       : stackcmd;
 signal appand_result_datain0   : bddhandle;
 signal appand_result_datain1   : bddhandle;
 signal appand_result_datain2   : bddhandle;
 signal appand_resulthandle     : bddhandle;
 signal appand_resultvalid      : std_logic;
 signal appand_start0           : std_logic;
 signal appnot_call_cmd         : stackcmd;
 signal appnot_call_datain0     : bdd_t;
 signal appnot_call_datain1     : bdd_t;
 signal appnot_call_datain2     : bdd_t;
 signal appnot_cam_field1       : camfield;
 signal appnot_cam_field2       : camfield;
 signal appnot_cam_field3       : camfield;
 signal appnot_cam_request      : std_logic;
 signal appnot_cam_resultin     : camfield;
 signal appnot_cam_rw           : std_logic;
 signal appnot_high0            : bddhandle;
 signal appnot_level0           : bddvar;
 signal appnot_low0             : bddhandle;
 signal appnot_port3_datain     : bdd_t;
 signal appnot_port3_handle     : bddhandle;
 signal appnot_port3_request    : std_logic;
 signal appnot_port3_rw         : std_logic;
signal appnot_result_cmd      : stackcmd;
signal appnot_result_datain0  : bddhandle;
signal appnot_result_datain1  : bddhandle;
signal appnot_result_datain2  : bddhandle;
signal appnot_resulthandle    : bddhandle;
signal appnot_resultvalid     : std_logic;
signal appnot_start0          : std_logic;
signal enableand_d            : std_logic;
signal enablenot_d            : std_logic;
signal visual_C1_Q            : std_logic;
signal visual_C2_Q            : std_logic;
component APPLY_NOT
  port (
      clk              : in std_logic;
      rst              : in std_logic;
      bddin            : in bddhandle;
      resulthandle     : out bddhandle;
      resultvalid      : out std_logic;
      nodemem_dataout  : in bdd_t;
      nodemem_busy     : in std_logic;
      nodemem_datavalid : in std_logic;
      nodemem_ack      : in std_logic;
      nodemem_handle   : out bddhandle;
      nodemem_datain   : out bdd_t;
      nodemem_rw       : out std_logic;
      nodemem_request  : out std_logic;
      cam_ack          : in std_logic;
      cam_busy         : in std_logic;
      cam_result       : in camfield;
      cam_resultvalid  : in std_logic;
      cam_found        : in std_logic;
      cam_request      : out std_logic;
      cam_rw           : out std_logic;
      cam_field1       : out camfield;
      cam_field2       : out camfield;
      cam_field3       : out camfield;
      cam_resultin     : out camfield;
      call_datain      : out bdd_t;
      call_dataout     : in bdd_t;
      call_full        : in std_logic;
      call_empty       : in std_logic;
      call_cmd         : out stackcmd;
      result_datain    : out bddhandle;
      result_dataout   : in bddhandle;
      result_full      : in std_logic;
      result_empty     : in std_logic;
      result_cmd       : out stackcmd;
      start            : in std_logic;
      mknode_start     : out std_logic;
      mknode_result    : in bddhandle;
      mknode_resultvalid : in std_logic;
      mknode_level     : out bddvar;
      mknode_lo        : out bddhandle;
      mknode_hi        : out bddhandle
      );
end component;
component apply
  port (
      clk              : in std_logic;
      rst              : in std_logic;
      lobddin          : in bddhandle;
      hibddin          : in bddhandle;
      resulthandle     : out bddhandle;
      resultvalid      : out std_logic;
      nodemem_dataout  : in bdd_t;
      nodemem_busy     : in std_logic;
      nodemem_datavalid : in std_logic;
      nodemem_ack      : in std_logic;
      nodemem_handle   : out bddhandle;
      nodemem_datain   : out bdd_t;
      nodemem_rw       : out std_logic;
      nodemem_request  : out std_logic;
      cam_ack          : in std_logic;
      cam_busy         : in std_logic;
      cam_result       : in camfield;
      cam_resultvalid  : in std_logic;
      cam_found        : in std_logic;
      cam_request      : out std_logic;
      cam_rw           : out std_logic;
      cam_field1       : out camfield;
      cam_field2       : out camfield;
      cam_field3       : out camfield;
      cam_resultin     : out camfield;
      call_datain0     : out bdd_t;
      call_datain1     : out bdd_t;
      call_dataout0    : in bdd_t;
      call_dataout1    : in bdd_t;
      call_full        : in std_logic;
      call_empty       : in std_logic;
      call_cmd         : out stackcmd;
      result_datain    : out bddhandle;
      result_dataout   : in bddhandle;
      result_full      : in std_logic;
      result_empty     : in std_logic;
      result_cmd       : out stackcmd;
      start            : in std_logic;
      mknode_start     : out std_logic;
      mknode_result    : in bddhandle;
      mknode_resultvalid : in std_logic;
      mknode_level     : out bddvar;
      mknode_lo        : out bddhandle;
      mknode_hi        : out bddhandle;
      operator         : in booleanop;
      operror          : out std_logic
      );
 end component;

 -- Start Configuration Specification
 for all : APPLY_NOT use entity bddlib.APPLY_NOT(APPLY_NOT);
 for all : apply use entity bddlib.apply(apply);
 -- End Configuration Specification

begin

 inst_APPLY_NOT: APPLY_NOT
  port map (
       clk => clk,
       rst => rst,
       bddin => bddin1,
       resulthandle => appnot_resulthandle,
       resultvalid => appnot_resultvalid,
       nodemem_dataout => node_dataout,
       nodemem_busy => node_busy,
       nodemem_datavalid => node_datavalid,
       nodemem_ack => node_port3_ack,
       nodemem_handle => appnot_port3_handle,
       nodemem_datain => appnot_port3_datain,
       nodemem_rw => appnot_port3_rw,
       nodemem_request => appnot_port3_request,
       cam_ack => cam_ack,
       cam_busy => cam_busy,
       cam_result => cam_result,
       cam_resultvalid => cam_resultvalid,
       cam_found => cam_found,
       cam_request => appnot_cam_request,
       cam_rw => appnot_cam_rw,
       cam_field1 => appnot_cam_field1,
       cam_field2 => appnot_cam_field2,
       cam_field3 => appnot_cam_field3,
       cam_resultin => appnot_cam_resultin,
       call_datain => appnot_call_datain0,
       call_dataout => call_dataout0,
       call_full => call_full,
       call_empty => call_empty,
       call_cmd => appnot_call_cmd,
       result_datain => appnot_result_datain0,
       result_dataout => result_dataout0,
       result_full => result_full,
       result_empty => result_empty,
       result_cmd => appnot_result_cmd,
        start => startnot,
        mknode_start => appnot_start0,
        mknode_result => mknode_result,
        mknode_resultvalid => mknode_resultvalid,
        mknode_level => appnot_level0,
        mknode_lo => appnot_low0,
        mknode_hi => appnot_high0);

inst_apply: apply
  port map (
        clk => clk,
        rst => rst,
        lobddin => bddin1,
        hibddin => bddin2,
        resulthandle => appand_resulthandle,
        resultvalid => appand_resultvalid,
        nodemem_dataout => node_dataout,
        nodemem_busy => node_busy,
        nodemem_datavalid => node_datavalid,
        nodemem_ack => node_port3_ack,
        nodemem_handle => appand_port3_handle,
        nodemem_datain => appand_port3_datain,
        nodemem_rw => appand_port3_rw,
        nodemem_request => appand_port3_request,
        cam_ack => cam_ack,
        cam_busy => cam_busy,
        cam_result => cam_result,
        cam_resultvalid => cam_resultvalid,
        cam_found => cam_found,
        cam_request => appand_cam_request,
        cam_rw => appand_cam_rw,
        cam_field1 => appand_cam_field1,
        cam_field2 => appand_cam_field2,
        cam_field3 => appand_cam_field3,
        cam_resultin => appand_cam_resultin,
        call_datain0 => appand_call_datain0,
        call_datain1 => appand_call_datain1,
        call_dataout0 => call_dataout0,
        call_dataout1 => call_dataout1,
        call_full => call_full,
        call_empty => call_empty,
        call_cmd => appand_call_cmd,
        result_datain => appand_result_datain0,
        result_dataout => result_dataout0,
        result_full => result_full,
        result_empty => result_empty,
        result_cmd => appand_result_cmd,
        start => startapply,
        mknode_start => appand_start0,
        mknode_result => mknode_result,
        mknode_resultvalid => mknode_resultvalid,
        mknode_level => appand_level0,
        mknode_lo => appand_low0,
        mknode_hi => appand_high0,
        operator => applyop,
        operror => applyerror);

process(enablenot_d,enableand_d,
appnot_resulthandle,
appnot_resultvalid,
  -- result signals
appand_resulthandle,
appand_resultvalid)
begin
if(enablenot_d = '1') then
resulthandle <= appnot_resulthandle;
resultvalid <= appand_resultvalid OR appnot_resultvalid;
else
resulthandle <= appand_resulthandle;
resultvalid <= appand_resultvalid OR appnot_resultvalid;
end if;

end process;

process(enablenot,enableand,
appnot_resulthandle,
appnot_resultvalid,
-- cam signals
appnot_cam_field1,
appnot_cam_field2,
appnot_cam_field3,
appnot_cam_resultin,
appnot_cam_request,
appnot_cam_rw,
-- node memory signals
appnot_port3_datain,
appnot_port3_handle,
appnot_port3_request,
appnot_port3_rw,
-- call register/stack frame
appnot_call_datain0,
appnot_call_datain1,
appnot_call_datain2,
appnot_call_cmd,
-- result stack frame
appnot_result_datain0,
appnot_result_datain1,
appnot_result_datain2,
appnot_result_cmd,
--mknode signals
appnot_start0,
appnot_level0,
appnot_low0,
appnot_high0,
-- apply and signals
  -- result signals
appand_resulthandle,
appand_resultvalid,
-- cam signals
appand_cam_field1,
appand_cam_field2,
appand_cam_field3,
appand_cam_resultin,
appand_cam_request,
appand_cam_rw,
-- node memory signals
appand_port3_datain,
appand_port3_handle,
appand_port3_request,
appand_port3_rw,
-- call register/stack frame
appand_call_datain0,
appand_call_datain1,
appand_call_datain2,
appand_call_cmd,
-- result stack frame
appand_result_datain0,
appand_result_datain1,
appand_result_datain2,
appand_result_cmd,
--mknode signals
appand_start0,
appand_level0,
appand_low0,
appand_high0)



 subtype visual_BIT_VECTOR_0_1_0 is BIT_VECTOR ( 0 to 1 );
begin

--case visual_BIT_VECTOR_0_1_0'(To_bitvector
(std_ulogic_vector'(startnot & startand),'0')) is
-- APPLY_NOT
--when "01" =>
if(enablenot = '1') then

-- result signals are handles in different process
-- cam signals
```

```
--cam_datain <= appnot_cam_datain;
cam_field1 <= appnot_cam_field1;
cam_field2 <= appnot_cam_field2;
cam_field3 <= appnot_cam_field3;
cam_resultin <= appnot_cam_resultin;
cam_request <= appnot_cam_request;
cam_rw <= appnot_cam_rw;
-- node memory signals
port3_datain <= appnot_port3_datain;
port3_handle <= appnot_port3_handle;
port3_request <= appnot_port3_request;
port3_rw <= appnot_port3_rw;
-- call register/stack frame
call_datain0 <= appnot_call_datain0;
call_datain1 <= appnot_call_datain1;
call_datain2 <= appnot_call_datain2;
call_cmd <= appnot_call_cmd;
-- result stack frame
result_datain0 <= appnot_result_datain0;
result_datain1 <= appnot_result_datain1;
result_datain2 <= appnot_result_datain2;
result_cmd <= appnot_result_cmd;
--mknode signals
start0 <= appnot_start0;
level0 <= appnot_level0;
low0 <= appnot_low0;
high0 <= appnot_high0;

elsif(enableand = '1') then

-- APPLY_AND
--when "10" |
--when others =>
  -- result signals are handled in different processs
-- cam signals
--cam_datain <= appand_cam_datain;
cam_field1 <= appand_cam_field1;
cam_field2 <= appand_cam_field2;
cam_field3 <= appand_cam_field3;
cam_resultin <= appand_cam_resultin;
cam_request <= appand_cam_request;
cam_rw <= appand_cam_rw;
-- node memory signals
port3_datain <= appand_port3_datain;
port3_handle <= appand_port3_handle;
port3_request <= appand_port3_request;
port3_rw <= appand_port3_rw;
-- call register/stack frame
call_datain0 <= appand_call_datain0;
call_datain1 <= appand_call_datain1;
call_datain2 <= appand_call_datain2;
call_cmd <= appand_call_cmd;
-- result stack frame
result_datain0 <= appand_result_datain0;
result_datain1 <= appand_result_datain1;
result_datain2 <= appand_result_datain2;
result_cmd <= appand_result_cmd;
--mknode signals
start0 <= appand_start0;
level0 <= appand_level0;
low0 <= appand_low0;
high0 <= appand_high0;

end if;
--end case;


end process;

enablenot_d <= (visual_C1_Q);


process (clk , rst)
begin
 if (rst = '0') then
   visual_C1_Q <= '0';
 elsif (clk'event and clk = '1') then
    visual_C1_Q <= (enablenot);


end if;
end process;


enableand_d <= (visual_C2_Q);


process (clk , rst)
begin
 if (rst = '0') then
   visual_C2_Q <= '0';
 elsif (clk'event and clk = '1') then
    visual_C2_Q <= (enableand);


end if;
end process;

end APPLYBLK;

library ieee;
use ieee.STD_LOGIC_1164.all;
library bddlib;
use bddlib.kernel.all;
library work;
use work.bdddebug.all;
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.ALL;


entity memctrl is
 generic (
      readdelay  : NATURAL          := 0;
      writedelay : NATURAL          := 0;
      memsize    : NATURAL          := bddmemsize
      );
 port (
     port3_request : in std_logic;
     port3_ack     : out std_logic;
     port3_handle  : in bddhandle;
     port3_datain  : in bdd_t;
     port3_rw      : in std_logic;
     port2_request : in std_logic;
     port2_ack     : out std_logic;
     port2_handle  : in bddhandle;
     port2_datain  : in bdd_t;
     port2_rw      : in std_logic;
     port1_request : in std_logic;
     port1_ack     : out std_logic;
     port1_handle  : in bddhandle;
     port1_datain  : in bdd_t;
     port1_rw      : in std_logic;
     datavalid     : out std_logic;
     dataout       : out bdd_t;
     busy          : out std_logic;
     clk           : in std_logic;
     rst           : in std_logic
     );

end memctrl;


architecture memctrl of memctrl is

 constant maxdelay    : NATURAL                 := readdelay; -- this
should be maxdelay which is max(readdelay,writedelay)
 signal  delaycnt    : NATURAL range 0 to maxdelay;
 signal  delaytmp    : NATURAL range 0 to maxdelay;
 signal  address     : bddhandle;
 signal  datain      : bdd_t;
 signal  read_write  : std_logic;
 signal  enable      : std_logic;
 signal  tp_writetable : BOOLEAN            := false;
 signal  port_ack    : std_logic_vector(1 to 3 );
 signal  ibusy       : std_logic;
 signal  startaccess  : std_logic;

 type visual_IDLE_states is (IDLE, delay);

 signal visual_IDLE_current, visual_IDLE_next : visual_IDLE_states;
 attribute STATE_VECTOR of memctrl :
        architecture is "visual_IDLE_current";


 type visual_GOT3_states is (GOT3, GOT1, GOT2);

 signal visual_GOT3_current, visual_GOT3_next :
visual_GOT3_states;

 signal  visual_delaycnt_next : NATURAL range 0 to maxdelay;
 signal  visual_ibusy_next : std_logic;

begin



 -- Combinational process
 memctrl_IDLE_comb:
 process (delaycnt, startaccess, read_write, port_ack,
visual_IDLE_current)
 begin
   port3_ack <= '0';
   port2_ack <= '0';
   port1_ack <= '0';
   enable <= '0';
   visual_ibusy_next <= '0';
   visual_delaycnt_next <= delaycnt;



   case visual_IDLE_current is
     when IDLE =>
      if ((startaccess = '1') and ((read_write = '1' and readdelay = 0) or (
         read_write = '0' and writedelay = 0))) then
        -- set the controls to memory
        enable <= '1' after 5 ns, '0' after 9 ns;
        datavalid <= read_write;
        port1_ack <= port_ack(1);
        port2_ack <= port_ack(2);
        port3_ack <= port_ack(3);
        visual_IDLE_next <= IDLE;
      elsif ((startaccess = '1') and (read_write = '1')) then
        visual_delaycnt_next <= readdelay - 1;
        visual_ibusy_next <= '1';
        visual_IDLE_next <= delay;
      elsif ((startaccess = '1') and (read_write = '0')) then
        visual_delaycnt_next <= writedelay - 1;
        visual_ibusy_next <= '1';
        visual_IDLE_next <= delay;
      else
        busy <= '0';
        datavalid <= '0' after 1 ns;
        port1_ack <= '0';
        port2_ack <= '0';
        port3_ack <= '0';
        visual_delaycnt_next <= 0;
        visual_IDLE_next <= IDLE;
      end if;

     when delay =>
      busy <= '1';
      -- ibusy <= '1';
      -- port1_ack <= '0';
      -- port2_ack <= '0';
      -- port3_ack <= '0';

      if (delaycnt = 0) then
        -- set the controls to memory
        enable <= '1' after 5 ns, '0' after 9 ns;
        datavalid <= read_write;
        port1_ack <= port_ack(1);
        port2_ack <= port_ack(2);
        port3_ack <= port_ack(3);
        visual_IDLE_next <= IDLE;
      else
        visual_delaycnt_next <= delaycnt - 1;
        visual_ibusy_next <= '1';
        visual_IDLE_next <= delay;
      end if;

     when others =>

      visual_IDLE_next <= IDLE;
   end case;
 end process;

 memctrl_IDLE:
 process (clk)
 begin

   if (clk'event and clk = '1') then
    if (rst = '0') then
      ibusy <= '0';
      visual_IDLE_current <= IDLE;
    else
      delaycnt <= visual_delaycnt_next;
      ibusy <= visual_ibusy_next;
      visual_IDLE_current <= visual_IDLE_next;
    end if;
   end if;
 end process;



 -- Combinational process
 memctrl_GOT3_comb:
 process (ibusy, port1_request, port1_handle, port1_datain, port1_rw,
       port2_request, port2_handle, port2_datain, port2_rw,
port3_request,
       port3_handle, port3_datain, port3_rw, visual_GOT3_current)
 begin



   case visual_GOT3_current is
     when GOT3 =>
      if ((ibusy = '0') and (port1_request = '1')) then
        startaccess <= '1';
        -- set values for the external signals
        port_ack <= "100";
        -- set the controls to memory
        address <= port1_handle;
        datain <= port1_datain;
        read_write <= port1_rw;
        visual_GOT3_next <= GOT1;
      elsif ((ibusy = '0') and (port2_request = '1')) then
        startaccess <= '1';
        -- set values for the external signals
        port_ack <= "010";
        -- set the controls to memory
        address <= port2_handle;
        datain <= port2_datain;
        read_write <= port2_rw;
        visual_GOT3_next <= GOT2;
      elsif ((ibusy = '0') and (port3_request = '1')) then
        startaccess <= '1';
        -- set values for the external signals
        port_ack <= "001";
        -- set the controls to memory
        address <= port3_handle;
        datain <= port3_datain;
   read_write <= port3_rw;
   visual_GOT3_next <= GOT3;
  else
   startaccess <= '0';
   visual_GOT3_next <= GOT3;
  end if;

 when GOT1 =>
  if ((ibusy = '0') and (port2_request = '1')) then
   startaccess <= '1';
   -- set values for the external signals
   port_ack <= "010";
   -- set the controls to memory
   address <= port2_handle;
   datain <= port2_datain;
   read_write <= port2_rw;
   visual_GOT3_next <= GOT2;
  elsif ((ibusy = '0') and (port3_request = '1')) then
   startaccess <= '1';
   -- set values for the external signals
   port_ack <= "001";
   -- set the controls to memory
   address <= port3_handle;
   datain <= port3_datain;
   read_write <= port3_rw;
   visual_GOT3_next <= GOT3;
  elsif ((ibusy = '0') and (port1_request = '1')) then
   startaccess <= '1';
   -- set values for the external signals
   port_ack <= "100";
   -- set the controls to memory
   address <= port1_handle;
   datain <= port1_datain;
   read_write <= port1_rw;
   visual_GOT3_next <= GOT1;
  else
   startaccess <= '0';
   visual_GOT3_next <= GOT1;
  end if;

 when GOT2 =>
  if ((ibusy = '0') and (port3_request = '1')) then
   startaccess <= '0', '1' after 1 ns;
   -- set values for the external signals
   port_ack <= "001";
   -- set the controls to memory
   address <= port3_handle;
   datain <= port3_datain;
   read_write <= port3_rw;
   visual_GOT3_next <= GOT3;
  elsif ((ibusy = '0') and (port1_request = '1')) then
   startaccess <= '0', '1' after 1 ns;
   -- set values for the external signals
   port_ack <= "100";
   -- set the controls to memory
   address <= port1_handle;
   datain <= port1_datain;
   read_write <= port1_rw;
   visual_GOT3_next <= GOT1;
  elsif ((ibusy = '0') and (port2_request = '1')) then
   startaccess <= '0', '1' after 1 ns;
   -- set values for the external signals
   port_ack <= "010";
   -- set the controls to memory
   address <= port2_handle;
   datain <= port2_datain;
   read_write <= port2_rw;
   visual_GOT3_next <= GOT2;
  else
   startaccess <= '0';
   visual_GOT3_next <= GOT2;
  end if;

 when others =>
   visual_GOT3_next <= GOT3;
 end case;
end process;

memctrl_GOT3:
process (clk)
begin

 if (clk'event and clk = '1') then
  if (rst = '0') then
   visual_GOT3_current <= GOT3;
  else
   visual_GOT3_current <= visual_GOT3_next;
  end if;
 end if;
end process;


process (enable, read_write, address, datain, tp_writetable)
  variable mem : bdd_vec_t(0 to memsize - 1 );
begin  -- process
  if enable = '1' then
   if read_write = '0' then
    --write
    mem(address) := datain;
   else
    -- read
    dataout <= mem(address);
   end if;
  end if;
-- this is for test purposes only
-- set the value of tp_writetable during
-- debug to dump the table to a file
if(tp_writetable) then
writenodetable(mem);
end if;

end process;
end memctrl;

--
-- When a request is recieved, the ack signal will go
-- high indicating that the data is ready.
-- This is a single cycle memory with no delay.
-- on a read, Ack indicates the data is valid at the output,
-- or the write has been completed.
-- All inputs must be held for a complete clock cycle.

library ieee;
use ieee.STD_LOGIC_1164.all;
library bddlib;
use bddlib.kernel.all;
use ieee.NUMERIC_STD.all;
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.ALL;


entity uniquemem is
 generic (
     readdelay  : NATURAL          := 0;
     writedelay : NATURAL          := 0;
     uniquesize : NATURAL          := bdduniquetablesize
     );
 port (
    port1_ack      : out std_logic;
    port1_busy     : out std_logic;
    port1_dataready : out std_logic;
    port1_dataout  : out bddhandle;
    port1_handle   : in bddhandle;
    port1_datain   : in bddhandle;
    port1_rw       : in std_logic;
    port1_request  : in std_logic;
    clk            : in std_logic;
    rst            : in std_logic
    );
```

```
end uniquemem;


architecture uniquemem of uniquemem is

 constant maxdelay  : NATURAL            := readdelay; -- this should
be maxdelay which is max(readdelay,writedelay)
 signal  delaycnt  : NATURAL range 0 to maxdelay;
 signal  address   : hashkey;
 signal  datain    : bddhandle;
 signal  dataout   : bddhandle;
 signal  read_write : std_logic;
 signal  enable    : std_logic;

 type visual_IDLE_states is (IDLE);

 signal visual_IDLE_current, visual_IDLE_next : visual_IDLE_states;
 attribute STATE_VECTOR of uniquemem :
        architecture is "visual_IDLE_current";


begin



 -- Combinational process
 uniquemem_IDLE_comb:
 process (rst, port1_request, port1_handle, port1_datain, port1_rw,
read_write,
        visual_IDLE_current)
 begin
  port1_ack <= '0';
  port1_busy <= '0';
  port1_dataready <= '0';
  address <= hashkey_zero;
  datain <= bddhandle_zero;
  read_write <= '1';
  enable <= '0';


  if (rst = '0') then
   -- reset all drivers to memory
   read_write <= '1';
   enable <= '0';
   address <= 0;
   datain <= 0;
   visual_IDLE_next <= IDLE;
  else

   case visual_IDLE_current is
    when IDLE =>
     if ((port1_request = '1') and (port1_rw = '1')) then
      port1_busy <= '1';
      port1_ack <= '1';
      -- set up the memory inputs
      enable <= '1' after 1ns;
      address <= port1_handle;
      datain <= port1_datain;
      read_write <= '1';
      port1_dataready <= read_write;
      visual_IDLE_next <= IDLE;
     elsif ((port1_request = '1') and (port1_rw = '0')) then
      port1_busy <= '1';
      port1_ack <= '1';
      -- set up the memory inputs
      enable <= '1' after 1ns;
      address <= port1_handle;
      datain <= port1_datain;
      read_write <= '0';
      port1_dataready <= read_write;
      visual_IDLE_next <= IDLE;
     else
      -- reset all drivers to memory
      read_write <= '1';
      enable <= '0';
      address <= 0;
      datain <= 0;
      visual_IDLE_next <= IDLE;
     end if;

    when others =>

     visual_IDLE_next <= IDLE;
   end case;
  end if;
 end process;

 uniquemem_IDLE:
 process (clk)
 begin

  if (clk'event and clk = '1') then
   if (rst = '0') then
    visual_IDLE_current <= IDLE;
   else
    visual_IDLE_current <= visual_IDLE_next;
   end if;
  end if;
 end process;


 -- this will eventually (before synthesis
 -- have to be moved outside of this unit
 -- so that it can represent an external memory
 uniquemem:
 process (enable, read_write, address, datain)
   variable mem : bddhandle_vec_t(0 to uniquesize - 1 );
  begin
 --dataout <= bddhandle_zero;
   if enable = '1' then
    if read_write = '0' then
     --write
     mem(address) := datain;
    else
      --read
     dataout <= mem(address);
    end if;
   end if;
 end process;

 port1_dataout <= dataout;
end uniquemem;

-- this is the make_node function.
-- it controls all creation and access
-- to thenode table.
-- garbage collection will also need
-- to manipulate the node table so
-- addtional controls will need to be
-- added later.
-- because VIsual on Linux is crashing when trying to
-- have concurrent machines in a sub level and
-- controls set to async outputs, this machine has some work arounds
-- using additional conditions in instate assignments

library ieee;
use ieee.STD_LOGIC_1164.all;
library work;
use work.kernel.all;
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.ALL;


entity mknodefsm is
 generic (
       uniquesize : NATURAL            := bdduniquetablesize
       );
 port (
     clk          : in std_logic;
    rst            : in std_logic;
    start          : in std_logic;
    level          : in bddvar;
    low            : in bddhandle;
    freehandle     : in bddhandle;
    freehandle_valid : in std_logic;
    high           : in bddhandle;
    resulthandle   : out bddhandle;
    resultnode     : out bdd_t;
    tookfreehandle : out std_logic;
    result_valid   : out std_logic;
    unique_handle  : out bddhandle;
    unique_datain  : out bddhandle;
    unique_rw      : out std_logic;
    unique_request : out std_logic;
    unique_dataready : in std_logic;
    unique_ack     : in std_logic;
    unique_busy    : in std_logic;
    unique_dataout : in bddhandle;
    nodemem_dataout : in bdd_t;
    nodemem_busy   : in std_logic;
    nodemem_dataready : in std_logic;
    nodemem_ack    : in std_logic;
    nodemem_handle : out bddhandle;
    nodemem_datain : out bdd_t;
    nodemem_rw     : out std_logic;
    nodemem_request : out std_logic
    );

end mknodefsm;


architecture mknodefsm of mknodefsm is

  signal hashval          : hashkey;
  signal firsthandle      : bddhandle;
  signal nodemem_handle_i : bddhandle;
  signal build_busy       : std_logic;
  signal start_build      : std_logic;

  type visual_wka_states is (wka);
  constant visual_wka_current : visual_wka_states := wka;

  type visual_wka_wka_IDLE_states is (IDLE, FINDNODE, FINDUN-
IQUE, WAITFORBUILD,
                         hash);

  signal visual_wka_wka_IDLE_current :
visual_wka_wka_IDLE_states;

  type visual_wka_wka_writenodeidle_states is (writenodeidle, writeno-
demem,
                            writenodewait);

  signal visual_wka_wka_writenodeidle_current :
    visual_wka_wka_writenodeidle_states;

  type visual_wka_wka_writeuniqueidle_states is (writeuniqueidle,
writeuniquemem
                            , writeuniquewait);

  signal visual_wka_wka_writeuniqueidle_current :
    visual_wka_wka_writeuniqueidle_states;


-- The three machines on this page are concurrent.
-- The writenodedle and writeuniqueidle machines
-- will be triggered when start_build is set to a 1.
-- The default value should be 0.
-- They will run concurrently until completion.
-- when both are complete build_busy should be 0;
-- when either is active build_busy should be active 1
begin
```

```
-- Synchronous process
mknodefsm_wka:
process (clk)
 variable uniqueaccess : NATURAL          := 0;
 variable uniquehit    : NATURAL          := 0;
 variable uniquemiss   : NATURAL          := 0;
 variable uniquechain  : NATURAL          := 0;
begin

 if (clk'event and clk = '1') then
  tookfreehandle <= '0';
  result_valid <= '0';
  case visual_wka_wka_IDLE_current is
   when IDLE =>
    -- reset all of the outputs

    if (rst = '0') then
     resulthandle <= bddhandle_zero;
     result_valid <= '0';
     tookfreehandle <= '0';
     -- -- turn off the memory interfaces

     -- unique mem outputs
     unique_handle <= bddhandle_zero;
     unique_datain <= bddhandle_zero;
     unique_rw <= '1'; -- read
     unique_request <= '0';
     -- nodemem outputs
     nodemem_handle_i <= bddhandle_zero;
     nodemem_datain <= bdd_t_zero;
     nodemem_rw <= '1'; -- read
     nodemem_request <= '0';
     visual_wka_wka_IDLE_current <= IDLE;
    elsif (start = '1') then
     result_valid <= '0';
     if (low = high) then
      resulthandle <= low;
      result_valid <= '1';
      -- -- turn off the memory interfaces

      -- unique mem outputs
      unique_handle <= bddhandle_zero;
      unique_datain <= bddhandle_zero;
      unique_rw <= '1'; -- read
      unique_request <= '0';
      -- nodemem outputs
      nodemem_handle_i <= bddhandle_zero;
      nodemem_datain <= bdd_t_zero;
      nodemem_rw <= '1'; -- read
      nodemem_request <= '0';
      visual_wka_wka_IDLE_current <= IDLE;
     else
      hashval <= bdd_hash((level,low,high,0,0),uniquesize);
      visual_wka_wka_IDLE_current <= hash;
     end if;
    else
     visual_wka_wka_IDLE_current <= IDLE;
    end if;

   when FINDNODE =>
    -- reset all of the outputs

    if (rst = '0') then
     resulthandle <= bddhandle_zero;
     result_valid <= '0';
     tookfreehandle <= '0';
     -- -- turn off the memory interfaces

     -- unique mem outputs
     unique_handle <= bddhandle_zero;
     unique_datain <= bddhandle_zero;
     unique_rw <= '1'; -- read
     unique_request <= '0';
     -- nodemem outputs
```

```
      nodemem_handle_i <= bddhandle_zero;
      nodemem_datain <= bdd_t_zero;
      nodemem_rw <= '1'; -- read
      nodemem_request <= '0';
      visual_wka_wka_IDLE_current <= IDLE;
    elsif (nodemem_ack = '1' and nodemem_dataready = '1') then
      nodemem_request <= '0';
      if ((nodemem_dataout.lo = low) and (nodemem_dataout.hi =
high) and (
          nodemem_dataout.level = level)) then
        resulthandle <= nodemem_handle_i;
        result_valid <= '1';
        uniquehit := uniquehit + 1;
        -- -- turn off the memory interfaces

        -- unique mem outputs
        unique_handle <= bddhandle_zero;
        unique_datain <= bddhandle_zero;
        unique_rw <= '1'; -- read
        unique_request <= '0';
        -- nodemem outputs
        nodemem_handle_i <= bddhandle_zero;
        nodemem_datain <= bdd_t_zero;
        nodemem_rw <= '1'; -- read
        nodemem_request <= '0';
        visual_wka_wka_IDLE_current <= IDLE;
      elsif (nodemem_dataout.nextbdd = bddhandle_zero) then
        if (build_busy = '1') then
          visual_wka_wka_IDLE_current <= WAITFORBUILD;
        else
          if (freehandle_valid = '1') then
            --start_build <= '1';
            resulthandle <= freehandle;
            result_valid <= '1';
            -- -- turn off the memory interfaces

            -- unique mem outputs
            unique_handle <= bddhandle_zero;
            unique_datain <= bddhandle_zero;
            unique_rw <= '1'; -- read
            unique_request <= '0';
            -- nodemem outputs
            nodemem_handle_i <= bddhandle_zero;
            nodemem_datain <= bdd_t_zero;
            nodemem_rw <= '1'; -- read
            nodemem_request <= '0';
            visual_wka_wka_IDLE_current <= IDLE;
          else
            visual_wka_wka_IDLE_current <= WAITFORBUILD;
          end if;
        end if;
      else
        nodemem_handle_i <= nodemem_dataout.nextbdd;
        uniquechain := uniquechain + 1;
        -- set up a read request to node memory
        nodemem_request <= '1';
        nodemem_rw <= '1'; -- read
        nodemem_datain <= bdd_t_zero;
        visual_wka_wka_IDLE_current <= FINDNODE;
      end if;
    else
      visual_wka_wka_IDLE_current <= FINDNODE;
    end if;

  -- wait until the unique memory access is complete

  when FINDUNIQUE =>
   -- reset all of the outputs

   if (rst = '0') then
     resulthandle <= bddhandle_zero;
     result_valid <= '0';
     tookfreehandle <= '0';
     -- -- turn off the memory interfaces

     -- unique mem outputs
     unique_handle <= bddhandle_zero;
     unique_datain <= bddhandle_zero;
     unique_rw <= '1'; -- read
     unique_request <= '0';
     -- nodemem outputs
     nodemem_handle_i <= bddhandle_zero;
     nodemem_datain <= bdd_t_zero;
     nodemem_rw <= '1'; -- read
     nodemem_request <= '0';
     visual_wka_wka_IDLE_current <= IDLE;
   elsif (unique_ack = '1' and unique_dataready = '1') then
     unique_request <= '0';
     uniqueaccess := uniqueaccess + 1;
     if (unique_dataout = bddhandle_zero) then
       if (build_busy = '1') then
         visual_wka_wka_IDLE_current <= WAITFORBUILD;
       else
         if (freehandle_valid = '1') then
           --start_build <= '1';
           resulthandle <= freehandle;
           result_valid <= '1';
           -- -- turn off the memory interfaces

           -- unique mem outputs
           unique_handle <= bddhandle_zero;
           unique_datain <= bddhandle_zero;
           unique_rw <= '1'; -- read
           unique_request <= '0';
           -- nodemem outputs
           nodemem_handle_i <= bddhandle_zero;
           nodemem_datain <= bdd_t_zero;
           nodemem_rw <= '1'; -- read
           nodemem_request <= '0';
           visual_wka_wka_IDLE_current <= IDLE;
         else
           visual_wka_wka_IDLE_current <= WAITFORBUILD;
         end if;
       end if;
     -- -- start setting up a node memory read
     -- -- must read from the handle (address) just
     -- -- found from the unique table
     -- -- first handle is needed when building a new node to
     -- -- put at beginning of chain

     else
       nodemem_handle_i <= unique_dataout;
       firsthandle <= unique_dataout;
       -- set up a read request to node memory
       nodemem_request <= '1';
       nodemem_rw <= '1'; -- read
       nodemem_datain <= bdd_t_zero;
       visual_wka_wka_IDLE_current <= FINDNODE;
     end if;
   else
     visual_wka_wka_IDLE_current <= FINDUNIQUE;
   end if;

  when WAITFORBUILD =>
   -- reset all of the outputs

   if (rst = '0') then
     resulthandle <= bddhandle_zero;
     result_valid <= '0';
     tookfreehandle <= '0';
     -- -- turn off the memory interfaces

     -- unique mem outputs
     unique_handle <= bddhandle_zero;
     unique_datain <= bddhandle_zero;
     unique_rw <= '1'; -- read
     unique_request <= '0';
     -- nodemem outputs
     nodemem_handle_i <= bddhandle_zero;
     nodemem_datain <= bdd_t_zero;
   nodemem_rw <= '1'; -- read
   nodemem_request <= '0';
   visual_wka_wka_IDLE_current <= IDLE;
  elsif (build_busy = '0') then
   if (freehandle_valid = '1') then
    --start_build <= '1';
    resulthandle <= freehandle;
    result_valid <= '1';
    -- -- turn off the memory interfaces

    -- unique mem outputs
    unique_handle <= bddhandle_zero;
    unique_datain <= bddhandle_zero;
    unique_rw <= '1'; -- read
    unique_request <= '0';
    -- nodemem outputs
    nodemem_handle_i <= bddhandle_zero;
    nodemem_datain <= bdd_t_zero;
    nodemem_rw <= '1'; -- read
    nodemem_request <= '0';
    visual_wka_wka_IDLE_current <= IDLE;
   else
    visual_wka_wka_IDLE_current <= WAITFORBUILD;
   end if;
  else
   visual_wka_wka_IDLE_current <= WAITFORBUILD;
  end if;

 when hash =>
  -- reset all of the outputs

  if (rst = '0') then
   resulthandle <= bddhandle_zero;
   result_valid <= '0';
   tookfreehandle <= '0';
   -- -- turn off the memory interfaces

   -- unique mem outputs
   unique_handle <= bddhandle_zero;
   unique_datain <= bddhandle_zero;
   unique_rw <= '1'; -- read
   unique_request <= '0';
   -- nodemem outputs
   nodemem_handle_i <= bddhandle_zero;
   nodemem_datain <= bdd_t_zero;
   nodemem_rw <= '1'; -- read
   nodemem_request <= '0';
   visual_wka_wka_IDLE_current <= IDLE;
  else
   -- start a read from the unique table memory
   unique_request <= '1';
   unique_rw <= '1'; -- read
   unique_datain <= bddhandle_zero;
   unique_handle <= hashval;
   firsthandle <= bddhandle_zero;
   visual_wka_wka_IDLE_current <= FINDUNIQUE;
  end if;

 when others =>

  visual_wka_wka_IDLE_current <= IDLE;
end case;
case visual_wka_wka_writenodeidle_current is
 when writenodeidle =>
  if (start_build = '1') then
   if (freehandle_valid = '1') then
    --nodememory write request
    nodemem_handle_i <= freehandle;
    nodemem_request <= '1';
    nodemem_rw <= '0'; -- write
    nodemem_datain <= (level,low,high,firsthandle,gc_zero);

    tookfreehandle <= '1';
    visual_wka_wka_writenodeidle_current <= writenodemem;
   else
    visual_wka_wka_writenodeidle_current <= writenodewait;
   end if;
  else
   visual_wka_wka_writenodeidle_current <= writenodeidle;
  end if;

 -- write the node to node memory

 when writenodemem =>
  if (nodemem_ack = '1') then
   resulthandle <= freehandle;
   -- turn of write request
   nodemem_request <= '0';
   nodemem_rw <= '1';
   -- tookfreehandle is turned off each
   -- clock by default
   visual_wka_wka_writenodeidle_current <= writenodeidle;
  else
   visual_wka_wka_writenodeidle_current <= writenodemem;
  end if;

 when writenodewait =>
  if (freehandle_valid = '1') then
   --nodememory write request
   nodemem_handle_i <= freehandle;
   nodemem_request <= '1';
   nodemem_rw <= '0'; -- write
   nodemem_datain <= (level,low,high,firsthandle,gc_zero);

   tookfreehandle <= '1';
   visual_wka_wka_writenodeidle_current <= writenodemem;
  else
   visual_wka_wka_writenodeidle_current <= writenodewait;
  end if;

 when others =>

  visual_wka_wka_writenodeidle_current <= writenodeidle;
end case;
case visual_wka_wka_writeuniqueidle_current is
 when writeuniqueidle =>
  if (start_build = '1') then
   if (freehandle_valid = '1') then
    --unique memory write request
    unique_request <= '1';
    unique_rw <= '0'; -- write
    unique_datain <= freehandle;
    unique_handle <= hashval;
    uniquemiss := uniquemiss + 1;
    visual_wka_wka_writeuniqueidle_current <= writeuniquemem;
   else
    visual_wka_wka_writeuniqueidle_current <= writeuniquewait;
   end if;
  else
   visual_wka_wka_writeuniqueidle_current <= writeuniqueidle;
  end if;

 -- write the handle to unique memory

 when writeuniquemem =>
  if (unique_ack = '1') then
   -- turn off write request
   unique_request <= '0';
   unique_rw <= '1';
   visual_wka_wka_writeuniqueidle_current <= writeuniqueidle;
  else
   visual_wka_wka_writeuniqueidle_current <= writeuniquemem;
  end if;

 when writeuniquewait =>
  if (freehandle_valid = '1') then
   --unique memory write request
   unique_request <= '1';
   unique_rw <= '0'; -- write
   unique_datain <= freehandle;
```

```
      unique_handle <= hashval;
      uniquemiss := uniquemiss + 1;
      visual_wka_wka_writeuniqueidle_current <= writeuniquemem;
     else
      visual_wka_wka_writeuniqueidle_current <= writeuniquewait;
     end if;

   when others =>

     visual_wka_wka_writeuniqueidle_current <= writeuniqueidle;
   end case;
  end if;
 end process;

 -- Combinational process
 mknodefsm_wka_comb:
 process (nodemem_ack, nodemem_dataready, nodemem_dataout, low,
high, level,
       build_busy, freehandle_valid, unique_ack, unique_dataready,
       unique_dataout, visual_wka_wka_IDLE_current,
       visual_wka_wka_writenodeidle_current,
       visual_wka_wka_writeuniqueidle_current)
 begin  -- Combinational process
  build_busy <= '0';
  start_build <= '0';

  case visual_wka_wka_IDLE_current is
   when FINDNODE =>
    if((nodemem_ack = '1') AND
    (nodemem_dataready = '1') AND
    NOT((nodemem_dataout.lo = low) AND
      (nodemem_dataout.hi = high) AND
      (nodemem_dataout.level = level))
    AND
    (nodemem_dataout.nextbdd = bddhandle_zero)
    AND
    (build_busy = '0') AND
    (freehandle_valid = '1'))
    then
    start_build <= '1';
    end if;

   -- wait until the unique memory access is complete

   when FINDUNIQUE =>
    if((unique_ack = '1') AND
    (unique_dataready = '1') AND
    (unique_dataout = bddhandle_zero) AND
    (build_busy = '0') AND
    (freehandle_valid = '1') )then
    start_build <= '1';
    end if;

   when WAITFORBUILD =>
    if ((build_busy = '0') AND
    (freehandle_valid = '1')) then
    start_build <= '1';
    end if;

   when others =>
    null;
  end case;
  case visual_wka_wka_writenodeidle_current is
   -- write the node to node memory

   when writenodemem =>
    build_busy <= '1';

   when writenodewait =>
    build_busy <= '1';

   when others =>
    null;
  end case;
  case visual_wka_wka_writeuniqueidle_current is
   -- write the handle to unique memory

   when writeuniquemem =>
    build_busy <= '1';

   when writeuniquewait =>
    build_busy <= '1';

   when others =>
    null;
  end case;
 end process;

 nodemem_handle <= nodemem_handle_i;
end mknodefsm;

-- this diagram contains the main functional units
-- for the BDD processor.
-- It connects the functional units with the memory
-- models.
-- Generics can be passed in from the above level to set
-- the memory sizes

library ieee;
use ieee.STD_LOGIC_1164.all;
library work;
use work.kernel.all;
library SYNOPSYS;
use SYNOPSYS.ATTRIBUTES.ALL;


entity mknodeblk is
 generic (
      nodememsize : NATURAL          := bddmemsize;
      camsize   : NATURAL          := bddcamsize;
      uniquesize  : NATURAL          := bdduniquetablesize
      );
 port (
    applyop           : in booleanop;
    bddin1            : in bddhandle;
    bddin2            : in bddhandle;
    clk               : in std_logic;
    enableand         : in std_logic;
    enablenot         : in std_logic;
    freehandle_valid_o   : out std_logic;
    init              : in std_logic;
    LOWONNODES           : out std_logic;
    tp_mknode_result     : out bddhandle;
    tp_mknode_resultvalid : out std_logic;
    mkselect          : in std_logic;
    start1            : in std_logic;
    level1            : in bddvar;
    low1              : in bddhandle;
    high1             : in bddhandle;
    OUTOFNODES          : out std_logic;
    resulthandle        : out bddhandle;
    resultvalid         : out std_logic;
    rst               : in std_logic;
    startapply         : in std_logic;
    startnot           : in std_logic
    );


end mknodeblk;


library bddlib;
architecture mknodeblk of mknodeblk is

 signal call_cmd         : stackcmd;
 signal call_datain0      : bdd_t;
 signal call_datain1      : bdd_t;
 signal call_datain2      : bdd_t;
 signal call_dataout0      : bdd_t;
signal call_dataout1      : bdd_t;
signal call_dataout2      : bdd_t;
signal call_empty         : std_logic;
signal call_full          : std_logic;
signal cam_ack            : std_logic;
signal cam_busy           : std_logic;
signal cam_busy_cnt       : NATURAL          := 0;
signal cam_field1         : camfield;
signal cam_field2         : camfield;
signal cam_field3         : camfield;
signal cam_found          : std_logic;
signal cam_request        : std_logic;
signal cam_result         : camfield;
signal cam_resultin       : camfield;
signal cam_resultvalid    : std_logic;
signal cam_rw             : std_logic;
signal freehandle         : bddhandle;
signal freehandle_valid   : std_logic;
signal high               : bddhandle;
signal high0              : bddhandle;
signal level              : bddvar;
signal level0             : bddvar;
signal low                : bddhandle;
signal low0               : bddhandle;
signal mknode_result      : bddhandle;
signal mknode_resultvalid : std_logic;
signal node_busy          : std_logic;
signal node_busy_cnt      : NATURAL          := 0;
signal node_dataout       : bdd_t;
signal node_datavalid     : std_logic;
signal node_port1_ack     : std_logic;
signal node_port2_ack     : std_logic;
signal node_port3_ack     : std_logic;
signal port1_datain       : bdd_t;
signal port1_handle       : bddhandle;
signal port1_request      : std_logic;
signal port1_rw           : std_logic;
signal port2_datain       : bdd_t;
signal port2_handle       : bddhandle;
signal port2_request      : std_logic;
signal port2_rw           : std_logic;
signal port3_datain       : bdd_t;
signal port3_handle       : bddhandle;
signal port3_request      : std_logic;
signal port3_rw           : std_logic;
signal post_init_clk_cnt  : NATURAL          := 0;
signal result_cmd         : stackcmd;
signal result_datain0     : bddhandle;
signal result_datain1     : bddhandle;
signal result_datain2     : bddhandle;
signal result_dataout0    : bddhandle;
signal result_dataout1    : bddhandle;
signal result_dataout2    : bddhandle;
signal result_empty       : std_logic;
signal result_full        : std_logic;
signal start              : std_logic;
signal start0             : std_logic;
signal start_cnt          : NATURAL        := 0;
signal tookfreehandle1    : std_logic;
signal unique_ack         : std_logic;
signal unique_busy        : std_logic;
signal unique_busy_cnt    : NATURAL          := 0;
signal unique_datain      : bddhandle;
signal unique_dataout     : bddhandle;
signal unique_dataready   : std_logic;
signal unique_handle      : bddhandle;
signal unique_request     : std_logic;
signal unique_rw          : std_logic;
component mknodefsm
  generic (
        uniquesize : NATURAL          := bdduniquetablesize
        );
  port (
      clk           : in std_logic;
      rst           : in std_logic;
      start            : in std_logic;
      level            : in bddvar;
      low              : in bddhandle;
      freehandle       : in bddhandle;
      freehandle_valid : in std_logic;
      high             : in bddhandle;
      resulthandle     : out bddhandle;
      resultnode       : out bdd_t;
      tookfreehandle   : out std_logic;
      result_valid     : out std_logic;
      unique_handle    : out bddhandle;
      unique_datain    : out bddhandle;
      unique_rw        : out std_logic;
      unique_request   : out std_logic;
      unique_dataready : in std_logic;
      unique_ack       : in std_logic;
      unique_busy      : in std_logic;
      unique_dataout   : in bddhandle;
      nodemem_dataout  : in bdd_t;
      nodemem_busy     : in std_logic;
      nodemem_dataready : in std_logic;
      nodemem_ack      : in std_logic;
      nodemem_handle   : out bddhandle;
      nodemem_datain   : out bdd_t;
      nodemem_rw       : out std_logic;
      nodemem_request  : out std_logic
      );
end component;
component uniquemem
  generic (
        readdelay  : NATURAL          := 0;
        writedelay : NATURAL          := 0;
        uniquesize : NATURAL          := bdduniquetablesize
        );
  port (
      port1_ack      : out std_logic;
      port1_busy     : out std_logic;
      port1_dataready : out std_logic;
      port1_dataout  : out bddhandle;
      port1_handle   : in bddhandle;
      port1_datain   : in bddhandle;
      port1_rw       : in std_logic;
      port1_request  : in std_logic;
      clk            : in std_logic;
      rst            : in std_logic
      );
end component;
component memctrl
  generic (
        readdelay  : NATURAL          := 0;
        writedelay : NATURAL          := 0;
        memsize    : NATURAL          := bddmemsize
        );
  port (
      port3_request : in std_logic;
      port3_ack     : out std_logic;
      port3_handle  : in bddhandle;
      port3_datain  : in bdd_t;
      port3_rw      : in std_logic;
      port2_request : in std_logic;
      port2_ack     : out std_logic;
      port2_handle  : in bddhandle;
      port2_datain  : in bdd_t;
      port2_rw      : in std_logic;
      port1_request : in std_logic;
      port1_ack     : out std_logic;
      port1_handle  : in bddhandle;
      port1_datain  : in bdd_t;
      port1_rw      : in std_logic;
      datavalid     : out std_logic;
      dataout       : out bdd_t;
      busy          : out std_logic;
      clk           : in std_logic;
      rst           : in std_logic
      );
```

```
end component;
component APPLYBLK
   port (
       applyerror        : out std_logic;
       start0            : out std_logic;
       level0            : out bddvar;
       low0              : out bddhandle;
       high0             : out bddhandle;
       applyop           : in booleanop;
       bddin1            : in bddhandle;
       bddin2            : in bddhandle;
       call_dataout0     : in bdd_t;
       call_dataout1     : in bdd_t;
       call_dataout2     : in bdd_t;
       call_full         : in std_logic;
       call_empty        : in std_logic;
       call_datain0      : out bdd_t;
       call_datain1      : out bdd_t;
       call_datain2      : out bdd_t;
       call_cmd          : out stackcmd;
       cam_request       : out std_logic;
       cam_rw            : out std_logic;
       cam_field1        : out camfield;
       cam_field2        : out camfield;
       cam_field3        : out camfield;
       cam_resultin      : out camfield;
       cam_ack           : in std_logic;
       cam_busy          : in std_logic;
       cam_result        : in camfield;
       cam_resultvalid   : in std_logic;
       cam_found         : in std_logic;
       clk               : in std_logic;
       enableand         : in std_logic;
       enablenot         : in std_logic;
       mknode_result     : in bddhandle;
       mknode_resultvalid : in std_logic;
       node_port3_ack    : in std_logic;
       node_port2_ack    : in std_logic;
       node_port1_ack    : in std_logic;
       node_datavalid    : in std_logic;
       node_dataout      : in bdd_t;
       node_busy         : in std_logic;
       port3_datain      : out bdd_t;
       port3_handle      : out bddhandle;
       port3_request     : out std_logic;
       port3_rw          : out std_logic;
       resulthandle      : out bddhandle;
       result_datain0    : out bddhandle;
       result_datain1    : out bddhandle;
       result_datain2    : out bddhandle;
       result_cmd        : out stackcmd;
       result_dataout0   : in bddhandle;
       result_dataout1   : in bddhandle;
       result_dataout2   : in bddhandle;
       result_full       : in std_logic;
       result_empty      : in std_logic;
       resultvalid       : out std_logic;
       rst               : in std_logic;
       startapply        : in std_logic;
       startnot          : in std_logic
       );
end component;
component freenodecntl
   generic (
         minhandle : bddhandle          := bdd_minhandle;
         maxhandle : bddhandle          := bdd_maxhandle
         );
   port (
       clk               : in std_logic;
       rst               : in std_logic;
       init              : in std_logic;
       tookfreehandle1   : in std_logic;
       freehandle        : out bddhandle;
       freehandle_valid  : out std_logic;
       LOWONNODES        : out std_logic;
       OUTOFNODES        : out std_logic;
       nodemem_busy      : in std_logic;
       nodemem_ack       : in std_logic;
       nodemem_dataready : in std_logic;
       nodemem_dataout   : in bdd_t;
       nodemem_request   : out std_logic;
       nodemem_handle    : out bddhandle;
       nodemem_datain    : out bdd_t;
       nodemem_rw        : out std_logic
       );
end component;
component handlestack
   generic (
         size : NATURAL          := 6
         );
   port (
       clk   : in std_logic;
       rst   : in std_logic;
       cmd   : in stackcmd;
       datain0 : in bddhandle;
       datain1 : in bddhandle;
       datain2 : in bddhandle;
       head0  : out bddhandle;
       head1  : out bddhandle;
       head2  : out bddhandle;
       full   : out std_logic;
       empty  : out std_logic
       );
end component;
component bddstack
   generic (
         size : NATURAL          := 6
         );
   port (
       clk   : in std_logic;
       rst   : in std_logic;
       cmd   : in stackcmd;
       datain0 : in bdd_t;
       datain1 : in bdd_t;
       datain2 : in bdd_t;
       head0  : out bdd_t;
       head1  : out bdd_t;
       head2  : out bdd_t;
       full   : out std_logic;
       empty  : out std_logic
       );
end component;
component cam
   generic (
         memsize   : NATURAL          := bddcamsize;
         readdelay  : NATURAL;
         writedelay : NATURAL
         );
   port (
       clk            : in std_logic;
       rst            : in std_logic;
       cam_request    : in std_logic;
       cam_rw         : in std_logic;
       cam_ack        : out std_logic;
       cam_busy       : out std_logic;
       cam_result     : out camfield;
       cam_resultvalid : out std_logic;
       cam_found      : out std_logic;
       cam_field1     : in camfield;
       cam_field2     : in camfield;
       cam_field3     : in camfield;
       cam_resultin   : in camfield
       );
end component;

-- Start Configuration Specification
for all : mknodefsm use entity bddlib.mknodefsm(mknodefsm);
for all : uniquemem use entity bddlib.uniquemem(uniquemem);
for all : memctrl use entity bddlib.memctrl(memctrl);
for all : APPLYBLK use entity bddlib.APPLYBLK(APPLYBLK);
```

```
 for all : freenodecntl use entity bddlib.freenodecntl(freenodecntl);
 for all : handlestack use entity bddlib.handlestack(handlestack);
 for all : bddstack use entity bddlib.bddstack(bddstack);
 for all : cam use entity bddlib.cam(cam);
 -- End Configuration Specification

begin
 freehandle_valid_o <= freehandle_valid;
 tp_mknode_result <= mknode_result;
 tp_mknode_resultvalid <= mknode_resultvalid;

 mknodectrl: mknodefsm
  generic map (uniquesize)
  port map (
        clk => clk,
        rst => rst,
        start => start,
        level => level,
        low => low,
        freehandle => freehandle,
        freehandle_valid => freehandle_valid,
        high => high,
        resulthandle => mknode_result,
        resultnode => open,
        tookfreehandle => tookfreehandle1,
        result_valid => mknode_resultvalid,
        unique_handle => unique_handle,
        unique_datain => unique_datain,
        unique_rw => unique_rw,
        unique_request => unique_request,
        unique_dataready => unique_dataready,
        unique_ack => unique_ack,
        unique_busy => unique_busy,
        unique_dataout => unique_dataout,
        nodemem_dataout => node_dataout,
        nodemem_busy => node_busy,
        nodemem_dataready => node_datavalid,
        nodemem_ack => node_port1_ack,
        nodemem_handle => port1_handle,
        nodemem_datain => port1_datain,
        nodemem_rw => port1_rw,
        nodemem_request => port1_request);

 inst_uniquemem: uniquemem
  generic map (0,
          0,
          uniquesize)
  port map (
        port1_ack => unique_ack,
        port1_busy => unique_busy,
        port1_dataready => unique_dataready,
        port1_dataout => unique_dataout,
        port1_handle => unique_handle,
        port1_datain => unique_datain,
        port1_rw => unique_rw,
        port1_request => unique_request,
        clk => clk,
        rst => rst);

 nodememory: memctrl
  generic map (0,
          0,
          nodememsize)
  port map (
        port3_request => port3_request,
        port3_ack => node_port3_ack,
        port3_handle => port3_handle,
        port3_datain => port3_datain,
        port3_rw => port3_rw,
        port2_request => port2_request,
        port2_ack => node_port2_ack,
        port2_handle => port2_handle,
        port2_datain => port2_datain,
        port2_rw => port2_rw,
        port1_request => port1_request,
        port1_ack => node_port1_ack,
        port1_handle => port1_handle,
        port1_datain => port1_datain,
        port1_rw => port1_rw,
        datavalid => node_datavalid,
        dataout => node_dataout,
        busy => node_busy,
        clk => clk,
        rst => rst);

APPLY_BLK: APPLYBLK
 port map (
        applyerror => open,
        start0 => start0,
        level0 => level0,
        low0 => low0,
        high0 => high0,
        applyop => applyop,
        bddin1 => bddin1,
        bddin2 => bddin2,
        call_dataout0 => call_dataout0,
        call_dataout1 => call_dataout1,
        call_dataout2 => call_dataout2,
        call_full => call_full,
        call_empty => call_empty,
        call_datain0 => call_datain0,
        call_datain1 => call_datain1,
        call_datain2 => call_datain2,
        call_cmd => call_cmd,
        cam_request => cam_request,
        cam_rw => cam_rw,
        cam_field1 => cam_field1,
        cam_field2 => cam_field2,
        cam_field3 => cam_field3,
        cam_resultin => cam_resultin,
        cam_ack => cam_ack,
        cam_busy => cam_busy,
        cam_result => cam_result,
        cam_resultvalid => cam_resultvalid,
        cam_found => cam_found,
        clk => clk,
        enableand => enableand,
        enablenot => enablenot,
        mknode_result => mknode_result,
        mknode_resultvalid => mknode_resultvalid,
        node_port3_ack => node_port3_ack,
        node_port2_ack => node_port2_ack,
        node_port1_ack => node_port1_ack,
        node_datavalid => node_datavalid,
        node_dataout => node_dataout,
        node_busy => node_busy,
        port3_datain => port3_datain,
        port3_handle => port3_handle,
        port3_request => port3_request,
        port3_rw => port3_rw,
        resulthandle => resulthandle,
        result_datain0 => result_datain0,
        result_datain1 => result_datain1,
        result_datain2 => result_datain2,
        result_cmd => result_cmd,
        result_dataout0 => result_dataout0,
        result_dataout1 => result_dataout1,
        result_dataout2 => result_dataout2,
        result_full => result_full,
        result_empty => result_empty,
        resultvalid => resultvalid,
        rst => rst,
        startapply => startapply,
        startnot => startnot);

freenodecntrl: freenodecntl
 generic map (bdd_minhandle,
          nodememsize - 1)
 port map (
        clk => clk,
        rst => rst,
        init => init,
        tookfreehandle1 => tookfreehandle1,
        freehandle => freehandle,
        freehandle_valid => freehandle_valid,
        LOWONNODES => LOWONNODES,
        OUTOFNODES => OUTOFNODES,
        nodemem_busy => node_busy,
        nodemem_ack => node_port2_ack,
        nodemem_dataready => node_datavalid,
        nodemem_dataout => node_dataout,
        nodemem_request => port2_request,
        nodemem_handle => port2_handle,
        nodemem_datain => port2_datain,
        nodemem_rw => port2_rw);

resultstack: handlestack
  generic map (1024)
  port map (
        clk => clk,
        rst => rst,
        cmd => result_cmd,
        datain0 => result_datain0,
        datain1 => result_datain1,
        datain2 => result_datain2,
        head0 => result_dataout0,
        head1 => result_dataout1,
        head2 => result_dataout2,
        full => result_full,
        empty => result_empty);

callstack: bddstack
  generic map (1024)
  port map (
        clk => clk,
        rst => rst,
        cmd => call_cmd,
        datain0 => call_datain0,
        datain1 => call_datain1,
        datain2 => call_datain2,
        head0 => call_dataout0,
        head1 => call_dataout1,
        head2 => call_dataout2,
        full => call_full,
        empty => call_empty);

inst_cam: cam
  generic map (camsize,
          0,
          0)
  port map (
        clk => clk,
        rst => rst,
        cam_request => cam_request,
        cam_rw => cam_rw,
        cam_ack => cam_ack,
        cam_busy => cam_busy,
        cam_result => cam_result,
        cam_resultvalid => cam_resultvalid,
        cam_found => cam_found,
        cam_field1 => cam_field1,
        cam_field2 => cam_field2,
        cam_field3 => cam_field3,
        cam_resultin => cam_resultin);

process(mkselect,start0,level0,low0,high0,start1,level1,low1,high1)
begin
if(mkselect = '0') then
start <= start0;
level <= level0;
low <= low0;
high <= high0;
else
start <= start1;
level <= level1;
 low <= low1;
 high <= high1;
 end if;
 end process;

 --synopsys translate off
 process(clk,rst)
 variable flag : boolean := FALSE;
 begin
 if(rst = '0') then
 node_busy_cnt <= 0;
 cam_busy_cnt <= 0;
 unique_busy_cnt <= 0;
 elsif(clk'event and clk = '1') then

 if(freehandle_valid = '1') then
 flag := true;
 end if;
 if(flag) then
 post_init_clk_cnt <= post_init_clk_cnt + 1;
 if(node_busy = '1') then
 node_busy_cnt <= node_busy_cnt+1;
 end if;
 if(cam_busy = '1') then
 cam_busy_cnt <= cam_busy_cnt+1;
 end if;
 if(unique_busy = '1') then
 unique_busy_cnt <= unique_busy_cnt+1;
 end if;

 end if;
 -- since all of the three start sigs are mutually
 -- exclusive just inc when any 1 is true
 if((startapply = '1')  OR (startnot = '1') OR (start1 = '1'))then
 start_cnt <= start_cnt + 1;
 end if;

 end if;

 end process;
 --synopsys translate on
end mknodeblk;
------------------------------------------
------------------------------------------
-- Date       : Wed May 10 14:35:34 2000
--
-- Author     : Bob Hatt
--
-- Company    : Portland State University
--
-- Description :
--
------------------------------------------
------------------------------------------
library ieee; use ieee.STD_LOGIC_1164.all;


library bddlib; use bddlib.kernel.all;

entity  mknodeblktestbench  is
generic (N : natural := 4);

port (clk : out std_logic ;
     high : out bddhandle ;
     init : out std_logic ;
     level : out bddvar ;
     low : out bddhandle ;
     rst : out std_logic ;
     startnot : out std_logic ;
     mkselect : out std_logic ;
     bddin1 : out bddhandle ;
     bddin2 : out bddhandle ;
     enablenot : out std_logic ;
     enableand : out std_logic ;
     start_mknode : out std_logic ;
```

```
    startapply : out std_logic ;
    applyop : out booleanop ;
    LOWONNODES : in std_logic ;
    OUTOFNODES : in std_logic ;
    mknode_result : in bddhandle ;
    mknode_resultvalid : in std_logic ;
    freehandle_valid_o : in std_logic ;
    applyresult_valid : in std_logic ;
    applyresult : in bddhandle ;
    testdone : out boolean );
end;

-----------------------------------------
-----------------------------------------
-- Date      : Wed May 10 14:35:34 2000
--
-- Author    : Bob Hatt
--
-- Company   : Portland State University
--
-- Description :
--
-----------------------------------------
-----------------------------------------
architecture Nqueen of mknodeblktestbench is
constant clkperiod : time := 10 ns;
constant clkperiodplusone : time := clkperiod + 1ns;
signal clki,rsti : std_logic;
signal queensig: bddhandle;

begin

-- clock with 50% duty 10 ns period
process
begin
clki <= '0';
wait for 5 ns;
while(TRUE) loop
clki <= NOT clki;
wait for 5 ns;
end loop;

end process;
clk <= clki;


--make reset active from 0-40
rsti <= '0', '1' after clkperiod *4;
rst <= rsti;

-- make init active from 60-80
init <= '0', '1' after clkperiod * 6, '0' after clkperiod * 8;

-- must wait for memsize*(memdelay+1) clocks until init of node mem
is done
-- for memsize = 256 memdelay = 0, 256 clocks

process
procedure mknodedirect( lvl,lo,hi : in bddhandle) is
begin
-- wait until clki = '1';
mkselect <= '1';

start_mknode <= '1';
level <= lvl;
high <= hi;
low <= lo;
wait for clkperiodplusone + 1ns;
start_mknode <= '0';
wait until mknode_resultvalid = '1';
end mknodedirect;

procedure ithvar(result: out bddhandle;i : in bddvar) is
begin
mknodedirect(i,bddhandle_zero,bddhandle_one);
result := mknode_result;
end ithvar;
procedure nithvar(result: out bddhandle;i : in bddvar) is
begin
mknodedirect(i,bddhandle_one,bddhandle_zero);
result := mknode_result;
end nithvar;

-- function hardbdd_ithvar( i : in bddvar) return bddhandle is
-- begin
-- mknodedirect(i,bddhandle_zero,bddhandle_one);
-- return(mknode_result);
-- end hardbdd_ithvar;
--
-- function hardbdd_nithvar( i : in bddvar) return bddhandle is
-- begin
-- mknodedirect(i,bddhandle_one,bddhandle_zero);
-- return(mknode_result);
-- end hardbdd_nithvar;

procedure apply_not(result: out bddhandle;handle : in bddhandle) is
begin
mkselect <= '0';
enablenot <= '1';

startnot <= '1';
applyop <= booleanop_not;
bddin1 <= handle;
bddin2 <= bddhandle_zero;
wait for clkperiodplusone;
startnot <= '0';
wait until applyresult_valid = '1';
result := applyresult;
enablenot <= '0';

end apply_not;

-- function hardbdd_apply_not(handle : in bddhandle) return bddhandle
is
-- begin
-- applynotdirect(handle);
-- return(applyresult);
-- end hardbdd_apply_not;


procedure apply(result: out bddhandle;handle1,handle2 : in bddhandle;
op : in booleanop) is
begin
mkselect <= '0';
enableand <= '1';
startapply <= '1';
bddin1 <= handle1;
bddin2 <= handle2;
applyop <= op;
wait for clkperiodplusone;
startapply <= '0';
wait until applyresult_valid = '1';
result := applyresult;
enableand <= '0';
end apply;

-- function hardbdd_apply(handle1,handle2: in bddhandle; op :in bool-
eanop) return bddhandle is
-- begin
-- applydirect(handle1,handle2,op);
-- return(applyresult);
-- end hardbdd_apply;
```

```
--

procedure init is
begin
start_mknode <= '0';
startnot <= '0';
enablenot <= '0';
startapply <= '0';
enableand <= '0';
wait until rsti ='1';
wait until freehandle_valid_o = '1';
wait for clkperiod/2;
end init;




-- local variables for this algorithm
constant boardsize : positive := N;
type bddhandle2d is array(natural range <>, natural range <>) of
bddhandle;
variable X : bddhandle2d(1 to boardsize,1 to boardsize);
variable queen,tmp1,tmp2: bddhandle;

procedure build(i,j : integer) is
variable a,b,c,d :bddhandle := bddhandle_one;
variable tmp1,tmp2 : bddhandle;
variable int1 : integer;
begin
-- no one in same column
for k in 1 to boardsize
loop
if(k /= j) then
apply_not(tmp1,X(i,k));
apply(tmp2,X(i,j),tmp1,booleanop_imp);
apply(a,a,tmp2,booleanop_and);
end if;
end loop;
-- no one in same row
for k in 1 to boardsize
loop
if(k /= i) then
apply_not(tmp1,X(k,j));
apply(tmp2,X(i,j),tmp1,booleanop_imp);
apply(b,b,tmp2,booleanop_and);
end if;
end loop;
-- no one in up right diagonal
for k in 1 to boardsize
loop
int1 := k-i+j;
if((int1 >= 1) AND (int1 <= boardsize)) then
if(k /= i) then
apply_not(tmp1,X(k,int1));
apply(tmp2,X(i,j),tmp1,booleanop_imp);
apply(c,c,tmp2,booleanop_and);
end if;
end if;
end loop;
-- no one in down right diagonal
for k in 1 to boardsize
loop
int1 := i+j-k;
if((int1 >= 1) AND (int1 <= boardsize)) then
if(k /= i) then
apply_not(tmp1,X(k,int1));
apply(tmp2,X(i,j),tmp1,booleanop_imp);
apply(d,d,tmp2,booleanop_and);
end if;
end if;
end loop;

apply(tmp1,a,b,booleanop_and);
apply(tmp1,tmp1,c,booleanop_and);
apply(tmp1,tmp1,d,booleanop_and);
apply(queen,queen,tmp1,booleanop_and);

end;
begin
init;

-------------- put the core algorithm here -----------------


queen := bddhandle_one;


-- initialize the board variables
for i in 1 to boardsize
loop
for j in 1 to boardsize
loop
ithvar(X(i,j),(i-1)*boardsize +(j-1));
nithvar(tmp1,(i-1)*boardsize +(j-1));
end loop;
end loop;

-- put a queen in each row
for i in 1 to boardsize
loop
tmp1 := bddhandle_zero;
for j in 1 to boardsize
loop
apply(tmp1,tmp1,X(i,j),booleanop_or);
end loop;
apply(queen,queen,tmp1,booleanop_and);
end loop;
ASSERT false report "queen now has in each row at each position"
severity note;
queensig <= queen;

-- build the constraints for each position
for i in 1 to boardsize
loop
for j in 1 to boardsize
loop
build(i,j);
assert false report "building contraints for i,j" severity note;
queensig <= queen;
end loop;
end loop;

-------------- end of program ------------------------------

queensig <= queen;
testdone <= true;



end process;
end;



library ieee;
use ieee.STD_LOGIC_1164.all;
library bddlib;
use bddlib.kernel.all;
library SYNOPSYS;
```

```
use SYNOPSYS.ATTRIBUTES.ALL;


entity testmknodeblk is


end testmknodeblk;


library bddlib;
architecture testmknodeblk of testmknodeblk is

 constant clkperiod    : TIME    := 10 ns;
 constant nodememdelay  : NATURAL := 0;
 constant N           : NATURAL := 4;
 signal  applyop       : booleanop;
 signal  applyresult   : bddhandle;
 signal  applyresult_valid : std_logic;
 signal  bddin1        : bddhandle;
 signal  bddin2        : bddhandle;
 signal  clk          : std_logic;
 signal  enableand     : std_logic;
 signal  enablenot     : std_logic;
 signal  freehandle_valid_o : std_logic;
 signal  high          : bddhandle;
 signal  init          : std_logic;
 signal  level         : bddvar;
 signal  low           : bddhandle;
 signal  LOWONNODES    : std_logic;
 signal  mknode_result : bddhandle;
 signal  mknode_resultvalid : std_logic;
 signal  mkselect      : std_logic;
 signal  OUTOFNODES    : std_logic;
 signal  rst           : std_logic;
 signal  start_mknode  : std_logic;
 signal  startapply    : std_logic;
 signal  startnot      : std_logic;
 signal  testdone      : BOOLEAN;
 component mknodeblktestbench
   generic (
        N : NATURAL            := 4
        );
   port (
      clk             : out std_logic;
      high             : out bddhandle;
      init             : out std_logic;
      level            : out bddvar;
      low              : out bddhandle;
      rst             : out std_logic;
      startnot          : out std_logic;
      mkselect          : out std_logic;
      bddin1            : out bddhandle;
      bddin2            : out bddhandle;
      enablenot         : out std_logic;
      enableand         : out std_logic;
      start_mknode      : out std_logic;
      startapply        : out std_logic;
      applyop           : out booleanop;
      LOWONNODES        : in std_logic;
      OUTOFNODES        : in std_logic;
      mknode_result     : in bddhandle;
      mknode_resultvalid : in std_logic;
      freehandle_valid_o : in std_logic;
      applyresult_valid  : in std_logic;
      applyresult        : in bddhandle;
      testdone           : out BOOLEAN
      );
 end component;
 component mknodeblk
   generic (
        nodememsize : NATURAL          := bddmemsize;
        camsize    : NATURAL          := bddcamsize;
        uniquesize  : NATURAL          := bdduniquetablesize
        );
   port (
        applyop           : in booleanop;
        bddin1            : in bddhandle;
        bddin2            : in bddhandle;
        clk              : in std_logic;
        enableand         : in std_logic;
        enablenot         : in std_logic;
        freehandle_valid_o  : out std_logic;
        init             : in std_logic;
        LOWONNODES        : out std_logic;
        tp_mknode_result    : out bddhandle;
        tp_mknode_resultvalid : out std_logic;
        mkselect          : in std_logic;
        start1            : in std_logic;
        level1            : in bddvar;
        low1              : in bddhandle;
        high1             : in bddhandle;
        OUTOFNODES        : out std_logic;
        resulthandle       : out bddhandle;
        resultvalid        : out std_logic;
        rst              : in std_logic;
        startapply         : in std_logic;
        startnot          : in std_logic
        );
 end component;

 -- Start Configuration Specification
 for all : mknodeblktestbench use entity bddlib.mknodeblktest-
bench(Nqueen);
 for all : mknodeblk use entity bddlib.mknodeblk(mknodeblk);
 -- End Configuration Specification

begin

 inst_mknodeblktestbench: mknodeblktestbench
  generic map (N)
  port map (
        clk => clk,
        high => high,
        init => init,
        level => level,
        low => low,
        rst => rst,
        startnot => startnot,
        mkselect => mkselect,
        bddin1 => bddin1,
        bddin2 => bddin2,
        enablenot => enablenot,
        enableand => enableand,
        start_mknode => start_mknode,
        startapply => startapply,
        applyop => applyop,
        LOWONNODES => LOWONNODES,
        OUTOFNODES => OUTOFNODES,
        mknode_result => mknode_result,
        mknode_resultvalid => mknode_resultvalid,
        freehandle_valid_o => freehandle_valid_o,
        applyresult_valid => applyresult_valid,
        applyresult => applyresult,
        testdone => testdone);

 C1: mknodeblk
  generic map (220000,
          220000,
          219983)
  port map (
        applyop => applyop,
        bddin1 => bddin1,
        bddin2 => bddin2,
        clk => clk,
        enableand => enableand,
        enablenot => enablenot,
        freehandle_valid_o => freehandle_valid_o,
        init => init,
        LOWONNODES => LOWONNODES,
        tp_mknode_result => mknode_result,
          tp_mknode_resultvalid => mknode_resultvalid,
          mkselect => mkselect,
          start1 => start_mknode,
          level1 => level,
          low1 => low,
          high1 => high,
          OUTOFNODES => OUTOFNODES,
          resulthandle => applyresult,
          resultvalid => applyresult_valid,
          rst => rst,
          startapply => startapply,
          startnot => startnot);
end testmknodeblk;
```