TÉCNICO LISBOA

SoC-FPGA Accelerated BDD-Based Model Checking

Rúben Alexandre Pereira Teixeira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Dr. Rui António Policarpo Duarte
Prof. Pedro Tiago Gonçalves Monteiro

Examination Committee

Chairperson: Prof. Nuno João Neves Mamede
Supervisor: Dr. Rui António Policarpo Duarte
Members of the Committee: Prof. Paulo Ferreira Godinho Flores

January 2021

Resumo

Verificação de Modelos é considerada uma das ferramentas mais importantes em verificação formal, geralmente usada para verificar requerimentos de hardware e software. Uma das mais importantes descobertas em verificação de modelos foi o uso de Diagramas de Decisão Binária (Binary Decision Diagrams) para melhorar significativamente o tempo de execução e permitir abordar problemas maiores. Devido ao problema fundamental associado a sistemas que exploram espaços de estados, Verificadores de Modelos são considerados demasiado dispendiosos em relação ao tempo gasto para serem amplamente utilizados.

Neste trabalho, propomos uma arquitetura HW/SW que usa um dispositivo SoC-FPGA para melhorar o tempo de execução de Verificador de Modelos baseados em BDD. Desenvolvemos uma arquitetura base e duas modificações para um total de quatro arquiteturas diferentes. O resultado final chega perto, mas não melhora as implementações de software em termos de performance relativa.

Palavras chave

Verificação de Modelos, Diagramas de Decisão Binária, SoC-FPGA, Sistema Hardware/Software.

Abstract

Model Checking is considered one of the most important tools in formal verification, commonly used to verify hardware and software requirements. One of most important breakthroughs in Model Checking was the use of Binary Decision Diagrams (BDDs) to significantly improve the run time and allowing to tackle larger problems. Because of the fundamental problem that plagues state space exploration systems, Model Checkers are considered too much time consuming in order to be widely used.

In this paper, we propose a HW/SW architecture that uses SoC-FPGA devices to improve BDD based Model Checkers runtimes. We develop a base architecture and two modifications for a total of four different architectures. The end result comes close but does not improve in terms of performance with regards to a full software implementation.

Keywords

Model Checking, Binary Decision Diagram, SoC-FPGA, Hardware/Software System.

Table of Contents

List of Figures

List of Tables

Listings

1 Introduction

Formal verification is the process of verifying whether a system satisfies some desired property. In the context of software and hardware systems, formal verification is used to prove the correctness of programs and the absence of bugs or faults. One example of the importance of formal methods comes from NASA, which created a research program entirely focused in researching formal methods and its applications[2], because "Digital systems can fail in catastrophic ways leading to death or tremendous financial loss (...), design errors are increasingly becoming the most serious culprit" [3]. A more comprehensive view of the history of Formal Methods in space missions can be found in [21].

What makes formal verification very attractive is the fact that it can provide a guarantee. The most common non-formal approach taken for software verification is based on Testing. The developer creates a test suite with the purpose of exercising a program for certain inputs and comparing with the expected output, looking for discrepancies. These methods can never guarantee that the program is bug-free, as it is always possible that the test suite is not exercising some portion of the program. Formal methods, on the other hand, guarantee that if some property is proved, then that property holds for all possible inputs. [4]

Some of the methods associated with formal verification include: theorem proving, where a person manually, often using **Proof assistants** or **Automated theorem proving** programs, construct mathematically proofs that implies the desired properties; **Program derivation** techniques, where a person describes the specifications of the system and then uses a tool to create the code that follows that specification and **Model Checking**, where the user provides the specifications of the system (called a model) and the properties it wants to verify, and then the Model Checker reports whether the properties hold in the model provided.

Model Checking is one of the most used and well known methods of formally verifying concurrent systems. In software, it has been used to detect difficult problems to reproduce, such as deadlocks [22] and race conditions [43]. In hardware, it has been primarily used to verify the correct behavior of logical circuits [6, 7]. More recently, it has been used to analyze properties of complex biological systems [3].

Compared to other formal methods, Model checking offers a unique set of advantages that make it one of the most attractive tools in formal verification:

1. Automatic. When compared to methods like Theorem proving, where a person needs to manually construct proofs until it proves the validity of the desired specification, Model Checkers require less human interaction, as the encoding of the model and the specification is significantly easier and faster.
2. Offers counter-examples. If a model does not satisfy a certain property, a Model Checker can offer an example of a possible execution that showcases the fault. This counter-example can then be used to refine the model until it is fixed.
3. Works with partial specifications. The model provided does not need to fully specify the system for the Model checker to function. This allows the developer to perform verification while still in the process of designing the component.

Model Checkers work by exhaustively exploring the state space of the model provided. The model encodes the possible initial states of the system as well as the state transitions. Repeatedly computing the states transitions is equivalent to exploring every possible state of the system, which eventually allows the Model Checker to prove, by exhaustion, the desired property. If, instead, a state is reached

[2] https://shemesh.larc.nasa.gov/fm/index.html
[3] https://shemesh.larc.nasa.gov/fm/fm-why-new.html
[4] http://users.ece.cmu.edu/~koopman/des_s99/sw_testing/

that does not satisfy the property, we can provide a counter-example by computing the transitions backwards from that invalid state, knowing that we eventually reach one of the initial states, thus providing a complete trace of the transitions that lead from the initial to the invalid state.

Model Checkers are divided into two types: Explicit and Symbolic. Explicit Model Checkers store and operate in states individually. A state is just a collection of values and the transition relation encodes how those values change from state to state.

Symbolic Model Checkers, on the other hand, store and operate on multiple states at the same time. This is performed by encoding sets of states, instead of each state individually. In this approach, a state is simply a element of the set, and the model checker performs operation with sets. The transition relation encodes a transformation from set to set such that, for every state in the input set, the output set contains the corresponding state transition. The advantage of this approach is that a symbolic model checker computes every state transition in a single step, instead of computing each individually.

In order to function, symbolic model checkers need to be able to encode boolean formulas in a space efficient manner while being able to perform boolean operations on them, like conjugation, negation, among others. A simple approach, like encoding the truth table, would not work: in the worst case, truth tables memory grows exponentially. The most important breakthrough in all of symbolic model checking was the discovery and usage of BDDs, with some additional constraints, to solve the boolean encoding problem. There exists Symbolic Model Checkers that use other data structures than BDDs, but nevertheless, they are still widely used and are still considered the basis for Symbolic Model Checking.

A BDD is a binary tree like data structure, where each node is either a terminal node or a decision node associated with a variable. Two important constraints imposed on BDDs allow nodes to be reused. This important fact is the reason BDDs are so fundamental for Model Checkers. In best case scenarios, the memory occupied by a BDD scales linearly with respect to the amount of input variables. Best case scenarios are rare, and in the worst case BDDs memory usage still grows exponentially, but it is still one of the most prevalent data structures used in Symbolic Model Checking. Boolean operations on BDDs can be performed directly, by means of tree traversal algorithms, that traverse both BDDs and produce a output BDD.

1.1 Motivation

The motivation for this thesis is to reduce the runtime of BDD Based Symbolic Model Checkers. Two factors impact the runtime:

1. The complexity of the model, which affects runtime by increasing the amount of BDD nodes needed to process. More possible states and more variable per state lead to an increase in the number of BDD nodes needed to represent them, which makes operations between BDDs more time consuming, as BDD operations need to traverse more nodes.
2. The complexity of the properties, which affects runtime by increasing the amount of states needed to explore. Exploring more states implies the computation of more transitions, which corresponds to performing more BDD operations.

Together, these factors contribute to one of the fundamental disadvantages commonly associated with Model Checkers: they tend to not scale up and the increase in computation time eventually becomes a detrimental factor for their usage. At the same time, these factors showcase how the majority of the Model Checking run time is dependent on the implementation of BDD operations.

Because of this, a considerable amount of work evolved in Model Checking is focused in improving the performance of BDD operations. This includes works based on exploiting the memory hierarchy by developing cache-friendly architectures, using multi-core and multi-computer systems, using GPUs and vector-processing supercomputers.

FPGA technology, on the other hand, has not seen much use, in regards to Symbolic BDD based approaches. The majority of work that utilizes FPGA technology to accelerate Model Checkers focus on the Explicit types and barely has been any work done for the Symbolic approaches. This is partially because Explicit Model Checking algorithms tend to be better suited for FPGA acceleration.

In recent years, FPGA technology has experience tremendous growth [41], making it an attractive choice for hardware accelerated computations. More specifically, the advances in SoC-FPGA technology made their use more viable for a wider range of applications.

What makes the use of SoC-FPGA a potential solution for improving Model Checkers performance is the fact that Model Checkers rely on only a few computationally intensive operations. This makes it possible to create a specialized SW/HW system, where the hardware portion implements the most intensive portions, the BDD operations, while the software implements the remainder of the Model Checking program.

1.2 Objectives

The main goal of this thesis is to explore the use of SoC-FPGA systems to improve the performance of BDD Based Model Checkers.

For that purpose, we:

1. Identify the most time consuming BDD operations, primary candidates for acceleration.
2. Propose a set of HW/SW architectures that accelerate BDD operations.
3. Integrate the architectures with an existing BDD Based Model Checker.
4. Create testing code to guarantee correct behavior.
5. Study the performance impacts and the scalability of the developed architectures.

1.3 Contributions

This paper presents a base HW/SW architecture as well as two modifications that can be implemented simultaneously or separately, for a total of four architectures.

The architectures proposed come close to matching the performance of the original software implementation. The architectures can be further improved as optimizations are still possible to be made.

The source code of the software and the hardware, as well as a small manual that explains how to run the implementations, is available in: https://github.com/zettasticks/Bdd-Fpga.

1.4 Thesis Outline

The rest of this dissertation is organized as follows: Chapter 2 gives a theoretical background that explains how model checkers work, what are BDDs, how the operations between BDDs are performed and finally what are SoC-FPGAs and how they function, to better understand the remaining chapters. We finish by presenting previous work related to Model Checking acceleration.

Chapter 3 presents the HW/SW architecture. It explains the communication between HW and SW, the partition of memory, the division of the architecture in modules and how those modules work. The architecture presented in this chapter is intended as a base for which we derive other architectures in the next chapter.

Chapter 4 presents modifications that change some parts of the base architecture proposed in the previous chapter. These modifications are different but have a possibility of improving the performance of the system. Two modifications are presented and since they can be implemented together or separately, a total of three modifed architectures is presented on this chapter.

Chapter 5 performs an evaluation of the architectures proposed. An implementation was made for each architecture and the performance evaluated with an already existing BDD package.

Chapter 6 concludes the dissertation and provides some insights on potential improvements for future work.

2 Background and Related Work

This section provides the bases needed to understand Symbolic Model Checking and SoC-FPGA devices. Section 2.1 talks about how a symbolic Model Checker works and what it means to be Symbolic. Section 2.2 explains what are Temporal Logics, what they can represent and how a Model Check verifies if a given formula is valid. Section 2.3 quickly explains what Binary Decision Diagrams are and why they are useful while section 2.4 explains how are the operations on them implemented. Section 2.5 explains what SoC-FPGA are and gives an overview of their capabilities. Section 2.6 gives an analysis on the approaches already developed to accelerate BDD operations.

2.1 Model Checker

A Model Checker is a verification tool designed to automatically verify certain properties of concurrent finite-state systems. The concept was developed separately in the 1980s by Clarke and Emerson and by Queille and Sifakis [14, 32]. At the time, the most common formal method used involved hand constructing proofs using a Floyd-Hoare style logic, which was a slow and tedious process [13]. Model Checkers work by exhaustively exploring all the possibles states of a system, which is guarantee to terminate since the system is finite.

The user only needs to provide a model, which describes the initial conditions of the system and how it evolves, and specifications, which encodes the properties the user wishes to check and the model checker will verify whether the model satisfies the specifications or not.

```
1 MODULE inverter(input)
2   VAR     output : boolean;
3   ASSIGN  init(output) := 0;
4           next(output) := !input;
5   FAIRNESS    running;
6 MODULE main
7   VAR     gate1 : process inverter(gate3.output);
8           gate2 : process inverter(gate1.output);
9           gate3 : process inverter(gate2.output);
```

Listing 1. A asynchronous system specified in SMV language.

The model is usually specified using a domain specific language. In Listing 1, a simple model written in the SMV language [25] is presented. Lines 1-4 declares a boolean inverter as a module that can be instantiated afterwards. Line 3 defines the initial value for the output while line 4 defines the next value, after a transition, which in this case it is the negation of the input. Lines 6-9 specify the main model, which corresponds to 3 inverters connected in a loop. The inverters are instantiated as parallel processes by using the keyword *process* in lines 7-9. Parallel processes follow an asynchronous model of concurrency where at each step, one is nondeterministically chosen and the assignments statements in its declaration executed. The statement in line 5 represents a fairness constraint, which basically guarantees that a inverter will get executed infinitely often, and it is needed because since the processes are selected nondeterministically, it is possible that one of the processes never gets selected.

The specifications are specified using a temporal logic (see Section 2.2) written in a domain specific language. As an example two specifications are shown in Listing 2. The first one states that ”it is possible to reach a state where gate1 output differs from gate3 output” while the second one that ”gate1 output will always be equal to gate3 output”.

```
1  SPEC (EF gate1.output = !gate3.output)
2  SPEC (AG gate1.output = gate3.output)
```

Listing 2. Two specifications written in SMV language for the model in 1

One of the most important properties of Model Checking is the ability to generate a counterexample when the specification is not satisfied. An example is provided in Fig. 1 which shows that the first specification in Listing 2 is `true`, while the second is `false`. Afterwards, an execution trace of the model, showcasing the state values and the inputs (which in this case corresponds to which process is selected), proves that the specification "gate1 output will always be equal to gate3 output" is false.

```
-- specification EF gate1.output = (!gate3.output) is true
-- specification AG gate1.output = gate3.output is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  gate1.output = 0
  gate2.output = 0
  gate3.output = 0
-> Input: 1.2 <-
  _process_selector_ = gate1
-> State: 1.2 <-
  gate1.output = 1
```

Fig. 1. Output from executing the NuSMV [11] Model Checker on the model and on the specifications listed in 1 and 2, respectively.

Model Checkers come in two varieties: explicit and symbolic. Symbolic Model Checkers represent sets of states by encoding the characteristic function of sets [25]. A state in a symbolic model checker is an assignment x of boolean variables and a set S is represented by a function f such that:

$$f(x) := \begin{cases} 0 \text{ if } x \notin S, \\ 1 \text{ if } x \in S. \end{cases}$$

Variables other than booleans, like integers and floating point numbers, can be encoded by using multiple boolean variables.

The transitions are calculated for all the states in a set, at the same time, instead of individually. This is accomplished by also representing the transition relation as a boolean relation $R(x, x')$, using the same encoding used above, where x are the variables of the state and x' are the variables of the next state [8]. The transitions can then be computed for every state in the set by calculating:

$$f(x') = \exists x f(x) \wedge R(x, x') \tag{1}$$

This operation is usually called *relational product* and it is one of the important operations in symbolic model checking [6] since this is the operation that traverses the state space.

The resulting function of the Relational Product is dependent on the next state variables. A variable substitution is perform to obtain the function using current state variables:

$$f(x) = f(x')\{x' -> x\} \tag{2}$$

2.2 Temporal logic

The main aspect that separates Model Checking from other verification techniques is its use of Temporal Logic to formulate specifications. Temporal logic is a logic system designed to reason about time without explicitly representing it. *Temporal operators* are used to specify statements like ”This will never happen” or ”This will eventually happen”.

Table 1. The temporal operators of CTL formulas. The operators must always be grouped in two, a path quantifier (A or E) followed by a temporal operator (X,G,F or U).

A Φ	For all	Along all paths
E Φ	Exists	Along at least one path
X ϕ	Next	ϕ has to hold in the next state
G ϕ	Globally	ϕ has to hold in the entire path
F ϕ	Finally	ϕ has to hold eventually
ϕ U δ	Until	ϕ has to hold until δ holds. δ has to hold eventually

There exists several temporal logics but the two most used in model checking are Computation Tree Logic (CTL) [12] and Linear Temporal Logic (LTL) [15]. CTL is whats known as a branching time logic, which means that it models time as a tree that branches in many paths, every path representing a possible future [2]. Table 1 lists the temporal operators in CTL. On the other hand, LTL is a linear time logic which models time as a linear path where at each point there is only one possible sucessor. Even thought LTL seems like a subset of CTL, they are incomparable in terms of expressiveness: there exists statements in LTL that CTL cannot express and vice-versa. Typically a Model Checker will support at least one of theses logics (or the CTL* logic which is a superset of LTL and CTL).

The syntax of CTL formulas is given by:

$$\phi, \delta ::= p | \neg\phi | \phi \wedge \delta | \phi \vee \delta$$
$$EX\phi | AX\phi | EF\phi | AF\phi | EG\phi | AG\phi | E[\phi U \delta] | A[\phi U \delta]$$

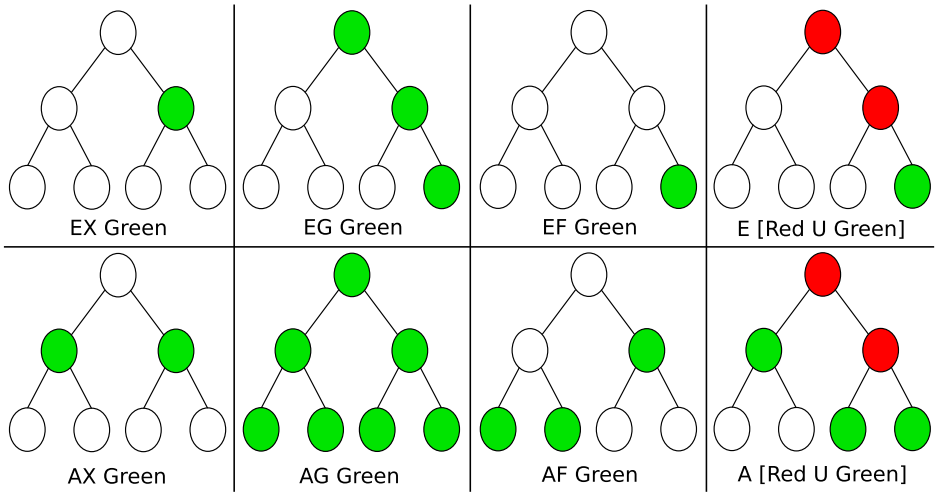

Fig. 2. Representation of the states that satisfy a given temporal operator

One property of the CTL operators that is commonly used in model checkers is the fact that some operators can be implemented in terms of others. An example of such equivalence is:

$$AX\phi = \neg EX\neg\phi. \tag{3}$$

In fact, the set of temporal operators $\{\neg,\wedge,EG,EU,EX\}$ is a minimal set, capable of representing all the formulas in the CTL language. A CTL based model checker only needs to implement these three operators in order to function.

Checking if a formula is satisfied consists in computing the set of states that satisfy the formula and then checking if the initial state of the system is one of those states. The simplest temporal operator to compute is EX. First, ϕ is encoded using the characteristic function encoding presented in the previous section. Second, the Relational Product (1) is used to compute all the parent states of ϕ. Obviously one of the next states of each parent of ϕ is ϕ itself, and so this set of parents is the result of computing $EX\ \phi$.

The other temporal operators, EG and EU, are usually computed based on the following *expansion laws*:

$$\begin{aligned} EG\ \phi &= \phi \wedge EX(EG\ \phi) & EG_0 &= \texttt{true} \\ E[\phi\ U\ \delta] &= \delta \vee (\phi \wedge EX(E[\phi\ U\ \delta])) & EU_0 &= \texttt{false} \end{aligned} \tag{4}$$

which, using EG as an example, leads to the following expansion:

$$\begin{aligned} EG_0 &= \texttt{true} \\ EG_1 &= \phi \wedge EX\ EG_0 \\ EG_N &= \phi \wedge EX\ EG_{N-1} \end{aligned}$$

Due to the repeated use of conjugation, the number of states that satisfy ϕ must decrease until it eventually reaches a fixpoint ($EG_N = EG_{N+1}$) at which point the resulting set satisfies $EG\ \phi$. The same reasoning applies to the EU case.

To perform these operations in a efficient manner, a fast and low memory usage data structure is needed to represent the boolean functions. The first Symbolic Model Checker used a data structure called Ordered Binary Decision Diagrams (OBDD) for this reason [25].

2.3 Binary Decision Diagrams

BDD are a tree-like data structure used to represent boolean functions. They are a restriction of Decision Diagrams (DD) where there exists two types of nodes, terminals and non-terminals such that:

1. Every non-terminal node is associated to a variable x and has 2 children, usually referred as low and high, one associated to the assignment $x = 0$ (0-edge) and $x = 1$ (1-edge), respectively.
2. There are only 2 possible terminal nodes, the constant 0 and the constant 1.

There is also two independent extensions to BDD that are commonly used:

1. Ordered: The variables are ordered such that, for every path possible, the variable of a node is always smaller than the variables of its children.
2. Reduced: Each node represents a distinct boolean formula, i.e redundant nodes, which have the same variable and children, do not exist. Ultimately, this means that nodes are shared between BDD and they can have multiple parent nodes.

A BDD that implements these two extensions at the same time is called Reduced Ordered Binary Decision Diagram (ROBDD) and is the preferred data structure used by Model Checkers to represent boolean functions. This is because, compared to plain BDD, ROBDD reduces the amount of memory needed, due to the node sharing, and it is a canonical representation of the function it represents. The canonical form property is used to simplify the process of comparing two ROBDD (only need to compare the roots) and checking if a given formula is satisfiable (the ROBDD of a non satisfiable formula equals the terminal 0).

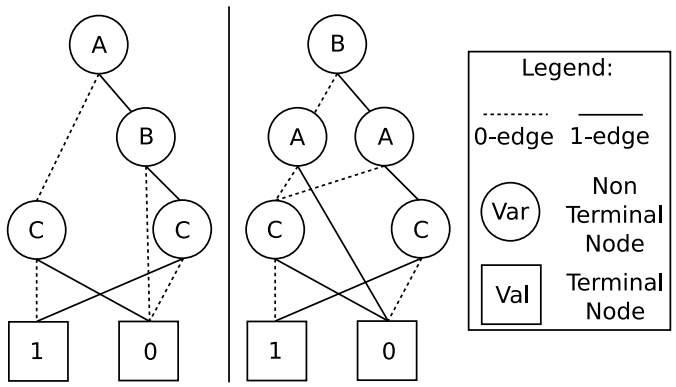

Fig. 3. Example of two BDD (more specifically two ROBDD). The left uses the order $A < B < C$ and the right uses $B < A < C$ but they both represent the function $f(A, B, C) = ABC + \overline{ABC} + \overline{A}B\overline{C}$.

In Fig. 3 an example of two BDDs is presented. One can calculate the value of the function represented by following the 0-edge or the 1-edge depending on the value of the variable associated. For example, the assignment $\{A = 1, B = 0, C = 1\}$ on the left BDD leads to the path $A \Rightarrow B \Rightarrow 0$, meaning that $f(1, 0, 1) = 0$.

An important aspect also present in Fig. 3 is the fact that the order of the variables influences the amount of nodes needed to represent the same function. The ROBDD on the right needs one more node than the one on the left. This disparity between node counts for different orders can grow rapidly for bigger functions. As an example, for a 8-bit comparator (16 variables), there exists a order which needs 26 nodes and another that needs 767.

Selecting an order that minimizes the number of nodes is not an easy task, as the problem is known to be NP-complete. Static methods exist that try to guess a good order based on the model provided [34, 37]. Another method commonly used, called Dynamic Reordering, changes the order of the variables after the BDD is constructed. There exists many heuristics for Dynamic Reordering but they are all based on the same operation: swapping adjacent variables (see Section 2.4. Dynamic Reordering usually happens when the Model Checker runs low on memory or when asked by the user.

A common optimization used when implementing BDD is called *Complement edges*. A complement edge is a edge that negates the BDD it points to. It usually takes one bit to implement (tagged pointers are commonly used[5]) and it leads to less memory used, since sub-trees of negated formulas can now be shared, but more importantly, it makes it so the negation of an entire BDD can be implemented in constant time [5]. Fig. 4 shows the use of complement edges to represent the same function as the one in Fig. 3.

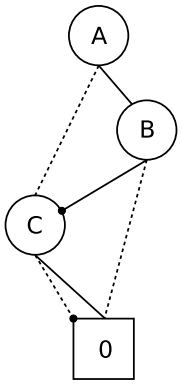

Fig. 4. An example of complement edges (the edges that end in a circle) used on the left BDD of Fig. 3.

2.4 Operations on BDD

Using BDD to encode boolean functions, the *Relational Product* (1) the temporal operators equivalences (3) and expansions (4), presented in Section 2.1 and Section 2.2, can then be computed by performing Negations, Conjugations, Disjunctions and Existential Quantification between BDD.

As seen in the previous section, the negation of a BDD can be accomplished by using complement edges. Complement edges do not interfere with other BDD operations: all the operations on BDD shown can be implemented taking complement edges into account [35, 38].

```
1  BDD BddAnd(f,g : BDD):
2  // Start of Apply
3  // Simple cases
4      if( f == ZERO | g == ZERO | f == Negate(g)):
5          return ZERO
6      if( f == g | g == ONE):
7          return f
8      if( f == ONE):
9          return g
10
11 // Check Operation cache
12     if(<f,g,And> in cache):
13         return cache[<f,g,And>]
14
15 // Compute top variable (the smallest variable of both operands)
16     int varF = GetVariable(f)
17     int varG = GetVariable(g)
18     int top = min(varF,varG)
19
20 // Fetch node data if variable equal to top
21     Bdd fv,fnv,gv,gnv
22     if(varF == top):
23         fv = GetNode(f).Then
24         fnv = GetNode(f).Else
25     else:
26         fv = fnv = f
27     if(varG == top):
28         gv = GetNode(g).Then
29         gnv = GetNode(g).Else
```

```
30      else:
31          gv = gnv = g
32  // End of Apply
33  // Recursively compute child values
34      Bdd t = BddAnd(fv,gv)
35      Bdd e = BddAnd(fnv,gnv)
36
37  // Reduce
38      Bdd res
39      if(t == e):
40          res = t
41      else:
42          res = FindOrInsert(e,t,top)
43
44      InsertIntoCache(<f,g,And>,res)
45      return res
```

Listing 3. Pseudocode for the And BDD operation.

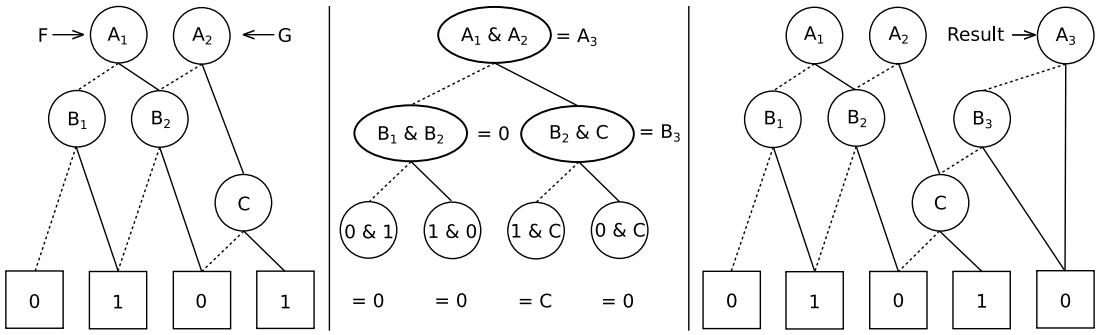

Fig. 5. Two BDD (more specifically two ROBDD) on the left; a look at the stack and the result of each call on the middle; the resulting BDD on the right (the 0 and 1 terminal nodes are replicated to simplify graphics)

The And operation between BDDs is shown in Listing 3 and an example of the operation is shown in Fig. 5. The implementation shown is divided into two phases: Apply and Reduce. The Apply phase checks for simple cases, if the Operation has already been computed and was stored in a cache, computes the top variable and fetches the nodes. The Operation Cache uses an hashing function over the operands as well as a tag value associated to each operation. The Operation Cache thus can store the result of different operations at the same time. After the Apply phase, the algorithm recursively computes the value of the child operations before performing the Reduce phase.

The Reduce phase computes the result of the Operation. If the result of both child operations are equal, then that value is the result. Otherwise the FindOrInsert function is called to insert a node with the values computed and that node becomes the result. Before returning the result, it is inserted into the Operation Cache.

The FindOrInsert function searches the existing nodes to find if a given node with the same fields already exists, in which case it is returned as the result. If no such node exists, a new one is create. This functions is responsible for guaranteeing the Reduced property mentioned in Section 2.3. In order to speed up this operation, every standard BDD package stores the nodes in a hashtable, called the UniqueTable, to improve the search operation.

```
1  // Simple cases
```

```
2  if(f == ZERO | f == ONE | top > lastVariableToQuantify):
3    return f
4  // [Rest of Apply]
5  if(quantify[top]): // If the variable is to be quantified
6    BDD t = BddExist(fv,quantify)
7    if(t == ONE): // Existential Optimization
8      return ONE
9
10   BDD e = BddExist(fnv,quantify)
11
12   return Negate(BddAnd(Negate(t),Negate(e))) // Equivalent to BddOr, by using De
         Morgan's laws
13 else:
14 // [Reduce]
```

Listing 4. Pseudocode for the Exist changes compared to the And pseudocode

The Existential Quantification operation follows a similar structure to the And operation. It takes as input a BDD node and a bitfield whose N bit indicates that the N variable is to be quantified. The changes from the And algorithm to the Exist algorithm are shown in Listing 4. Other than the different simple cases, the change from the And algorithm to the Exist algorithm is present in the case that a variable is to be quantified, as shown in line 6. Two differences are present:

1. The result of the quantification is the Or of the values computed in the child requests. The Or implementation can be computed by using the And and Negate operations, according to the De Morgan's rules.
2. Since the result is computed by a Or operation, a optimization exists, and is present on line 7, that saves the computation of the second child request by checking if the result of the first request was One. In this case the Or result would be One regardless of the second result and as such it can return earlier. We call this the Existential Optimization.

The And and Exist algorithms together can be used to implement the Relational Product operation. Most common BDD packages also offer a specialized algorithm that combines both operations into one: It performs both a And and a Exist operation at the same time. This algorithm is more efficient since it reduces the amount of nodes created in temporary BDDs.

```
1  // Simple cases
2  if(f == ZERO | g == ZERO | f == Negate(g)):
3    return ZERO
4  if(f == ONE & g == ONE):
5    return ONE
6  // Relational Product simplifications
7  if(top > lastVariableToQuantify):
8    return BddAnd(f,g)
9  if(f == ONE | f == g):
10   return BddExist(g,quantify)
11 if(g == ONE):
12   return BddExist(f,quantify)
13 // [Rest of Apply]
14 BDD res,e,t
15 if(quantify[top]):
16   t = BddRelationalProduct(fv,gv,quantify)
```

```
17
18    if(t == ONE | t == fnv | t == gnv): // Existential optimization
19      return t
20
21    // Relational Optimization
22    if(t == Negate(fnv)):
23      e = BddExist(gnv,quantify)
24    else if(t == Negate(gnv)):
25      e = BddExist(fnv,quantify)
26    else:
27      e = BddRelationalProduct(fnv,gnv,quantify)
28
29    res = Negate(BddAnd(Negate(e),Negate(t)))
30  else:
31  // [Rest of Reduce]
```

Listing 5. Pseudocode for the Relational Product.

The pseudocode for the Relational Product operation is shown in Listing 5. The pseudocode for the complete version, including Negation, is shown in Appendix A. Compared to the And and the Exist operations, the Relational Product operation introduces three new optimizations: Relational Product simplifications in line 6, Existential optimizations on line 17 and Relational Optimizations in line 20. The Relational Product simplifications check if the current operation can be performed by using the simpler And or Exist operations.

The Existential and the Relational optimizations in line 17 and 20 make use of a property of the Relational Product. If the result of the first child request is equal to one of the operands to the second request:

$$(f_{x=0} \wedge g_{x=0}) = t \; (where \; t = f_{x=1} \; or \; t = g_{x=1})$$

then we can perform the following reduction, by noticing the fact that if the function when one of the variables takes a given value is equal to itself, then it implies that it does not depend on that variable:

$$(f_{x=0} \wedge g_{x=0}) \vee (f_{x=1} \wedge g_{x=1}) = t \vee (f_{x=1} \wedge g_{x=1}) = t$$

without having to compute the second case. This optimization is an important time save, since cutting one branch of computation can potently save performing thousands of operations. The Relational Optimizations uses the fact that:

$$t \wedge (!t \vee g) = t \wedge g$$

to further simplify the operation that computes the second request, if possible. Note that the Relational Optimization is not related to the Relational Product simplification: the comparison in the Relational Product is made with the result of the first Operation, while the simplification compares the arguments of the request themselves.

The Substitute operation performs variable substitution in a BDD. It takes as input the root of a BDD and an array that maps the current variable to its substitution.

A general Substitute implementation is capable of substituting any variable with another, but for Model Checking, a general implementation is not needed. The use of susbstitution in Model Checking is

to perform the substitution of next state variables with current state variables. By encoding these pairs of next state and current state variables adjacently, we can use a simpler implementation of the Substitute Operation.

```
1  BDD Substitution(F : BDD, NextVar : int[]){
2     if(Constant(F))
3        return F;
4
5     BDD Then = Substitution(F.Then, Substitute);
6     BDD Else = Substitution(F.Else, Substitute);
7
8     int newVar = NextVar[F.Var];
9     Bdd result = FindOrInsert(newVar, Then, Else);
10
11    return result;
12 }
```

Listing 6. Pseudocode for the Simple Substitution operation.

The pseudocode for the Simple Substitution operation is shown in Listing 6. This implementation is only capable of performing substitution if for every variable x, either NextVar[x] = x or NextVar[x] = x + 1. When building the BDDs, the Model Checker can assign Next and Current state variables such that Current = Next + 1, thus allowing the use of the Simple Substitution operation to perform variable Substitutions.

2.5 SoC-FPGA

A SoC-FPGA is a board that integrates a Field-Programmable Gate Array (FPGA) and a Hard Processing System (HPS) into a single device. The FPGA and the HPS divide the system into two regions, where each region has accessed to different components. The regions are connected and can transfer data between themselves.

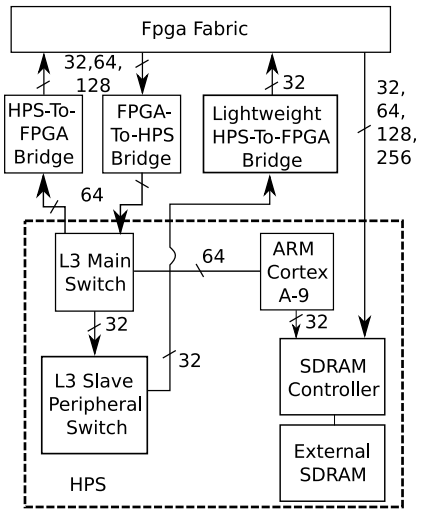

Fig. 6. A simplified diagram of the Cyclone V SoC HPS and system integration.[5]

[5] The full diagram can be found in the Cyclone V Hard Processor System Technical Reference Manual, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_54001.pdf, Fig.2-2 in page 2-4.

Fig. 6 showns a simplified diagram of the connection between the FPGA and the HPS systems in the De1-SoC board. The arrows represent a Master/Slave relationship between components: The component that connects to another component with an arrow starts the memory transfers. The connection between FPGA and HPS is performed by 3 bridges and a port that connects the FPGA directly to the HPS SDRAM Controller. Two bridges are mastered by the HPS side, while the FPGA is the master of the remaining bridge, the FPGA-To-HPS bridge and the SDRAM port. The HPS-To-FPGA and the FPGA-To-HPS are intended for high throughput transfers. They can be programmed to transfer 32, 64 or 128 bits at a time. The Lightweight bridge offers a lower throughput but also a lower latency.

The FPGA Fabric can utilize both the FPGA-To-HPS bridge or the SDRAM port to perform data transfers. The difference between these methods are related to cache coherence. The FPGA-To-HPS bridge passes through the ARM Cortex A-9 device, which contains a Accelerator Coherency Port (ACP). Accesses to memory performed using the FPGA-To-HPS bridge are thus cache coherent with the CPU. If the FPGA performs accesses using the SDRAM port, then it accesses the SDRAM directly and as such it bypasses the L3 Main Switch and the CPU cache. The SDRAM port has a higher throughput but memory must be carefully managed to avoid reading invalid data.

A FPGA is a integrated circuit that can be programmed to implement complex digital circuits. It is composed by logic blocks, that can be programmed to implement logic functions, memory, to temporarily store the results of a computation, and a configurable interconnect to connect the logic blocks.

As an example, the CycloneV FPGA uses Adaptive Logic Module (ALM) as logic blocks. A ALM is composed by a 8 input Look-up Table (LUT) block whose output can then be connected to a full adder, a register or as the output of the ALM block which connects to another ALM. The LUT are the fully programmable components of the ALM, capable of implementing any logical 8 bit function.

FPGAs also contain memory blocks, with a storage capability greater than registers. A memory block is addressable, meaning that it associates memory to an address. A block contains read and write ports that can be configured to output memory in various sizes. The CycloneV FPGA uses M10K memory blocks, capable of storing 10000 bits, and has two read and two write ports. The M10K memory block has registered inputs and thus a memory read (or write) takes a minimum of two cycles: The first cycle stores the address and the input; The second cycle reads the data stored in the address.

The process of programming a FPGA starts by writing a specification that encodes logic circuits. This is usually done by using a Hardware Description Language (HDL), either Verilog or VHDL. The process of compilation of this language produces a bitstream, that is capable of programming the programmable components of the FPGA.

The way HW/SW systems are created for FPGA involve the creation of Modules, called IPs. The IPs are then connected between themselves. The final system is then generated and compiled into a bistream that programs the FPGA.

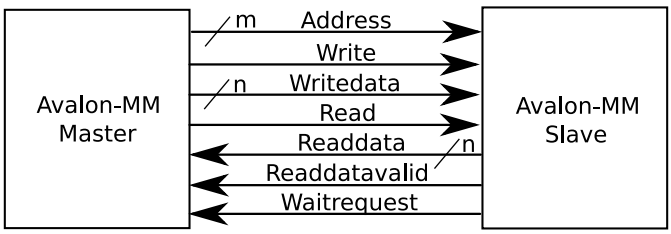

Fig. 7. Avalon Memory Mapped Interface. The Address and Data size (equal for both Readdata and Writedata) are user defined.

These modules communicate by means of standardized interfaces. Two common interfaces used by Altera IPs are the Avalon Memory Mapped (Avalon-MM) and the Avalon Streaming (Avalon-ST) interfaces. The Avalon-MM Interface, represented in Fig. 7, provides an abstraction that is similar to a memory access: The Master asserts either the Write or Read signal to indicate the type of operation and sets the Address to select the register. The Slave either performs the write operation or, in case of a read operation, provides the data using the Readdata signal, as well as the Readdatavalid signal to indicate that the Readdata value is valid. If the Slave is incapable of performing the data transfer in the current cycle, it can delay it by asserting the Waitrequest signal. This signal indicates to the Master that it needs to keep the signals constant for the next cycle, which effectively stalls the transfer for each cycle the Waitrequest signal is asserted.

The Avalon-MM interface allows the pipelining of Read operations by allowing the Master to assert multiple reads before having to wait for their results. The Slave uses the Readdatavalid signal to indicate when the Readdata value is valid and returns Readdata in the same order the Master posted the Reads.

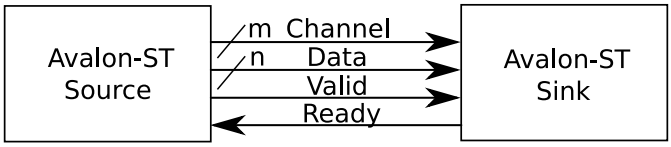

Fig. 8. Avalon Streaming Interface. The Channel and Data signals size are user defined.

The Avalon-ST provides a Stream like interface. It is represented in Fig. 8. The Source asserts the Valid signal to indicate that the Data is valid. The Sink asserts the Ready signal to indicate it is capable of receiving new data. When both signals are asserted in the same cycle a data transfer occurs. An additional signal, called Channel, allows to identify the Data and to logically separate different Data.

In a SoC-FPGA, the HPS system is provided as an IP itself, that we can connect to our custom made IPs. The software can then communicate with our IP by accessing memory, which in the FPGA will lead to the use of the Memory Mapped interface. So, even thought the communication between HPS and FPGA fabric is performed through bridges, from the FPGA perspective, the HPS communicates through the use of a Memory Mapped Master Interface.

In SoC-FPGA, it is common for both systems to possess their own memory. The De1-SoC device, for example, has a 1GB SDRAM memory reserved for use by the HPS system, and a 64MB SDRAM memory for use by the FPGA. The device also provides a special interface, that allows the direct access from FPGA to the HPS memory. By using a Direct Memory Access (DMA), it is possible to transfer large quantities of memory from FPGA to HPS and vice versa.

Designing hardware circuits to run in a FPGA requires some extra care. This is because we are limited by the amount of logic circuits, memory and routing resources available. Also, the performance of the design depends on the frequency at which we can run the circuit clock. The higher frequency we can run the design the better performance we expect to have (assuming we do not increase cycles needed). For these reasons, circuits expected to run in a FPGA are expected to be heavily pipelined: not only it improves maximum frequency, but it also facilitates fitting the design.

There are two types of on-chip memories in a FPGA: registers and memory blocks. Registers are small and fast. They are used to store the outputs of a MLAB unit if they are configured to do so. A memory block is a bigger and addressable unit of memory. One example of a memory block is the M10K unit, which is capable of storing up to 10240 bits. The M10K block has registered inputs and takes 2 cycles to perform a read operation.

2.6 Related Work

Some works focused in providing efficient implementations of BDD for single core systems. Implementations where divided between depth first approaches [5, 35, 39], which had small memory overhead, and breadth first approaches [1, 28, 36], which had better memory locality.

Hybrid approaches [9, 10] also started to emerge to get the best of both worlds. These works are particularly useful for this project as memory is a limited resource when dealing with SoC-FPGA.

One work implemented a BDD processor in hardware [44]. It only computed binary operations and was depth first which does not scale very well.

But the majority of papers tried to accelerate the computations by exploring parallel BDD operations. They were almost always breadth first approaches, due to the inherent sequential nature of depth first approaches (even though one work [40] managed to parallelize depth first by partitioning the BDD). They were further divided based on whether it was multi-core [16, 17, 18, 23, 24, 33] or distributed computing [4, 19, 20, 26, 27, 30, 31].

Most of the techniques developed in distributed computing are not useful for this work as they focused in reducing the communication overhead associated with distributed systems or in better ways of partitioning BDD. For this project, these techniques mean very little since SoC-FPGA offers high bandwidth connections.

Multi-core works also barely apply to this project since most focused in distribution of work and reducing synchronization (lock-free and concurrent data structures). Again, the techniques developed in these papers are not really useful for this work as synchronization and work distribution problems are very easy to solve when using digital circuits.

A more recent work used a GPU to accelerate the processing of large BDD [42]. This work was not associated with Model Checking but it obtained good results. Coupled with another work, that used vector supermachines [29], it shows that a Single Instruction, Multiple Data (SIMD) architecture might have its merits.

3 Proposed HW/SW Architecture

In this chapter we propose a HW/SW architecture designed to accelerate BDD Based Model Checkers. The hardware system implements a accelerator that performs four BDD operations: And, Exist, Relational Product and Substitution. The software system runs a Model Checker that offloads these operations to the hardware controller.

In the first section we present a overview of this system. We showcase the HW system, composed of several modules, and how it is connected to the SW system.

The second section describes how memory is partitioned between systems and how data transfers are managed. This includes the approach taken to guarantee that both systems have access to coherent data needed. This section also describes one hardware component of our architecture, the Memory Access IP, that performs memory management on the hardware side as well as a Kernel Module Driver that interacts with this component from the HPS side.

In the third section we present the Depth Controller IP. This IP implements the BDD operations using a Depth-first approach.

The final section presents a set of supporting software functions that were developed in order to test the HW/SW system. These functions implement the algorithms in software and are used during the evaluation to compare the performance of the HW/SW system to a standalone SW system.

3.1 HW/SW system overview

The proposed HW/SW system divides the Model Checking process into two halves: the BDD operations that are used to compute the state transitions are implemented in hardware, while the remaining portion of the Model Checker process runs in software. The software side initiates BDD operations by programming the hardware and then waits for the hardware to complete the operation.

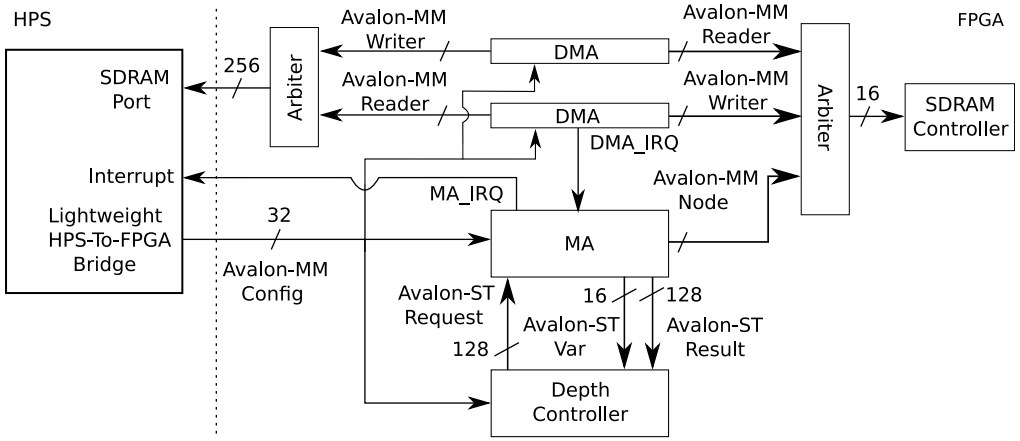

Fig. 9. The schematic of the hardware controller implementation.

The hardware is further divided into modules. A overview of the system is represented in Fig. 9. The SDRAM Controller IP interfaces with a SDRAM that is directly accessible by the FPGA. Two DMAs are used to perform data transfers between the systems RAMs, since each DMA is only capable of transferring data in one direction. The Controller and Memory Access IP are the proposed modules of this dissertation.

The Memory Access IP is the module that manages memory and provides an abstraction over the memory to the Controller. The Controller is the module that performs BDD operations and is decoupled

from direct memory access. When the Controller needs access to data, it sends a request to the Memory Access IP through the use of the Avalon-ST Request interface. The Memory Access IP fetches, or writes, the data and then sends it to the Controller using the two Avalon-ST interfaces: Variable and Result. The specifics of this process are further detailed in section 3.2.2.

Both modules receive configuration data from the HPS through the use of the HPS-to-FPGA Lightweight bridge. The Memory Access IP receives data necessary to coordinate memory transfers and the Controller receives data related to the BDD operation to perform. It is through this interface that the HPS can initiate a BDD operation, by sending the data for the first request to the Controller, and read the final result.

The DMA units connect the HPS SDRAM with the FPGA SDRAM. The SDRAM port is used and configured to the maximum transfer size possible of 256 bits, to provide the maximum throughput possible.

The interruption lines are connected in order to provide the Memory Access IP the capability to request memory transfers. As seen in Fig. 9, the DMA modules can only be configured by the HPS. Thus, in order for the Memory Access IP to perform memory transfers, it must be able to perform a request to the HPS. This is accomplished by the MA-IRQ line. The DMA-IRQ line is connected to the Memory Acccess IP unit to indicate when the memory transfer finishes. The HPS unit does not need to know when the transfer from HPS-To-FPGA finishes. A full description of the memory transfer process is further detailed in section 3.2.

3.2 HW/SW memory overview

The proposed HW/SW system uses both RAMs. From the point of view of the HW system, the HPS RAM is used as a secondary storage unit while the FPGA RAM is used as the primary storage unit.

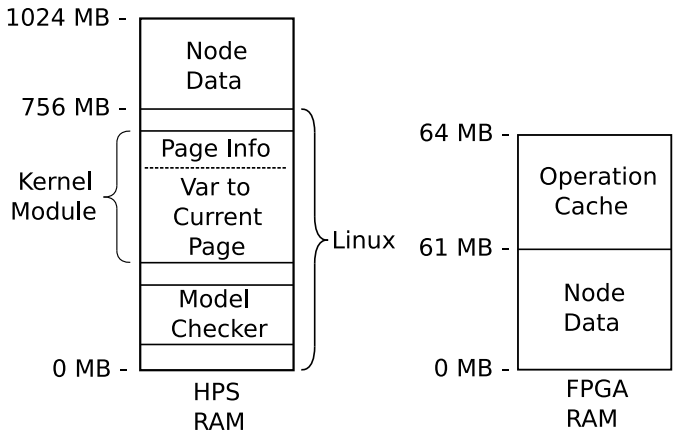

Fig. 10. The Memory Map for the HPS and the FPGA systems. The first 756 MB of the HPS memory is managed by the Linux OS. The Kernel Module further allocates memory inside the Linux Kernel to store Page Related Information. The Model Checker executes in user space.

Fig. 10 shows the memory map of our architecture. The FPGA RAM is used to store node data and a Operation Cache. The HPS memory is divided into two zones: memory used by the OS and memory used by our architecture. This memory is allocated by using the Linux Kernel arguments *mem=768M*, *memmap=768M$256M* and *vmalloc=300M*. These arguments instruct the Linux Kernel to not use that region of physical memory. A Kernel Module is then used to provided access to that region of memory to userland programs running in the OS. More details are provided in section 3.2.3.

The Page info array is used to implement a paging scheme that divides the node memory into statically sized arrays of fixed size called Pages. We use the standard size of modern OS of 4Kb as the Page size. Accessing a node is performed by fetching the Page info the node belongs to and then using the page address plus the offset to access the node. The Page index and the offset are grouped together in a data structure called BDDIndex.

In our architecture we do not store the variables of the Nodes inside them. Variables are associated to Pages and the variable of a Node is determined by the Page it is stored in.

Memory transfers are performed by a cooperation between the Memory Access IP and the HPS. During the execution of a BDD operation, the Controller sends memory requests to the Memory Access, which contain a BDDIndex. The Memory Access first verifies a cache if it contains information associated to the Page encoded in the BDDIndex. If it does, it proceeds to perform the memory operation, read or write, and returns the result to the Controller.

When the page information is not stored in the cache then it means one of two things: The page is not currently present in the FPGA SDRAM or the page is present but the Memory Access page info cache does not have the information about that Page. In either the cases, the Memory Access makes a request to the HPS.

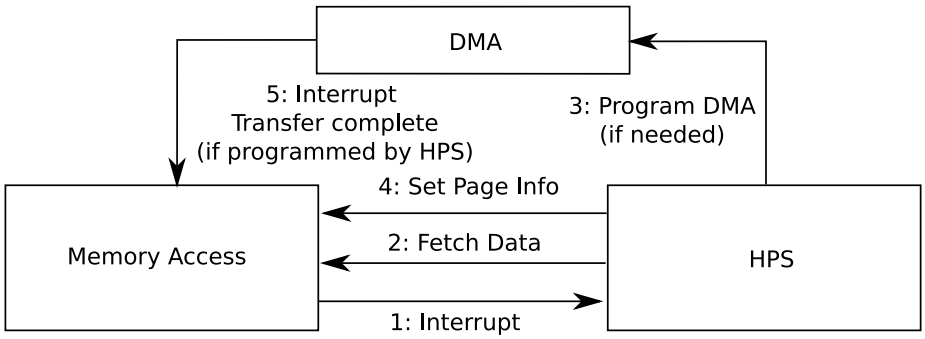

Fig. 11. The steps involved in a Memory Access Request to the HPS.

This whole process is illustrated in Fig. 11. The Memory Access asserts the interrupt line, indicating that it needs to access a page that it currently does not have stored in its cache. A interrupt handling routine is triggered in the HPS. This routine is responsible for handling the page transfer. It fetches the data regarding which page is needed by using the Avalon-MM Config interface. It then accesses the Page info array to get information about the page and sends in to the Memory Access by using the Avalon-MM interface again.

In the cases where the Page is not present in the FPGA RAM, the routine configures the DMA to perform the memory transfer and when sending the information back to the Memory Access, it indicates the needed to wait for the end of the DMA operation. This ensures that the Memory Access does not perform any memory operation before the Page has finished being transfered.

A special case in the process is when the Memory Access needs to insert a new Node. In this case, the Memory Access might not have the information for a Page that is both free and associated to that variable. To handle this case, the request sent from the Memory Access to the HPS is capable of requesting a Page that has free space and is associated to a variable. In this case, instead of sending a page, the Memory Access sends the variable to the HPS, as well as one bit set at one to indicate that it is requesting a page associated to a variable, and not to a BDDIndex. The interruption routine either sends the info to a page that is free and associated to the variable needed, or it associates a new page to the variable needed and uses it.

3.2.1 Data Structures

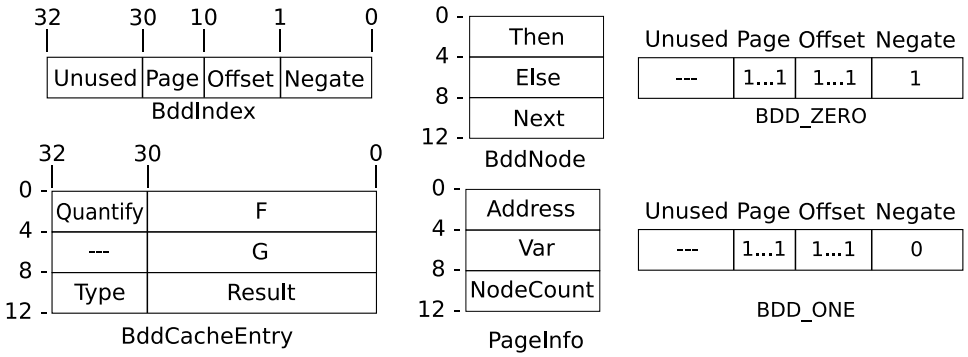

Fig. 12. Memory layout of BDDIndex, BDDNode and BDDCacheEntry. Also memory layout for BDD_ZERO and BDD_ONE.

The simple data structures used by our architecture are shown in Fig. 12. A BDDNode uses three BDDIndexes: two for the childs and another for the Unique Table chain. It occupies 12 bytes meaning that a Page is capable of storing up to 341 BDDNodes. Thus, the BDDIndex needs 9 bits to encode the offset, plus one bit to encode complement edges and 20 bits to encode the Page index. This layout is capable of accessing the full range of a 32 bit system and two bits are left unused. These free bits are used in the BDDCacheEntry, which is the data structure that stores the result of a BDD operation in the Cache. The structure is composed by three BDDIndexes, which store the arguments and the result. Since we only handle 4 different BDD operations, we can store the operation in two bits. One other bit is used to store whether variable quantification was performed.

3.2.2 Memory Access IP

The Memory Access IP is the Module that abstracts the Controller from having to perform direct memory access. Instead, it provides an interface that is capable of fetching the BDDNodes and the variables that belong to a BDDIndex. It is also capable of fetching the BDDCacheEntry that belongs to a given hash.

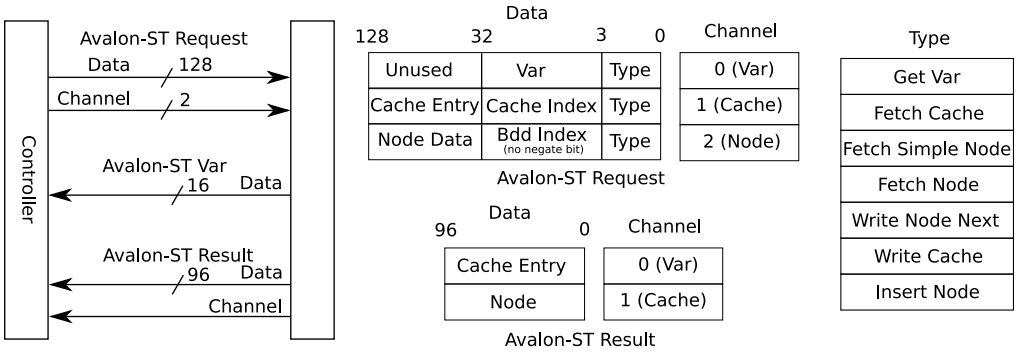

Fig. 13. The Memory Access IP Interface. The Controller makes memory requests using the Avalon-ST Request interface. The channel associates the Request to one of three types: Var, Cache or Node.

The interface of the Memory Access IP is shown in Fig. 13. The memory request sent by the Controller specifies a type of operation, an ID and a Data field whose contents depend on the type of

operation. The ID field is used when sending the results back to the Controller, to match the result with the request that originated it. The operations that produce a result are the Fetch, GetVar and Insert operations. This result is then sent through the use of the Avalon-ST interface along with the ID that lets the Controller associate the result with the operation. The GetVar operation does not need to perform SDRAM access and therefore is capable of sending the result sooner. For this purpose the Avalon-ST Var interface is used. The Write* operations do not return results.

The type of data needed by each operation is as follows: the Cache operations need a cache index, the InsertNode operation needs a variable and the remaining operations need a BDDIndex.

The InsertNode operation is the only operation that simultaneously writes and produces a result. The result of this operation is the BDDIndex where the new node was inserted.

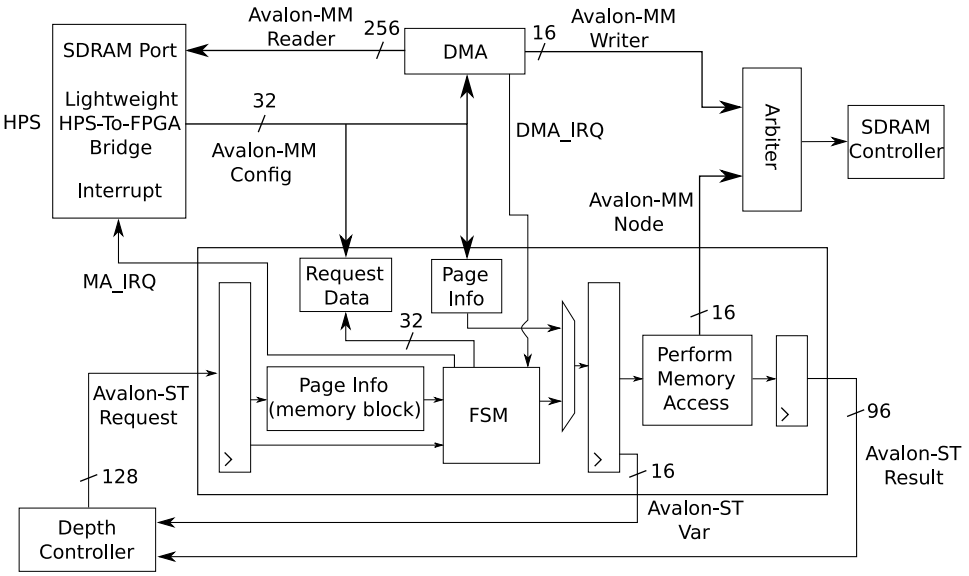

Fig. 14. The Memory Access IP and the connections to the HPS and the DMA. The FSM coordinates the transfer in case the page is not stored in the Page Info memory block.

Fig. 14 represents the Memory Access IP. The Depth Controller sends a Request using the Avalon-ST Request. The first stage in processing the Request is to fetch the Page info, in case it is a Var or a Node Request. In the cases that the page info is not currently stored, a FSM coordinates the making of a request to the HPS to provide the info and perform a page transfer if needed. In any case, the next stage performs the memory access, unless it is a Var request, in which case the variable that was fetched in the previous stage can be immediately sent to the Depth Controller.

The Memory Access tries to utilize the ability to post multiple read requests to the SDRAM Controller. A small FIFO inside the Perform Memory Access allows the stage to send as many memory reads requests as the SDRAM Controller allows before sending any result. This stage also waits for the current amount of bytes to return and/or write, so that it can send the entire data to the Controller. When all the data is read it is stored in a register that immediately sends it to the Depth Controller.

3.2.3 Linux Kernel Module

The Linux Kernel Module performs two functions. It provides to the Model Checker access to the region of physical memory allocated and it coordinates with the Memory Access IP to perform page transfers during a BDD operation.

The Model Checker runs inside a Linux based operative system. In order to provide direct memory access to a region of memory outside of the OS, we must use a Kernel Module, which has access to functions that are not available for user land programs.

Our Kernel Module utilizes the mmap interface to accomplish this. The Model Checker opens our device and then calls the mmap function. Our implementation of mmap utilizes the *remap_pfn_range* function to map the physical region into the Model Checker virtual memory. Inside the Kernel Module, we use the *ioremap_nocache* function to provide access to the Kernel Device.

Both *remap_pfn_range* and *ioremap_nocache* perform a mapping that does not utilize the cache. This reduces performance when accessing the memory but guarantees that the data is coherent.

The second function performed by the Kernel Module is to coordinate page transfers with the Memory Access IP. When the hardware detects the need for a Page transfer, it signals the HPS through the use of interrupts. In a Linux operative system, user land programs cannot handle interruptions. Thus, we implement the interrupting handling in our Kernel Module.

There are two situations where the Memory Access IP raises an interruption. Either it needs access to a specific Page or it needs to insert a Node, situation where we need to provide a Page with free space associated to the variable needed.

In either the situations, the routine accesses the Page Info table, either using the Page index for the first situation, or by making a search for a page with the requested variable.

With the information of the Page needed the final step consists in sending the information to the Memory Access IP using the interfaces provided. If the Page is not currently present in the FPGA SDRAM, the routine also programs a DMA to perform a transfer.

In the case that the FPGA SDRAM does not have free space avaiable, we select one Page to save back into HPS SDRAM to free up space before programming the DMA. The selection process mirrors a FIFO scheme: we keep a counter that is increment every time a Page is transferred back and we select that Page to be freed when needed.

One optimization done when transferring Pages back to HPS SDRAM is to first fetch the number of nodes that Page has. This is accomplished by using the Memory Access IP Config Interface. When programming the DMA, we only need to transfer the amount of nodes actually present in the page, we do not transfer the whole page unless it is full.

3.3 Depth Controller IP

The Depth approach implements the simplest method of performing BDD operations. Each request is processed individually, one at a time. The child requests are processed fully, one at a time, and their results are then used in the Reduce phase.

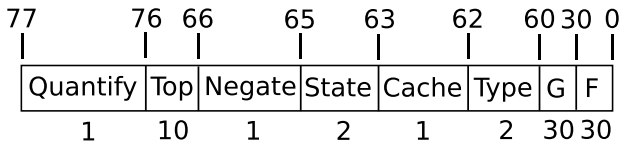

Fig. 15. Depth Request

We implement an Iterative algorithm that performs all the 4 BDD operations. We keep track of the operation and the arguments in a Request data structure as shown in Fig. 15. We keep track of the state of an operation by using two bits, which take on the following values: Apply, Return0, Return1 and Reduce.

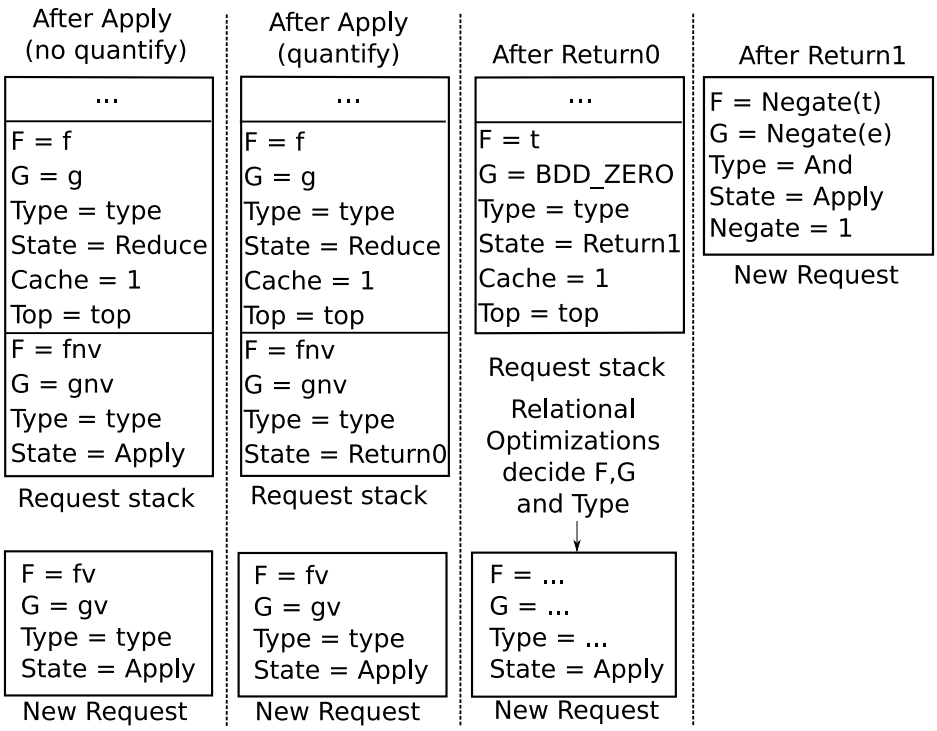

Fig. 16. The Request creation after each iteration. The meaning of the variable names matches their usage in the pseudocode presented for each operation in section 2.4

After performing an Apply operation, new Requests are created according to Fig. 16. Every operation can create at most three new requests. Since one of these Requests is immediately set to be processed, we only need to store two new Requests in the stack. Because we impose a limit on the maximum number of variables, the Request stack is thus bounded in size to be twice the maximum number of variables.

The Request creation process can be divided into two, depending on whether we are performing a quantification or not. In the case we do not perform a quantification, the Requests created are either in the Apply or Reduce state.

The Return0 and Return1 states are used exclusively to perform the quantification and implement the Existential and Relational optimizations mentioned in section.2.4. In a Exist or a RelationalProduct computation that needs to perform quantification of a variable, we calculate each Request one at a time. We use the Return0 state to verify if the first Request has the correct value, if not, we insert the Return1 request and we calculate the other Request. The Return1 state then performs the call to the And operation used at the end of a quantification. In order to save a Request, we embedded in the Request structure the Negate bit that indicates that the result is to be negated before insertion.

3.3.1 Controller

The Controller IP is the module that performs BDD operations. It interfaces with the Memory Access IP to request memory needed, such as the variables, BDDNodes and BDDCacheEntries.

The schematics for the DepthController are shown in Fig. 17. The implementation resembles the implementation of a simple pipelined processor. The first stage keeps track of the current request as well as it stores the Requests and Results in a Lifo. This data is sent to the Apply or Reduce modules depending on the Request state. After the Request is processed, a group of Control Signals performs a writeback by inserting new Requests, new Results or setting a new Request to then be processed. The

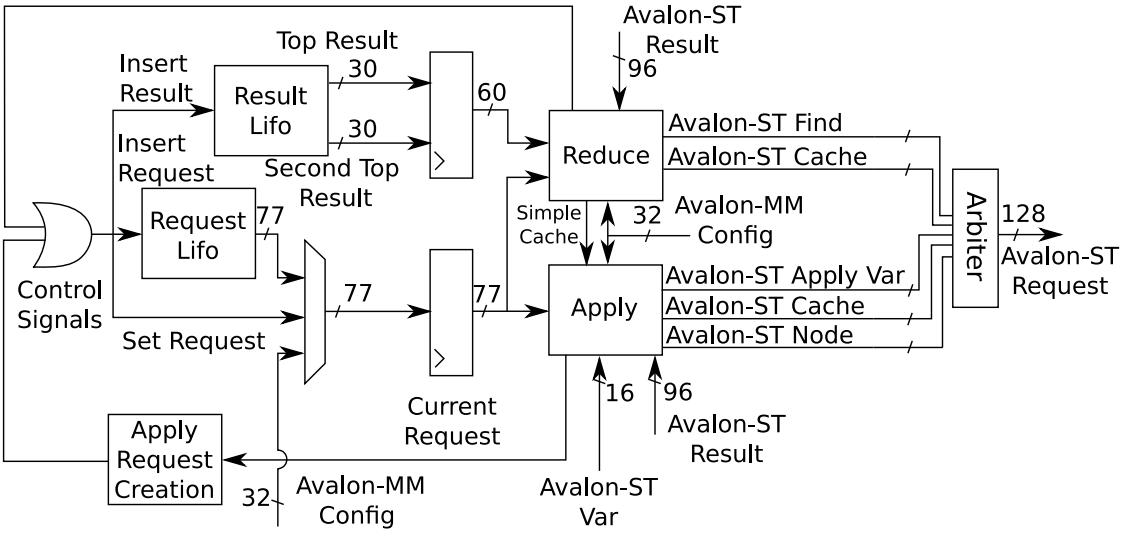

Fig. 17. Depth Controller

Apply Request Creation module sets the necessary control signals to insert Requests as depicted in Fig. 16. The Reduce operation uses the Control Signals to pop and insert results and to pop Requests from the Lifo to be processed in the next iteration.

Data specific to each operation can be configured by using the Avalon-MM Config interface. This corresponds to the quantification data used by the existential operations, the NextVar data used by the Substitution operation and the UniqueTable values used by all the operations.

After configuring the operation, the initial Request is sent using the Avalon-MM Config interface. The request is composed by the two BDDIndex arguments, the second being BDD_ZERO for the operations that only use one argument, and the type. Sending this initial operation starts the Controller. The HPS reads the final result by polling the Controller through the Avalon-MM Config interface until it reads a valid value.

3.3.2 Apply

The Apply module is design to perform the Apply phase of the 4 operations. The Apply phase of these operations tend to possess similar steps, as can be seen in section 2.4 and by having all in one module we can reuse logic between operations.

The schematics for the Apply module is shown in Fig. 18. The input to the module is the two BDDIndexes plus the type of operation. The output is the result of a typical apply phase: if the operation was a simple case or was found in cache, the result is a BDDIndex; Otherwise, the output is the data that is later used to create the child Requests. Since only one Request is being processed at a time, the Simple Case result can be outputted immediately.

The Apply module defines three Avalon-ST interfaces to make memory requests. These interfaces are then connected to the Memory Access IP.

The Avalon-MM Config interface is used to receive configuration data from HPS. This data is sent before starting a BDD operation and corresponds to quantification or NextVar values. This values are stored using on-chip memory.

The Simple cache wires are used to implement a small operations cache inside the Apply module. A Operation Cache needs 96 bits to store one entry and as such the amount of memory needed to implement a Operation Cache using on-chip memory would be prohibited. In our HW/SW system, we

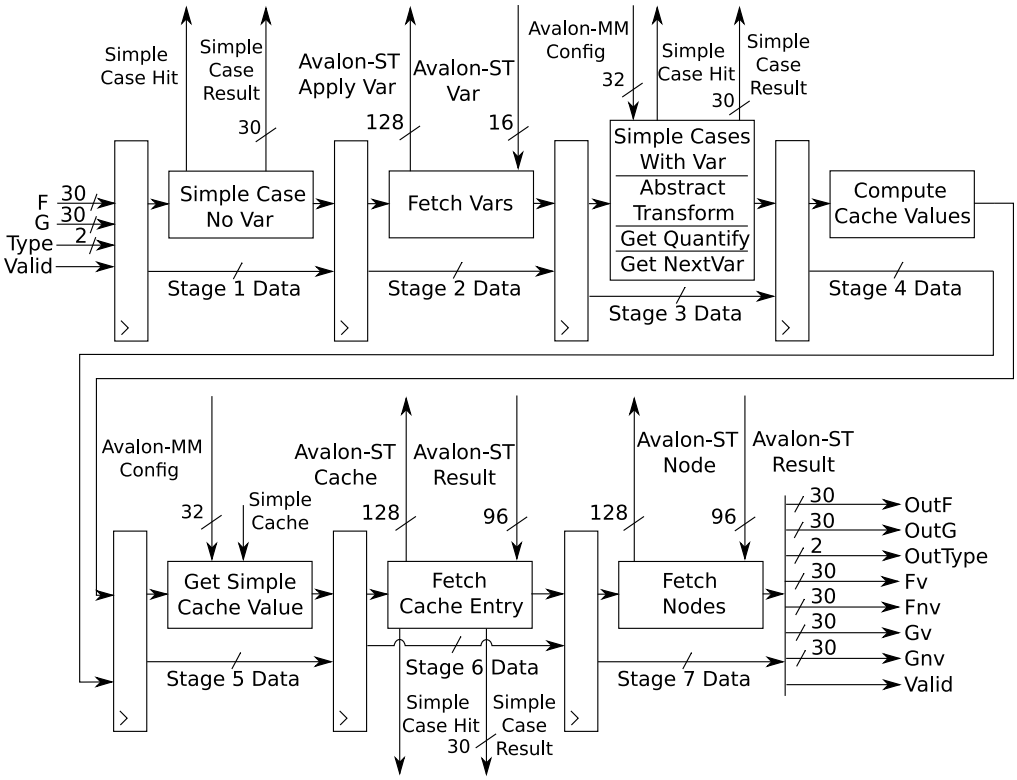

Fig. 18. The Apply Module.

use the SDRAM to store cache entries, but in order to improve caching performance, we implement a smaller cache.

This smaller cache, called a Simple Cache, stores a small hash computed using a different function from the one that calculates the index to store the entry. This value stored is much smaller than the size of a BDDCacheEntry: 2 bits. What this value accomplishes is a small test that rules out the need for a memory access: If the value computed is not equal to the value stored, then it is impossible for the cache entry stored in FPGA RAM to also be equal. Therefore the cache entry does not need to be checked and we save a memory access.

3.3.3 Reduce

The Reduce operation is divided into three different stages depending on the state of the Request.

The Return0 state corresponds to checking the Existential and the Relational Optimizations. In the case that the check fails, the second child request is created. The type and the arguments of this second request depend on the Operation. Section 2.4 explains the reason for the different types and arguments.

The Return1 state collects the results and sets the New Request to compute the Negated And of the Negate operands (effectively computing a Or). This state performs the same operation regardless of whether it is a Exist or a Relational Product operation.

The Reduce state performs the Reduce operation as mentioned in Section 2.4. It verifies if the arguments are equal, at which point it can simple store one of them, and if they are not equal it performs the FindOrInsert operation. In the end the result is stored into the cache, by storing the BDDCacheEntry in SDRAM, and the small hash of the entry is stored in the small cache as previously mentioned.

3.3.4 FindOrInsert

The FindOrInsert module performs the search of a BDDNode by using the UniqueTable and, if not found, inserts the Node.

```
1  BDD FindOrInsert(e,t : BDD, int var):
2      BDD ptr = UniqueTable(<e,t,var>)
3
4      BDD previous = BDD_ZERO
5      while(ptr != BDD_ZERO):
6          BddNode node = GetNode(ptr)
7
8          if(node.Then == t && node.Else == e):
9            return ptr
10
11         if(node.Else > e | node.Else == e & node.Then > t):
12           break;
13
14         previous = ptr;
15         ptr = node.next;
16
17     BDD res = Insert(e,t,var)
18     if(previous == NULL):
19       UniqueTable(<e,t,var>) = res
20     else:
21       previous.next = res
22
23     return res
```

Listing 7. FindOrInsert pseudocode

The implementation of the Module is accomplished by a FSM that follows the pseudocode in Listing 7. The search of a BDDNode is performed by a continuous traversal of the linked list, up until finding a BDDNode whose integer value is greater than the BDDNode looking for. The linked list is kept sorted so that finding a bigger BDDNode indicates that the searched BDDNode does not exist.

The UniqueTable index is computed by a hash function that takes both the child BDDIndexes and the variable of the BDDNode. Notice that, during the search, we do not check to see if the variable of the BDDNode fetched is equal to the variable of the BDDNode currently in search. This optimization, which reduces the need to perform variable requests, is possible as long as the number of entries in the UniqueTable is bigger than the number of possibles variables, since by having more entries, we make it so that the hash function will held different values for different variables.

After Inserting a BDDNode, we receive back the BDDIndex from the Memory Access IP. This BDDIndex is then inserted into the UniqueTable chain, at which point only two situations can occur: Either the BDDIndex needs to be inserted into the UniqueTable itself, situation where the BDDNode becomes the first element of the chain, or the BDDNode is inserted in the middle of the linked list, situation where we need to update the BDDIndex of another BDDNode. This is done by making another request to the Memory Access IP to write over the Next field of the previous node in the chain.

3.4 Supporting software

In order to test and evaluate our architecture we developed additional software that allowed our implementation to run using data from a Model Checker.

The Model Checker uses an already functional BDD package, called CuDD, to implement the BDD operations. We implemented functions that performed a conversion from the CuDD format into our format, in order to test our implementations.

The Conversion performs a full conversion, meaning that every BDD Node in the CuDD package is converted into our format. The CuDD package is also capable of storing ADD Nodes and other constants nodes. These we do not convert. The end result is that after a conversion, the CuDD might still have more nodes that our own, but the amount of BDD nodes are the same.

We also implemented the architectures in software. This implementations where then used to create the testbenches that verified the correctness of the hardware implementations.

4 Alternative architectures

In this chapter, we propose two modifications to the previous HW/SW architecture.

The first modification changes how the BDDNode data is stored and processed. We remove the ability of the hardware controller to access BDDNodes, opting to implemented all operations that necessitate BDDNodes in software. This modification is called CoBDD and is further explained in section 4.1.

The second modification augments the Depth approach with the capability to store and process more than one Request at a time. The resulting algorithm, called Bounded-Depth, is designed to make better use of the pipelining already present in the architecture. This modification is presented in section 4.2.

The previous modifications are orthogonal, in the sense that both can be implemented at the same time. The last section briefly explains the architecture where both modifications are implemented.

4.1 Depth CoBDD

The CoBDD modification changes the proposed architecture by removing the storage of BDDNode data in the FPGA SDRAM. Operations that require BDDNodes are instead performed in software. The hardware performs requests to the HPS to perform the operations that require BDDNodes: the last stage of Apply, Fetch Nodes, and the FindOrInsert operation.

```
1  while(True):
2      Request req = CoBddMemoryInterfaceRead();
3      switch(req.type):
4      case Apply_One_Node:
5          Result.NodeF = GetNode(req.f);
6      case Apply_Two_Nodes:
7          Result.NodeF = GetNode(req.f);
8          Result.NodeG = GetNode(req.g);
9      case Find_Or_Insert:
10         Result.Index = FindOrInsert(req.e,req.t,req.top);
11     case Finished:
12         return req.finalResult;
13
14     CoBddMemoryInterfaceWrite(Result);
```

Listing 8. The CoBDD algorithm running in the HPS

This modification adds a new Avalon-MM Slave interface to the Depth Controller. The HPS connects the HPS-to-FPGA bridge, using an Avalon-MM Master interface, to send and receive those requests. The pseudo code running on the HPS is Listing 8. After starting the BDD Operation, the HPS enters a loop where it reads the request, performs the operation encoded by it and then writes back the result. The Finished type indicates that the BDD Operation has terminated and the final result, encoded inside that request, is returned.

The modifications in hardware are contained in a module, called CoBDD, represented in Fig. 19. This module coordinates the transfer with the HPS. It performs the same functions as Apply stage 7 and FindOrInsert and as such it receives the data intented for those stages as input. The module then performs the following steps:

1. It processes the data, calculating the type and the order of the arguments, and stores it in a register.
2. It then awaits for the HPS to perform a read operation. In the case that the HPS reads before the CoBDD has valid data, it asserts the waitrequest signal to delay the operation. After finishing the read, the module waits for the HPS to write the result into another register.

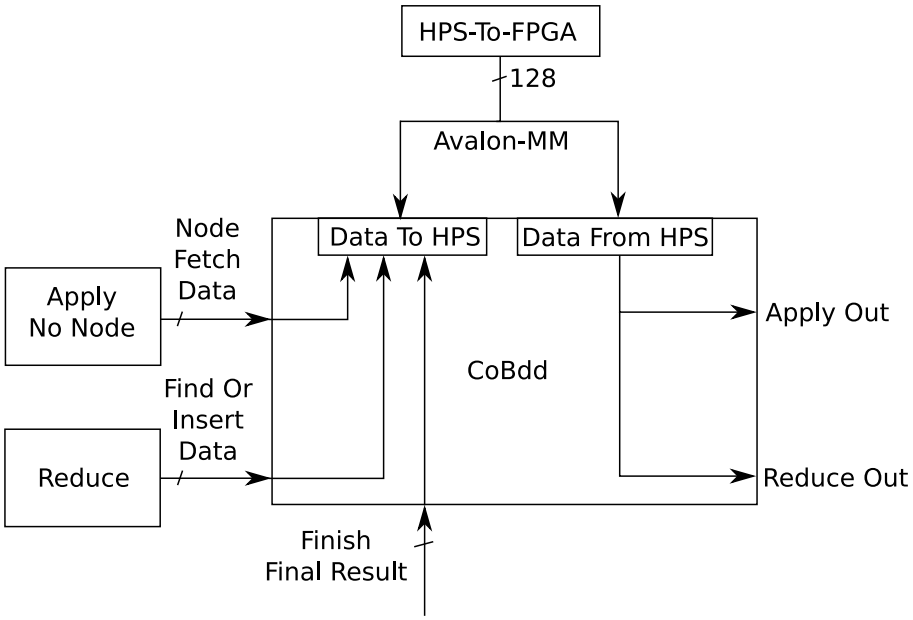

Fig. 19. The CoBDD module

3. After the write operation, the result is unpacked and, based on the type of request, set as the modules output.

4.2 Bounded-Depth

The Bounded-Depth architecture augments the Depth architecture proposed by changing the way requests are stored and processed. Instead of processing one request at a time, we process multiple requests using the pipelined modules already proposed.

We define a data structure called a Context which stores multiple requests that have the same depth. This structure also stores the state of those requests in two bitfields: validApply and validReduce. The validApply bit N indicates that the N Request is waiting to be processed in the Apply phase and similarly for the Reduce phase.

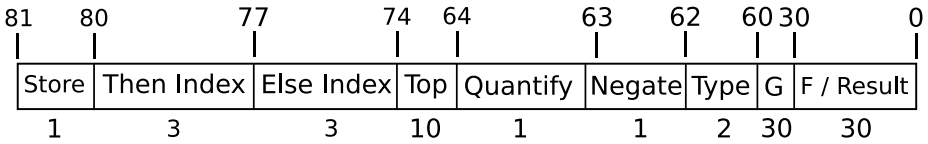

Fig. 20. Bounded-Depth Request. Compared to the Depth Request, this structure stores the child Request indexes and the result of the Operation. Since the arguments are not needed after computing the Result, it is stored using the space for the first argument.

We define the Apply and Reduce operation on a Context as performing the Apply and Reduce operation over every Request whose valid bit is set for the corresponding operation. During the Apply phase, new Requests that are created are stored in the Next Context, since their depth is one bigger. The Reduce phase fetches the results from the child Requests stored in the Next Context. The result of the Reduce phase is stored inside the Request themselves. The Bounded-Depth Request stores three more fields: The Result and two indexes that store the position of the Then and Else Requests. The Request structure is shown in Fig. 20.

```
1  i = 0
2  context[0].requests[0] = InitialRequest
3  context[0].validApply[0] = True
4  action = Apply;
5  while(i >= 0):
6      if(action == Apply):
7          Apply(context[i],context[i+1])
8          if(createdChilds):
9              i += 1
10         else:
11             i -= 1
12             action = Reduce
13     else:
14         Reduce(context[i],context[i+1])
15         if(ApplyNextContext):
16             action = Apply
17             i += 1
18         else if(context[i].validApply != 0):
19             action = Apply
20         else if(i == 0):
21             return context[0].requests[0].result;
22         else:
23             i -= 1
```

Listing 9. Pseudocode for the Bounded-Depth implementation

The pseudocode for the Bounded-Depth algorithm is shown in Listing 9. The Apply and Reduce phases are performed over all the valid requests that belong to the current Context being processed. After each phase, the next action and context depends on their results. Line 8 checks if the Apply phase produced any new Request, meaning that the Next Context needs to be Applied. Otherwise, if no new Request was created, it implies that every Request was a simple case and the next action is to Reduce the previous Context.

After the Reduce phase there are 3 different cases. The case on Line 20 is the terminal case: if the Reduce action was performed in the first Context, then the final result was computed and the algorithm terminates. The case on Line 15 is specific to the Existential optimizations and is explained further below. The Line 18 case checks if the Current Context still has valid Requests to perform Apply. Since the amount of Requests possible to store is limited (and each Request can produce up to two new child Requests) it is possible that, during an Apply phase, Requests are not processed.

The Controller implementation differs from the Depth implementation not only on the way Requests are stored and processed but on the implementation of the modules, Apply and Reduce.

The schematics for the Bounded-Depth Controller are shown in Fig. 21. The Controller implements the algorithm shown in Listing 9. The processing of Contexts occurs as follows: The Current Context and Next Context buffers store the requests for the current and next context respectively, as well as the valid Apply and Reduce bitfields. Apply or Reduce are performed by sending the valid Requests one at a time to the Apply or Reduce modules. Alongside the Request is sent an index to identify it. At the output of the modules, the result is stored into the New Current and New Next buffers. The FSM that controls the operations waits for every request to finish processing, which means that it awaits for the number of requests that passed through the pipeline to match the number of requests sent.

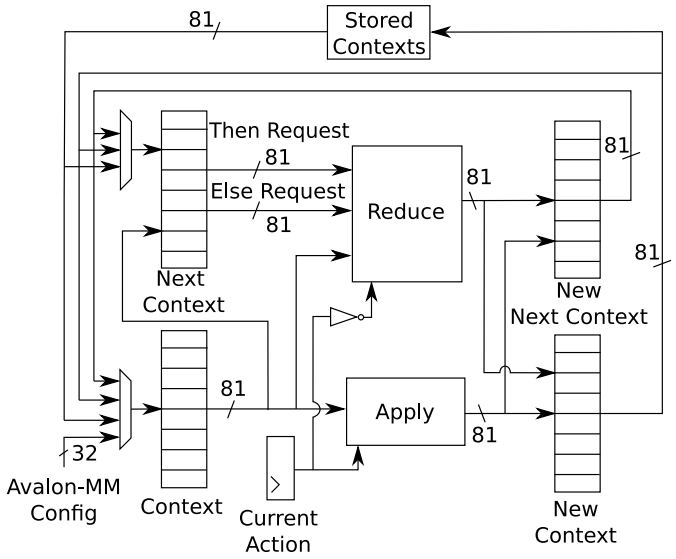

Fig. 21. The Bounded-Depth Controller. The processing of Requests are performed by sending them into the Apply or Reduce modules one at a time and then awaiting for the result. A FSM controls the transfer of Requests.

Table 2. The data transfers after an operation. Depending on how the Context Index changes, the New data is copied to the Current and Next Context buffers from these sources.

Context Index Change	Current Context	Next Context
i += 1	New Next Context	-
i stays the same	New Context	New Next Context
i -= 1	Stored Context	New Context

At the end, when every Request that was sent as finished processing, the FSM moves data from the New Buffers into the Current buffers, as well as store it into the Stored Context buffer. The data movement is depicted in Table 2. The Context Index changes match the changes present in Listing 9.

4.2.1 Apply

Compared to the Depth approach, the Apply module presents two modifications: The way existential optimizations are handled and a new stage is added that checks if a new Request is not already present in the Next Context.

The existential optimizations are handled differently, compared to the Depth architecture. The Apply phase, when performing an Exist or a Relational Product operation, and needs to perform variable quantification, it inserts two requests, but sets the left Request in an inactive state, by setting the Store bit to one, as well as not setting it valid to perform Apply. Afterwards, during the Reduce phase, it is possible to check if the Request has been computed or not, by checking the state of the Store bit and thus figure out in which state, before or after the Existential Optimizations, the Request is in.

The new stage added after the Apply checks to see if we can reuse any of the Requests present in the Next Context, instead of reserving and creating a new one with the same arguments. This is only possible for Requests that do not have the store bit set, as this requests can have their arguments changed due to the Relational Optimizations.

The end result is that this new stage saves some space in a Context to allow more Requests to be inserted. It also guarantees that no two Requests with the same arguments are present in the same Context, which simplifies the Caching of results: If two equal requests where in the same Context, it

might happen that both would be processed at the same time. This would lead to their child requests to be process at the same time and so on.

4.2.2 Reduce

The reduce module is the one that suffers more dramatic changes when comparing this implementation with the Depth Controller.

Unlike the Depth Approach, the Bounded-Depth Reduce needs to keep track and process multiple requests at the same time. Therefore a pipelined approach is used. This pipeline processes the requests in the following way:

1. The simple reduce cases are checked. These correspond to the cases where the e and t arguments are equal and to the quantify simple Reduce cases.
2. In the case that it is not a simple case and it is not a quantify operation, perform FindOrInsert.
3. Store the result of the FindOrInsert in the Operation and in the Simple cache. If it was a quantify operation, create the new requests.

The existential optimizations are divided into two problems, with regards to the Bounded-Depth Approach: The creation of the else Request and performing the And Request after computing both child requests.

The first problem is resolved by having the Apply module store the else Request in an inactive state. During the Reduce operation, if the Then Request result is not a simple case, this else Request is then set to active, and the algorithm performs an Apply on the next Context Index. The inactive state is set by setting the Store bit to one in the Request.

The second problem is resolved by reusing the same index to store the And Request. The Negate bit is set to true so that the result is negated afterwards.

4.3 Bounded-Depth CoBDD

The Bounded-Depth CoBDD architecture combines both modifications previously mentioned. The CoBDD algorithm performed by the HPS remains the same one presented in Listing 8, since the Bounded-Depth modifications do not change the data structures exchanged with the HPS.

In hardware, the only change needed is to augment the CoBDD module with the ability to stall, since it is now integrated into a pipeline that processes more than one Request at a time. The modification involves adding two more signals, busyOut and busyIn, which are used to stall the data transfers between stages if necessary. The busy out signal is asserted when a transfer is in process and the busyIn signal stalls the output of data.

5 Results and Evaluation

In this chapter we analyse and evaluate the architectures proposed.

We perform a experimental evaluation of our architectures by running an implementation on a SoC-FPGA board.

5.1 Experimental evaluation

The evaluation of the proposed architecture and the alternative architectures are performed on a De1-SoC board, which contains a Dual-core ARM Cortex-A9 HPS and a Cyclone V SoC 5CSEMA5F31C6 FPGA Device. The tools used to compile the designs where the Quartus Prime Lite edition, version 18.1.

5.1.1 Methodology

The ideal approach to evaluate the architecture would be to run a Model Checker that implements our HW/SW architecture and compare runtimes. This is not possible since our package does not fully implement all the operations necessary by the Model Checker (used to build the transition BDD associated to the Model, for example). Because our package also does not implement a garbage collector simulating various operations would inevitably lead to a poor performance, as the garbage collector also serves to improve the performance of BDD operations.

The approach taken was to make use of the Conversion function developed to convert the BDD from CuDD into our format, and then measure the runtime for the operation individually. Since the Relational Product is the most time consuming operation performed, we profiled an example that came with the NuSMV Model Checker, called *msi_wtrans*, and identified 8 times where the Relational Product took a considerable amount of the runtime (more than one second). We then tested our proposed architectures with these 8 operations and measured the results. We also tested the Substitution algorithm using a similar approach: we tested our architecture for the first 8 Substitution operations that are performed in the msi_wtrans example.

Values that do not depend on the characteristics of the board, only on the architecture, where calculated in software. The software was used as an integration test for the architecture and as such it is guaranteed that the values calculated are the same in the hardware.

5.1.2 Evaluation

The proposed architectures were implemented on a De1-SoC device. They were compiled using Quartus Prime Lite Edition 18.1 with compiler settings to optimize for speed. The Depth and Depth CoBDD architecture can reach 115 MHz but in order to provide a comparison between architectures, all implementations use the frequency of the slowest architecture, the Partial CoBDD architecture, which is 100 MHz.

In every architecture proposed, the critical path is present in one of our modules and as such there is still potential for improving the maximum frequency.

Table 3 presents the resources used by the architectures proposed. The Bounded-Depth CoBDD architecture uses more ALM units because, as mentioned earlier, this architecture is the bottleneck in terms of frequency. The other architectures have more leeway and the Quartus tool performs less optimizations after reaching the desired frequency.

The amount of M10K Blocks needed measures how many M10K blocks are used if every bit could be utilized. Since the blocks have a fixed size and location, not every bit can be used and the amount of blocks actually used is higher.

Table 3. Resource usage by implementation

Architecture	ALM	M10K Blocks needed	M10K Blocks used
Depth	7,562	281	368
Depth CoBDD	6,619	89	124
Bounded-Depth	9,144	312	394
Bounded-Depth CoBDD	12,652	124	165
Total	32,070	397	397

Table 4. The time results, in seconds, for 8 Relational Product operations in the *msi_trans* example.

Operation Number	CuDD	Depth (no preload)	Depth (preload)	Depth CoBDD	Depth CoBDD (50 MHz)	Bounded-Depth (no preload)	Bounded-Depth (preload)	Bounded-Depth CoBDD
37	1.499	2.056	2.036	2.135	3.371	2.186	2.182	2.058
41	1.347	1.750	1.669	1.652	2.511	1.841	1.828	1.712
81	1.173	1.835	1.693	1.608	2.385	1.856	1.818	1.562
85	2.497	2.861	2.816	2.769	4.411	3.011	2.997	2.637
89	2.675	3.042	3.009	3.042	4.703	3.181	3.173	3.053
93	3.223	3.482	3.466	3.433	5.261	3.663	3.650	3.417
97	1.635	1.803	1.791	1.890	2.999	1.940	1.937	1.959
101	1.174	2.078	1.925	1.743	2.470	2.201	2.161	1.845

The time taken to perform these 8 Relational Product operations is shown in Table 4. As we can see, the architectures proposed are not far off from equaling the runtime of the CuDD library. The preload and no preload indicate whether the data was already present in the FPGA RAM or not. Thus the preload times demonstrate better how a full run of the architecture would look like, as it would try to keep as much data in FPGA RAM as possible.

The Depth architectures slightly outperform the Bounded-Depth architectures. The reason is likely that the few cycles gained by having the requests being processed in a pipelined fashion do not offset the times where data is fetched but ultimately not used because the next Context is full.

The 50 MHz Depth CoBDD was an implementation created to measure the interaction between HPS and CoBDD module by using the Signal Tap Logic Analyzer tool from Quartus. This tools limits the maximum frequency to around 70 MHz in our implementations, and so we could not use this tool while running at 100 MHz. We measure its performance to compare how half the frequency affects the runtime. As we can see, the time taken increases by around 50%.

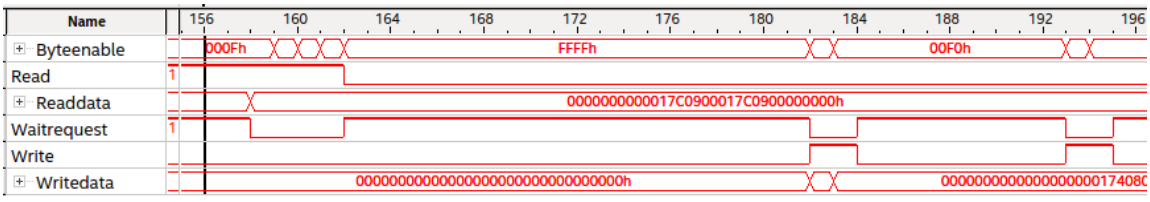

Fig. 22. CoBDD read followed by a write for the Apply Fetch 1 Node type.

Figures 22 and 23 are the results of the CoBDD Apply Fetch 1 or Fetch 2 nodes, respectively. Fetching 1 node takes 20 cycles and fetching 2 nodes takes 23 cycles (from the moment the read finishes and the write starts). The amount of cycles is measured from the perspective of the hardware, running at 50 MHz, and it improves at higher frequencies.

The writing of the result occurs over two sets of two cycles. It takes 13 cycles in both types to finish the read operation. Thus the amount of cycles it takes for a full transfer of data, from the moment the HPS gets valid data to the moment the CoBDD modules gets the result is around 40 cycles, from the

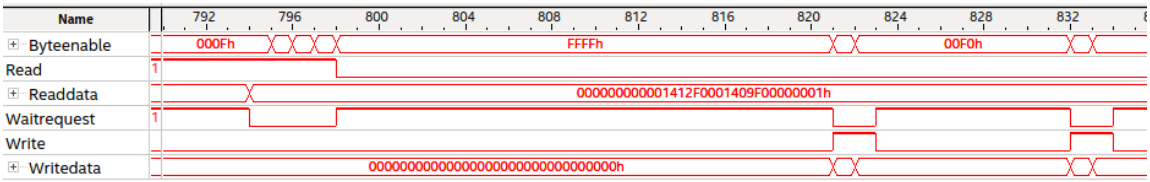

Fig. 23. CoBDD read followed by a write for the Apply Fetch 2 Node type.

perspective of the hardware, running at 50 MHz. Since this cannot be pipelined, this might explain the reason the CoBDD architectures had worse results then the non-CoBDD architectures.

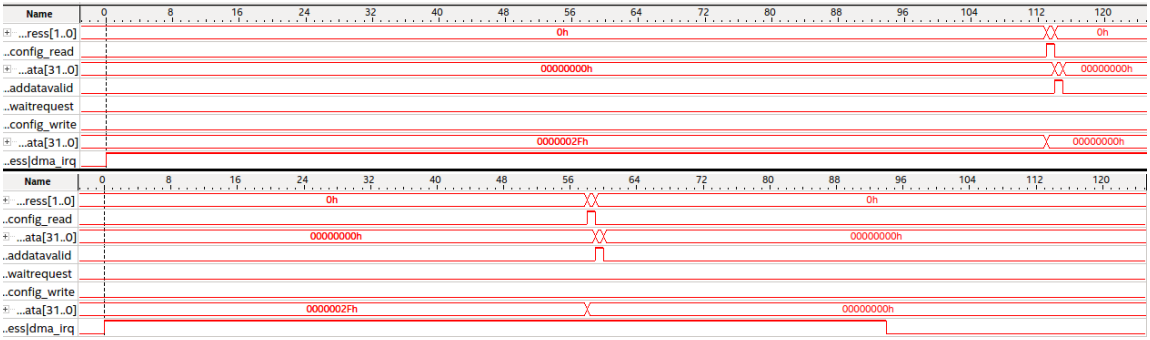

Fig. 24. Cycles taken until the interrupt routine starts (asserting the read signal), measured using Signal Tap. The longest amount of cycles measured was 113 (top picture) and the shortest was 58 (bottom picture).

The latency from the moment the Memory Access IP asserts the IRQ signal and the interrupt routine performs the first read was measured using Signal Tap on an implementation running at 50MHz. In Fig. 24 we show the results for longest and the shortest amount of cycles from the moment the DMA_IRQ signal is asserted to the moment the read signal is asserted. A read is the first operation that the interrupt routine performs and as such we are measuring the time it takes for the routine to start.

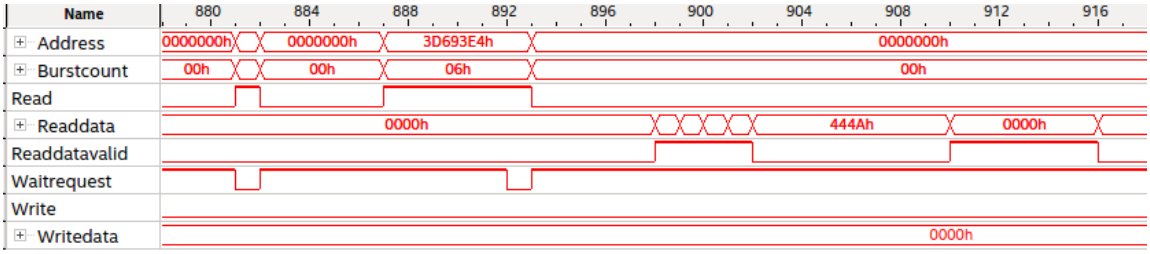

Fig. 25. The Memory Access module using posting reads before receiving the data from a previous read

In Fig. 25 we can see the Memory Access IP performing read operations to the SDRAM module. It shows two reads being posted before the first one returns valid data. In this case, it took 8 cycles to provide a read value after the previous read value finished returning. Considering that the first read value only returned after 16 cycles, it is easy to see how this form of interaction with the SDRAM improves throughput.

The simple cache mentioned in section 3.3.2 was evaluated in software. The results for the simple cache hits are presented in Table 5. The implementation used to perform the measurements was the Depth architecture. The Cache Saved indicates that the Simple Cache prevented an unneeded memory

Table 5. Simple Cache results. The higher the Cache Saved and the Cache Hit values the better, performance wise.

Operation	Cache Saved (%)	Cache Hit (%)	Cache Miss (%)
37	49	34	17
41	53	30	17
81	50	34	16
85	50	33	17
89	51	32	17
93	50	33	17
97	51	32	17
101	54	28	18

access. Cache Hit means that the cache entry was fetched but it was an actual hit and as such the memory access was needed. The Cache miss corresponds to the times where the Simple Cache had the same hash but the Operation Cache was a miss. This means that a wasteful memory access was performed.

Table 6. The Average of the amount of valid Requests present in a Context at a start of an iteration (Apply or Reduce).

Operation	Valid Requests Average
37	2.39
41	2.36
81	2.36
85	2.41
89	2.41
93	2.41
97	2.40
101	2.35

The impact of the Bounded-Depth modification can be measured by how much Valid Requests are valid at the start of an iteration. This measures how much work an iteration performs. The higher the average the better. The results shown in Table 6 indicate that for the Bounded-Depth modification, using a Context size of 8, that on average, each iteration processed around 2.4 requests. This low value is the result of how the Existential Optimizations are implemented, which frequently lead to a Context being processed with a low amount of Valid Requests. Still, the Depth implementation can be thought as only having a Valid Request Average of 1, meaning that this implementation at least can provide the groundworks for a better one.

5.2 Discussion

The architectures proposed almost reached an equal runtime to the software only CuDD library. It should be possible to improve the frequency and arrive at an almost equal time to the software implementation. At this point, even thought the architectures would not improve the performance, only match it, they could offload the work from the HPS system.

Also it is to be noted that the implementations do not make full use of the capabilities available in order to allow the comparison between them in evaluation. As an example, the CoBDD implementations could use the 64MB of SDRAM as Cache and store variable data using on-chip memory. This would clearly improve the performance of the architecture.

Regardless of optimizations, it is clear that memory throughput is the most vital aspect when trying to improve BDD operations. The iteration of a BDD operation barely performs any operation that does

not depend in performing a memory access. Simple cases, calculating top variables, computing hash to later perform a cache access and calculating the Requests to insert are the only operations that do not require memory access. Everything else needs to fetch data.

Therefore, for future works, the maximization of memory throughput should therefore be the priority when creating an accelerator for BDD operations.

6 Conclusion

In this document I presented four different architectures to improve the performance of BDD operations. In the end, the architectures did not end up improving the performance, but the difference in time is close.

Further optimizations are possible to perform and it should be possible to improve the architectures further both in terms of reducing the amount of cycles it takes to perform operations as well as increasing the maximum frequency.

6.1 Future work

The architectures proposed where designed with the objective of having them run in a low-end SoC-FPGA device. Because of this, some design choices were not considered, because of the lack of resources necessary for their implementation.

The most important resource, when discussing BDD operations, is memory. Because of this, FPGAs that possess a higher amount of memory and allow a higher throughput could potently implement more efficient architectures.

The Apply module shown divides the memory requests into three different types: Var, Node and Cache. If these types were stored in different memories, it would be possible to parallelize the Apply module further. As an example, memory requests to Nodes could be done even before the Var result returned. Exploring an architecture that provides a division of these three components could lead to a more efficient architecture.

The act of accessing SDRAM memory can also be improved. The SRAM memory throughput improves when performing burst reads and writes, as well as continuously accessing positions of memory closer to each other. One improvement to this architecture could be to have multiple requests being process in the same phase. These multiple requests would make memory requests but these would be resolved out-of-order, such that requests that are closer in address or whose page is currently present in memory could be resolved first. This would improve throughput at a small cost of extra logic resources which we have plenty.

References

[1] Ashar, P., Cheong, M.: Efficient breadth-first manipulation of binary decision diagrams. In: ICCAD (1994)

[2] Ben-Ari, M., Pnueli, A., Manna, Z.: The temporal logic of branching time. Acta Informatica 20(3), 207–226 (Sep 1983)

[3] Beneš, N., Brim, L., Pastva, S., Safránek, D.: Model Checking Approach to the Analysis of Biological Systems, pp. 3–35 (06 2019)

[4] Bourahla, M., Benmohamed, M.: Efficient partition of state space for parallel reachability analysis. In: The 3rd ACS/IEEE International Conference onComputer Systems and Applications, 2005. pp. 21– (Jan 2005)

[5] Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a bdd package. In: Conference proceedings on 27th ACM/IEEE design automation conference - DAC 90. ACM Press (1990)

[6] Burch, J.R., Clarke, E.M., Long, D.E., McMillan, K.L., Dill, D.L.: Symbolic model checking for sequential circuit verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13(4), 401–424 (April 1994)

[7] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L.: Sequential circuit verification using symbolic model checking. In: 27th ACM/IEEE Design Automation Conference. pp. 46–51 (June 1990)

[8] Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 1020 states and beyond. Information and Computation 98(2), 142–170 (Jun 1992)

[9] Bwolen Yang, Yirng-An Chen, Bryant, R.E., O'Hallaron, D.R.: Space and time-efficient bdd construction via working set control. In: Proceedings of 1998 Asia and South Pacific Design Automation Conference. pp. 423–432 (Feb 1998)

[10] Chen, Y.A., Yang, B., Bryant, R.E.: Breadth-first with depth-first BDD construction: A hybrid approach (Mar 1997)

[11] Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: Nusmv: a new symbolic model checker. STTT 2, 410–425 (03 2000)

[12] Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems 8(2), 244–263 (Apr 1986)

[13] Clarke, E.M.: The birth of model checking. In: 25 Years of Model Checking, pp. 1–26. Springer Berlin Heidelberg (2008)

[14] Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Logics of Programs, pp. 52–71. Springer-Verlag (1981)

[15] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: LTL model checking. In: Lecture Notes in Computer Science, pp. 385–418. Springer Berlin Heidelberg (2007)

[16] van Dijk, T., Laarman, A., van de Pol, J.: Multi-core BDD operations for symbolic reachability. Electronic Notes in Theoretical Computer Science 296, 127–143 (Aug 2013)

[17] van Dijk, T., van de Pol, J.: Multi-core decision diagrams. In: Handbook of Parallel Constraint Reasoning, pp. 509–545. Springer International Publishing (2018)

[18] Gai, S., Rebaudengo, M., Reorda, M.S.: A data parallel algorithm for boolean function manipulation. In: Proceedings Frontiers '95. The Fifth Symposium on the Frontiers of Massively Parallel Computation. pp. 28–34 (Feb 1995)

[19] Grumberg, O., Heyman, T., Guy, N., Schuster, A.: Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. vol. 3725, pp. 129–145 (10 2005)

[20] Heyman, T., Geist, D., Grumberg, O., Schuster, A.: A scalable parallel algorithm for reachability analysis of very large circuits. Formal Methods in System Design 21, 317–338 (11 2002)

[21] Johnson, C.: The natural history of bugs: Using formal methods to analyse software related failures in space missions. In: FM (2005)
[22] Kaveh, N.: Using model checking to detect deadlocks in distributed object systems. In: EDO (2000)
[23] Kimura, S., Clarke, E.: A parallel algorithm for constructing binary decision diagrams. In: Proceedings., 1990 IEEE International Conference on Computer Design: VLSI in Computers and Processors. IEEE Comput. Soc. Press (1990)
[24] Lee, I., Rajasekaran, S.: Fast parallel algorithms for model checking using bdds. In: [1993] Proceedings Seventh International Parallel Processing Symposium. IEEE Comput. Soc. Press (1993)
[25] McMillan, K.L.: Symbolic Model Checking. Springer US (1993)
[26] Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R.M., Gopalakrishnan, G.: Parallel and distributed model checking in eddy. International Journal on Software Tools for Technology Transfer 11(1), 13–25 (Nov 2008)
[27] Milvang-Jensen, K., Hu, A.J.: BDDNOW: A parallel bdd package. In: Formal Methods in Computer-Aided Design, pp. 501–507. Springer Berlin Heidelberg (1998)
[28] Ochi, H., Yasuoka, K., Yajima, S.: Breadth-first manipulation of very large binary-decision diagrams. In: Proceedings of 1993 International Conference on Computer Aided Design (ICCAD). pp. 48–55 (Nov 1993)
[29] Ochi, H., Ishiura, N., Yajima, S.: Breadth-first manipulation of SBDD of boolean functions for vector processing. In: Proceedings of the 28th conference on ACM/IEEE design automation conference - DAC (91). ACM Press (1991)
[30] Oortwijn, W., Dijk, T.v., Pol, J.v.d.: Distributed binary decision diagrams for symbolic reachability. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. pp. 21–30. SPIN 2017, ACM, New York, NY, USA (2017)
[31] Parasuram, Y., Stabler, E., Chin, S.K.: Parallel implementation of bdd algorithms using a distributed shared memory. In: Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences HICSS-94. IEEE Comput. Soc. Press (1994)
[32] Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Lecture Notes in Computer Science, pp. 337–351. Springer Berlin Heidelberg (1982)
[33] Ranjan, R., Gosti, W., Brayton, R., Sangiovanni-Vincenteili, A.: Dynamic reordering in a breadth-first manipulation based BDD package: challenges and solutions. In: Proceedings International Conference on Computer Design VLSI in Computers and Processors. IEEE Comput. Soc (1997)
[34] Rice, M., Kulhari, S.: A survey of static variable ordering heuristics for efficient bdd / mdd construction (2008)
[35] Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: Proceedings of 1993 International Conference on Computer Aided Design (ICCAD). IEEE Comput. Soc. Press (1993)
[36] Sanghavi, J.V., Ranjan, R.K., Brayton, R.K., Sangiovanni-Vincentelli, A.: High performance bdd package by exploiting memory hierarchy. In: 33rd Design Automation Conference Proceedings, 1996. pp. 635–640 (June 1996)
[37] Siminiceanu, R., Ciardo, G.: New metrics for static variable ordering in decision diagrams. pp. 90–104 (03 2006)
[38] Somenzi, F.: Binary decision diagrams. International Journal on Software Tools for Technology Transfer (1999)
[39] Somenzi, F.: Efficient manipulation of decision diagrams. International Journal on Software Tools for Technology Transfer 3, 171–181 (2001)
[40] Stornetta, T., Brewer, F.: Implementation of an efficient parallel bdd package. pp. 641–644 (07 1996)

[41] Trimberger, S.M.: Three ages of fpgas: A retrospective on the first thirty years of fpga technology. Proceedings of the IEEE 103(3), 318–331 (March 2015)
[42] Velev, M.N., Gao, P.: Efficient parallel GPU algorithms for BDD manipulation. In: 2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE (Jan 2014)
[43] Wang, C., Yang, Y., Gupta, A., Gopalakrishnan, G.: Dynamic model checking with property driven pruning to detect race conditions. vol. 5311 (10 2008)
[44] Yoneda, T., Ishigaki, T.: Hardware acceleration for bdd manipulations (01 2000)

7 Appendixe A - Relational Product

```
1  BDD BddRelationalProduct(f,g : BDD, quantify : bool bitarray):
2    // Simple cases
3    if(f == ZERO | g == ZERO | f == Negate(g)):
4      return ZERO
5    if(f == ONE & g == ONE):
6      return ONE
7
8  // Compute top variable (the smallest variable of both operands)
9    int varF = GetVariable(f)
10   int varG = GetVariable(g)
11   int top = min(varF,varG)
12
13   // Relational Product simplification
14   if(top > lastVariableToQuantify):
15     return BddAnd(f,g)
16   if(f == ONE | f == g):
17     return BddExist(g,quantify)
18   if(g == ONE):
19     return BddExist(f,quantify)
20   // Check Operation cache
21   if(<f,g,quantify[top],RelationalProduct> in cache):
22     return cache[<f,g,quantify[top],RelationalProduct>]
23
24 // Fetch node data if variable equal to top
25   BDD fv,fnv,gv,gnv
26   if(varF == top):
27     fv = GetNode(f).Then
28     fnv = GetNode(f).Else
29     if(IsNegated(f)):
30       fv = Negate(fv)
31       fnv = Negate(fnv)
32   else:
33     fv = fnv = f
34   if(varG == top):
35     gv = GetNode(g).Then
36     gnv = GetNode(g).Else
37     if(IsNegated(g)):
38       gv = Negate(gv)
39       gnv = Negate(gnv)
40   else:
41     gv = gnv = g
42
43   BDD res,e,t
44   if(quantify[top]):
45     t = BddRelationalProduct(fv,gv,quantify)
46
47     if(t == ONE | t == fnv | t == gnv): // Existential optimization
48       return t
49
```

```
50      // Relational Optimization
51      if(t == Negate(fnv)):
52        e = BddExist(gnv,quantify)
53      else if(t == Negate(gnv)):
54        e = BddExist(fnv,quantify)
55      else:
56        e = BddRelationalProduct(fnv,gnv,quantify)
57
58      res = Negate(BddAnd(Negate(e),Negate(t)))
59    else:
60      t = BddRelationalProduct(fv,gv,quantify)
61      e = BddRelationalProduct(fnv,gnv,quantify)
62
63      if(t == e):
64        res = t
65      else:
66        if(IsNegated(t)):
67          res = Negate(FindOrInsert(Negate(e),Negate(t),top))
68        else:
69          res = FindOrInsert(e,t,top)
70
71    InsertIntoCache(<f,g,quantify[top],RelationalProduct>,res)
72    return res
```

Listing 10. Pseudocode for the complete Relational Product Operation