TECHNICAL UNIVERSITY OF CRETE
DEP. OF ELECTRONIC AND COMPUTER ENGINEERING
MICROPROCESSOR AND HARDWARE LABORATORY

Modeling Structures for Recursion in Reconfigurable Hardware

M.Sc. Thesis of Spyridon Ninos

Committee:
Prof. Apostolos Dollas *(advisor)*
Assoc. Prof. Dionisios Pnevmatikatos
Asst. Prof. Ioannis Papaefstathiou

to the most wonderful recursion ever existed:

to freedom.

(**FREE**dom **D**oesn't **O**bey **M**anipulation)

ABSTRACT

Recursion is used for solving problems that present repeating patterns. The high computing power which is demanded for the implementation of certain recursive algorithms has driven the effort to implement them in integrated circuits of reconfigurable logic (FPGAs). To date there is no general way to implement recursion in FPGAs, as it is considered of low performance and demanding in resources.

In this thesis we studied a general way to implement structures for recursion in reconfigurable hardware. A step by step implementation methodology is analyzed and we propose a new approach to the subject, with respect to older proposals. This approach solves many of the problems that exist in those proposals.

The recursive algorithms, that are implemented based on this methodology, present small area consumption and good performance, with respect to the faster modern processors. They have been implemented in FPGA development boards, and the results have confirmed the performance estimations of the resulted circuits.

ΕΠΙΤΟΜΗ

Η αναδρομή χρησιμοποιείται για επίλυση προβλημάτων τα οποία παρουσιάζουν επαναλαμβανόμενη δομή. Οι υψηλές απαιτήσεις υπολογιστικής ισχύος που απαιτούνται για την εφαρμογή ορισμένων αναδρομικών αλγορίθμων, έχουν οδηγήσει στην προσπάθεια εφαρμογής τους σε ολοκληρωμένα κυκλώματα αναδιατασσόμενης λογικής (FPGAs). Μέχρι σήμερα δεν υπάρχει γενικός τρόπος για την εφαρμογή αναδρομής σε FPGAs, καθώς θεωρείται χαμηλής απόδοσης και υψηλών απαιτήσεων σε πόρους.

Στην παρούσα διατριβή μελετήθηκε ένας γενικός τρόπος για την εφαρμογή δομών αναδρομής σε αναδιατασσόμενη λογική. Αναλύεται βήμα προς βήμα η μεθοδολογία εφαρμογής και προτείνεται μια καινούρια προσέγγιση στο θέμα σε σχέση με παλαιότερες προτάσεις. Η προσέγγιση αυτή λύνει πολλά από τα προβλήματα που υπάρχουν στις προτάσεις αυτές.

Οι αναδρομικοί αλγόριθμοι που εφαρμόζονται με βάση την μεθοδολογία, παρουσιάζουν μικρές απαιτήσεις σε χώρο και καλή απόδοση, σε σχέση με τους ταχύτερους σύγχρονους επεξεργαστές.

Έγινε εφαρμογή σε πλακέτες ανάπτυξης αναδιατασσόμενης λογικής (FPGA development boards), και τα αποτελέσματα επιβεβαίωσαν τις εκτιμήσεις απόδοσης των παραγόμενων κυκλωμάτων.

ACKNOWLEDGEMENTS

This thesis is a result of hard work and countless nights of study. It is also the result of the support of certain people, which I would like to thank through these lines.

First of all, I thank my parents for the endless support and faith in me. Parents, the only thing I know is being part of your wonderful family. I wouldn't want to know any other way.

I truly thank my advisor, prof. Apostolo Dolla, for his remarkable patience against my peculiar way of doing things. Professor, you taught me all those things that every engineer needs to know: strict professionalism, plan Bs and project management among other things. But most important, you trusted my last-minute "come-backs"; this is something that few people have the luxury to afford. Thank you for everything.

I would also like to thank assoc. prof. Dionisio Pnevmatikato and asst. prof. Ioanni Papaefstathiou, for accepting the participation in my thesis committee. Professors, taking your courses was a great chance to discover obscure niches in computer engineering. You are wonderful teachers.

Finally, a big "thank you" goes to all the friends that supported me during the endless nights of designing and debugging things. Guys and gals, there is no match for your presence in my life.

INTRODUCTION

Purpose

Since the early days of electronic engineering, companies have strived to provide high quality Electronic Design Automation (EDA) tools. This need was dictated by the long time-to-market periods and high non-recurring engineering (NRE) costs enforced by the manual design processes. One of the key features that EDA tools have to provide to the designer, is the capability to use high level algorithm concepts during the design period. One such concept is recursion, which has been proved to be a powerful technique in problem solving. This is the reason why recursion found its way to software engineering, boosting the productivity of programmers.

With the advent of hardware description languages (HDLs), the need to utilize recursion in hardware designs has increased. HDLs provide the engineer a more productive way to describe a specified circuit and many efforts have been made to adopt techniques from the software engineering discipline to the hardware one. Unfortunately, there is no known general way to implement recursion in hardware to date. Even though the literature has been enriched with some research on the subject, widely accepted methods have yet to be proposed. This thesis studies certain ways to implement recursion in field programmable gate arrays (FPGAs). Its purpose is to propose a generic way to implement recursion in hardware and to bridge the gap between today's needs and EDA products.

Contribution

In this thesis we study the most common ways to implement the recursive functionality of algorithms in hardware. A thorough study of previous research on the field has been done, and some pitfalls and drawbacks have been identified. The contribution of this thesis is the proposal of a new method to implement recursive functionality in reconfigurable hardware. We show that our method, is simpler that the others and produces faster circuits, retaining the area consumption to small figures. We also set a frame which facilitates the implementation of recursive algorithms in FPGAs, which can be used by future developed EDA tools.

Thesis organization

The first chapter, Fundamental Concepts, is an introduction to the two main concepts behind this thesis: recursion and FPGAs. We introduce basic facts about the theory surrounding each of them, and we present the capabilities to use them in a variety classes of applications. In the second chapter we present the previous research in the field and analyze the three major previous contributions. In chapter 3 we study the most prevalent proposal for recursion implementation, along with the results that have been presented in international conferences. The method proposed in this thesis is analyzed in the fourth chapter, where we show the steps involved in transforming a high-level language recursive solution to the low level concepts of an HDL. The fifth chapter is dedicated to some hardware design patterns that can be used during the design and implementation of recursion to FPGAs. Finally, the sixth chapter presents case studies, implementations and results of our proposed method to some of the most known recursive problems.

CHAPTER 1: FUNDAMENTAL CONCEPTS

1.1 Introduction to Recursion

Using the Russian dolls that fit inside one another as an example [1], we can have a basic visualization of recursion. The smaller doll is inside a bigger one, and this in turn is inside a bigger one. All of the small ones fit inside the biggest doll. All dolls are cut in half, in a way that when you separate the two halves of the biggest doll a smaller one appears from inside it. When you separate the two halves from the second doll then a third one, smaller, appears from inside the second doll. This procedure is repeated until you reach the smallest of the dolls, which has no doll inside it.

The above paradigm is an every day life example of recursion; when you open a doll, a same one appears. This could be the fundamental definition of recursion; something that reproduces itself. This notion can be extended to programming. A recursive function is a function that calls itself, making a ***recursive call.*** Recursive solutions are ideal when designing algorithms that have a repeating solving pattern. For example, a binary search algorithm of a tree can be solved using the same small search function repeatedly, until the whole tree is searched. This simplifies the solutions that can be designed, since using recursion we can reduce the problem from searching the whole tree to identifying the repeating pattern and applying a smaller solution to it recursively.

1.2 Fundamental Theory and Applications of Recursion

Recursion can be applied directly or indirectly. ***Direct recursion*** is the case where a function calls itself directly. An example of a direct recursive function follows [2] (in C programming language):

```
void hanoi(disks, from, to, using)
{
        If (disks>0)
        {
                Hanoi(disks-1, from, using, to);
                Printf("%d to %d\n", from, to);
                Hanoi(disks-1, using, to, from);
        } / * if */
} /* Hanoi() */
```

Example 1: Direct recursion

The above example is the Towers of Hanoi problem which, given the true condition in the IF statement, the function proceeds with calling itself, with different parameters. The code shown in Example 1 will be used in Section 4.5, as a software reference.

The ***indirect recursion*** is the case where a function A calls a function B and in turn B calls A, as illustrated in the example bellow:

```
Void A(void)
{
        B();
}
Void B(void)
{
        A();
}
```

Example 2: Indirect Recursion

Every recursive call has two main cases: the base case and the recursive case. The ***base case*** is the case for which the solution can be stated non-recursively. The ***general or recursive case*** is the case for which the solution is expressed in terms of a smaller version of itself [1]. In Example 1 the base case is the false IF condition, when disks are equal to zero (0), and the recursive case is the body of the IF condition, when the *disks* variable is greater than zero. Recursive functions have to follow two general rules: they should be called each time with a different set of parameters, and they must have a terminating case. If either of the conditions is not met, the recursive function will call itself ad infinitum, until an external factor terminates the execution (i.e. resource quota restrictions or memory allocation error).

There are several different kinds of recursion, and many ways to categorize functions according to the recursion type they use. The most common recursion types are the linear, tail, binary, exponential, nested and mutual recursions [3]. The categorization of recursion to the above classification is based on how many times and in what way recursive calls happen inside the body of a recursive function. Based on the type of recursion, there can be some optimizations. For example, tail recursion, which is the case where after the last recursive call there are no other instructions to be executed, can be reduced to simple iteration rendering the solution faster and cheaper in memory consumption.

Despite the fact that recursion is a very elegant way for solving problems, it cannot be applied directly to hardware implementations. Although *hardware description languages* (HDLs) have promoted a more *software like* design process to the hardware domain, they still fail to provide the designer with the full software capabilities. One such failure is the lack of a general way to describe recursive algorithms in hardware. Efforts have been made to bridge the gap between hardware description and hardware implementation but the gap is still wide. One of the first attempts was to embed recursive

functionality in subprograms of the VHDL[1] language [4]. The recursive usage of the VHDL language can be simulated, but cannot be synthesized [5]. This is a result of the important distinction between *describing recursively a circuit* and *describing a recursive algorithm*. In the first case, we use a recursive way to describe a circuit that is a multiple repetition of the same small circuit. For example in [6] a demonstration of a recursive description of a fanout tree is made. For the description of a recursive algorithm there is no standard supported way for EDA tools to implement, so to date there is no method for Computer Aided Design (CAD) tools to support recursion to be applied to hardware, except by *ad hoc* design.

1.3 Introduction to Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays (FPGAs) are the state of the art in today's hardware design processes. They are the first choice for hardware prototyping along with Complex Programmable Logic Devices (CPLDs). FPGAs are integrated circuits that can be programmed multiple times in the field of their usage; this is the reason why they are called *field programmable*. The ability to be programmed multiple times during the design phase has boosted the productivity of hardware designers and reduced the design costs. Design, simulation, implementation, verification and redesign constitute a design cycle that was once very expensive and time-consuming.

Although FPGAs and CPLDs have similar programming capabilities, they differ a lot in the domain of structural architecture and target area. FPGAs offer capabilities to design multimillion gate equivalent circuits, whereas CPLDs do not. On the other hand CPLDs offer lower power consumption than FPGAs, making them more suitable to limited power consumption applications. Despite that, FPGAs are the preferred choice for mainstream applications, especially for computationally intensive algorithm designs; their structures consist of many digital building blocks (i.e. Digital Signal Processors, adder/subtracters, multipliers etc) that make them the ideal choice for arithmetic calculations and other similar applications that need excessive computer capabilities (i.e. signal processing, search algorithms etc.). Moreover, with the introduction of extremely high density FPGAs, the concept of System on a Chip (SoC) has emerged, which has driven the embedded technology to be one of the most prominent areas of interest today.

1.4 Fundamental Theory and Applications of FPGAs

A general categorization of the *programmable logic devices* (PLDs) is given in **Fig. 1** ([7]). In this figure we can see the major PLD categories, which are the Simple PLDs and the Complex PLDs. In the SPLD subcategories we can see all the basic structures that can be found in the market, including *programmable read only memories* (PROMs), *erasable programmable read only memories* (EPROMs), *electronically erasable programmable read only memories* (EEPROMS) and FLASHes, along with *programmable logic arrays* (PLAs) and *programmable array logic* (PALs). CPLDs on the other hand, have a more complex structure; they mainly consist of a lattice of SPLDs (i.e. PALs) and a routing network that connect SPLDs with each other.

[1] Other HDLs may have similar functionality, but the author is familiar with VHDL, so every reference to a description language will be VHDL-oriented.

FPGAs and CPLDs have many common characteristics but the few existent differences suffice for them to be in separate categories. FPGAs have a similar general structure to CPLDs, with two major differences; one is that the basic building unit of an FPGA is not an SPLD, but a more complex unit often called "***logic primitive***" or "***logic element***". The second difference is that FPGAs have many more storage units than CPLDs, many of which are due to the D flip flops connected with each logic element.

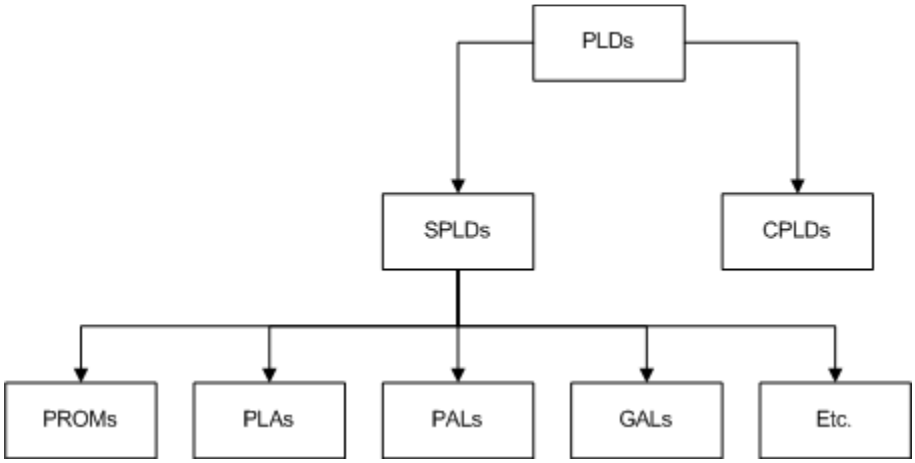


Figure 1: Basic categorization of PLDs

In **Fig. 2** we can see a basic logic element of an FPGA, which consists of a 4-to-1 ***Look-Up Table (LUT)***, a 2-to-1 multiplexer (MUX) and a D flip flop (DFF).

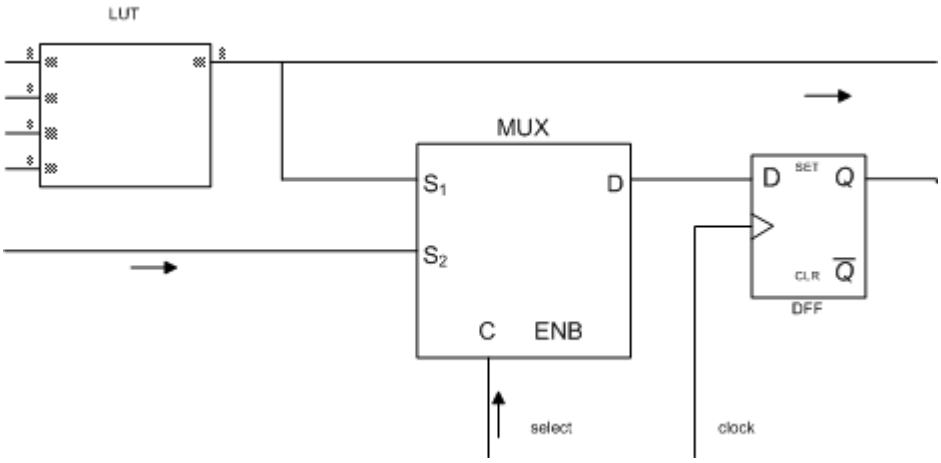


Figure 2 : A basic logic element structure

In the figure above, the multiplexer selects between the output of the LUT and an immediate input, the DFF provides the capability for the logic element to be synchronously read and the LUT plays the role of a function generator. This functionality is achieved by configuring the LUT to provide as output the output of a logic function. For example, if we wanted the LUT to perform the function below:

$$y = (a \,\&\, b) \mid !c$$

(where **&** denotes the logical AND, **|** denotes the logical OR and **!** denotes the logical NOT) then the LUT should be configured as below [7]:

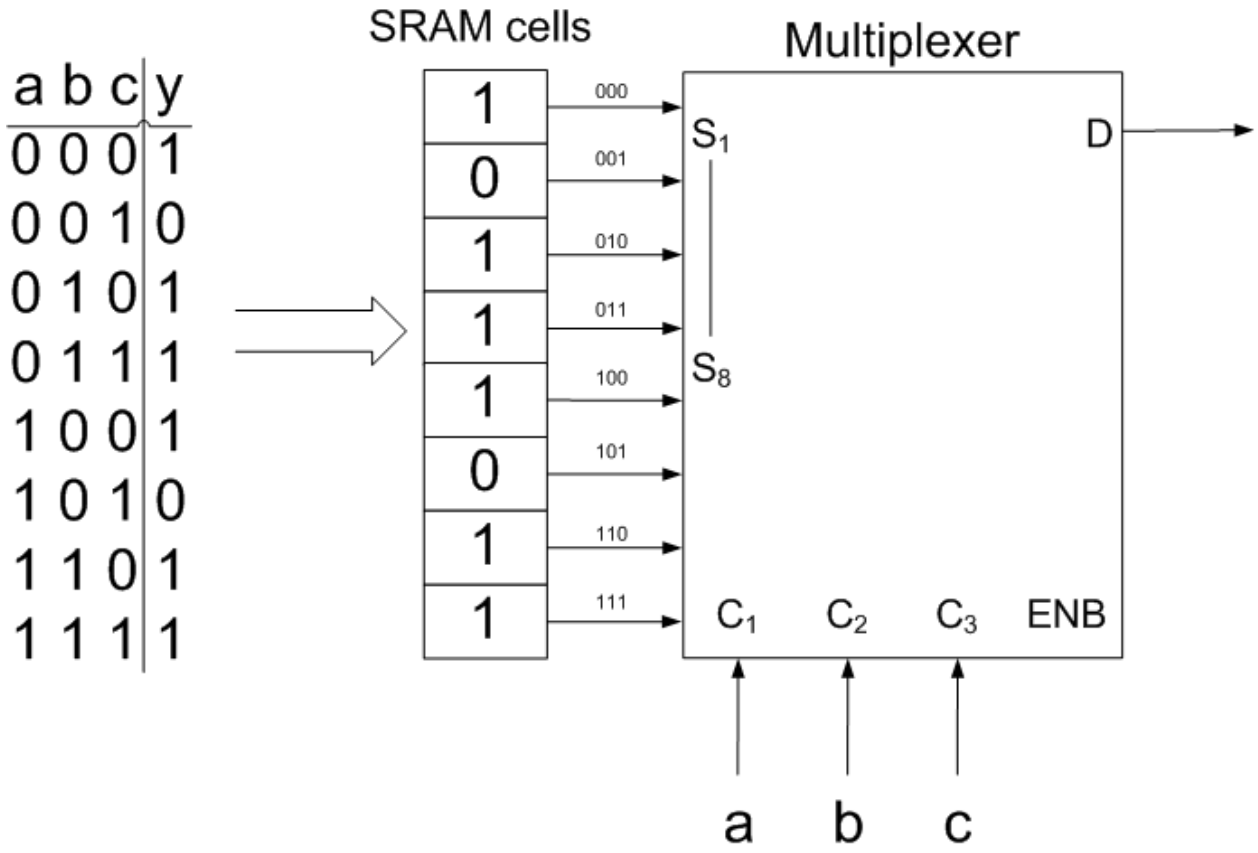


Figure 3 : Configuring a LUT

The LUT consists of several SRAM cells and a multiplexer, which selects the output from the cells and gives it to the LUT output, according to the selection signals *a, b* and *c*. Thus, the LUT can perform as a function generator for any logical function with any inputs (in **Fig. 3** with three inputs) to one output. LUTs can also be used as different functional units, depending on the FPGA. For example, the Spartan 3 FPGA family from the Xilinx Company provides the ability for the LUT to be used as a function generator, as a 16 bit shift register or as a simple 16 bit Distributed RAM[2] unit.

Depending on the FPGA family, several logic element configurations have been deployed; usually they are packaged by two or four, constructing a programmable logic block. Programmable logic blocks come with several names, vendor depended; Xilinx uses the term ***Configurable Logic Block (CLB)*** whereas Altera uses the term ***Logic Array Block (LAB)***. In order to avoid the terminology clutter, we will use the Xilinx introduced terminology. A package of two logic elements makes a ***slice***. Two slices are

[2] We will explain more on Distributed RAMs on chapter four.

combined to create a single CLB. FPGA structures resemble closely a lattice, where CLBs are interconnected by a routing network through **programmable interconnects (PICs)** (**Fig. 4**).

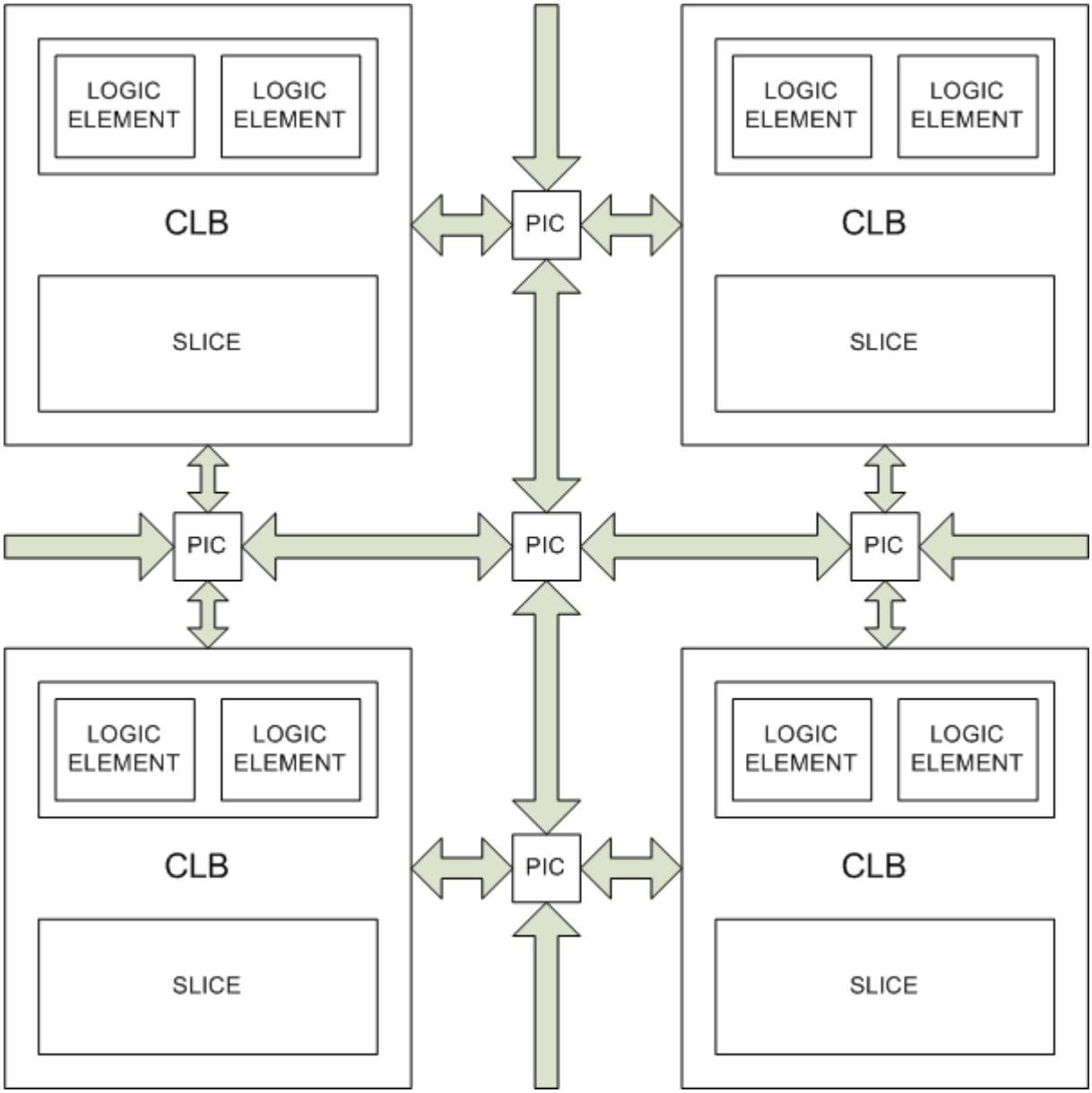


Figure 4 : Example of an FPGA structure

Depending on the number of features the CLB provides, the FPGA is said to be ***fine-grained*** when CLBs are very small, ***medium-grained*** and ***coarse-grained*** when CLBs are medium or big size respectively.

An FPGA is also divided to the reconfigurable portion of the FPGA, or ***fabric***, and the processors that it may have. Today's FPGAs have the ability to provide one or more processors in their fabric, offering this way the opportunity to create large scale SoC designs. Processors can be hard processors,

meaning that they are made directly at the integrated circuit layout of the FPGA, or soft processors, meaning they can be configured and downloaded to the FPGA according to the user's needs. For example, Xilinx Virtex II Pro FPGAs have two PowerPC hard processors [8] and can be configured to include several instances of the Microblaze or Picoblaze soft processors [9], [10].

Finally, the technology upon which the FPGAs are based is worth mentioning. FPGAs can be fuse-, antifuse-, flash- or SRAM-based. The names come from the way the FPGA is manufactured. ***Fuse-based*** FPGAs have fuses in the place of the programmable interconnects (PICs, see **Fig. 4**) which are burned in order to program the FPGA. The fuse-based FPGA once programmed, retains its status; even after the voltage supply is removed. Same goes for the ***antifuse*** technology, except that instead of burning the fuses, antifuses are "grown" in the desired connections and afterwards retain their status. With ***flash-based*** technology the configuration of the FPGA is stored in flash portions. Those portions have the ability to be reprogrammed (whereas the fuse and antifuse can not be reprogrammed) and to retain their configuration status even when the power supply has been removed. Finally, ***SRAM-based*** technology uses SRAM cells to store the configuration of the FPGA. This gives the advantage that the reprogramming of the FPGA takes place very fast; however, once power supply is removed the configuration is lost. Nevertheless, the SRAM technology is at the forefront of technology making the FPGAs ideal platforms for implementing ideas in the latest trend of technology [7]. All of the above, plus many other factors that concern the manufacturing process, have rendered the SRAM-based FPGAs to be the dominant technology today.

The above analysis indicates clearly one important point; since FPGAs are fast, cheap and provide all the above capabilities (rapid system prototyping, SoC, small time to market and non recurring engineering etc) they are ideal for implementing old and new concepts. This goes from commercial products to computational experiments or newly devised algorithms. Given that more scientific groups follow this direction every day, we need to closely study new ways of extending the capabilities offered by the FPGA technology. One of these capabilities is how to implement high-level concepts to fabric, without the time consuming procedure that consists of the transformation from high- to mid-level and from there to low-level implementation; the most obvious example of such a capability is the concept of recursion.

CHAPTER 2: PREVIOUS RESEARCH ON RECURSION IN FPGAs

This chapter will present previous research made in the field of recursion implementation in reconfigurable hardware. It is divided in three major sections: in the first section we describe the work of Maruyama, Takagi and Hoshino who implemented the knapsack problem and the knight's tour. In the second section we describe the work of Ferizis and ElGindy, who implemented the recursion using pipeline and runtime reconfiguration. In the last section we describe the work made by Sklyarov et. al, who introduced the theory of hierarchy graph schemes, a language that facilitates the implementation of recursion in hardware.

3.1 Maruyama, Takagi and Hoshino

The group has implemented two recursive algorithms; the knapsack problem and the knight's tour. Their research was focused mainly to the optimization of the recursive calls and loops that exist in software recursive functions [11]. In their research, they studied two implementation techniques that can be applied to recursive algorithms; the first, multi-threaded execution, has been implemented to the knapsack problem and the second, speculative execution, has been implemented to the knight's tour algorithm.

In the knapsack problem, they presented the main function of their software implementation and optimize the last recursive call; they transformed it to iteration because it's a tail recursion. Then they broke the algorithm implementation to several main portions, that were used as a reference to the hardware implementation structure (see **Fig. 5**).

```
loop:
    cal-A:
  if (cond-A)
        recursive-call();
  cal-B;
  if (cond-B)
        goto loop;
```

Figure 5 : Implementation structure separation example from [11]

Using the basic blocks depicted in **Fig. 5** they have implemented a simple recursive hardware algorithm, which comprised of the blocks and a stack. The purpose of this implementation was to have a reference design for a later, multi-threaded implementation. The step by step recursive execution of the knapsack problem is shown in **Fig. 6**. In this sequential execution of the algorithm we can follow the order of how the stages become active; stages from S0 to S3 activate sequentially, on S3 we choose whether the recursive call will be made or not; then execution continues sequentially from S4 to S7. The logical mapping from the pseudo-code presented in **Fig. 5** and the stages in **Fig. 6** is that S0 and S1 execute the blocks before the condition A (cal-A). We have two stages in the dataflow (**Fig. 6**) because in the original code presented in [11] the pre-condition actions were two. Continuing on **Fig. 6**, when the S2 stage is executed, then the decision is made whether the condition A is true or false, and, thus, if a recursion call will be made. If the condition turns out to be true, then on S3 the current local variables

(which are called *environment variables* in [11]) are pushed on to the stack and the newly calculated local variables are forwarded to the S0 stage; then execution restarts from S0. If, on the other hand, condition A is found to be false, then execution continues from S4 until it reaches stage S6. There, the condition B is checked. If it is found to be true, then the current local variables are forwarded from stage S7 to S0 and execution is re-initiated from there. If condition B is found to be false, then the stored environment variables are popped from the stack and execution continues from the stage S4.

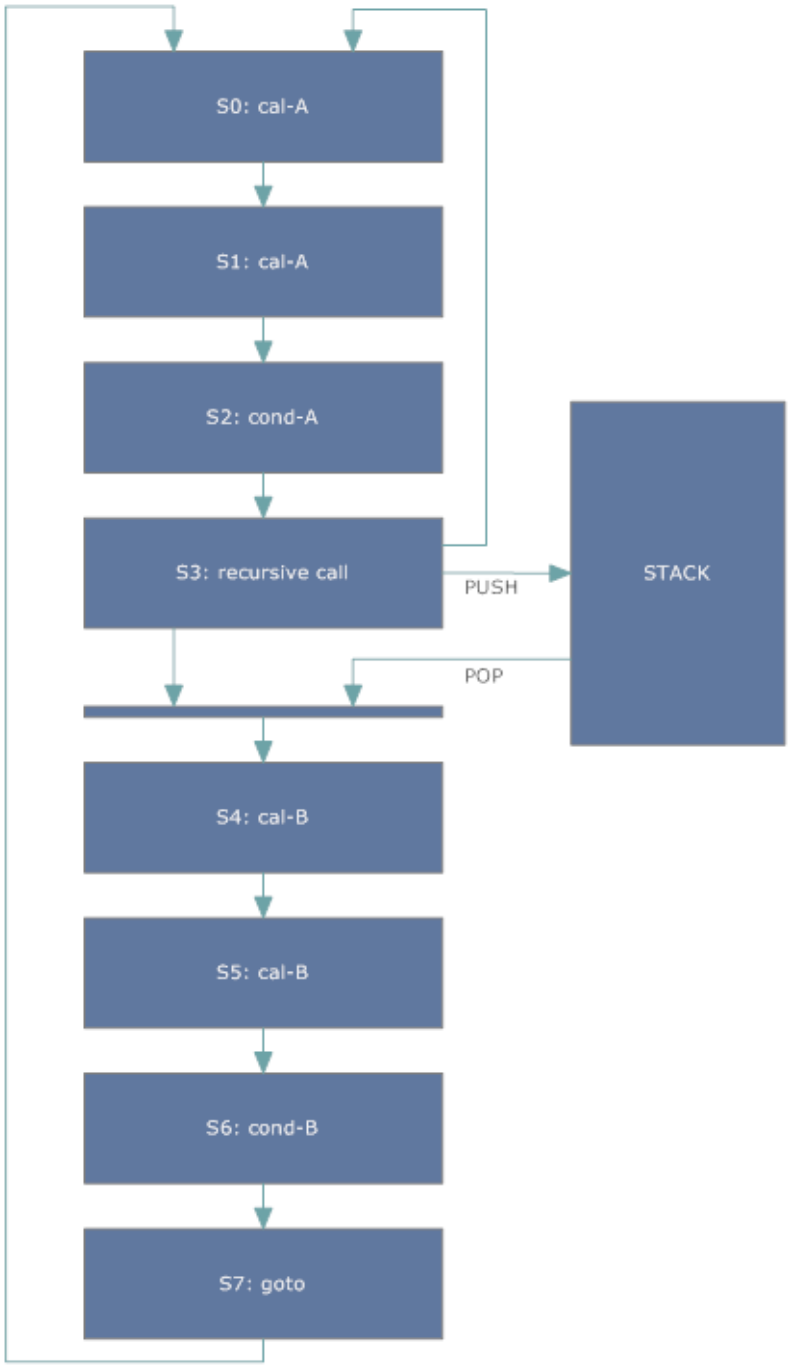


Figure 6 : A sequential recursive execution of the knapsack problem

The stage transition flow diagram is presented in **Fig. 7**. It is presumed than when on stage S2 for the first time, then the condition A is true, when on S2 for the second time then condition A is false, when on S6 for the first time then condition B is false and when on S6 for the second time condition B is true. Thus, the sequence of stages is as follows: a push is made from the first S3 stage, execution is continued uninterrupted from the second S3, a pop is performed from the first S7 and from the second S7 we go to the stage S0.

Figure 7 : Sequential execution flow diagram for the knapsack problem

In **Fig. 8** the sequential execution has been replaced by the multi-threaded model, where multiple threads of the same tree search can be executed on the same circuit. The concept was that by modifying the circuit that performed the sequential execution, a transit to a new structure could be made that would favor multiple threads of the search. In this model the same stages as in **Fig. 6** are used, except that the execution sequence differs; when condition A in S2 is true and in S3 a recursive call is made, then stages S4 through S7 are executed using the current environment variables. Then:

1. If S7 is idle when we are in S3, the new environment variables are forwarded to stage S0. This happens because if S7 is idle, then S0 will not be occupied on the next clock cycle.
2. If S7 is not idle, then the new environment variables are pushed to the stack.
3. If both S3 and S7 are idle, then the old environment variables from the stack are popped and forwarded to the S0 stage and execution continues from there.

The multi-threaded execution has the positive trait that all stages are active at the same time, processing different data according to the three *executional conditions* described above. The stage transition flow diagram described above is shown in **Fig. 9**. The execution begins from stage S0. When it reaches stage S3 the execution continues in two branches: on the first (sequence one) it continues normal execution to the next state (S4) using the current environment variables. The second branch (sequence two) signifies that a recursive call was made with the new environment variables and execution flow restarts from S0. In **Fig. 9** the recursion is shown with the arrow from sequence one to sequence two. When the sequence one reaches S6, then three things happen simultaneously:

1. Since at the same time S0 and S7 are idle, a pop is performed and execution is resumed from stage S0 (sequence three)
2. On the second sequence which is on stage S3 at that time there was a “true” result for the condition for recursion, and a push is performed. Thus, execution is resumed normally from stage S4 (shown in transition point 1, from S3 at sequence two to S4 at sequence one).
3. On the sequence two, the execution continues normally using the current environment variables, and proceeds to the stage S4.

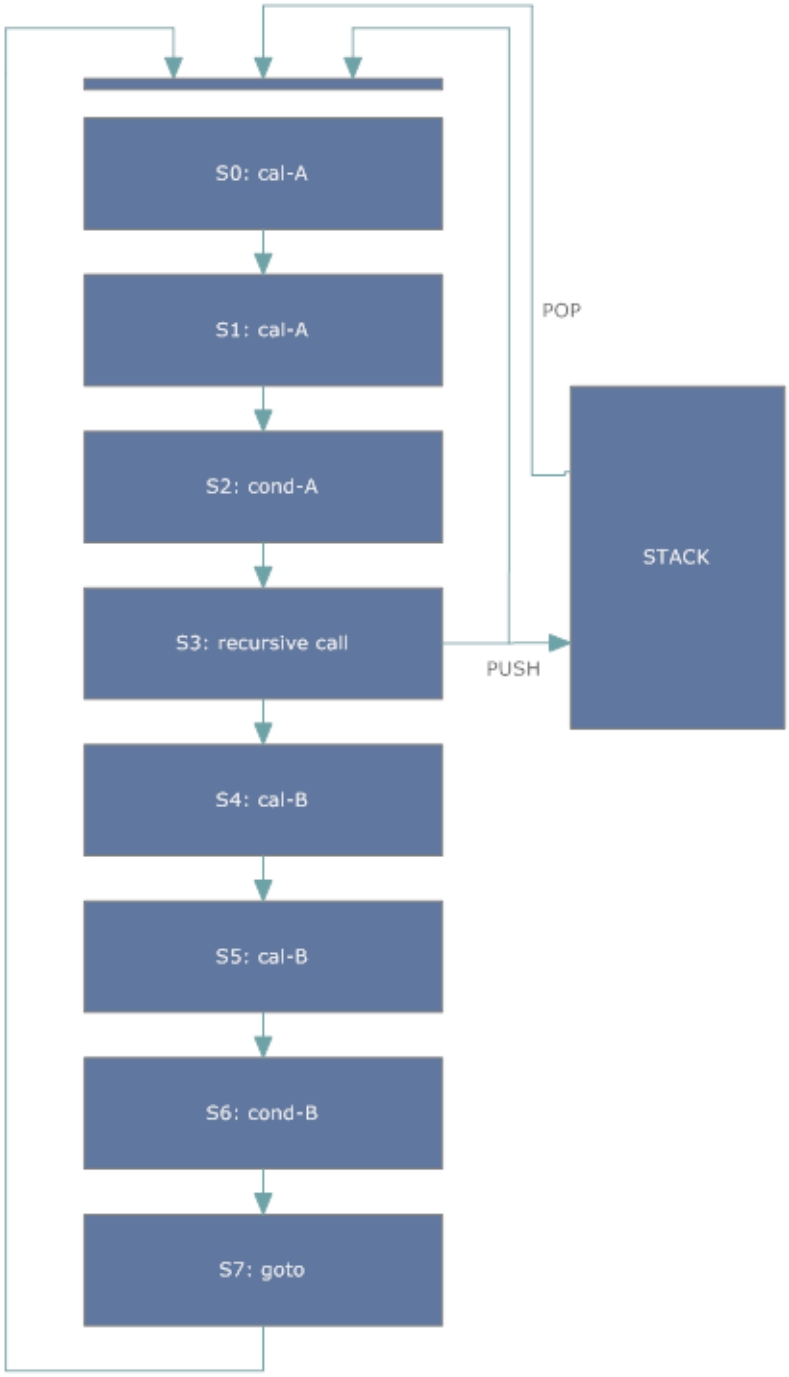


Figure 8 : Multi-threaded execution flow diagram for the knapsack problem

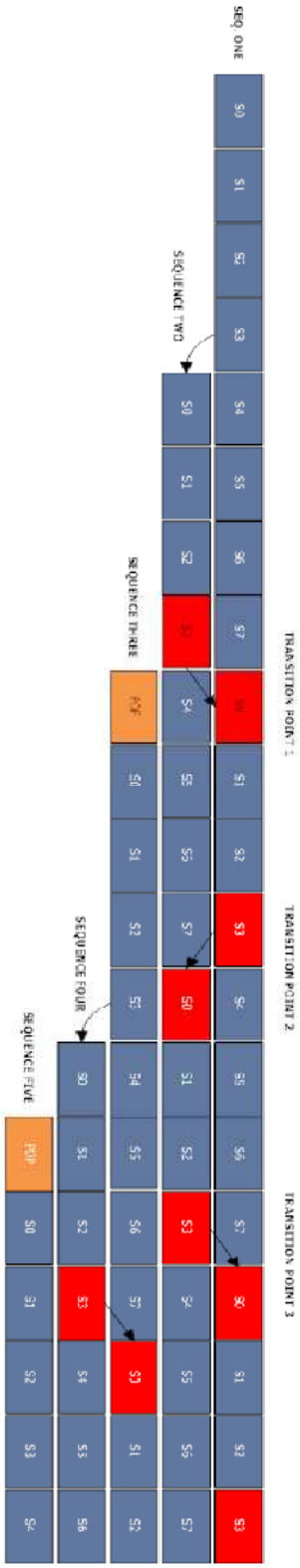


Figure 9 : Multi-thread execution stages transition

It is important to observe that transitions from stages S3 to S0 using push operations (indicated at transition points in **Fig. 9**) happen because the first executional condition is satisfied and pop transitions happen because of the third condition.

In order to implement the needed isolation among the different stages, we need to insert pipeline registers. This way a pipeline architecture is constructed, which increases the throughput of the circuit. It is stated in [11] that the more complex the programs we implement, the more pipeline stages are needed and the higher performance is expected. The maximum speedup achieved from this method is analogous to the depth of the pipeline. The pipeline architecture does not introduce signinficant area consumption to the sequential circuit, and there should be no overhead in hardware size and clock speed. The results presented by this method are shown in **Table 1**. Comparison is made with an average of 100 runs for the knapsack problem, using an Ultra-SPARC at 200 MHz. The different pipeline stages are considered because of the capability to parallelize some operations; the 8 stages are the normal pipeline datapath, whereas at the 4 stage architecture the S0 and S4, S1 and S5, S2 and S6, S3 and S7 are executed at the same time.

Table 1 : Comparison of Computation Speed for the Knapsack Problem

Implementation	Speedup	Time
Ultra-SPARC 200 MHz	1.00	0.84 secs
Sequential execution (8 stages) 35 MHz	0.74	1.13 secs
Sequential execution (4 stages) 35 MHz	1.55	0.55 secs
Multi-thread execution (8 stages) 35 MHz	6.68	0.13 secs
Multi-thread execution (4 stages) 35 MHz	6.70	0.13 secs

The method that was used in order to solve the Knight's Tour was the speculative execution. This means that for every condition met, the processor calculated both taken and non-taken conditions, performing the classic speculative execution of the algorithm. In **Fig. 10** we see a similar block identification to the knapsack program, that is used for the composition of the different implementation stages. In **Fig. 11** the stage transitions for the speculative execution are presented. In this method there is a loop index, the variable ***i***, which is incremented in each clock cycle. Stages S0 to S2 execute continuously for different values of ***i***. When condition A becomes true at stage S2, then all the other computations (for ***i+1***, ***i+2*** and ***i+3*** stages) are disregarded and recursive call is made to S3. In this case, the current environment variables are pushed to the stack, and the new environment variables for the recursion call are forwarded to S0. Index variable ***i*** is re-initialized to zero (0) and execution begins. When condition A is false for all values of ***i***, then environment variables are popped from the stack at stage S3 and execution continues at stage S4. The stage transitions are shown in **Fig. 12**. The comparison results for a Knight's Tour search, for chessboard size of 8x8 and an Ultra-SPARC at 200 MHz, are shown on **Table 2**.

```
for (i=0; i<N; i++)
{
  cal-A;
  if (cond-A)
  {
    recursive_call();
    cal-B;
  }
}
```

Figure 10 : Structural identification for the Knight's Tour program

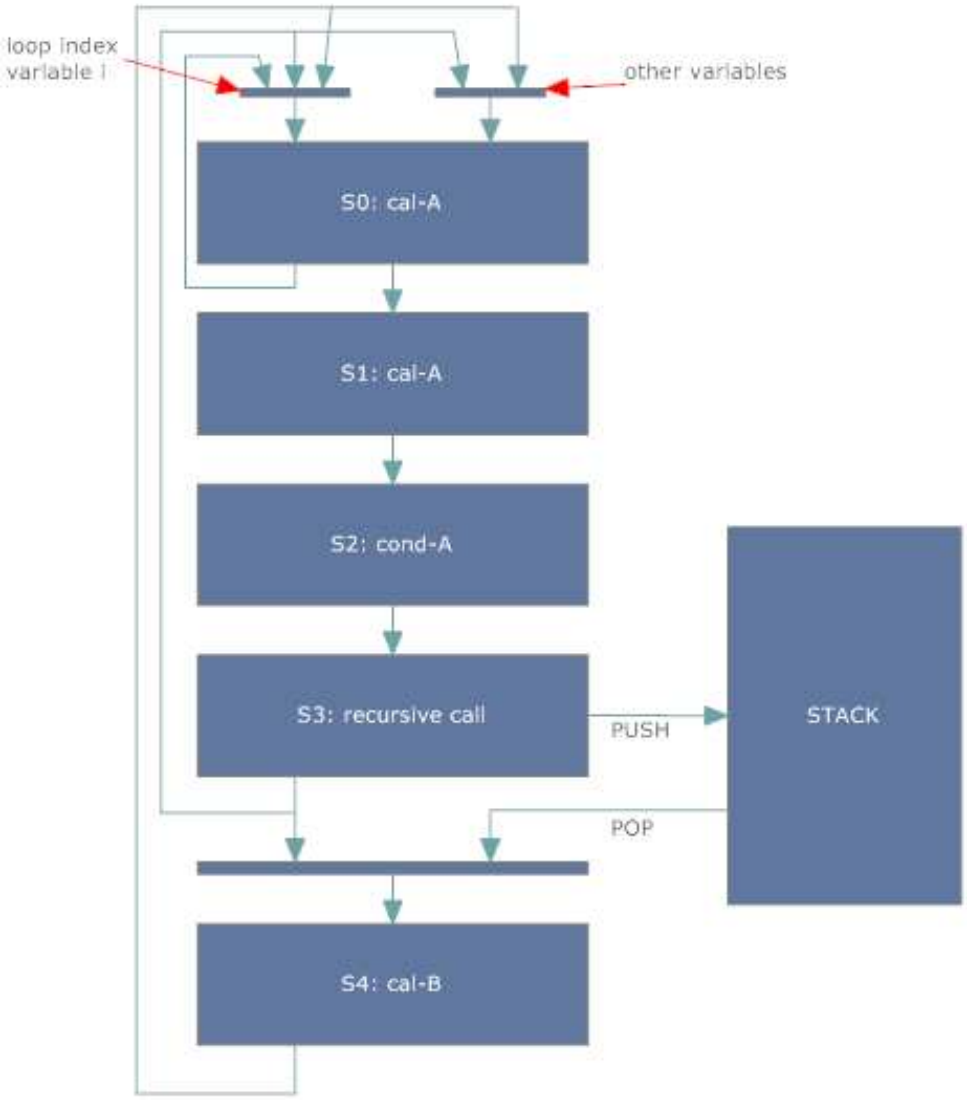


Figure 11 : Speculative execution stage transitions flow diagram

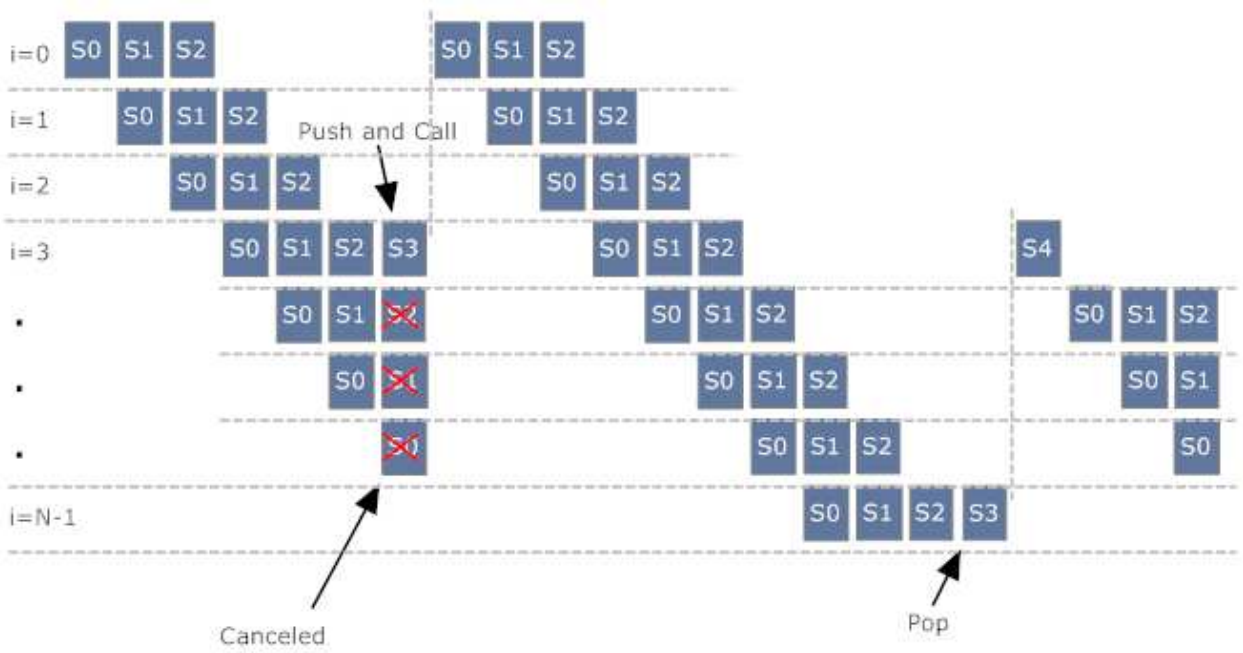


Figure 12 : Stage transition for the Knight's Tour speculative execution

Table 2 : Comparison of Computation Speed of Knight's Tour problem

Implementation	Speedup	Time
Ultra-SPARC 200 MHz	1.00	15.1 secs
Sequential execution 31 MHz	1.72	8.78 secs
Speculative execution 31 MHz	4.06	3.72 secs

3.2 Ferizis and ElGindy

In this study, a different approach has been considered in order to implement recursion in hardware. Ferizis and ElGindy [12] divided a recursive function in three main parts, implemented those parts in reconfigurable hardware using pipeline and then used runtime reconfiguration to achieve theoretically unlimited recursion depth.

The division of the recursive function was based on the theoretical background of the recursion that was introduced on *Chapter 1: Fundamental Concepts*; every recursive call is divided in two parts: the recursive part – where all the code that makes a recursive call resides – and the non-recursive part – where the base case code resides. In [12] this notion was extended to a further step; the recursive step was divided in two separate functions. The first function is called ***pre-recursive*** and the second one is called ***post-recursive***. The first function includes all the operations that are executed before the recursive call is made and the second includes all the operations that are executed after the recursive

call. Because the two functions have the property that they process the same set of data, in [12] state data were transmitted from the pre-recursion to the respective post-recursion operation.

The algorithm to identify which parts constitute which function among the pre-, post- and non-recursive possibilities uses a flowgraph introduced in [13] by Muchnick (with minor differences). Every recursive function is in a block that has no other statements; this block is named ***recursive block***. A flowgraph example for the Fibonacci function is shown in Fig. 13.

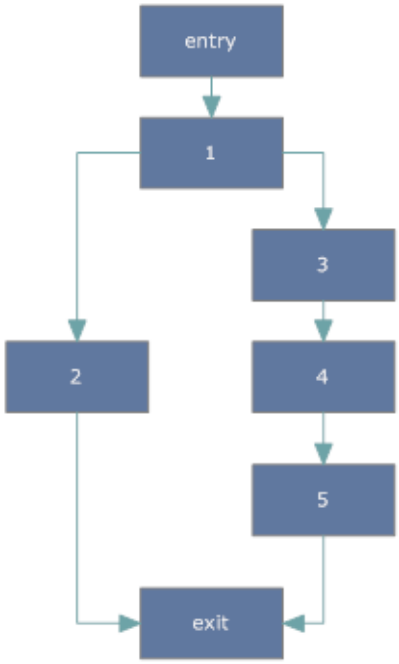


Figure 13 : Flowgraph for the recursive Fibonacci function

In this figure, five (5) blocks with operations are depicted, along with the *entry* and *exit* blocks. In order to understand the graph decomposition and identify the parts that constitute the pre-, post- and non-recursive function, in [12] a graph G' is defined, which results from the graph shown in **Fig. 13** if all recursive blocks are removed. Then they define:

1. **Non-recursive function**: this function ($G_{non\text{-}recursive}$) is derived from the G' graph and contains all blocks that are reachable from the *entry* block and can reach the *exit* block.
2. **Pre-recursive function**: this function ($G_{pre\text{-}recursive}$) depends on the graph shown in **Fig. 13**, if we remove all blocks except those that are reachable from the *entry* block and the blocks that can reach any other recursive block, including the recursive blocks.
3. **Post-recursive function**: this function ($G_{post\text{-}recursive}$) depends on the graph shown in **Fig. 13**, if we include all blocks that can reach the *exit* block and are reachable from any recursive block.

According to the above definitions, the functions include the following blocks:

- G'{entry, 1, 2, 5, exit}
- $G_{non\text{-}recursive}${1,2}

- $G_{pre\text{-}recursive}\{1,3,4\}$
- $G_{post\text{-}recursive}\{5\}$

The *pre-recursive* and *post-recursive* functions are connected together so that state data can be transmitted between each member of a pair; this connection is show in **Fig. 14**.

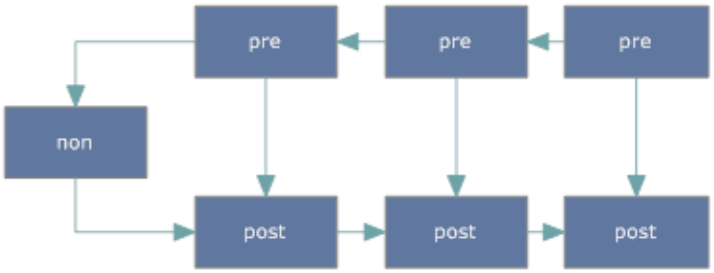


Figure 14 : Connection between pre- and post- recursion functions

For the method to be applied, a new term has been defined; the ***process growth rate***. It is defined as the *ratio of work done between the root node and its children; it is the sum of the work done for each child node divided by the work done by the root node* [12]. This ratio is given by the mathematical formula shown below:

$$process\ growth\ rate\ = \left\lceil \frac{f(D)}{\sum_{i=1}^{N} f(Di)} \right\rceil$$

The ceiling function of the above equation is the number of function units that need to be allocated to compute the result for all the children nodes with the same throughput as the previous node. The symbols D and Di are shown in **Fig. 15**. They represent the D sized input that is received by a node in a recursive tree; the node has N children where the i^{th} child receives Di sized input.

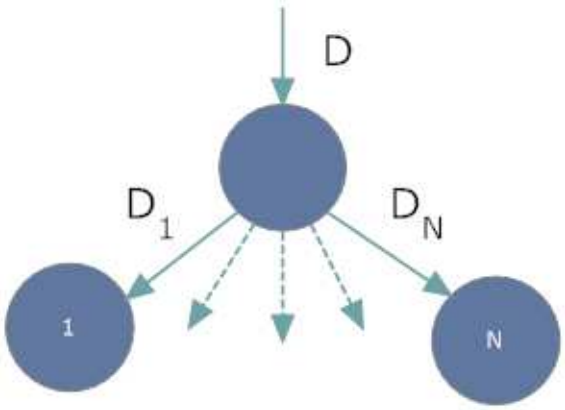


Figure 15 : Node with D size input and N children

Algorithms with a process growth rate of one require a single function unit to be allocated per level of recursion. This reduces the complexity of mapping the function to hardware to that of a recursive function with a single recursive call. Such algorithms must match four criteria:

1. f(A) ≥ f(B), for every A≥B
2. D ≥$\sum_{i=1}^{N} Di$
3. f(0) = 0
4. f'(A) ≥ f'(B), for every A≥B

The symbols used in the equations above are not clarified in [12]. However we assume that A and B represent the functions' domains, N is the number of children a node can have and f' is the derivative function of f. Two major problems arise from the concept illustrated above; the impact of the reconfiguration and the placement and routing. In order to reduce the impact of the reconfiguration, a circuit was implemented that was used to predict the need for reconfiguration. The circuit managed to reduce the time to reconfigure the device to a minimum, which in theoretical terms was 15ms in a XC2V1000 Xilinx device. The reconfiguration prediction circuit could manage with great accuracy the reconfiguration needs, especially when it was used with functions with predictable behavior; for example the Fibonacci sequence. In cases where the algorithm had not a predictable behavior, then between the best and worst cases (i.e. in quicksort the best case is O(log(N)) and worst case is O(N)) the case with the least FPGA space demands was assumed for the reconfiguration prediction. In this case, an extra runtime reconfiguration stage is allocated at all times, which reduces the negative impact but still results in a system stall while waiting for reconfiguration to occur.

The results from this method were presented in [12] and showed a speedup of about six (6) for the quicksort algorithm, for about eight thousand (8.000) samples. Nonetheless, the metrics of these results are in clock cycles without explicit account of clock period for post placed and routed designs.

3.3 Sklyarov

Valery Sklyarov proposed the implementation of recursive algorithms to reconfigurable hardware through the use of Hierarchical Graph Schemes (HGS) [14][15]. The concept in his proposal is

to divide the algorithm to a discrete number of modules (labeled z_i, i.e. z_1, z_2 etc), each of which will have a number of discrete states (labeled a_i, i.e. a_1, a_2 etc). There are three separate stacks, shown in **Fig. 16**, one to store the current module executed (the ModuleStack), one to store the current state of the current module (the StateStack) and one to store the data of each operation (the DataStack). There is also a combinational circuit, which is connected to the three stacks and operates on the input that it receives from the stacks, producing the appropriate output. It is worth noting that the two stacks (ModuleStack and StateStack) use the same stack pointer.

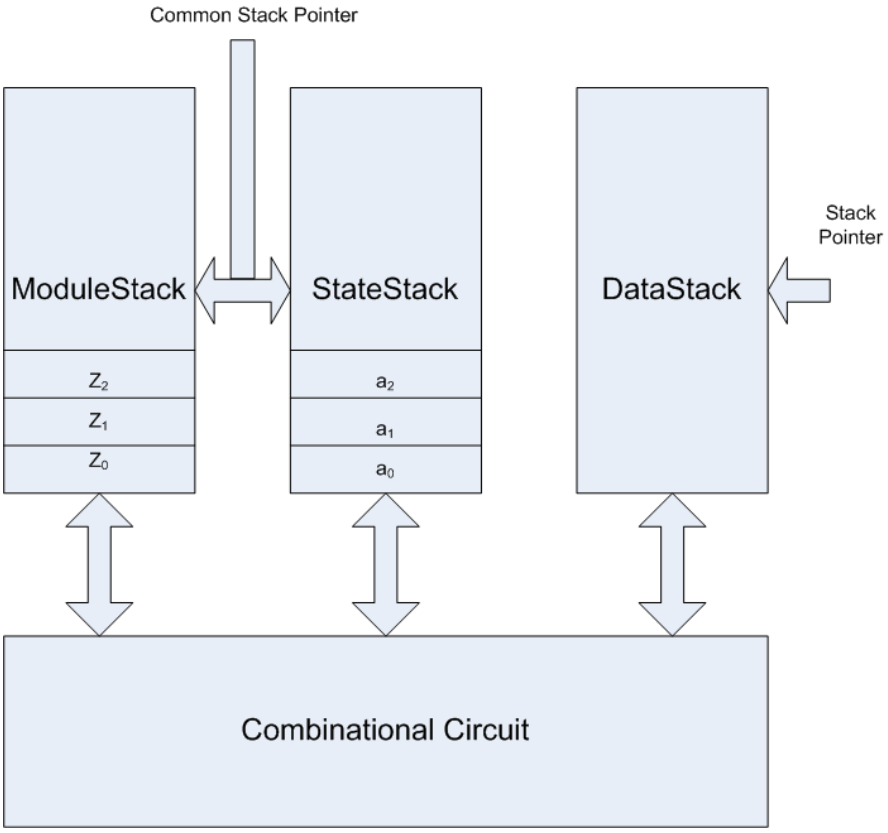


Figure 16 : General structure of a recursive algorithm implementation

Every module has its states (a_i), its input (x_i) and output (y_i). Each module begins with state a_0, which is the Begin state, and ends with the End state, which is labeled according to the module's number of states (in this example a_3). Two examples are shown in **Fig. 17** and **Fig. 18**. The module (z_1) depicted in **Fig. 17** is simple because it makes references to z_2 and z_4, whereas the module (z_2) depicted in **Fig. 18** is recursive because it contains a self-reference.

The function of the circuit is as follows; at every state the module produces the outputs (y_1, y_2, etc) that are inputs to the combinational circuit. At every decision point, the module's inputs (x_i) are the outputs of the combinational circuit. Every time a module is called from another module (i.e. module z_2 from module z_1) then the common stack pointer is incremented by one, the new module is saved at the

ModuleStack, the new Begin state is also saved at the StateStack and the new module is executed. Every time a module state is changed, the new state is stored at the StateStack, overwriting the previous state stored at the same location. Using this concept, the recursive algorithm can be easily described in hardware, as it is easy for the circuit to determine new states, modules and recursive calls using the stacks.

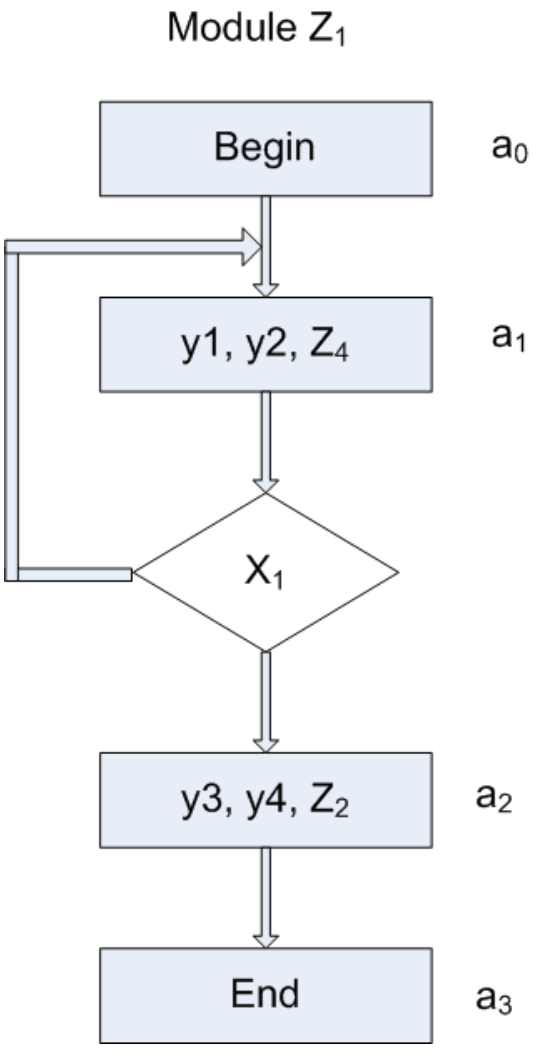


Figure 17 : Example of Z_1 module

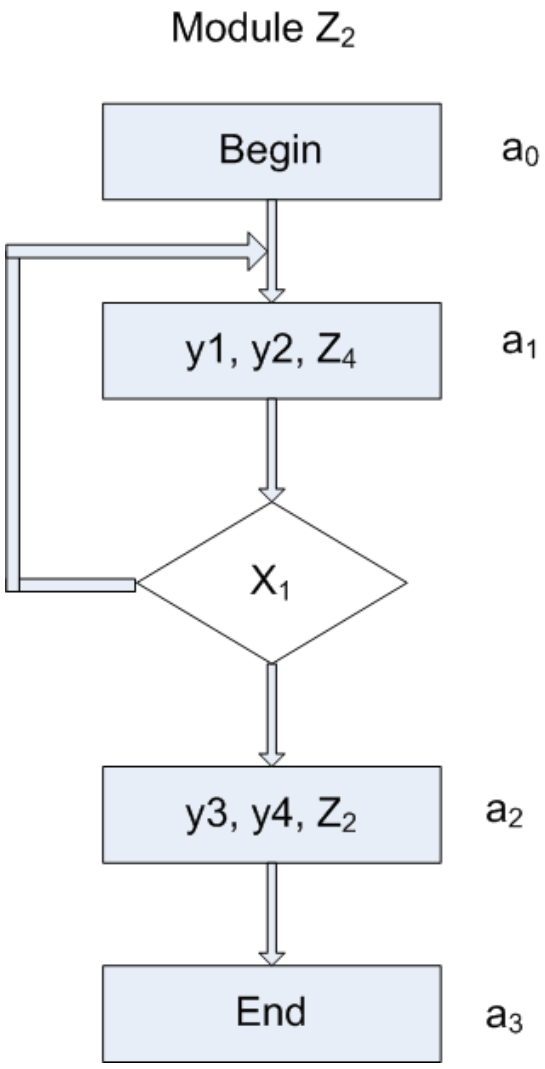


Figure 18 : Example of Z_2 module with recursion call

All the proposals mentioned in this chapter gave a significant base for further development. Although the approaches followed were very different in nature, the common characteristic was that each proposal tried to solve a specific subproblem of the recursion i.e. processing throughput, stack elimination or designing process. Thus, solution proposed so far cannot solve the problem of recursion implementation as a whole. However, the theory presented by Sklyarov is considered the most important, because he tried to solve the problem from the designing level. In the next chapter we will analyze further the theory of Sklyarov and explain the advantages and disadvantages of his theory.

CHAPTER 3: SKLYAROV ALGORITHM ANALYSIS

In this chapter we analyze further the algorithms proposed by Sklyarov. We shortly present the theory around the hierarchical graph schemes, the recursive form of the schemes which are the recursive hierarchical graph schemes, and finally we study some implementation results presented in international literature.

3.1 Hierarchical Graph Schemes (HGS)

A **hierarchical graph scheme (HGS)** is a directed graph which contains nodes of rectangular and rhomboidal shapes [15]. Every HGS has an entry point, which is represented by a rectangular node named **Begin**, and an exit point, which is also a rectangular node named **Exit**. The circuit described by the HGS is able to perform **micro operations** $y_1, y_2, \ldots, y_i$. These micro operations are combined to create the set Y of **micro instructions** Y that the circuit can also execute. Formally, a micro instruction is given by: $Y = \{y_1, y_2, \ldots, y_i\}$. Except for the nodes **begin** and **exit**, each HGS can contain other rectangular nodes, which contain micro instructions or micro operations. A micro operation is an output signal that causes the circuit datapath to perform a simple action.

A **macro operation** is given by the set $z_1, z_2, \ldots, z_q$ and is described by another HGS of a lower level. A set of macro operations constitute a **macro instruction**, that is $Z = \{z_1, z_2, \ldots, z_q\}$. In [15] it is assumed that each macro instruction includes only one macro operation.

There is also defined the set of **logic conditions** $X = \{x_1, x_2, \ldots, x_L\}$ and the set of **logic functions** $\Theta = \{\theta_1, \theta_2, \ldots, \theta_m\}$. The set $X \bigcup \Theta$ is contained in each rhomboidal node. A logic condition is an input signal which presents the result of a test. A logic function is calculated by executing a set of predefined sequential steps that are described in a HGS at a lower level. Inputs and outputs from the nodes are directed lines (arcs) that have the same meaning as in a normal graph.

An example of all the above terms is presented in **Fig. 19**; we can see clearly all the factors that are used in the graph in order to create the circuit. Every state is given a distinct label. In this figure, states are given names from a_0 to a_8. In every state a_i we perform a micro instruction Y_i, that is composed by the micro operations $y_1, y_2, \ldots, y_i$. For example, in state a_2 the micro instruction Y_1 is composed by the micro operations y_1 and y_4. This essentially means that when execution sequence is in state a_2, the control signals y_1 and y_4 are activated in order to cause the datapath to perform an action. Another example is state a_3, where the micro instruction Y_2 is executed. The Y_2 is composed by the micro operation y_1 and the invocation of the macro operation Z_2; this macro operation is actually the invocation of another HGS graph which follows the same principles as depicted in **Fig. 19**. When on the first rhomboidal node, the logical condition X_1 is checked; whatever the value, the steps below the check point (as we look at **Fig. 19**) are executed. Each branch of the logical check constitute a logic function Θ – as described above. As we can see, the X signal in the check is an input signal whereas the path that will be followed, depending on the X signal, is the value of Θ.

A final observation is that HGS can be easily implemented in hardware using finite state machines (FSMs). The fact that instructions, operations and conditions are so straightforward in conception and implementation, the HGS is considered suitable for immediate translation from a graph

to a control unit – datapath solution. In [15] is defined that every FSM which implements a Hierarchical Graph Scheme, should be named **Hierarchical Finite State Machine (HFSM)**.

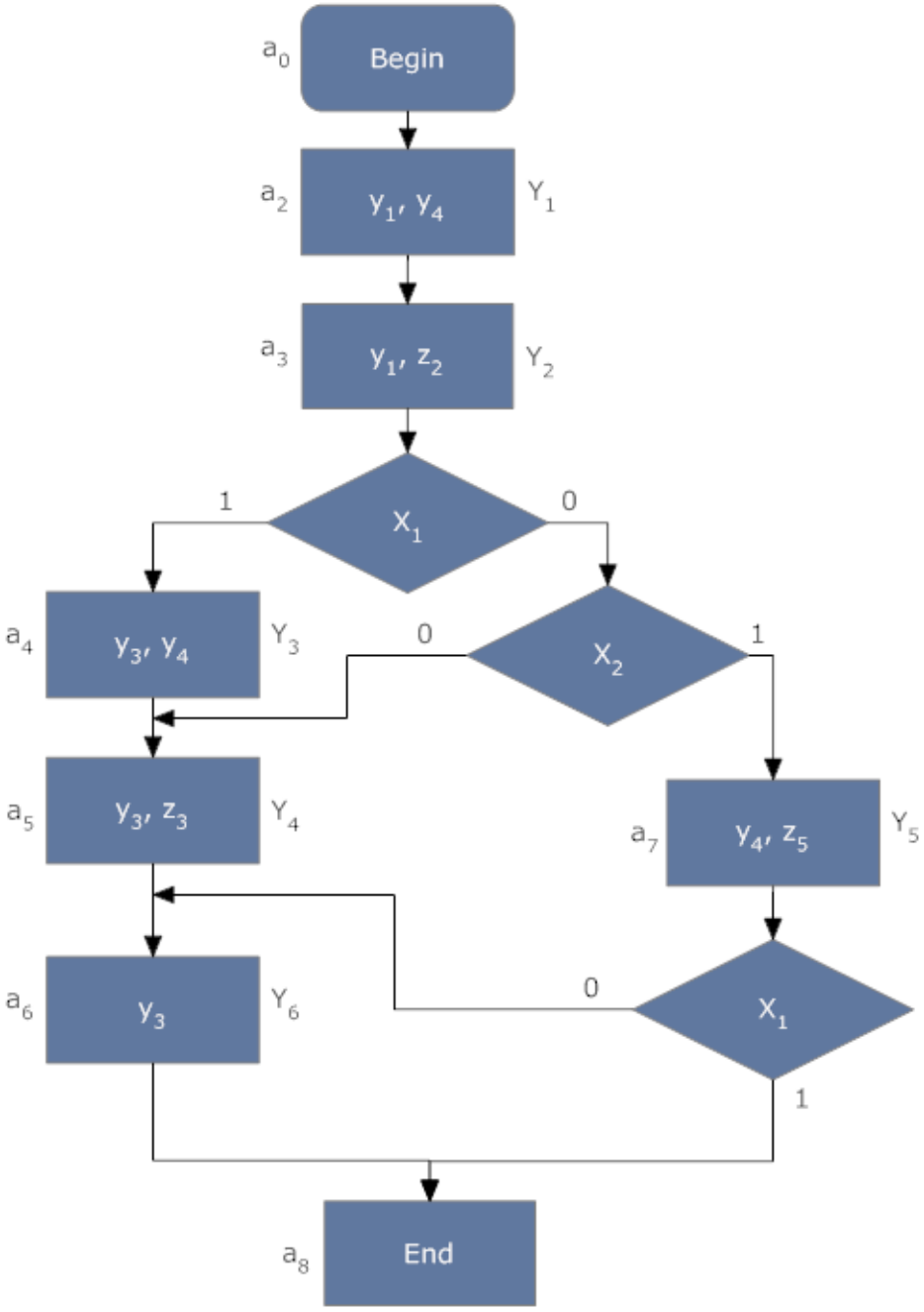


Figure 19 : A HGS example

3.2 Recursive Hierarchical Finite State Machines (RHFSM)

The concepts of HGS and HFSMs have been extended in [14], in order to better describe and implement recursion. Even though recursive algorithms can be implemented using HFSMs, there are some drawbacks in using this primary model; one of them being the book keeping overhead. In order to

supersede the disadvantages that follow the use of HFSMs, an extended concept has been proposed, which is the use of **Recursive Hierarchical Finite State Machines (RHFSMs)**. This concept can be applied to a wide variety of cases, but the most important application presented by Sklyarov et al. is the different tree search problems [16]. In this area, recursion is found to be extremely efficient.

The main example for the explanation of how the RHFSMs work is shown in **Fig. 20**.

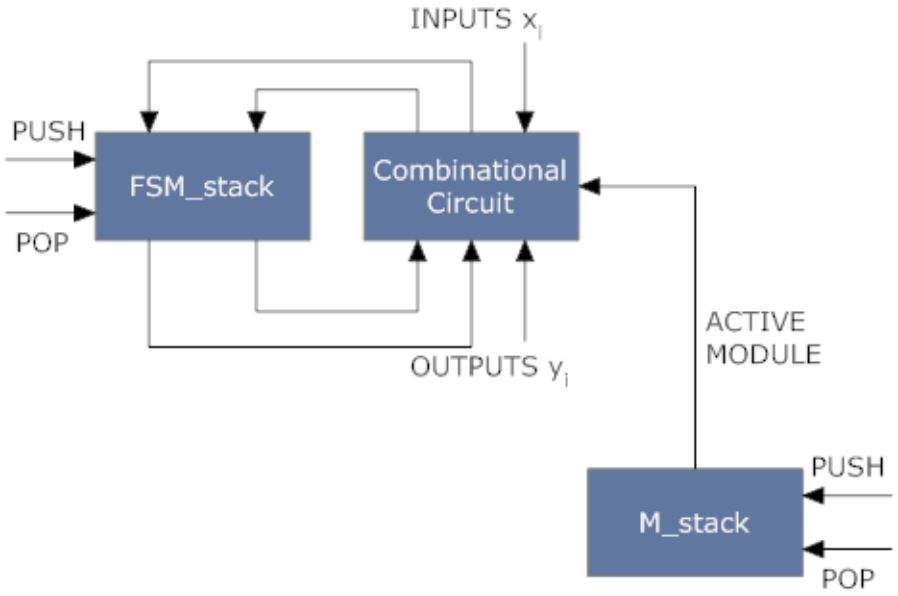


Figure 20 : Recursive Hierarchical Finite State Machine (RHFSM) basic structure

As was explained in the previous paragraph, an HFSM is decomposed in a control unit and a datapath. The same concept is followed for the RHFSMs. The **Fig. 20** should be studied having in mind the **Fig. 16**, which is more representative in terms of functional equivalence to the theory. More specifically, stack pointers for both FSM_stack and M_stack are the same, so the signal wires **PUSH** and **POP** in **Fig. 20** are practically the same.

The RHFSM is constructed by a combinational circuit and two stacks: the FSM_stack and the M_stack. The combinational circuit is responsible for accepting the inputs X_i and for producing the outputs Y_i; essentially it produces the state transitions in the active module indicated by the M_stack. The FSM_stack is responsible for storing the current state in the active module and the M_stack is used in order to store the current module we execute. It is important to understand that the stack pointers for the two stacks combine; they are the same. This is used probably for simplicity, since two stacks with different stack pointers would need a more complicated control unit. The common stack pointer has given the following functionality to the circuit: for every module change, the stack pointer is incremented. For every module state change, the stack pointer stays unchanged. Instead, the new state is stored at the same place as the current one, overwriting existing stored state data. This has the consequence that the FSM_stack is not a normal stack as we usually define it, since **PUSH** operations do not always result in a stack pointer increment.

In order to understand how the circuit in **Fig. 20** fully works, we shall use the example provided in **Fig. 17**. Note that in **Fig 17**, every state inside the same module must have a different name, whereas among different modules we can use the same state names. When we enter module Z_0 (**Fig. 17**), the M_stack stores the identifier[3] of Z_0 and the identifier of **Begin** state (a_0) in the FSM_stack. When we reach state a_1, the module Z_1 is called; this produces the appropriate input to the combinational circuit so that the stack pointer is incremented, the identifier Z_1 is stored in the M_stack and the state a_0 is stored in FSM_stack. Thus, the current active module indicated by the M_stack is Z_1 and states transit according to the combinational circuit. When Z_1 module is finished, then the stack pointer is decremented by one, indicating as active module the previous one (Z_0 in our case) and the current state as the state that we left from (a_1 in our case). From this state data (active module and current module state) the combinational circuit enters the rhomboidal node, which is a check. If X inputs are true, then the combinational circuit indicates as the next state to be the a_2; else the state a_1. The next state then, is stored in the FSM_stack, in the place where the previous state was stored; in our case, the next state overwrites the a_1 state. Finally, if the current state is a_2 then the module Z_2 is called, performing all the actions described above. When the Z_2 module returns, then we reach the **End** state, and control is transferred to the module that called the Z_0 module; if the Z_0 module was the first module to be executed, then the algorithm has been completed.

As stated in [14], the main differences between a HFSM and a RHFSM are:

1. The RHFSM does not require a code converter because of the module stack (M_stack). This is the reason why we can use the same state names in different modules.
2. The design is pure hierarchical in a RHFSM since now the HGS combinational circuit can be independently designed from other HGSs. In HFSMs the combinational circuit and the code converter depend on all the HGSs that describe the algorithm.
3. In the **Begin** and **End** nodes we can have micro operations in RHFSMs, whereas in HFSMs we cannot. This eliminates the need for dummy states.

Finally, RHFSMs need to have an execution unit that implements the recursive algorithms. This is because, except from the module and module state data that define the current and next state, there are also some local registers in the datapath that hold important values. These registers have to be stored when a call to another module is made, and restored when a return to the previous module is performed. All this procedure is performed by an execution unit. In the example of a binary tree traversal given in [14], the execution unit is shown in **Fig. 21**. In this figure we can distinguish a RAM, a local stack, a register and an output stack. This general execution unit can be used in any binary tree traversal. The register stores the RAM-addresses that store the local register values and the local stack stores the data (register addresses) that enable the circuit to traverse the tree. This means that the stack pointer from the local stack indicates the current module state and the contents of the local stack address that are pointed by the local stack pointer are loaded to the local register. The value of the

[3] The identifier cannot be the ASCII letters Z0, but usually a binary number in the form of one-hot or weighted binary which is assigned to the module during synthesis.

register is an address to the RAM; the address shows the appropriate data that are stored in the RAM. The pointed data are the stored values of the datapath registers for the current state.

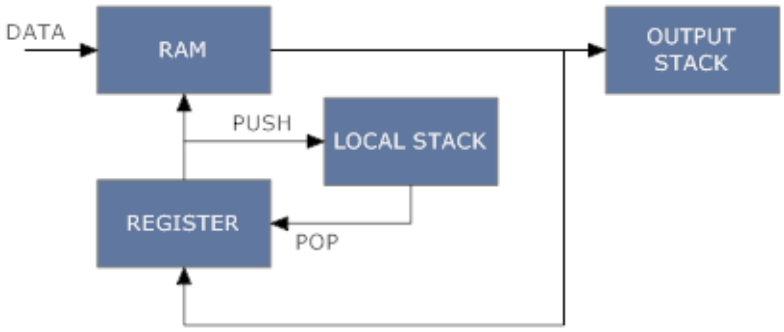


Figure 21 : Example of an execution unit for a binary tree traversal

3.3 Implementations Based on Sklyarov's Algorithm

In [14] an implementation is proposed for a binary tree sorting algorithm. The proposal contains a software model and a hardware model. The tree described in those models has nodes with four fields: the pointer to the left node, the pointer to the right node, the value of the node and a counter which indicates how many times a value occurs in the tree. The software model is given in the C++ programming language, introducing the class templates that form the binary tree, as long as the recursive functions that add nodes to the tree and traverse the tree. The hardware model comes with some restrictions:

1. There is no counter field.
2. All nodes have different values.
3. Values will be considered unsigned integers with a predefined size (number of bits).
4. There is no capability for dynamic memory allocation.
5. The circuit that adds nodes to the tree will construct the tree in a RAM with the root node being in the first RAM address (0x0..0);
6. The circuit that sorts the tree will put the results in the output stack (depicted in **Fig. 21**).

Using the architecture of **Fig. 21** we can fully implement the circuit that creates and sorts the tree. In [14] the full transformation from C++ to HGS and VHDL is given. The above methodology has been used in order to implement a Huffman coding circuit; the algorithm demands for a binary tree construction and the sorting of it.

Also, in [16] N-ary search problems have been studied using HGS and RHFSMs. Since this class of problems is generally solved by constructing a N-ary search tree, using the Sklyarov's algorithm for the decomposition of the problem, and the RHFSMs to implement the solution, makes the process easy. At the theoretical level, exhaustive search of a N-ary tree is an extremely slow process; some techniques need to be applied in order to reduce the search space. Some of them fall under the tree-pruning optimizations category. Another method, which is studied in [16], is the tree reduction. This method is

based on replacing the current situation with a simpler one, without sacrificing any feasible solution. The tree reduction method is not always feasible to be applied. Thus, other methods should be employed, most important of them being the divide-and-conquer technique. Using this method, a N-ary problem is divided to N simpler problems and each of them is solved separately. The main task is to find the minimum N number. In practice, the divide-and-conquer algorithm is applied to N-ary search trees following the steps below:

1. Apply reduction rules
2. Verify intermediate results. This leads to:
    a. Pruning the current sub-tree and backtracking to the nearest branching point.
    b. Store the current solution in case it is the best.
    c. Sequential execution of points 3 and 4.
3. Apply the selection rules; this means to divide the problem in sub-sections and select one of them.
4. Select one of the two points below
    a. Execute the next iteration of the sequence of one of the sub-problems.
    b. Execute recursively the same algorithm over one for the sub-problems.

Because of the nature of the algorithm that was described above, the RHFSMs are very easy to be used in order to implement it, as was demonstrated in [16]. The implementation has to support multiple entry points to sub-algorithms and fast stack unwinding.

The multiple entry points are supported through various dedicated signals which indicate each time which point in the algorithm the call was made from. The fast stack unwinding is a proposed method in which the depth of the stack is stored in a variable. When there is a need to make a multiple pops from the stack, with the last pop to be the terminating operation of the algorithm, then there is no need for the algorithm to perform these pops; instead from the first pop we know that the algorithm will be terminated. Thus, we need to return the stack pointer to the initial value, rather that to deduct its value from sequential pops. With the fast stack unwinding we perform a multi-step pop without having to wait multiple cycles.

Also, in [16] Sklyarov et al. propose the use of embedded block RAMs (BRAMs) as storage means in order to avoid the resource consumption that the distributed RAM (DRAM) results in. This method is essentially equivalent to microprogramming; since all the micro operations and instructions that program the datapath are stored in a BRAM, this is essentially an approximation to microprogramming.

Finally, in [17] implementations were presented, according to the HGS method. The implementations were: a binary tree sorting, approximation methods for discovering a minimal row cover for a binary matrix, an exact binary search tree for solving the knapsack problem and the computation for the greatest common divisor.

3.4 Examples and Measurements from Sklyarov's Implementations

The results that were achieved from the implementations based on Sklyarov's algorithm are presented below; for the sake of representation simplicity, in [17] the problems have been identified as:

1. P1: the binary tree sorting,
2. P2: the approximation method,
3. P3: the exact binary search algorithm,
4. P4: the computation of the greatest common divisor.

In **Table 3** we can see the results of the above implementations.

Table 3 : Results from experiments

Problem/MI/BM/DM	N_s/F/N_{clock}/ET	
	Recursive	**Iterative**
P1 (t)		443/35.1/70/1994
P1/MI (t)	623/74.8/72/963	599/76.0/87/1145
P1/MI/BM (t)	474/52.2/72/1379	473/70.2/87/1239
P1/MI/DM (t)	477/58.8/72/1224	
P3 (t)		153/59.9/88/1469
P3/MI (t)	165/37.3/62/1662	
P3/MI/BM (t)	149/40.3/62/1538	
P3/MI/DM (t)	150/43.1/62/1438	
P4 (c)		448/41.3/9.217
P4/MI (c)	515/42/11/261	
P4/MI/BM (c)	454/43.5/11/252	
P4/MI/DM (c)	454/42.5/11/259	

Where MI stands for Modular Iterative implementation, BM stands for Block Memory, DM stands for Distributed Memory, N_s is the number of FPGA slices, F the maximum frequency speed in MHz, N_{clock} the number of clock cycles required in order to solve a problem and ET stands for the execution time in ns derived from the frequency F. Note that DM and BM are not expressed in a specific unit in [17]; we assume that DM is expressed in slices and BM in number of block RAMs occupied, even though it is not clear from **Table 3** how many block RAMs each problem occupies. The correspondence among acronyms and expanded names are shown in **Table 4**. Better illustrations of the results from **Table 3** are shown in **Charts 1, 2** and **3** for the problems P1, P3 and P4 respectively. Numbers in x-axis represent the four cases covered for each problem in **Table 3** i.e. in **Chart 1** number 1 is the first line for P1, number 2 the second line (P1/MI), number 3 the third line (P1/MI/BM) etc.

Table 4 : Acronyms and explanations

MI	Modular Iterative
BM	Block Memory
DM	Distributed Memory
N_s	Number of Slices
F	Frequency in MHz
N_{clock}	Clock cycles required to solve a problem
ET	Estimated Time

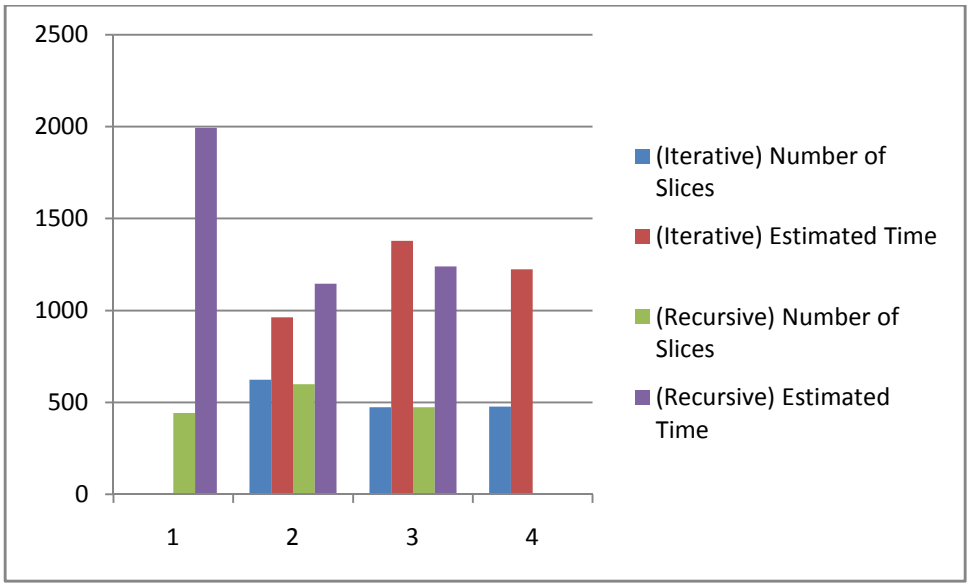


Chart 1 : Recursive and Iterative implementation comparison for P1

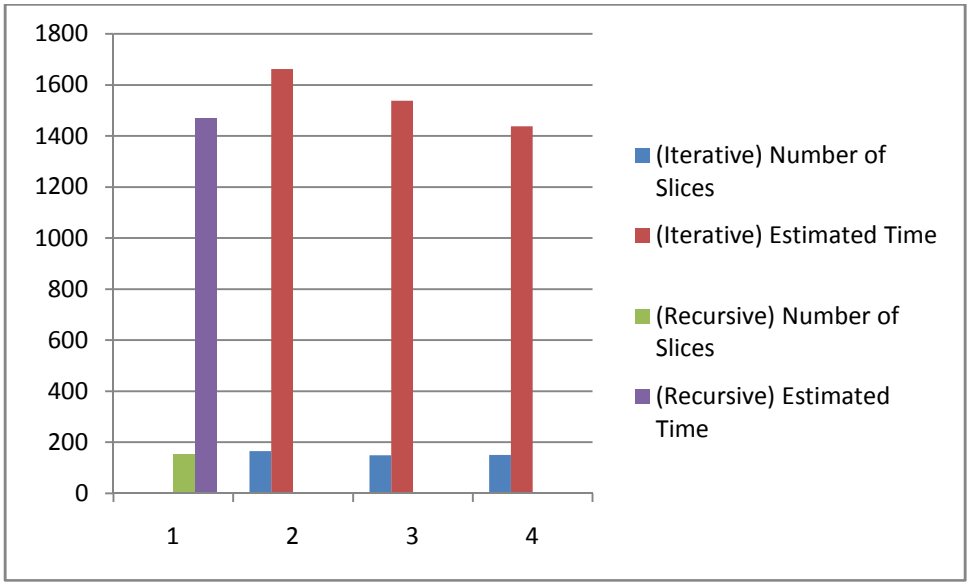


Chart 2 : Recursive and Iterative implementation comparison for P3

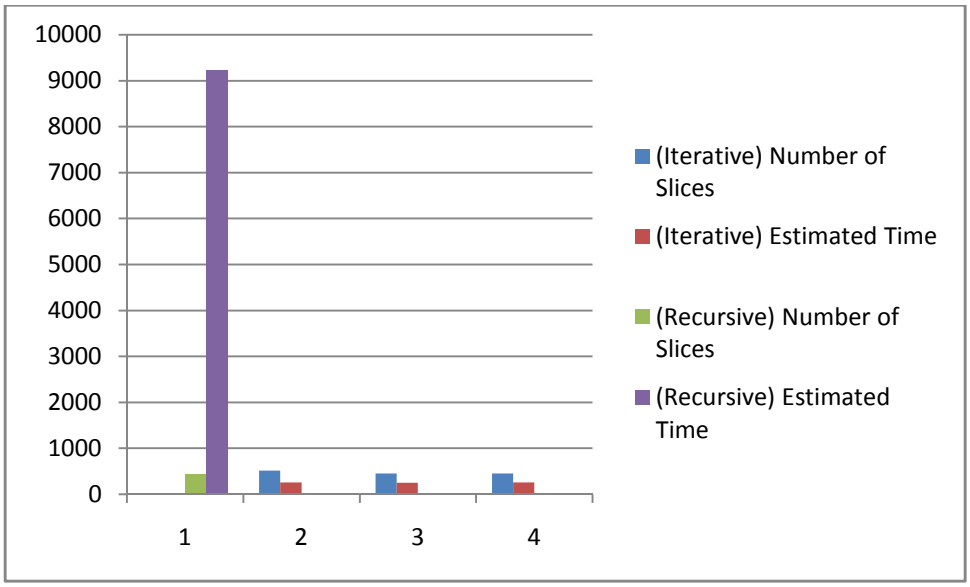


Chart 3 : Recursive and Iterative implementation comparison for P4

In [16] Sklyarov et. al have implemented optimizations to the algorithms presented in [17], which were mainly the fast stack unwinding possibility, faster hierarchical returns and the use of embedded memory blocks. The results of these optimizations are shown in **Table 4**.

Table 5 : Results from implementations in [16]

Problem from [16]	N_s/N_{clock}			
	Implementation from [15]	Block RAM	Distributed RAM	Multiple Entries
P1	192/72	50/72	53/72	189/59
P3	68/62	17/62	19/62	67/51
P4	49/11	15/11	16/11	47/9

A graphical representation of the results illustrated in **Table 4** is given in **Chart 4**. Note that measurements n. 1 concern the implementation column from **Table 4**, measurements n. 2 concern the Block RAM column, measurements n.3 concern the Distributed RAM and measurements n.4 concern the Multiple Entries column. As it can been seen from the results, a slightly improved performance was gained from the hardware optimizations that were performed in [16].

The algorithm that was proposed from Sklyarov is an important step towards the implementation of recursion in reconfigurable hardware. Implementations based on recursive hierarchical graph schemes are simplified compared to previously manual designs, and results from [16] and [17] show that recursive hierarchical graph schemes are especially beneficial in tree search algorithms against iterative implementations.

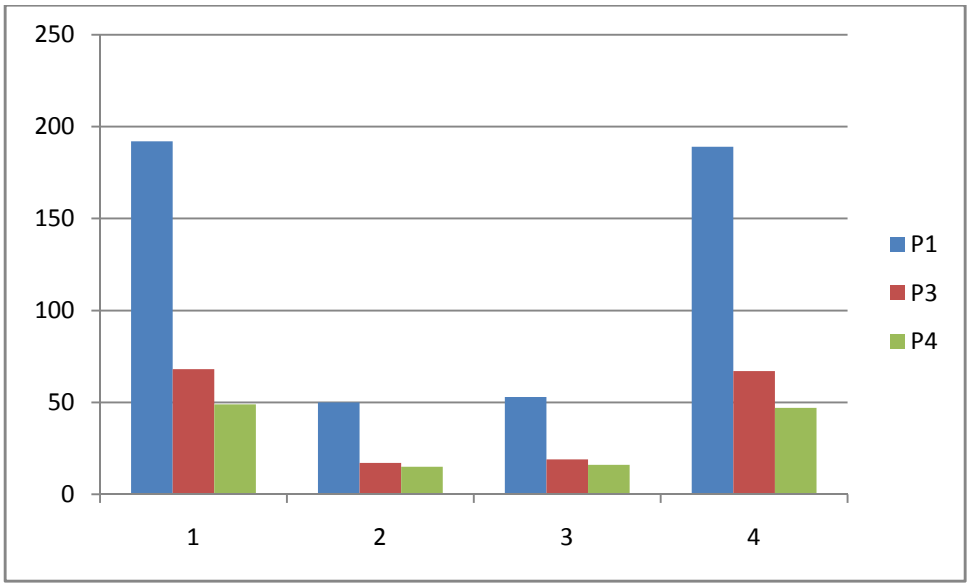


Chart 4 : Implementation results from [16]

CHAPTER 4: A NEW ALGORITHM FOR RECURSION IMPLEMENTATION

4.1 A Closer Look at Programmable Logic Structures

In order to achieve optimized results from any implementation of any algorithm in a FPGA, it is essential that the structure of the reconfigurable fabric is understood. Because today's FPGAs include processing units that are optimized to perform a single operation (i.e. multiplication), a good implementation has to take into account those processing units; thus, it is almost certain that we will achieve good levels of speed and area performance. The dedicated processing units of a FPGA are often called ***macros*** or ***mega-functions***[4]. The most often used processing units are multipliers, adders and RAMs, which are the most common macros found in FPGAs. In order to understand how these macros are distributed in an FPGA fabric, we will closely examine the structure of a Spartan 3 FPGA from the Xilinx Company [18]. In **Fig. 22** we see a structure diagram as is given in the Spartan 3 data sheet [19]:

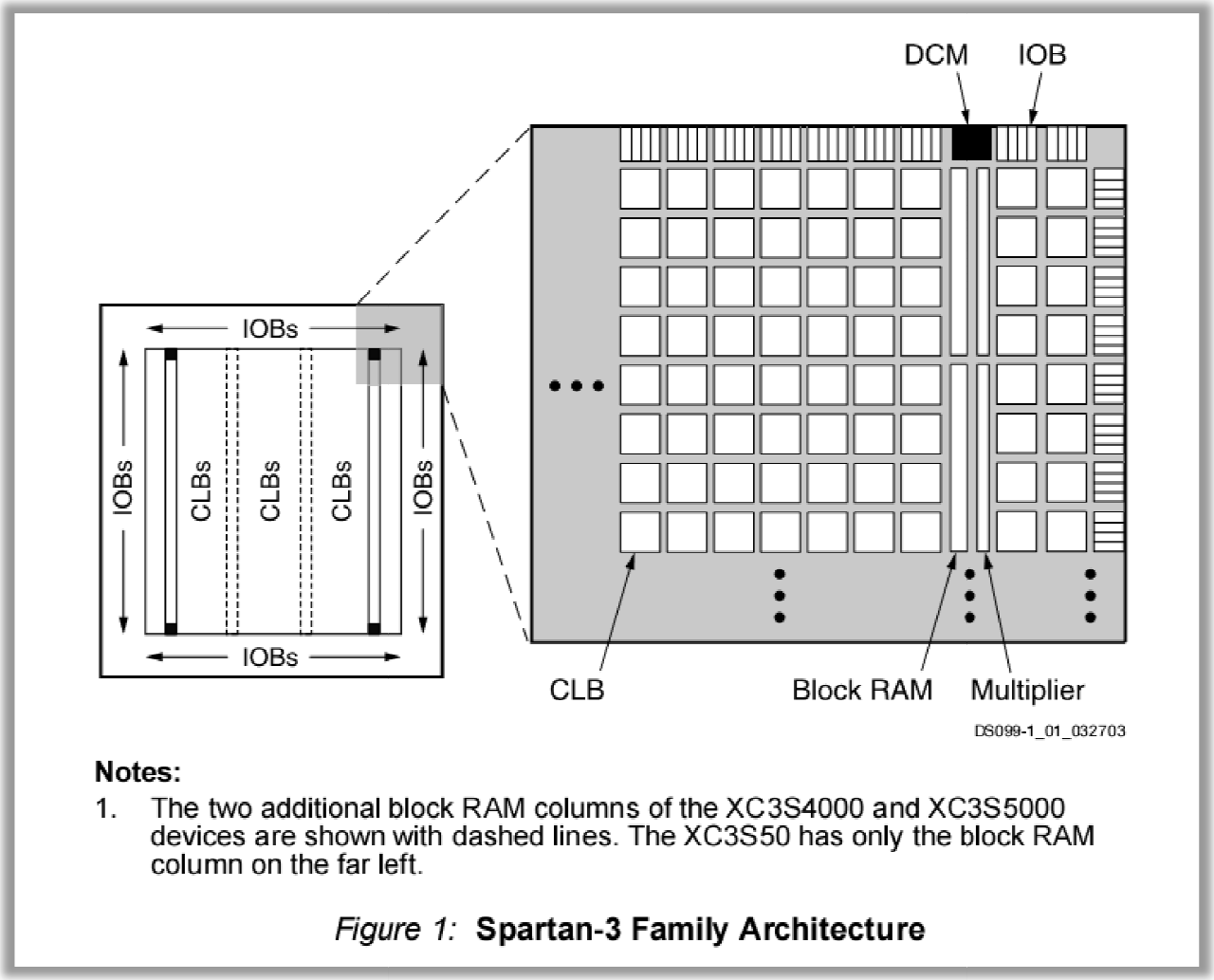


Figure 22 : Spartan 3 structure from [19]

[4] Macro is the term used by Xilinx. We will use this term.

From the figure above we can see that Spartan FPGAs offer Block RAMs and dedicated multipliers among other macros (i.e. Digital Clock Managers, IO Blocks etc). Also, every logic element has special hardware that can make easy to construct multiple-bit-adders. In the Spartan 3 specification [19] it is reported that the specific FPGA family can provide[5] 1 million system gates, 120 Kbits[6] of Distributed RAM, 432 Kbits of Block RAM and 24 dedicated 18x18 multipliers. These numbers change depending on the specific FPGA chip of the family.

It is important to notice the structural difference between the ***Distributed RAM (DRAM)*** and the ***Block RAM (BRAM)***. As has already been mentioned in the first chapter, the DRAM consists of the LUTs offered by each logic element. This means that every LUT is considered as a small part of a bigger RAM.However, instead of being concentrated in a certain area of the fabric, the DRAM is distributed all over the FPGA. On the other hand, BRAMs are SRAMs concentrated in certain spots of the fabric, as already indicated from the **Fig. 22**.

4.2 Implementation Problems Introduced by Previous Proposals

All the proposals mentioned in the previous chapter have several drawbacks, especially when it comes to FPGA resource consumption. Even though the proposed solutions achieve a very good level of overall performance, they manage it at some costs and tradeoffs.

Maruyama, Takagi and Hoshino in [11] put the separate algorithm steps in a pipeline, and used a control unit to check the different states of the pipeline. If the pipelined processing is favored then the control unit permits it, else it allows a semi-sequential processing. This allows for a speed increase at the expense of area. Even though the transformation from sequential to pipelined architecture does not produce much hardware overhead, the control unit to control the pipeline does. Furthermore, as stated by the authors, the maximum speedup provided equals the depth of the pipeline. This means that for small enough recursions there is no speed gain, yet there is more hardware area consumption because of the controller.

The method proposed by Sklyarov [14] introduces the software clarity of algorithm design to hardware. The use of HGS [15] simplifies the design and implementation of many recursive algorithms [17]. There are, however, drawbacks in the proposed solution; the use of three stacks per algorithm implementation suggests greater area consumption vs. a conventional single stack solution.

The extra area that is demanded may vary from some registers to many FPGA logic primitives. If for example we have three modules and each module can have five states, it means that we need two bits for the encoding of the modules and three bits for the states (using weighted binary encoding). This totals to five bits multiplied by the depth of the module stack. In this case the memory utilization for the two extra stacks is not a problem, but if we had i.e. thirty five modules and each could have thirty five states, that would total to twelve bits times the module stack. That implies that, for deep recursion or complex recursive algorithms, the two extra stacks may introduce area limitation problems. In [16] an

[5] For the X3S1000 chip, which is the IC used as the implementation FPGA in this thesis.
[6] K equals to 1024.

alternative practice was proposed, which included embedding the implementations of HGS for the recursion to block RAMs (BRAMs). This effectively triples the area utilization in terms of memory.

In addition, by inserting the state transitions in a stack, we bound the depth of recursion by the data stack depth, along with the module stack depth. This can be a much more serious problem vs. the module and state stack area consumption; by defining the module stack depth we pre-decide an upper bound for module calls. Given that for every direct or indirect recursion call, or even a simple different module call, we need to store the new module to the module stack, the times a call will be made is constrained by the module stack depth. That constraint may be much more important than the data stack boundaries as every recursion call implies data push or pop and a module to be pushed or popped from the module stack. That means that the module stack needs to be at least as large as the size of the data stack with the obvious area overhead.

The recursion mapping through pipeline proposed by Ferizis and ElGindy [12] has also an impact to performance. The obvious drawback, which is also stated by the authors, is the case where the recursion is deeper than the pipeline. That leads to employment of two other solutions; the use of run-time reconfiguration (RTR) and reconfiguration prediction.

The use of RTR is a very valuable concept, since it can result in a very deep pipeline, and thus a major speedup in terms of the effective pipeline depth. The prerequisite for the above is the possibility to have many pipeline stages stored in a fast memory and the ability to perform the RTR fast enough so as not to stall the processing. The major drawbacks to this concept are that run time reconfiguration at such a fine grain is not necessarily fast enough, and, since not all FPGAs support RTR the proposed solution is vendor specific. In addition, RTR is not as fast as the processing capabilities offered by the FPGAs, so the branch prediction has to be deployed; this results in a very complicated design that leads to a potentially error-prone design procedure and a high concept-to-implementation time. Even with the reconfiguratio prediction, RTR still imposes a penalty to the overall performance. The penalty becomes a serious problem to the processing, in the case where the FPGA is filled with pipeline stages and it needs constant swaps of reconfiguration units. Finally, this solution cannot be used in a System-On-a-Chip (SOC) design. Since it is important to use the entire FPGA in order to deploy the pipeline, the design cannot exist in the same reconfigurable fabric with other designs. This is a severe drawback that limits its uses, since recursion may be needed in SOCs, i.e. for compression algorithms such as Huffman.

All of the above methods address the problem of recursion implementation in hardware to various degrees. They provide significant insight in the field of recursions, pointing out the problems that exist. However, more efficient methods are needed, such as the one we propose in the next section.

4.3 Introduction to Recursion Simplification

Taking into account all the restrictions that the existing solutions impose, the way to design the recursion algorithms can be improved; we need a transition from a *processing-oriented* to a ***data-oriented*** design. All the solutions presented so far used the data stack solely for storing the processed

data. The general concept was to decide from the current state and conditionals which will be the next state, i.e. whether it will be a recursive call or not.

The alternative approach, proposed in this thesis, is to let the data stored in the data stack to decide which will be the next state; when there is a recursive call or a recursive return. That way we gain three benefits: **(1)** we simplify the logic and hardware needed for transitions, **(2)** we allow for a more compact design since most of the data of the algorithm are in the stack and **(3)** we minimize data transactions since on one action (push or pop) we can have both the data for processing and the data for the next state.

In order to implement the data-oriented approach to recursive algorithms, an operation has to be performed that is defined as the ***recursion simplification***. This operation involves identifying which conditions in the algorithm "decide" whether a recursion call will take place or not. These conditions are defined as the ***conditions for recursion***. The conditions for recursion are of two types; the ***recursion termination conditions*** and the ***recursion scheduling conditions***. The first condition type identifies the circumstances where the recursion has reached its base case; where the recursion stops the unrolling and the reverse procedure is initiated. The recursion scheduling conditions are the conditions that "decide" whether a recursive call will be made. Although it seems that the conditions may have a relation of opposition, this is not always the case. A recursion scheduling condition may not exist (i.e. unconditionally recurse as in the Towers of Hanoi algorithm) but the recursion termination condition must exist; otherwise the algorithm will recurse until an external factor interrupts the algorithms' execution. There may be cases where the scheduling conditions can be used as recursion termination conditions. A more detailed study of this will be given in the analysis of the binary tree search example. Once the conditions for recursion have been identified, we must decide which data uniquely define the state from where the recursion call takes place. This step is defined as the ***local data identification***, since the data that uniquely defines a state are local to that state. The local data identification may be divided in two other steps; if the data can define uniquely every recursive call, then these data are the local data for the call. If the data do not suffice for the unique definition of every recursive call, then local data are divided in two categories; local processing data and local operational data. The ***local processing data*** are the data that are transmitted to every recursive call for processing purposes; they are called also ***parameter signature***. This name comes from the fact that the parameters in a function derive from the function signature excluding the return type of the function. The ***local operational data*** are identifiers that uniquely characterize the specific recursive call; this is used in case where recursive calls in different depth of the algorithm use the same data for processing. The type of the identifiers is usually an integer, encoded in weighted binary. After data identification is performed, the recursion simplification is complete.

In order to illustrate the concept of the different cases of the local data identification, we will use two paradigms; the Knight's Tour and the Towers of Hanoi. When implementing the algorithms, for every recursive instance of a function we must answer a question: *are the data supplied to the recursive function adequate to identify uniquely the current recursive call?* If yes, then those data are the local data and the local data identification step has finished. This is the case with the Knight's Tour; with every recursive call we supply the current position in the board and the next position to search. Since every

square in the board must be visited only once, these data suffice to identify every recursive call uniquely. But with the Towers of Hanoi this is not the case; as we can see from Example 1 in Chapter 1 (page 2) the two recursive calls interchange the same data, which results in calling the same recursive functions with the same parameters in different recursion depth (this concept is better illustrated in **Fig. 28** on page 45). Thus, apart from the processing data (which are the local processing data) we need to identify each recursive call with data that characterize the current operation. Using both types of data, we will be able to distinguish if we return from multiple recursive calls (the second instance of *hanoi* function in Example 1) or an arbitrary recursive call in the body of the algorithm (the first instance of the *hanoi* function in Example 1).

The local data identification can be used efficiently as a part of a more generic algorithm, which can be used in any case to transform the recursive algorithm to a hardware implementation:

1. Identify the condition for recursion:
    a. *Recursion termination condition step*: find which condition terminates the recursion. This will be the condition that will reverse the unrolling of the recursion.
    b. *Recursion scheduling condition step*: find which conditions decide whether the algorithm will make a recursion call.
2. Perform the local data identification:
    a. *Local processing data identification step*: For each recursive call identify the parameter signature. If the parameter signature is adequate for identifying the specific recursive call uniquely in the whole recursion tree, then step 2.b can be skipped.
    b. *Local operational data identification step*: If there are more than one recursive calls that have the same parameter signature, then assign identifiers to each recursive instance so as to create a combination that will be unique for every call. The identifiers can be simple integers; for example, if there are N recursive calls from within a function, then assign the numbers 0…N-1 to the recursive calls. The combination of the numbers with the parameter signature creates a unique data set for that specific call.

Once the above algorithm has been implemented and the recursion simplification is made, the recursion is transformed to a sequence of simple operations, as illustrated for example in **Fig. 23**. In this figure we show the recursion simplification operation to a function that comprises of four basic steps. Let's assume that on step 2 there is a condition which invokes the recursion. Then all local variables are pushed to the stack and the function is rescheduled for execution. When the function reaches step 2 again, we assume for the sake of the example that the condition is not met and the function proceeds with the execution of the other steps.

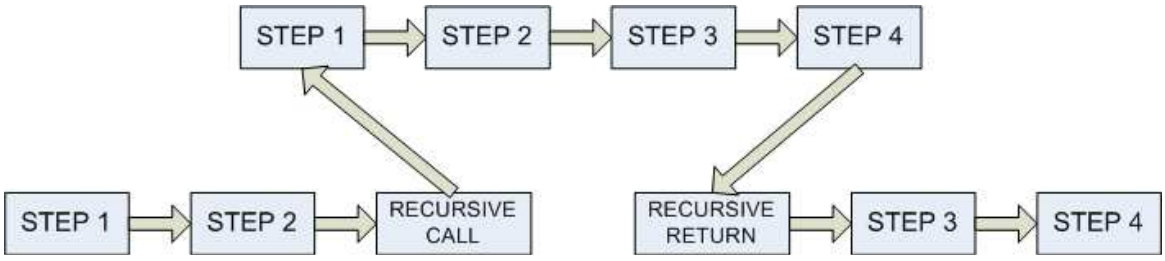


Figure 23 : Simple recursion example

When step 4 is reached and the function returns, the local variables from the stack are restored and the caller portion resumes execution. This is what normal recursion does. The key point here is to identify what are the condition for recursion and the data that uniquely identify the current state so as to perform the recursion simplification. As soon as the simplification has been done, the recursion can be thought of as a conditional flow; if the condition for recursion is met, then push local data and restart, else continue.

In **Fig. 24** we use a state diagram to illustrate the concept of recursion. The use of state diagrams helps to better understand the recursion in low level terms (i.e FSMs). The concept in **Fig. 24** is the same as in **Fig. 23**; there is a function that consists of five steps or states. The function begins normal operation and follows the transitions A and B until it reaches state 3.

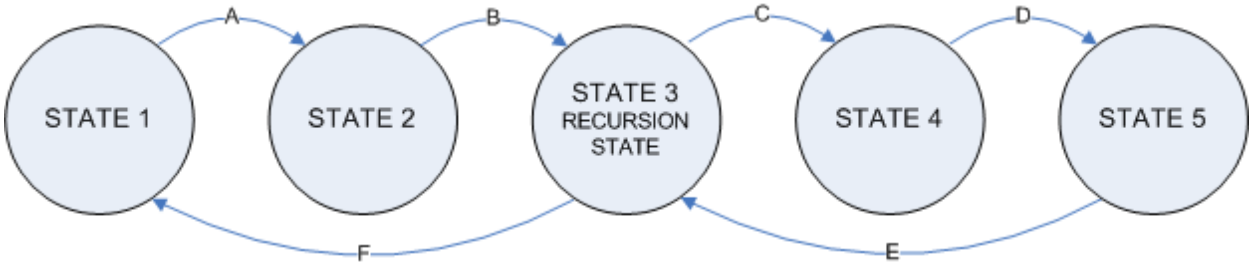


Figure 24: State diagram example for recursion simplification

When in state 3, which is the state that decides if a recursion call will be made or not, if the condition for recursion is met then local data are pushed to the stack and a transition to state 1 is made (transition F). If the condition is not met then the execution flow continues by following the transitions C and D until the last state. When state 5 is reached, if the current execution is a recursion call then a recursion return is performed by following the E transition and restoring the previous local data from the stack; else execution is ended.

The recursion simplification also works well when implementing recursion algorithms that have a base case and unconditional recursion calls. For example in order to implement the Fibonacci numbers recursively we consider a base case and the recursion case. The base case is the check if we calculate the first two numbers in the Fibonacci sequence. The recursion case is the general case that applies to calculating the rest of the sequence. This form of recursion can also be easily implemented in hardware,

although it is not considered the optimal choice. By using the recursion simplification operation we see how recursion is implemented when there is no recursion scheduling condition; the function recurses unconditionally until it meets the recursion termination condition. The local data identification step uses the current Fibonacci number for the local processing data and identifiers for the local operational data.

This fairly simple concept is easy to be expanded to as many recursion calls in the same module as we need. The data-oriented recursion design solves many of the problems that are present to the other proposals. First of all, the area consumed is minimal. Given that block RAMs (BRAMs) exist in every modern FPGA, using them as the base for stacks does not impose any major area overhead. Also, by storing the local data for the state identification in the stack, we save much more space than if we were to store them in the fabric. Most common configurations for BRAMs allow for a wide range of data widths. The local data are usually small compared to the other data of the design, so adding them to the stack does not introduce extra overhead.

Moreover, comparing to the recursion stacks ([15]) or the pipelines that depend on the recursion depth ([11], [12]), our throughput remains nearly constant; it depends only on the throughput of the design, which is a common constraint for all the previously proposed methods. Finally, the proposed method does not depend on specific FPGA characteristics, which makes it vendor independent and portable to a wide range of FPGA families. The compact form of our design makes it suitable for SOC-applications, which is an important factor for future designs.

4.4 Example 1: The Knight's Tour

In this section we will study further the details of the implementation of the Knight's Tour. This problem is about finding a Hamiltonian path on a square chessboard. A Hamiltonian path is a path where every square of the board is visited until the knight passes through all the squares of the board; the knight however, cannot visit the same square twice. The knight's tour algorithm implementation works as follows; given initial coordinates, the knight performs a Depth First Search (DFS) using a circular search pattern. The valid locations around a knight are numbered from zero to seven, clock wisely (**Fig. 25**). Every time a valid move is identified, the current location is stored and the knight proceeds to the new location. The knight tests every possible move around the current location, until it reaches the last of the valid possibilities.

In **Fig. 26** we show a code snippet in the C programming language which is used to identify the local data and the condition for recursion. The local data that need to be pushed to the stack before the recursion call are the x, y and the number of the current check for every call. The condition for recursion each time is the expression checked in the "IFs".

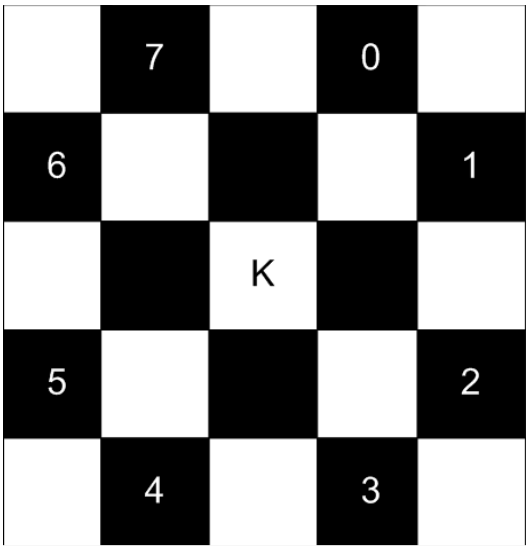


Figure 25: A Knight's eight possible moves

As a cross-reference to the previous section, where the knight's tour was introduced, we will analyze further the local data identification step. In the code snippet below, we see the conditional recursive calls:

```
y_new = y + 1;
x_new = x - 2;
if ( (y_new < size) && (x_new > -1) && (visited[x_new][y_new] == 0) )
       knights_tour(x_new, y_new, size, pinakas, visited, moves);

y_new = y + 2;
x_new = x - 1;
if ( (y_new < size) && (x_new > -1) && (visited[x_new][y_new] == 0) )
       knights_tour(x_new, y_new, size, pinakas, visited, moves);

y_new = y + 2;
x_new = x + 1;
if ( (y_new < size) && (x_new < size) && (visited[x_new][y_new] == 0) )
       knights_tour(x_new, y_new, size, pinakas, visited, moves);

y_new = y + 1;
x_new = x + 2;
if ( (y_new < size) && (x_new < size) && (visited[x_new][y_new] == 0) )
       knights_tour(x_new, y_new, size, pinakas, visited, moves);
```

Using the algorithm analyzed in the previous section, we proceed as follows:

1. *Identify the condition for recursion*
    a. *Recursion termination condition step*: the condition that terminates the recursion is whether the stack meets the edge conditions; full or empty. We should however note that empty stack does not indicate necessarily termination of the knight's tour algorithm, since we may have to backtrack to the first position and recheck the board following a totally different path.

b. *Recursion scheduling condition step*: we have eight conditions that schedule recursion calls; they are the assignment of the new coordinates along with the checks whether they are valid or not

2. *Identify the local data*
   a. *Local processing data identification step*: the coordinates x_new and y_new identify uniquely every recursive call and the search number that we perform. Every coordinate supplied to the recursive function must be unique to the chessboard so, coordinates are unique among the different sequential calls. The coordinates have to be combined with the current search point (in the circular search pattern shown in **Fig. 25**) so as to produce a unique identifier for every recursive call. This is important, since the same coordinates are unique for the recursive call, but without the search point we will not be able to distinguish whether we make a recursive call or backtrack.
   b. *Local operational data identification step*: since we can distinguish every recursive call using the step 2.a, this step is skipped and operational data are not needed.

Using the above algorithm, we can implement the knight's tour easily in hardware. The algorithmic stage diagram of the FSM derived by the conditions identified in the recursion simplification algorithm is shown in **Fig. 26**.

In **Fig. 27** we also show the state transitions that could represent the code in C for the recursion point, along with VHDL code that checks for the next valid location and some of the transitions needed for push/pop operations. Note that the "rot_sig" signal is a number that represents the number of check for valid next location that is currently performed. According to the recursion simplification, "rot_sig" is considered to be one of the local data. It can be seen from this example that using the data-oriented approach to map recursive functionality to FPGAs is very simple and straight forward.

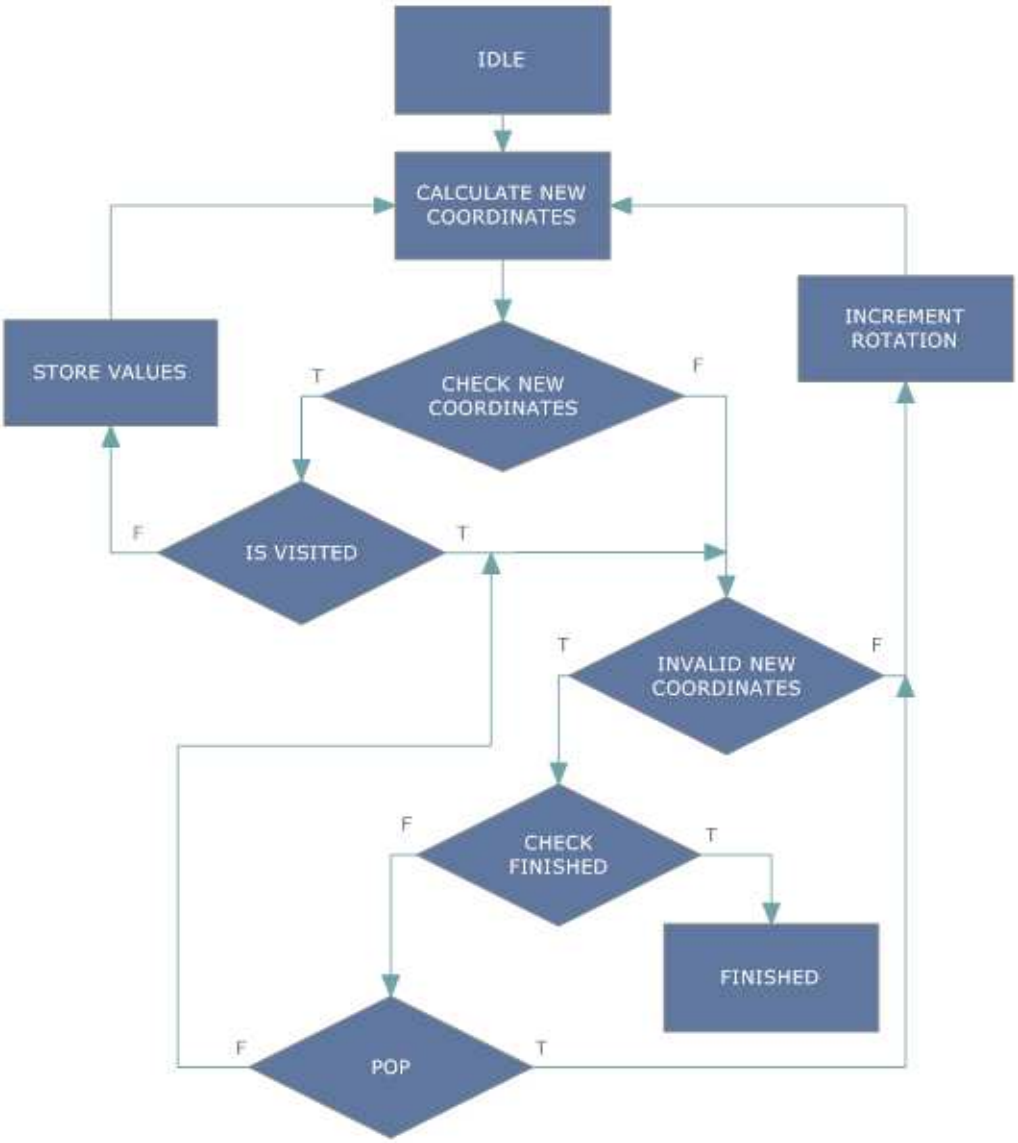


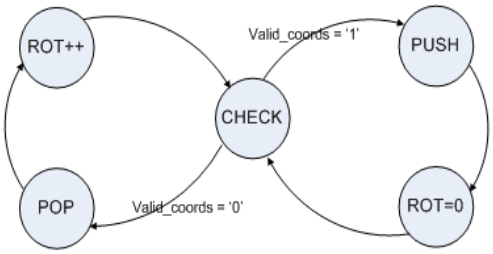
Figure 26 : Knight's tour algorithmic state diagram

```
[…]
        y_new = y + 1;
        x_new = x - 2;
        if ( (y_new < size) && (x_new > -1) && (visited[x_new][y_new] == 0) )
                knights_tour(x_new, y_new, size, pinakas, visited, moves);

        y_new = y + 2;
        x_new = x - 1;
        if ( (y_new < size) && (x_new > -1) && (visited[x_new][y_new] == 0) )
                knights_tour(x_new, y_new, size, pinakas, visited, moves);

        y_new = y + 2;
        x_new = x + 1;
        if ( (y_new < size) && (x_new < size) && (visited[x_new][y_new] == 0) )
                knights_tour(x_new, y_new, size, pinakas, visited, moves);

        y_new = y + 1;
        x_new = x + 2;
        if ( (y_new < size) && (x_new < size) && (visited[x_new][y_new] == 0) )
                knights_tour(x_new, y_new, size, pinakas, visited, moves);
[…]
```




```
[…]
when CALC_NEW_COORDS =>
        case rot_sig is
                when 0 =>  x_new_sig <= x_curr_sig - 2;
                           y_new_sig <= y_curr_sig + 1;
                when 1 =>  x_new_sig <= x_curr_sig - 1;
                           y_new_sig <= y_curr_sig + 2;
                when 2 =>  x_new_sig <= x_curr_sig + 1;
                           y_new_sig <= y_curr_sig + 2;
                when 3 =>  x_new_sig <= x_curr_sig + 2;
                           y_new_sig <= y_curr_sig + 1;
                when 4 =>  x_new_sig <= x_curr_sig + 2;
                           y_new_sig <= y_curr_sig - 1;
                when 5 =>  x_new_sig <= x_curr_sig + 1;
                           y_new_sig <= y_curr_sig - 2;
                when 6 =>  x_new_sig <= x_curr_sig - 1;
                           y_new_sig <= y_curr_sig - 2;
                when 7 =>  x_new_sig <= x_curr_sig - 2;
                           y_new_sig <= y_curr_sig - 1;
        end case;
[…]
when CHECK_NEW_COORDS =>
        if ( (x_new_sig > -1) AND (y_new_sig > -1) AND
        (y_new_sig < BOARD_SIZE) AND (x_new_sig < BOARD_SIZE) ) then
        -- coords_valid
        -- go to checking visited table
        visited_addr <= std_logic_vector(to_unsigned(x_new_sig, COORDS_WIDTH))
&
                        std_logic_vector(to_unsigned(y_new_sig, COORDS_WIDTH));
        end if;
[…]
when CALC_NEW_COORDS =>    next_state <= CHECK_NEW_COORDS;
when CHECK_NEW_COORDS =>
        if ( (x_new_sig > -1) AND (y_new_sig > -1) AND
        (y_new_sig < BOARD_SIZE) AND (x_new_sig < BOARD_SIZE) ) then
                next_state <= IS_VISITED;
        else
                next_state <= INVALID_NEW_COORDS;
        end if;
```

Figure 27 : Sample C code, state diagram to show the push/pop operations and sample VHDL code to show implementation of a similar transition

4.5 Example 2: The Towers of Hanoi

In this section we will study how to use the recursion simplification in order to implement the Towers of Hanoi. We will have as a software guide the Example 1 (in page 2). As we can see from the main body of the hanoi function there are two different recursive calls, separated by a print of the results. In order to apply our method, we need to identify the local data that characterize uniquely each recursive call. The parameter signature does not suffice because in different depths the same parameter signature is used by other recursive calls. Thus, by using solely the parameter signature as the local data, we cannot uniquely identify each recursive call. This can be seen in the recursive call/return representation for 4 disks, shown in **Fig. 28**. Several recursive calls/returns have been pointed out, and we can see that all squared calls use the same parameter signature but they are in different stack depths. There are two general cases; one is that a recursive call is followed by a single return and a print (boxes marked with the letter A) and the second is that a recursive call is followed by multiple returns (boxes marked with the letters B, C and D). The first case happens when the hanoi function is called as the first recursive call in the program and the second case happens when the hanoi function is called as the second recursive call in the program. From **Fig. 28** it can be seen that it is impossible to conclude from the parameters only which recursive call is made, the first or the second.

This leads to the use of both the local processing and operational data; the parameter signature is the local processing data for each recursive call and the operational data will be an identifier that will inform us whether a recursive return is taking place from the first call or the second recursive call. Thus the local data identification are the FROM, TO, USING parameters (which are modeled as constants, mapped as 2-bit vectors in hardware) and a flag named “last_call” which identifies whether it is a return from the second instance of the Hanoi function or not. The algorithm implementing the concept described is:

1. *Identify the condition for recursion*
    a. *Recursion termination condition step*: the condition that terminates the recursion is whether the disk number has reached the value zero.
    b. *Recursion scheduling condition step*: there is no condition of this type. As we can see from the code in Example 1 in page 2, if the recursion termination condition is not met, then we make two recursive calls unconditionally; there is no specific prerequisite for the recursion to take place, as we can see in code snippets already cited for the Knight’s Tour problem.
2. *Identify the local data*
    a. *Local processing data identification step*: the FROM, TO and USING variables along with the disk number (the parameter signature).
    b. *Local operational data identification step*: we have two recursive calls in the main body, so we identify the first call with the value zero (0) and the second call with the value one (1). This can be mapped directly to hardware.

The ***algorithmic state machine (ASM)*** is shown in **Fig. 29**. The concept behind the implementation is almost a direct mapping from high level to implementation. The RELOCATE 0/1 state,

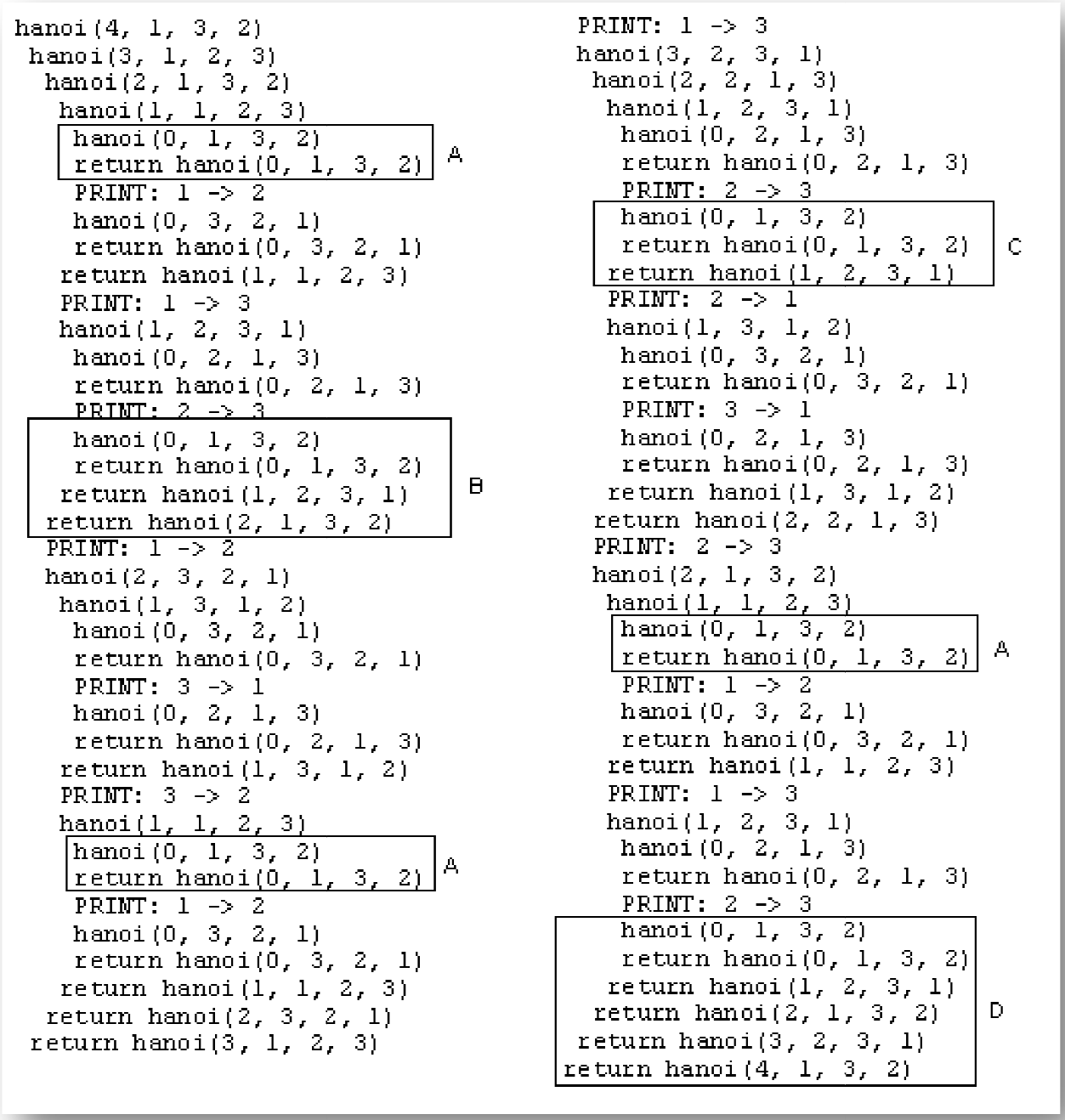


Figure 28 : The hanoi recursive call/return stack for 4 disks

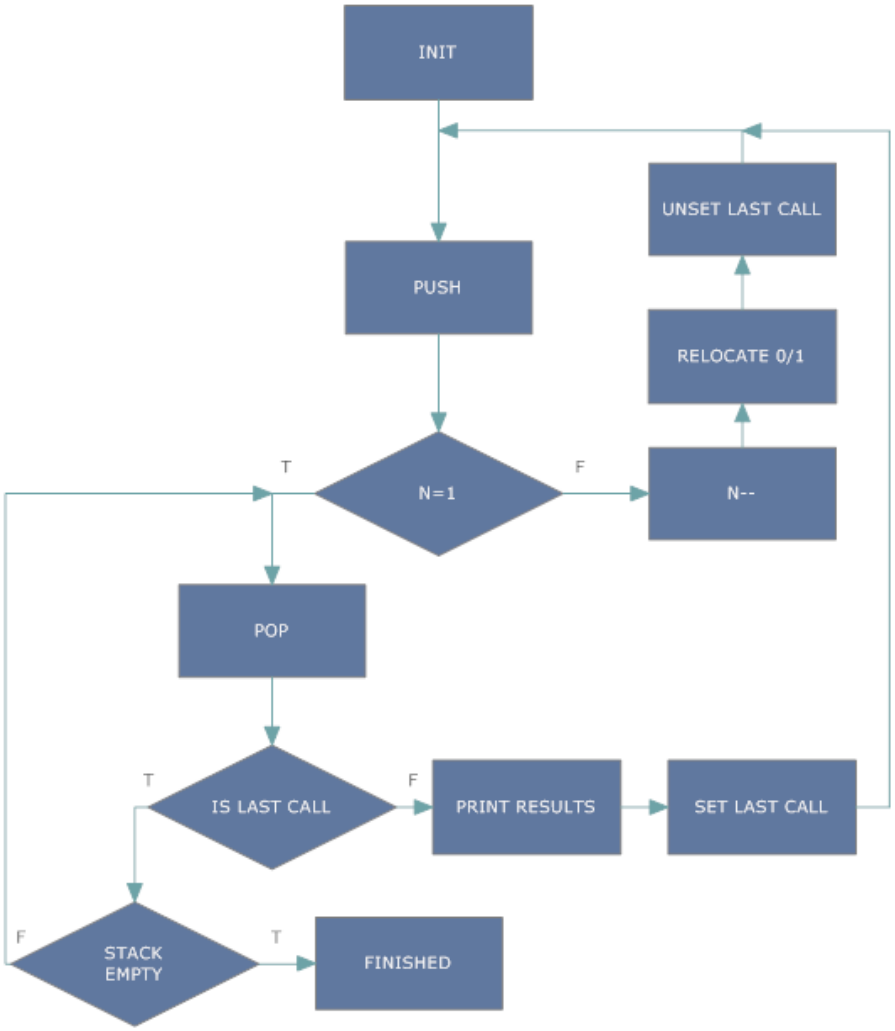


Figure 29 : The Towers of Hanoi ASM

makes the permutation of the *from, to* and *using* variables, according to which recursive call is being made (0 for the first call and 1 for the last call). The local operational data are used at states SET/UNSET LAST CALL. The rest of the implementation details are straightforward.

4.6 Example 3: Binary Tree Search

The next implementation is a binary tree traversal algorithm. The algorithm performs the ***Depth First Search (DFS)*** on a binary tree. Even though the algorithm was tested on a complete but not ordered tree, it was designed to traverse a non complete tree as well. The snippet of code that implements the recursive functionality (written in the C programming language) is shown in **Fig. 30**. Ignoring all the code that has nothing to do with the traversal *per se*, we can see that the code snippet can be clearly processed using the algorithm for the recursion simplification:

1. *Identify the condition for recursion*
    a. *Recursion termination condition step*: the condition that terminates the recursion is if the current node contains the wanted number.

b. *Recursion scheduling condition step*: the condition that schedules the recursive calls is the existence of child nodes. The checks that are performed offer to the program the capability to traverse an incomplete tree too; if one of the children does not exist but the others do, then recursion can proceed normally.

2. *Identify the local data*
   a. *Local processing data identification step*: the current node is the only data that is needed in order to uniquely identify the current call. In every tree each node is unique and no two nodes can have the same parents and/or children.
   b. *Local operational data identification step*: we have two recursive calls in the main body, so we identify the first call with the value zero (0) and the second call with the value one (1). This can be mapped directly to hardware, exactly the same way as with the Towers of Hanoi.

There is a special case that must be examined closely; when the wanted number is not found and there is no child of the current node to visit. One could possibly include the dissatisfaction of the scheduling conditions in the recursion termination condition. This possibility is provided, but it may lead to improper algorithm interpretation and implementation. For example, if **in Fig. 30** we just wanted to traverse the entire tree, then the termination condition (checking the wanted number) is not needed. In this case the scheduling conditions play the role of the termination conditions as well. In practical applications however, a termination condition should exist separately, since it usually represents the purpose of the data processing; for example in the knight's tour the condition is whether we have found the Hamiltonian path (stack full), in the binary search if we have found the number or object that we are looking for, etc.

By identifying the local data that characterize each recursive call, we are now able to implement the algorithm in hardware. The algorithmic state machine depicted in **Fig. 31**, shows how straightforward the implementation of the Towers of Hanoi is after the recursion simplification is performed. From **Fig. 31** we observe that we follow strictly the structure of the code in **Fig. 30**; we first check if we have found the node, then we go left and in the end we go right. This sequence is not mandatory, but describes the DFS algorithm. If we change the order by which those three steps are performed, we can change the way the tree is traversed; we can have ***in-order, pre-order*** and ***post-order*** traversal.

```
void search_tree(struct node *currNode, int wanted_num)
{
        if(debug_flag == 1)
                printf("Searching node: %d\n", currNode->value);

        if (currNode->value == wanted_num)
        {
                printf("Number value found!\n");
                time_end = time(NULL);
                printf("Time elapsed for search: %5.0fsecs\n",
                                                        difftime(time_end, time_begin));
                printf("Done.\nProgram ended.\n");
                exit(0);
        }
        else
        {
                if (currNode->left != NULL)
                        search_tree(currNode->left, wanted_num);
                if (currNode->right != NULL)
                        search_tree(currNode->right, wanted_num);
        }
        return;
} /* search_tree() */
```

Figure 30 : Recursive function for a binary tree search

4.7 Solving the "Chicken and Egg" Problem in Local Data Identification

The recursion simplification procedure, presented in this thesis, has a significant problem: we are not able to know *a priori*, whether the parameter signature (see page 36) of every recursion call suffices to uniquely identify the call throughout the recursion. We have to perform the recursion in order to find out if two or more calls have the same parameter signature. For example in **Fig. 28** we could not possibly know that the boxed calls have the same parameters, unless we performed the recursion. This is a very restrictive problem; if we are not able to know whether the parameter signature can uniquely identify a recursion call, then the recursion simplification cannot be implemented.

The solution to the "chicken and egg" problem is to generally use the concept of both the processing and operational data, in order to assure that unique identifiers are binded with each recursive call. The overhead introduced is extremely small and can be practically neglected; it can only introduce problems when we have a lot of recursion calls in a function. This notion will be better understood through the following example: if we use weighted binary encoding to enumerate the recursion calls made by a function, then the bits needed to encode a number is given by $\log_2 N$ (where N is the number of recursion calls). This means that the number of bits required for encoding are significant less than the number of the encoded recursion calls. Let's suppose that we have local processing data that are represented by D bits. If we were to use only those bits in the stack, then the

stack width W would be W=D bits. If we were to use the local processing data and the local operating data in

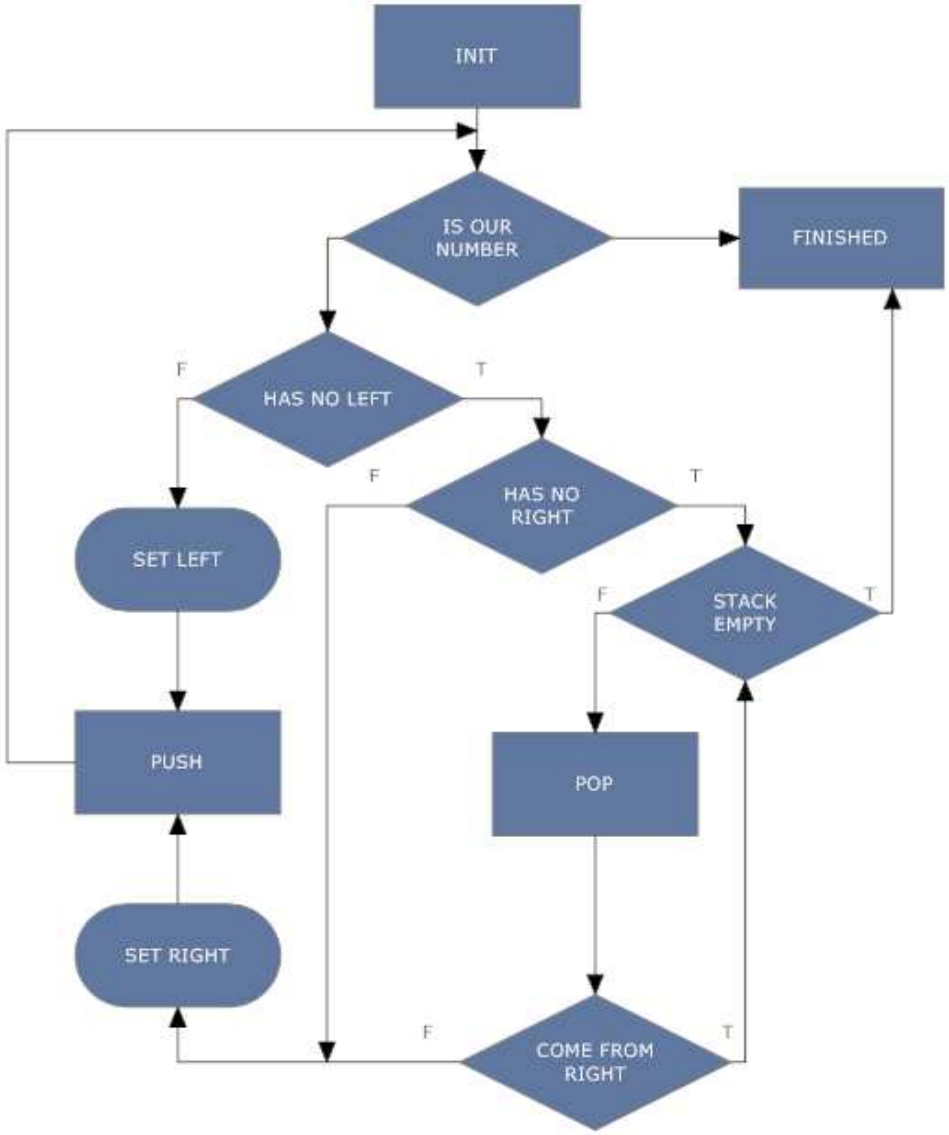


Figure 31 : The ASM of a binary tree traversal algorithm

the stack, then the stack width would be $W = D + \log_2 N$ bits. As it turns out, the growth in the stack width is very small, as compared to the number of recursive calls that can be represented. For example, if we had 256 recursive calls in a function, we would only need 8 bits of local operating data to be added in the stack width, which is a small amount. Furthermore, in practical applications rarely someone implements algorithms with more that a few recursive calls. This means that even for i.e. eight (8)

recursive calls, the amount of bits needed is only three (3), which does not introduce any significant overhead; in fact, it is negligible.

The concept of local processing and operating data can be easily implemented to every recursive algorithm that present simplified parameter signature. Simplified parameter signature means that every parameter is a scalar value and not expressions or functions. Most algorithms that fall under this condition can be effectively described by the recursion simplification procedure. For example the Knight's Tour problem that was described in a previous section can be implemented using the two kinds of local data instead of the processing data only. The modified algorithm that can describe the Knight's Tour with the two kinds of local data is:

1. *Identify the condition for recursion*
    a. *Recursion termination condition step*: the condition that terminates the recursion is the traversal of the entire chessboard, condition that is indicated by the fact that the stack is full.
    b. *Recursion scheduling condition step*: there are several conditions that can schedule the recursion, namely every valid displacement from the current coordinates.
2. *Identify the local data*
    a. *Local processing data identification step*: the data of this kind are the current coordinates, which are unique to every recursion call.
    b. *Local operational data identification step*: instead of using a circular search pattern as the algorithm was designed previously, we could enumerate each of the eight recursive instances. This enumeration constitutes the operation data.

Note that the two implementation concepts are functionally equivalent; whether we use the circular search pattern or we bind each recursive instance with an integer, we end up using in hardware a 3-bit signal that is combined with the coordinates. The difference, though, lies in the concept; it is easier to have a general way to describe recursions than to use application specific designs.

CHAPTER 5: DESIGN AND IMPLEMENTATION OF THE NEW ALGORITHM

5.1 Efficient Structures for Recursion: an Introduction

In order to implement recursive algorithms in hardware, we need to understand the capabilities that the fabric has in algorithmic terms. In other words what kind of algorithmic strategies the hardware can support. The most obvious data structure that we need is one for the stack. Before we analyze how we can implement such a structure, we should first define the stack capabilities that are the most important. Even though a stack could have many capabilities, the most important from a functional point of view, which leads to the most area and speed optimizations, are:

1. Push
2. Pop
3. Stack full indicator
4. Stack empty indicator
5. Stack reset possibility (ignore stack's contents and re-initialize the stack pointer)

The final point for the structure of a stack is whether it will support error correction as a unit or it will be left to the operator. By error correction we mean the two circumstances where we have a full or empty stack and a push or pop is made respectively; these types of errors are defined as ***completeness-type errors***. There is a big difference, both in area and speed, if we choose to implement completeness-type error handling or not. The decision to implement them in the stack results in an area and speed penalty by a notable factor. Nevertheless, the obvious advantage is that stack can function safely as a unit, and the controlling circuit is much more simplified than if it had to check for stack errors. This is considered a significant advantage; rapid design overweights speed and area penalty by a small factor.

A very important concept for the stack is also the sequence of actions and their reversibility. This means that fundamental operations, such as push and pop, constitute a sequence of actions which are reversed when we switch from one operation to the other. The sequence has to be defined in the design of the stack so as to achieve the reversibility of actions. In our implementations the push operation is defined as the following sequence:

1. Store data to the current pointed location in memory.
2. Increment stack pointer by one.

Then pop should be the reverse sequence:

1. Decrement the stack pointer by one.
2. Show the contents of the current pointed location.

The sequences presented above indicate that the implementation of the stack should include the notion of time. This problem will be further analyzed in subsequent sections.

We also need to study the implementation of comparators. In ***Application Specific Integrated Circuits (ASICs)***, comparators result in gates with many inputs, which imply high fan-in values and lower

operating frequency. Even though in FPGAs there is not such a problem, since comparator circuits are emulated using LUTs, wide width comparators still pose a problem due to their implementation in tree form; they need many LUTs in order to be realized, which leads to the utilization of many logic elements and the surrounding routing network which is the reason for speed decrease. This leads us to consider approaches to the problems that eliminate the need for comparators. Those approaches are discussed below (in the section 5.3: "*Arithmetic and positional comparators*").

5.2 BRAM vs DRAM

As was mentioned in the previous chapter (Chapter 4, "*A closer look to structures of programmable logic*") RAM capability offered by many FPGA chips is divided in two major categories: Distributed RAM (DRAM) and Block RAM (BRAM). Besides the obvious fact that the two memory structures have area occupation differences, there are also three other non-obvious, equally serious differences; the read mode, the handling of simultaneous reads and writes to the same memory location and the memory multi-port capability.

The read mode is synchronous in BRAMs and asynchronous in DRAM [8], [20][7]. This means that in BRAMs once the data are written in the current clock edge, the data will be available to the output in the next clock edge. On the other hand, in DRAMs, at the same clock edge that we write data, those data are disposed for reading. We can emulate synchronous reading from DRAM if we use the output D flip flop (DFF) from each logic element (see **Fig. 2** in page 5) to store the output data from the LUT (which is the local portion of the DRAM).

The second difference, the handling of simultaneous reads and writes, is one of the most important differences. In DRAM when someone tries to read and write at the same time from a given location, the result is as expected from an ordinary register; we write the new value and read the previous one. In BRAMs there is the possibility to configure the memory to select one of three modes:

1. Read before write (READ_FIRST)[8]: this means that we write the new data but we read the previous data at the same clock edge. This behavior resembles the behavior of a register.
2. Read after write (WRITE_FIRST): this means that we write the new data and the new data are available immediately to the output, ignoring the old RAM data.
3. No read on write (NO_CHANGE): this means that we disable the possibility to read and write from the same location at the same time. In case this should happen, either the data to be written are ignored or the output of the memory are read *unknown ('X')*, depending on which operation we desire to be executed at the time specified.

The above configurable difference is important for the simple reason that if we demand an asynchronous read from the stack, then the stack will be implemented in DRAM. If we demand a similar functionality to a BRAM, the synthesis tool implements the functionality inferring combinational logic or

[7] Although the references point to specific vendor devices, this assertion is true for all FPGA families, as far as the author knows.

[8] Inside the parentheses the Xilinx terms are given, but are considered to be confusing. We use more descriptive terms. Altera uses similar terminology [21].

we should explicitly define the read before write functionality to the tool. As we can understand, the resulted RAM is slower and consumes more area than BRAMs.

Finally, as far as the multi-port capability, BRAMs are true dual-port macros; their original structure offers two ports for two different concurrent operations. On the other hand, DRAM is by nature single port and as such, in order to provide dual-port capability we must duplicate its size. One could consider BRAMs with the dual-port as disadvadageous, since when we need single port configuration we must ignore half the size of it. But the fact is that BRAMs exist anyway in dual-port configuration, whether or not we select this feature does not impose any area penalty; DRAM doubles the area consumption if we want it to be dual-port.

As a last note, a major drawback for the DRAM is the mere fact that it is distributed. As such, routing latency is unavoidable with its implementation as storage unit; the bigger the DRAM we need, the higher the latency we experience. Thus, we gain a cycle from the asynchronous read, but we loose from the latency associated with the nature of the memory. BRAMs do not have the same problem; since they are macros offered from the FPGA, their structure is optimized for fast access and low processing latency. Nevertheless, if RAMs needed by our design are adequately small, then all RAM functionality goes to DRAM. That is because it is considered as area waste from the synthesis tools to use a whole BRAM only for small memory usage. Having in mind the small size of the DRAM that is deployed in these cases, latency and routing delays are not considered a big issue.

5.3 Arithmetic and Positional Comparators

In the implementations that are described in this thesis, it was imperative that we used comparators in order to perform some actions. For example, in the knight's tour problem we had to compare the new knight's coordinates with the chessboard borders; this had to be done with comparators. Also, in the towers of Hanoi problem we had to compare the current disk number to one, in order to derive the condition for recursion; this also had to be done using a comparator. A comparator, however, is a sub-circuit that poses a great speed penalty to the whole circuit. The wider the comparison, the bigger the penalty paid.

This led to the need to find circuits that could substitute the comparators and though performing the same operation, would not impose such a downgrade in the speed of our circuit. The solution to this was based in the distinction between the two kinds of comparators; in the knight's tour problem the comparator is a positional one whereas in the towers of Hanoi the comparator is an arithmetic one. A **positional comparator** is a circuit that compares bit positions. An **arithmetic comparator** is a circuit that compares numbers. There is not much to be done in order to avoid an arithmetic comparator, since there is not a general way to transform number comparison to something equivalent. However, for the comparisons in the knight's tour, this transformation is easy. We will illustrate the fact with the following example:

```
y_new = y + 1;
x_new = x - 2;
if ( (y_new < size) && (x_new > -1) && (visited[x_new][y_new] == 0) )
       knights_tour(x_new, y_new, size, pinakas, visited, moves);
```

In the code snippet above, we observe that the *y_new* and *x_new* get new values and are compared to the maximum size of the board (*size* variable) or the lowest boundary of the board (-1, since 0 is a legitimate position in an axis). If we translate this directly to hardware, we face the problem of comparators. In the case of -1 this is not a problem, but in the case where the *size* variable gets a high value (i.e. bigger than 8 bits wide) then the comparator gets big enough. This is not so costly, as it turns out, since comparators are implemented in 4-to-1 LUTs[9] of a logic element, 4 logic elements constitute a CLB, which makes that a single CLB could easily emulate an 8 bit wide comparator. If we have multiple instances of the same comparator then the problem gets bigger. In circumstances when resource sharing is employed during synthesis, the speed performance is downgraded in favor of area consumption. If we disable resource sharing, then theoretically the routing latency for the interconnection of the many comparator instances could be the bottleneck.

All the above have led to the following solution. Let's suppose that we actually don't care whether the new coordinates are bigger than the maximum or smaller than the minimum; we instead wanted to know just if we are out of bounds. With this in mind we have constructed the circuit showed in **Fig. 32**. This circuit calculates if the next coordinates will be in-bounds for one axis (i.e. x or y). For two axis coordinates we use two of such circuits; one for x and one for y.

The circuit illustrated in **Fig. 32** is for a 4x4 board and has five distinct sub-circuits; an encoder from weighted binary to one-hot, a register that stores the one-hot result, a circuit that depends on the operation supplied and performs the respective modification to the one-hot stored result, a second register that holds the modified one-hot result and a circuit that checks for out-of-bound condition. The concept is very simple; instead of comparing absolute numbers in order to find out whether the new coordinates are in-bounds, we transformed the new coordinates to one-hot form. Depending on the current position in the axis (x or y) the respective bit is set and the others are zero. For example, if the current position is 1, then the register 0 holds a zero, register 1 holds a 1 and registers 2 and 3 hold zeros; the one-hot encoding for the number "1" would be 0100. This is the ***positional equivalent numbers***. According to the operation supplied, the one-hot encoding will be shifted as many positions as needed. For example, suppose that the current x-axis coordinate is zero (0). The positional equivalent number is "1000". Suppose the next position is position zero with respect to current coordinates (see **Fig. 25** on page 40). This means that the number one (1) should be added to the current x coordinate (assuming positive augmentation towards the right in **Fig. 25**). Thus, the operation "+1" should be supplied to the position modification circuit in **Fig. 32**, which will transform the addition by one to one-bit right shift. In this case, the resulted number in the register after the position modification circuit will be the 00010000, which is in-bounds. If we were to transit from current position zero (0) to the fifth (5) next position (as enumerated in **Fig. 25**) then the operation to be supplied to the position modification circuit would be the "-2", which would transform the "1000" to the "10000000". The resulted number would activate the out-of-bounds signal, since now the active signal lies within the four registers that are checked for the out-of-bounds condition.

[9] For a Spartan 3 FPGA, that was used as the verification platform FPGA in this thesis.

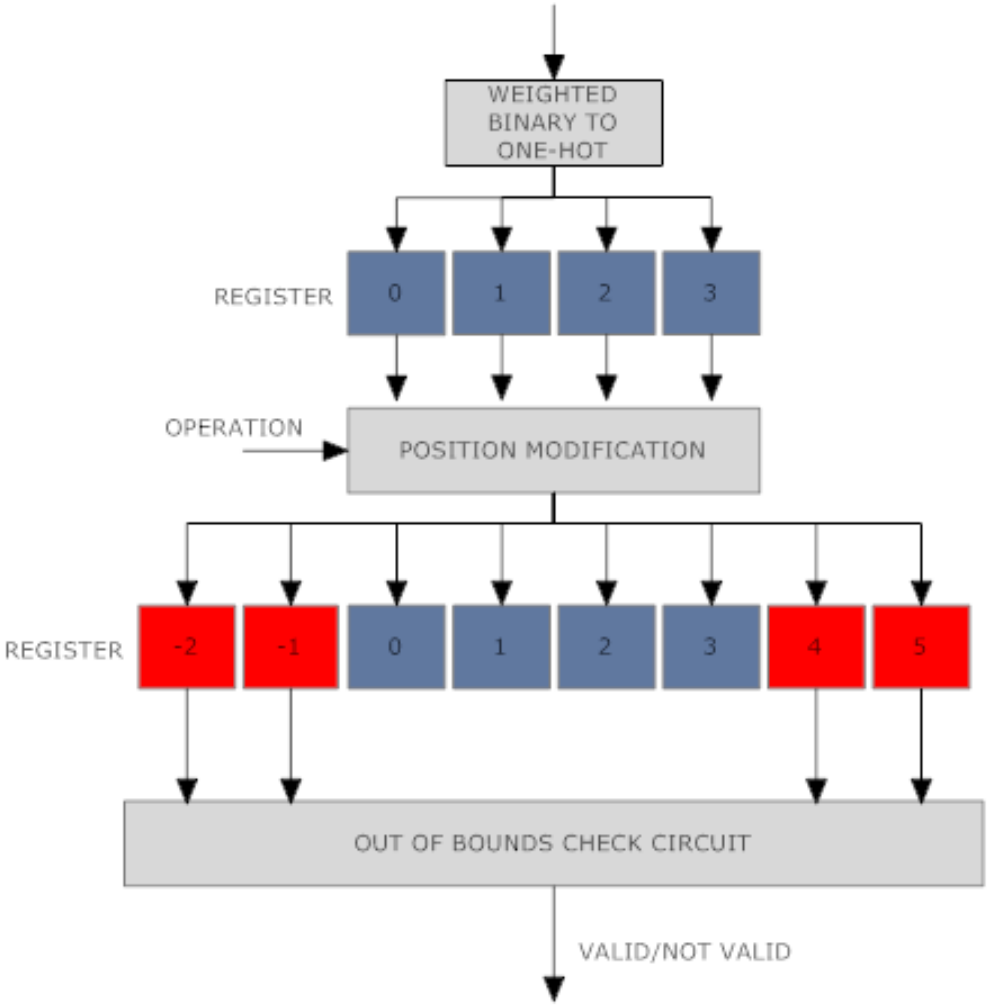


Figure 32 : Positional comparator substitute circuit

This same concept has been used in the stack, in order to eliminate the comparison for full or empty conditions. As was stated above, comparing to a number is a very costly operation if the number exceeds a certain number of bits. In this case, full condition for the stack is a costly operation. By substituting the arithmetic comparator with a positional one, instead of comparing the value of the stack pointer register to conclude the stack full and empty conditions, we compare a single bit in the top or the bottom of the shift register that accompanies the stack (see **Fig. 33**). Thus, when an active bit is found in the first flip flop (marked with the number one (1)) then the stack is not empty. If a zero bit is found there then the stack is empty. If an active bit is found in the last flip flop (marked with the number two (2)) then the stack is full, else the stack is not full.

The major drawback of this method is that it does not use the dedicated carry logic found in every logic element. This dedicated logic is especially designed in order to offer fast adders in the FPGAs. By using the circuit described in this section we loose the capability to utilize the carry logic; our circuit is implemented using LUTs. The results however are equally to or better than if we were to use adders, in terms of speed as was found after the place and route operation on a Spartan 3 FPGA.

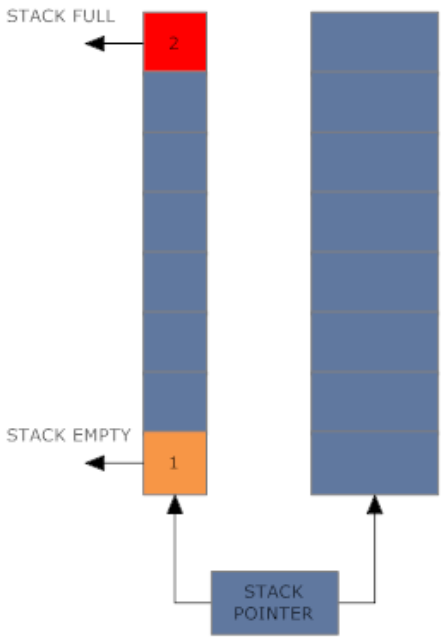


Figure 33 : Stack arithmetic substitution with a positional comparator

5.4 Arithmetic units' substitutes

The same problem as the comparators exists with the arithmetic operators. An adder or a subtracter is a circuit that poses a speed penalty to the overall circuit. Thus, a much simpler circuit would be to transform the weighted binary to one-hot form, perform the respective arithmetic operation as a shift and then retransform the resulted one-hot to weighted binary. This way, a faster circuit has been produced, without great area consumption. Of course, the speed decreases rapidly, proportionally to the bit width of the shift register. This concept is illustrated in **Fig. 34**.

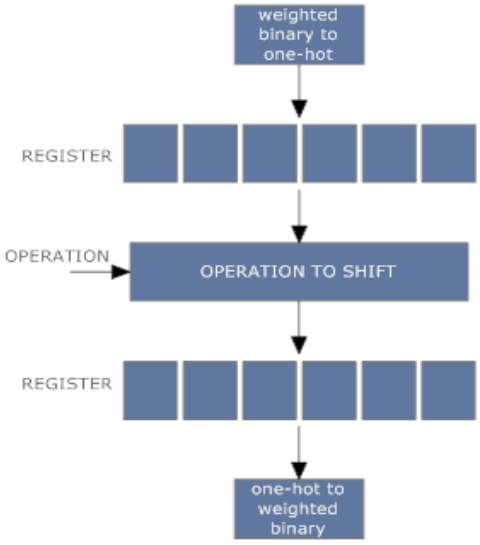


Figure 34 : Arithmetic circuits substitute

5.5 Single-cycle stacks

A single-cycle stack is a stack that can perform both the push and pop operations in once clock cycle. This is especially difficult, since such a capability inevitably leads to complicated circuits with high latency. One example of such an implementation is given if **Fig. 35**. This is made possible only because of two hardware tweaks; locate the adder/subtracter before the stack pointer and use DRAM as the main stack storage type. This is justified as follows: since the adder is located before the stack pointer, then when the push operation is performed the already stored address in the stack pointer will be read by the DRAM; the stack pointer though, will store the incremented by one address. Both operations will take place in the same clock edge. If the pop operation is performed then the subtracter stores the decremented by one address to the stack pointer and, since the memory is DRAM, it supports asynchronous reads and thus the new data will show up to the output in the same clock cycle. With this implementation, however, we are not able to deploy all the above mentioned circuits, and the single cycle stack ends up to be extremely slow.

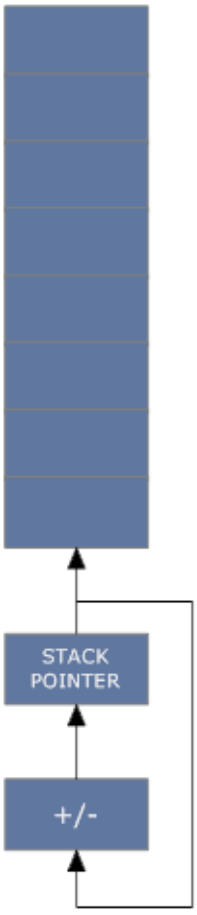


Figure 35 : Single cycle stack implementation

5.6 Multi-cycle stacks

Multi-cycle stacks are easier to implement because we can utilize every synchronous design we have already mentioned in the previous sections. A multi-cycle stack is based on an FSM that controls

the execution of the desired operations. Depending on how many or how complicated operations a stack will be able to execute, a multi-cycle stack may need from a few to several cycles in order to complete execution. After the definition of the requirements that were described in section 5.1, a good compromise between cycles and functionality is a two-cycle stack. The FSM that implements this stack must have the following states: IDLE, PUSH, PUSHED, POP and POPPED. The state transition diagram is shown in **Fig. 36**.

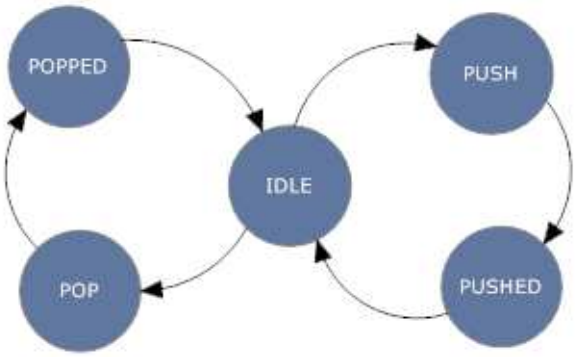


Figure 36 : State diagram for a multi-cycle stack

The above design has been proven to be extremely fast in one of the smallest FPGAs available today, with increasing performance as it was implemented to faster FPGAs. ***Post Place and Route (PPAR)*** implementation details are shown in **Table 5**, although the BRAM usage is not an objective criterion for memory consumption, since from device to device the size of a BRAM block differs; this, for example means that Virtex 5 has 32 BRAMs and Virtex 4 has 48, but the size of a BRAM in Virtex 5 is bigger than in Virtex 4. A simulation run is shown in **Fig. 37** in order to show the timing diagram of the operation of the stack.

Table 6 : Implementation details for a multi-cycle 32x64 stack

FPGA family	Speed in MHz (PPAR)	Area consumption		Cycles needed for PUSH op.	Cycles needed for POP op.
		Slices	BRAMs		
Spartan 3 XC3S1000-4	215,285	41/7.680 (1%)	1/24 (4%)	2	2
Virtex II Pro XV2P30-6FF896	386,548	39/13.696 (1%)	1/136 (1%)	2	2
Virtex 4 XC4VLX15-10SF363	397,931	74/6.144 (1%)	1/48 (2%)	2	2
Virtex 5 XC5VLX30-1FF324	454,339	21/4.800 (1%)	1/32 (3%)	2	2

The compromise assumed above is considered the best when we have to utilize ***symmetric stack functionality***; when the push and pop operations have to delay the same amount of clock cycles in order to complete. In practice though, there is no known reason why a stack must have this kind of restriction, so ***asymmetric stack functionality*** can be used. With asymmetry, the two operations delay different amounts of clock cycles. In our implementations the most obvious ratio between push and pop cycles is 1:2; 2 cycles for pop and 1 cycle for push. This way we can have the characteristics of a single-cycle and

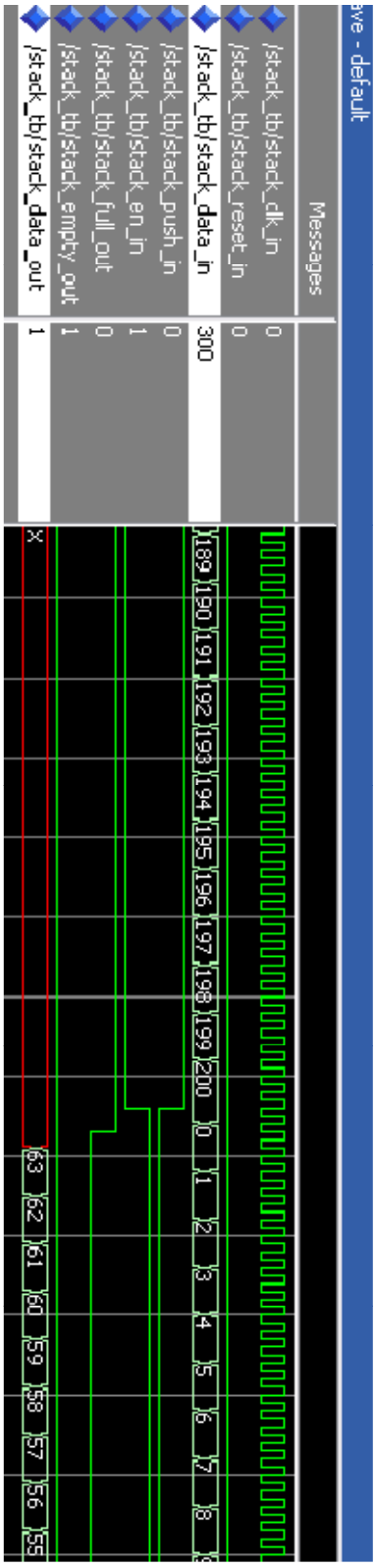


Figure 37 : Simulation snapshot showing some pushes and pops in a multi-cycle stack

a multi-cycle stack combined. The implementation results for an asymmetric functionality stack are shown in **Table 6**.

Table 7 : Implementation Results for an asymmetric stack 32x64

FPGA family	Speed in MHz (PPAR)	Area consumption		Cycles needed for PUSH op.	Cycles needed for POP op.
		Slices	BRAMs		
Spartan 3 XC3S1000-4FT256	210,837	44/7.680 (1%)	1/24 (4%)	1	2
Virtex II Pro XV2P30-6FF896	313,185	41/13.696 (1%)	1/136 (1%)	1	2
Virtex 4 XC4VLX15-10SF363	372,301	79/6.144 (1%)	1/48 (2%)	1	2
Virtex 5 XC5VLX30-1FF324	435,730	22/4.800 (1%)	1/32 (3%)	1	2

The advantage of using asymmetric functionality stacks is that we have to only wait for a single cycle in order to perform a push operation, and two cycles in order to perform a pop operation. Thus, from a protocol communication side of view, we have the minimum delay. The state diagram of an asymmetric stack is shown in **Fig. 38**. Notice that although there are two states in order to perform the push operation, the data are pushed during the transition from the IDLE state to the PUSH state. When performing the pop operation, the data are available when returning to the IDLE state after the POP state.

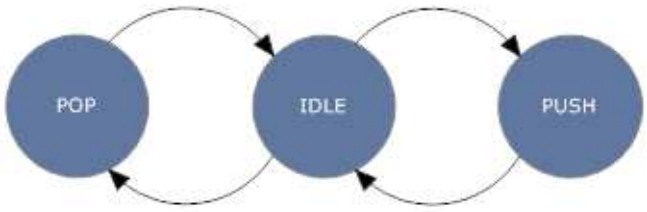


Figure 38 : Asymmetric stack state diagram

In **Table 7** we see stack implementations for big depths. This table demonstrates how the area scalability factor plays an important role in the circuit speed. The stack size is considered to be 32 bits wide to a defined depth (D) which is indicated in **Table** 7.

Table 8 : Stack post place and route implementation results for a variety of depths

Stack depth (D)	BRAMs	Slices	Period (ns)	Frequency (MHz)
256	1/24 (1%)	161/7680 (2%)	5,540	180,505
512	2/24 (8%)	316/7680 (4%)	5,958	167,842
1024	4/24 (16%)	639/7680 (8%)	6,402	156,201
2048	8/24 (33%)	1271/7680 (16%)	7,756	128,932
4096	16/24 (66%)	2472/7680 (32%)	8,387	119,232

5.7 Design Verification Methodology

The design verification methodology that was followed consisted of three major steps:

1. Functional simulation using the Modelsim simulator.
2. Implementation in the Spartan 3 Board [22] taking actual output measurements for a proof-of-concept time.
3. Implementation in the Spartan 3 Board for an indicative run and projection to the time needed.

The three methods above were used whenever possible in the verification procedure. The first step was used during the initial design phases, where actual runs were not as important and functional correctness was a primary target. The second step was used for all the designs, although it was not possible to have actual measurements from all the designs due to the nature of the problems. For example the knight's tour provided results from simulation, which were within the time scale of milli-seconds. Increasing the chessboard size results in exponential increment in the time, many orders of magnitude. Thus, in the first case time frame is too small to be observed on the Spartan 3 Starter board, whereas in the second case time frame is too big for practical measurement. In this case, the design was run on the board as a proof of correctness, and long runs were estimated by projecting the simulation results in time.

CHAPTER 6: PERFORMANCE VALIDATION AND EVALUATION

6.1 Performance Evaluation Issues: Scaling of Time and Space

The performance of the designs described in the previous sections is subject to the size of the input parameters of the problem examined. This means that the bigger the input to the problem we implement, the lower performance we achieve. This is a problem that cannot be avoided because it is due to the nature of the FPGA structure; since logic functions are packet together in order to be implemented in LUTs (see page 6), then the packaging is done by group of four inputs (since LUTs are usually 4-to-1). This means that incrementing the input by four bits, one more logic element is added to the implementation, which in turn means that the routing network is incremented. This produces degradation of the performance by a significant factor, especially when this concerns the implementation of comparators. Thus, the scaling of area in FPGAs is a factor that causes many problems in performance, a fact that is not observed in the implementation of the same problems in general purpose processors.

Another factor that depends on the input size is the time scaling. Time scales non-linearly according to size input, because of the area scaling problems; when size gets bigger, the area has to confront routing problems, which are not analogous to the size. This means that doubling the input does not mean that the routing delays will double themselves; since the whole circuit must be interconnected, delays will be longer. This in turn causes a major degradation in the time needed to complete the operation. It is possible that when the time for an operation to complete is estimated, the projection might have a declination from reality.

In the sections that follow we present experimental results from the implementation of the three problems analyzed in previous sections. The tools that were used in order to design, implement and verify the results were the Xilinx ISE Foundation versions 9.1i and 10.1 as implementation software, the Modelsim simulator version 6.3f for simulation and the verification board was the Spartan 3 Starter Board from Digilent Inc. [22]. For the software runs, the 64-bit Intel Pentium 4 Dual Core at 3,4 GHz speed with 3 GBytes DDR2 SDRAM main memory was used.

6.2 The Knight's Tour

The Knight's Tour has been implemented for all chessboards from 5x5 up to 8x8. The design has been simulated for all mentioned boards, it has been downloaded to the FPGA development board where it was proven that it run properly. However, practical measurements could not be taken, due to the exponential time scaling that the problem presents. In **Table 8** we present the results from the place and route procedure: the board for which the design was implemented, the speed achieved, the slices and BRAMs occupied for the Spartan 3 FPGA. Notice that all designs do not present big difference in speed, because all board sizes up to 8x8 can be represented using the same amount of bits (3 in this case). The only factor that changes the design performance is the stack, because it is implemented in DRAM (see footnote on page 63). Thus, the routing network introduces different delays in each case.

Table 9 : Implementation Results for the Knight's Tour problem

Post Place and Route Implementation Results for the X3S1000-4FT256 device			
Board Size	**Speed (MHZ)**	**Slices**	**BRAMs**
5x5	146,263	415/7.680 (5%)	N/A[10]
6x6	144,802	508/7.680 (6%)	N/A[10]
7x7	147,732	611/7.680 (7%)	N/A[10]
8x8	145,560	750/7.680 (9%)	N/A[10]

The simulation results are presented in **Table 9**, where we show the problem size (board size) for which the simulation is run, the number of cycles it took to produce the results and an estimated time that needed to complete the operations. The initial coordinates provided to the circuit were known to provide a solution.

Table 10 : Simulation Results for the Knight's Tour problem

Board Size	Num. Of Cycles	Time to complete
5x5	1219	8,3 μs (microseconds)
6x6	227.757	1,57 ms
7x7	48.275.810	326,344 ms
8x8	140.011.361	961,878 ms

In **Table 10** we present comparison results that were obtained by the Xilinx ISE tool, versions 9.1i and 10.1. The tools differ in the place and route algorithms, so the results are different. Software measurements were taken from the 64-bit Intel Pentium 4 Dual Core at 3,4 GHz, with every measurement to be an average of a hundred (100) runs for each size.

Table 11 : Software-Hardware results and comparison for the Knight's Tour

	Software	**Hardware (ISE 9.1i)**	**Hardware (ISE 10.1)**	**Speed Up (SW/HW)**	
				ISE 9.1i	**ISE 10.1**
5x5	>1ms	7,7 microsecs	8,3 microsecs	(Not accurate to report)	
6x6	1 ms	1,4 ms	1,57 ms	1,40	1,57
7x7	874 ms	303,7 ms	326,344 ms	0,35	0,38
8x8	222 ms	880,8 ms	961,878 ms	3,97	4,33

Results from **Table 11** show that the speed up of software against hardware is not very big. Given the fact that software implementation was run on a microprocessor with great processing capabilities and the hardware implementation was run on a very small FPGA, then a performance speed up of about four (extreme case of an 8x8 board) indicates promising results from the hardware implementations.

[10] Due to the small size of the stack, the memory has been implemented in DRAM type (automatic selection of the synthesis tool). The DRAM has been counted in the number of slices occupied by the design.

6.3 Binary Search Algorithm on a Complete and Non-Ordered Tree

The binary search algorithm has been implemented to be used with all types of binary trees; a complete and non-ordered tree, a complete and ordered one, a non-complete and ordered and a non-complete and non-ordered tree. The implementation considers that all the data are found in a ROM inside the FPGA; each ROM-address is the number of a node, and each ROM address contains data that correspond to the data structure depicted in **Fig. 39**.

```
struct node {
        struct node *left;
        struct node *right;
        int value;
        };
```

Figure 39 : The binary tree data structure (in the C programming language)

The ROM is organized in such a way that the left portion of the data points to the left node, the right portion of the data points to the right node and the middle portion of the data represents the integer value. The portions' size depends on the depth of the tree. Every leaf has zeros in the left portion to indicate that it has no child on the left, zeros to the right to indicate that it has no right child. The left, right and middle portions of the ROM data are depicted in **Fig. 40**, where the left and right boxed portions represent the addresses to the left and right node children respectively and in the middle, the un-boxed portion, is the integer value that represents the node value. The tree constructed by such a ROM structure is shown in **Fig. 41**, where a tree of depth four (4) is given.

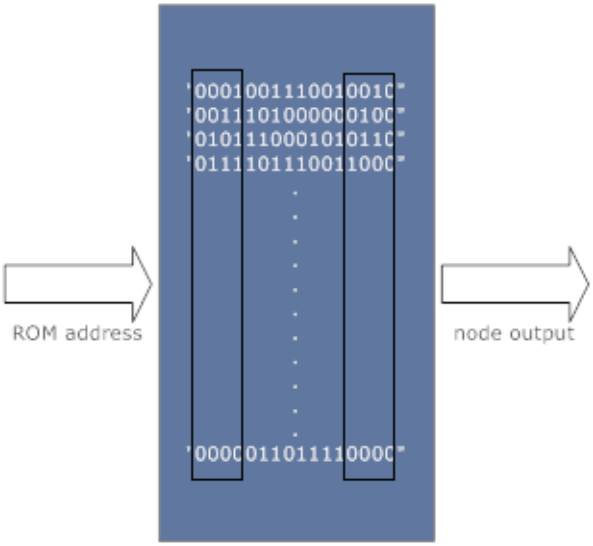


Figure 40 : ROM contents example for a binary tree of depth 4

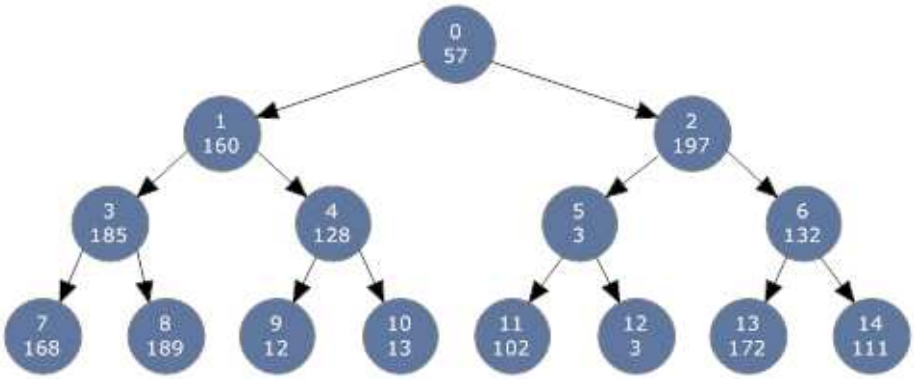


Figure 41 : A binary tree example for depth 4

In **Fig. 41**, each node has two values; the upper value indicates the node number, corresponds to the ROM address, and the value below indicates the integer that each node contains. The tree depicted in **Fig. 41** is a complete and ordered tree, but the implemented algorithm has the ability to traverse all the types of binary trees. The algorithmic state diagram of the implemented algorithm is shown in **Fig. 31**. Note that in the diagram two checks are performed; whether there is a left child and a right one. If we add more checks we could easily expand the algorithm to traverse *any N-ary tree* we want.

The binary tree search algorithm has been simulated, implemented and downloaded to the Spartan 3 board. The implementation results are shown in **Table 11**. The "Time to Complete" measures the time it took to traverse the full tree, searching for a number that it was known not to exist in the tree.

Table 12: Post place and route implementation results from the binary tree algorithm

Tree depth	Speed (MHz)	Slices occupied	Cycles to complete	Time to complete
4	164,123	33/7.680 (1%)	110	670,23 ns
8	127,081	152/7.680 (1%)	1.910	15029,79 ns

We should notice the impact that the area scalability factor has in the performance of the circuit. The algorithm that performs the traversal does not change among the implementations; the ROM's size that holds the tree to be traversed is increased exponentially according to the input. As we can see from the **Table 11**, we have a speed downgrade of about 37 MHz just by doubling the depth. Doubling the depth causes the ROM to have multiple times more size; Going from tree depth 4 to depth 8 causes the ROM to go from 15 storage positions to 255 storage positions. The stack size is not considered to be a problem, since its size should be equal to the tree depth which is small enough. Even though the design was downloaded to the FPGA board, it was impractical to take actual time measurements due to the small time run.

Also, we could not increase the depth of the tree because we needed a much bigger FPGA device. Using longer depths for the tree, so as to have long run times in the board, it can be seen that for a depth of 27 a ROM with depth of 27 bits is needed. This translates to 134.217.727 ROM addresses,

which, assuming 27 bits for each node pointer and 27 bits for numbers as values to the tree nodes, gives us a little more that 10Mbits in space. The ROM space needed is very big, which Spartan 3 platform does not dispose. Also in software implementations, when a tree for a depth of 28 was created, the process was killed from the Linux kernel, due to memory limitation factors – even though resource quotas to the process were unlimited (2GB swap memory filled and process was terminated).

6.4 Towers of Hanoi

The last algorithm that was implemented was the Towers of Hanoi. In contrast with the binary search tree described in the previous section, this design was very dependent on the input size. The only memory module used by the algorithm was the stack RAM, which increased analogous to the number of disks provided. **Table 13** shows the implementation results for different number of disks, and **Table 14** shows a comparison between software and hardware. The speed performance is downgraded in hardware, whereas the software performance remains constant for all inputs. Note that BRAM metrics are not provided due to the fact that the synthesis tool implemented the stack in DRAM type because of the small stack size. This has been accounted for in the slices occupied number.

Table 13: Post place and route results for the Towers of Hanoi

Number of Disks	Speed (MHz)	Slices Occupied
10	164,150	68/7.680 (1%)
29	143,885	84/7.680 (1%)
34	148,104	109/7.680 (1%)

Finally, even though the design achieved the speed mentioned in **Table 13**, on the Spartan board it run using the board clock provided, which was 50MHz. Thus, in **Table 14** time measurements are given for both the run in 50 MHz and an average circuit speed of 150 MHz which was reported by the implementation tools.

Table 14: Software - hardware comparison

Number of Disks	Software	Hardware (50 MHz)	Hardware (150 MHz)	Speed Up (SW/HW) 50 MHz	Speed Up (SW/HW) 150 MHz
29	6 secs 320 ms	150 secs	50 secs	23,73	7,91
34	3 mins 22 secs	81 mins	27 mins	24,77	8,25

As we can see from **Tables 13** and **14**, even for big numbers of disks, the problem can be solved in hardware in an extremely efficient way; a 3,4GHz processor is approximately 22,7 times faster (ignoring the memory advantage and the wide bus capability) than a circuit operating at an average speed of 150 MHz. However the speed up of such a processor is shown to be only about eight (8) times.

CHAPTER 7: CONCLUSIONS

In this thesis a new theory was presented that demonstrated a new approach to implement recursive algorithms in reconfigurable hardware. The key point to the theory is the recursion simplification which is very simple to be applied to a broad variety of recursive algorithms. Although the algorithms implemented have simple parameters as a common characteristic, as opposed to parameters that are expressions or functions, the concept of the recursion simplification can be expanded to more complicated forms of recursion.

The results that were produced by the algorithm implementations were promising, in area and speed terms. Area consumption by the circuits produced is minimal for all the algorithms; in the knight's tour problem, area consumption reached 10% in Spartan 3, while in all the other algorithms it reached about 1% of the Spartan's total fabric. This gives a very good perspective for the unit level parallelism, where we can implement multiple times the circuit and parallelize the operation. In the case of the knight's tour, since the circuit consumes 10% of the fabric, we can place the circuit approximately 10 times in the same FPGA. For the other algorithms, that consume about 1% of the total CLB resources, we can place the circuit as many times as there are BRAMs. In an extreme case, we could implement a hybrid solution; we could place as many circuits as there are BRAMs, and the rest of the circuits can use DRAM. This way, the number of circuits that can be placed inside the FPGA's fabric can increase significantly.

We must also underline two very important points that this thesis highlighted. The first is that recursion is possible to be implemented in reconfigurable hardware, and produce efficient circuits that can be compared to many modern microprocessors. The software reference platform used was a GNU/Linux system on a 64 bit Intel Pentium 4 Dual Core running at 3,4 GHz, having 3GB of DDR2 SDRAM. The hardware reference was a Spartan 3 Starter Kit [22] that has an estimated speed limit of about 250MHz. The stacks which were produced, along with the circuits that used them, operated very close to that speed limit. As has been demonstrated from the measurements taken, and the comparison among the software and hardware implementations, the speed benefit offered by the microprocessor is doubtful. Having as a sole criterion the processor's speed, the Pentium was found to be almost 23 times faster than the hardware circuits solving the same problem. Yet, the speed up reached only a factor of eight (8) in some cases. This speed up can be shown to be less beneficial if we consider several other factors: the Pentium's power consumption is extremely high (95W) compared to the Spartan's (for XC3S1000-4FT256 device, power consumption is in the order of 24 mW [23]); the circuits implemented in reconfigurable hardware were simple, whereas Pentium optimizes software programs in order to exploit deep pipelines and other sophisticated circuitry; the software platform had huge amounts of memory as compared to the FPGA platform: the L2 cache of the specific Pentium (2MB for each core, total 4MB) is greater than the memory offered by the Spartan 3 FPGA (432Kbits).

The second point is that all recursive algorithm implementations have a similar structure. There exist software design patterns that are used in order to produce optimized and formally verified recursive algorithms. In the same way we can use similar design patterns in order to produce hardware recursive implementations. The algorithms implemented in this thesis followed this notion, and the

control units that resulted had identical structure. This is a very important point, since this similarity can be exploited in order to produce CAD tools that automate the production of recursive algorithms, bringing recursion one step closer to the hardware designer's toolbox.

BIBLIOGRAPHY

[1] Nell Dale, *C++ Plus Data Structures 3d ed.*, Jones and Bartlett Publishers Inc., 2003

[2] http://www.kernelthread.com/hanoi/html/c.html (page last visited on June, the 27th, 2008)

[3] http://www.sparknotes.com/cs/recursion/whatisrecursion/section2.rhtml (page last visited on June, the 27th, 2008)

[4] David Pellerin, Douglas Taylor, *VHDL made easy!*, Prentice Hall PTR, 1997

[5] IEEE Standard 1076.6 for VHDL Register Transfer Level (RTL) Synthesis

[6] Peter J. Ashenden, *Recursive and repetitive hardware models in VHDL,* Technical Report TR 160/12/93/ECE

[7] Clive "Max" Maxfield, *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*, Newnes, 2004

[8] Virtex II Pro: http://www.xilinx.com/support/documentation/virtex-ii_pro_data_sheets.htm (page last visited on June, the 27th, 2008)

[9] http://www.xilinx.com/products/design_resources/proc_central/microblaze.htm (page last visited on June, the 27th, 2008)

[10] http://www.xilinx.com/products/ipcenter/picoblaze-S3-V2-Pro.htm (page last visited on June, the 27th, 2008)

[11] T. Maruyama, M. Takagi, T. Hoshino, "*Hardware Implementation Techniques for Recursive Calls and Loops*", Lecture Notes in Computer Science 1673 – The Ninth International Workshop, FPL '99, Glasgow, UK, 1999, pp. 450-455.

[12] G. Ferizis, H. ElGindy, "*Mapping recursive functions to reconfigurable hardware*", Field Programmable Logic and Applications, FPL '06, pp. 1-6, 2006

[13] S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997

[14] V. Sklyarov, "FPGA-*based implementation of recursive algorithms*", Microprocessors and Microsystems, Special Issue on FPGAs: Applications and Designs, vol. 28/5-6, pp. 197-211, 2004

[15] V. Sklyarov, "*Hierarchical finite-state machines and their use for digital control"*, IEEE Transactions on VLSI Systems, Vol 7, No 2, pp. 222-228, 1999

[16] V. Sklyarov, I. Skliarova, "*Recursive and Iterative Algorithms for N-ary Search Problems"*, in IFIP International Federation for Information Processing, Volume 218, Professional Practice in Artificial Intelligence, eds. J. Debenham, (Boston: Springer), pp. 81-90, 2006

[17] V. Sklyarov, I. Skliarova, B. Pimentel, "*FPGA-based implementation and comparison of recursive and iterative algorithms"*, Proceedings of FPL'05, Tampere, Finland, pp.235-240, 2005

[18] Spartan 3 datasheets: http://www.xilinx.com/support/documentation/spartan-3_data_sheets.htm (page last visited on June, the 27th, 2008)

[19] Spartan 3 FPGA datasheet: http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf (page last visited on June, the 27th, 2008)

[20] Stratix II FPGA datasheet: http://www.altera.com/literature/hb/stx2/stx2_sii5v1_01.pdf (page last visited on June, the 27th, 2008)

[21] Quartus II HDL coding styles: http://www.altera.com/literature/hb/qts/qts_qii51007.pdf (page last visited on June, the 27th, 2008)

[22] Spartan 3 Board: http://www.digilentinc.com (page last visited on June, the 27th, 2008)

[23] http://www.xilinx.com/cgi-bin/power_tool/power_Spartan3 (page last visited on June, the 27th, 2008)

PUBLICATIONS FROM THIS WORK

- S. Ninos, A. Dollas, "*Modeling recursion data structures for FPGA-based implementation",* International Conference on Field Programmable Logic and Applications, FPL '08, 8-10 September, Heidelberg, Germany