Study of Recursion Structures and Quicksort Implementations on Reconfigurable Logic

Advisor: Apostolos Dollas
Author: Nikos Kallitsis

February 9, 2009

Contents

Abstract

To date there is no general known way to implement **recursion on hardware**, only some proposals and experiments on the subject. This study is a research on hardware implementation models for algorithms described by recursion. It briefly states how recursion has been implemented on hardware until now and maps the advantages and disadvantages of the proposed models. Although, for over four decades, high level languages support recursion, still there is no equivalent Hardware Description Language. Initiating from these previous model proposals' results, an implementation of **quicksort** is presented as this study experimental basis. The study aims through theoretical and practical experiments on software and hardware designs, of the described by recursion quicksort algorithm, to evaluate each solution. The comparison of time and space expenses as well as the implementation costs, leads to a complete exposition of the advantages and disadvantages of each case studied. Comparisons made between software and hardware or between different hardware models help define the most profitable solution and their financial viability. Taking into consideration that memory usage poses bottlenecks in the design, various models are proposed in order to distinguish the optimum solution.

Chapter 1

Introduction

Preview of relative terms an short analysis of methods and techniques that will be mentioned later in this study.

1.1 Dialecting (Devide n Conquer)

A common method to simplify a problem is Dialecting. By using **Dialecting** method we divide the problem into sub-problems of the same type. We solve the main problem by solving the sub-problems running the same procedure for each sub-problem until all are solved.

In computer science there are two ways to do this, by **iteration**[21] or by **recursion**[19].

Iteration for some problems can be slightly faster than recursion because we dont have to pay the **function-call-overhead** as many times as recursive function calls, whose overhead in many languages is high. There are other types of problems whose solutions are inherently recursive because they need to keep track of the prior states, e.g. tree traversal, divide n conquer algorithms. All these algorithms can be implemented by using iteration but then use of stack is required which nullify the advantages of the iterative solution.

Although recursion is a powerful technique to solve problems with repeating patterns, and is a fundamental structure in software today programming languages provide less stack space to each thread than the space available in heap a fact that can lead us to iteration use instead of recursion.

However in the special case of **tail recursion**[20], in which recursion call is at the end (tail) of the algorithm and no more operations need to take place after the function returns, the caller's return position doesnt have to be saved on the call stack. Therefore, on compilers which support tail recursion optimization, tail recursion saves both space and time and can provide a better solution than

iteration.

Divide and conquer is a powerful tool for solving conceptually difficult problems and all it requires is a way of breaking the problem into sub-problems, of solving the trivial cases and of combining sub-problems to the original problem. The divide and conquer paradigm often helps in the discovery of efficient algorithms. Divide and Conquer was the key, for example, the quicksort and mergesort algorithms, and fast Fourier transforms.

1.2 Sorting Algorithms??

In computer science and mathematics, a sorting algorithm[15] is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the implementation of other algorithms (such as search and merge algorithms) that require sorted lists to work. Often it is also useful for normalizing data and producing a human-readable output. More formally, the output must satisfy two conditions:

1. The output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order).

2. The output is a permutation, or reordering, of the input.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956. Although many consider it a solved problem, useful new sorting algorithms are still being invented (for example, library sort was first published in 2004). Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space trade offs, and lower bounds.

1.3 Reconfigurable Logic Devices

A reconfigurable logic device is an electronic component used to build reconfigurable digital circuits. Unlike a logic gate, which has a fixed function, a RLD has an undefined function at the time of manufacture. Before the RLD can be

used in a circuit it must be programmed (i.e. reconfigured). Some types of Reconfigurable logic Devices are PLDs, PALs, GALs, CPLDs and FPGAs. Usually RLDs come with a set of peripherals on PCB (printed circuit board).

1.3.1 Field Programmable Gate Array F.P.G.A.

A **field-programmable gate array** (FPGA)[22] is a semiconductor device that can be configured by the customer or designer after manufacturing, hence the name "field-programmable". To program an FPGA you specify how you want the chip to work, either with a logic circuit diagram, either with a source code in a hardware description language (HDL). FPGAs can be used to implement any logical function that an application-specific integrated circuit (ASIC) could perform, but the ability to update the functionality after shipping offers advantages for many applications.

FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together" somewhat like a one-chip programmable breadboard. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

For any given semiconductor process, FPGAs are usually slower than their fixed ASIC counterparts. They also draw more power and generally achieve less functionality using a given amount of circuit complexity. But their advantages include a shorter time to market, ability to re-program in the field to fix bugs, and lower non-recurring engineering costs. Vendors can also take a middle road by developing their hardware on ordinary FPGAs, but manufacture their final version so it can no longer be modified after the design has been committed.

1.4 Application Specific Integrated Circuit A.S.I.C.

An **Application-specific integrated circuit** (ASIC)[23] is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use. For example, a chip designed solely to run a cell phone is an ASIC. In contrast, the 7400 series and 4000 series integrated circuits are logic building blocks that can be wired together for use in many different applications. Intermediate between ASICs and standard products are application specific standard products (ASSPs).

As feature sizes have shrunk and design tools improved over the years, the maximum complexity (and hence functionality) possible in an ASIC has grown from 5,000 gates to over 100 million. Modern ASICs often include entire 32-bit processors, memory blocks including ROM, RAM, EEPROM, Flash and other large building blocks. Such an ASIC is often termed as **SoC** (System-on-a-chip). Designers of digital ASICs use a hardware description language (HDL), such as Verilog or VHDL, to describe the functionality of ASICs.

Field-programmable gate arrays (FPGA) are the modern-day technology for building a breadboard or prototype from standard parts; programmable logic blocks and programmable interconnects allow the same FPGA to be used in many different applications. For smaller designs and/or lower production volumes, FPGAs may be more cost effective than an ASIC design even in production, beacuse the non-recurring engineering cost of an ASIC can run into the millions of dollars.

1.5 Memory

1.5.1 Random Access Memory

Random-access memory (usually known by its acronym, **RAM**) is a form of computer data storage. Today it takes the form of integrated circuits that allow the stored data to be accessed in any order (i.e., at random). The word random thus refers to the fact that any piece of data can be returned in a constant time, regardless of its physical location and whether or not it is related to the previous piece of data.

This contrasts with storage mechanisms such as tapes, magnetic discs and optical discs, which rely on the physical movement of the recording medium or a reading head. In these devices, the movement takes longer than the data transfer, and the retrieval time varies depending on the physical location of the next item.

The word RAM is mostly associated with volatile types of memory (such as DRAM memory modules), where the information is lost after the power is switched off.

1.5.2 Dynamic Random Access Memory

Dynamic random access memory (DRAM) is a type of random access memory that stores each bit of data in a separate capacitor within an integrated circuit. Since real capacitors leak charge, the information eventually fades unless the

capacitor charge is refreshed periodically. Because of this refresh requirement, it is a dynamic memory as opposed to SRAM and other static memory.

The advantage of DRAM is its structural simplicity: only one transistor and a capacitor are required per bit, compared to six transistors in SRAM. This allows DRAM to reach very high density. Unlike Flash memory, it is volatile memory (cf. non-volatile memory), since it loses its data when the power supply is removed.

1.5.3 Cache

In computer science, a cache is a collection of data duplicating original values stored elsewhere or computed earlier, where the original data is expensive to fetch (owing to longer access time) or to compute, compared to the cost of reading the cache. In other words, a cache is a temporary storage area where frequently accessed data can be stored for rapid access. Once the data is stored in the cache, future use can be made by accessing the cached copy rather than re-fetching or recomputing the original data, so that the average access time is shorter.

A cache has proven to be extremely effective in many areas of computing because access patterns in typical computer applications have locality of reference. There are several kinds of locality, data that are accessed close together in time (**temporal locality**) and data located physically close to each other (**spatial locality**).

1.6 C for Speed

In computing, C is a general-purpose, cross-platform, block structured, procedural, imperative computer programming language originally developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories for use with the Unix operating system.

Although C was designed for implementing system software, it is also widely used in developing application software.

It is widely used on a great many different software platforms and computer architectures, and several popular compilers exist. C has greatly influenced many other popular programming languages, most notably C++, which originally began as an extension to C.

One of the most important functions of a programming language is to provide facilities for managing memory and the objects that are stored in memory. C provides three distinct ways to allocate memory for objects:

- Static memory allocation: space for the object is provided in the binary

at compile-time; these objects have an extent (or lifetime) as long as the binary which contains them is loaded into memory.

- Automatic memory allocation: temporary objects can be stored on the stack, and this space is automatically freed and reusable after the block in which they are declared is exited.

- Dynamic memory allocation: blocks of memory of arbitrary size can be requested at run-time using library functions, such as malloc from a region of memory called the heap; these blocks persist until subsequently freed for reuse by calling the library function free.

To date programming language C is still the fastest high level programming language and also gives the developer the freedom to control memory without automated and complex memory cleaning functions, that may protect from memory leaks but retard the processing time of the algorithm.

So for memory control and speed C!

1.7 Goals of this Thesis

The first goal of this thesis is to study the previous implementers and take advantage of their knowledge in order to avoid their mistakes and continue the study. This thesis main goal is to design hardware to be optimal and run a recursive algorithms such as quicksort. We will prove that our hardware design can be faster than software but also we will define when and why our solution is optimal considering cost as well as the speedup profit and the area utilization.

1.8 Organization of this Thesis

This thesis is organized in 6 chapters starting from a sort introduction of basic terms, which are needed to understand what this study dwell with.

The second chapter is dedicated to the research done before we began to design and implement. This research was on quicksort and on previous implementers and their thesis in order to shortly present their work and map the advantages and disadvantages of their proposals and their designs.

On Chapter 3 we present our designs and the choices made for implementation. Afterwards in Chapter 4 we describe the experiments done and their results.

At last on chapter 5 we commit to paper all the conclusions that came up from the experimental results and we close this thesis with some proposals for further future implementation and experimenting.

Chapter 2

Research

Implementing requires previous research on relative subjects and study of to date known implementations and their results. This chapter is a collection and short description of the previous research on recursive algorithms and their implementations.

2.1 Recursive Algorithm Implementations on Hardaware

To date there is not any known general way to apply a recursive solutions on hardware and it is considered difficult to implement. Still there are some theoretical as well as some practical approaches to the subject.

Recursive algorithms gives an easy way to understand and deploy solutions for certain classes of problems. On the other hand recursion may be a heavy source consumer for time and memory utilization, so it should be used only when it benefits against an iteration solution. Recursion is advantageous for some problem classes and some types of recursion provides even more advantages (see 1.1).

As reconfigurable (programmable) hardware evolves with higher speed and transistor density, its efficient usage for more complex data structures becomes an important issue for research and experimenting.

Since the ALGOL 68, developed in 1973, recursion was supported from **high level languages**. Furthermore today most of the high level languages supports recursion but there is no standard hardware description language, like Verilog and VHDL, that can translate it. Because of this incognizance of recursion from **Hardware Description Languages (HDL)** there have been certain approaches to the matter, that will be briefly described in this section.

2.1.1 Maruyama, Takagi and Hoshino

The first reported method to map recursion to hardware was proposed by Tsutomu Maruyama, Masaki Takagi and Tsutomu Hoshino. The Authors performed a study on techniques for optimizing and implementing recursive calls and loops on hardware and they described a technique that uses stacks. As expected through their study they show that, by using multi-threaded execution of recursion calls instead of simple sequential execution, greater performance is gained.

Dividing the problem (see 1.1) in smaller problems and using the **pipiline** technique to parallelize the sub-problems and they were able to use all the independent stages among recursion calls and save idle cycles. By implementing the knapshack problem the authors showed that their method can gain 6.7 time speed up against software solutions. Experimenting on the case of speculative execution of parts of the knight's tour problem they showed that their method improved overall performance by a factor of 4.1. [6]

2.1.2 Valery Sklyarov

Valery Sklyarov proposed an implementations of recursion algorithms to reconfigurable logic with the use of **hierarchical graph schemes (HGS)**. The concept in these proposal is to divide the algorithm to a discrete number of modules (labeled $\mathbf{z_i}$ i.e. $\mathbf{z_1}$, $\mathbf{z_2}$, $\mathbf{z_3}$, etc) and each one will have a number of discrete states (labeled $\mathbf{a_i}$, i.e. $\mathbf{a_1}$, $\mathbf{a_2}$, $\mathbf{a_3}$, etc). Executing something like this, the use of three separate stacks is required.

These stacks will be:

1. the stack to store the current module executed, the **Module Stack**
2. the stack to store the current state of the current module, the **State Stack**
3. the stack to store the data of each operation, the **Data Stack**

the Module Stack and State Stack use the same stack pointer.

There is also a combinational circuit connected to the three stacks which operates as an interface receiving data from the stacks and produces the appropriate output for the rest of the design. See figure 2.1.

Every module has its states ($\mathbf{a_i}$), its inputs ($\mathbf{x_i}$) and outputs ($\mathbf{y_i}$). Each module begins with the state ($\mathbf{a_0}$), which is the Begin state and ends with the End state, which is labeled according to the module's number of states (in this

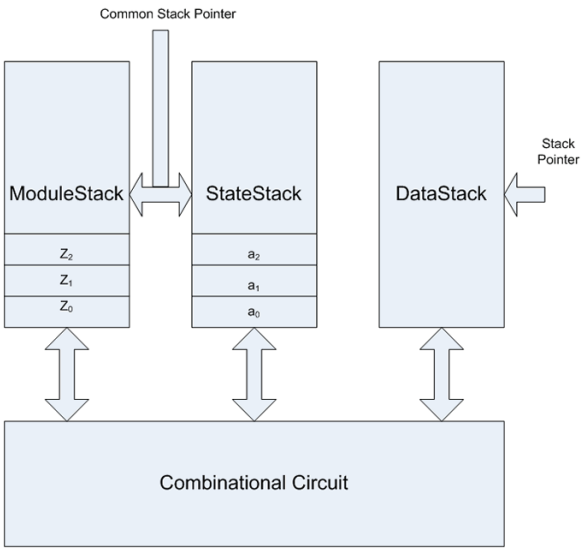

Figure 2.1: Sklyarov's Triple Stack Model

example ($\mathbf{a_3}$)). Figures 2.2 and 2.3 are two graphical examples. The module ($\mathbf{z_1}$) depicted in Fig. 2.2 is simple because it makes references to $\mathbf{z_2}$ and $\mathbf{z_4}$, whereas the module ($\mathbf{z_2}$) depicted in Fig. 2.3 is recursive because it contains a self-reference.

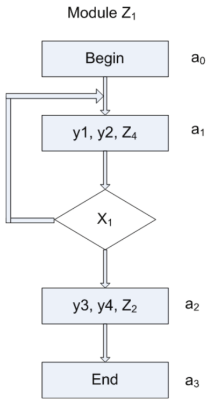

Figure 2.2: Sklyarov's Example 1

The function of the circuit is as follow: at every state the module produces the outputs ($\mathbf{y_1}$, $\mathbf{y_2}$, etc) that are inputs to the combinational circuit. At every decision point, the modules inputs ($\mathbf{x_i}$) are the outputs of the combinational circuit. Every time a module is called from another module (i.e. module $\mathbf{z_2}$ from

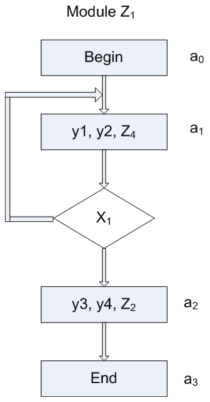

Figure 2.3: Sklyarov's Example 2

module $\mathbf{z_1}$), 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. [7],[8],[9],[10]

2.1.3 George Ferinzis and Hossam El Gindy

George Ferizis and Hossam El Gindy have proposed a different approach, avoiding completely the use of stacks in their implementation. Besides the base case and recursive case respectively, they define that the recursive part is further divided to the pre-recursive and post-recursive functions; they are respectively the part before and after the actual recursive call is made. By dividing a recursive function to these parts (non-, pre- and post-recursive) they are able to distinguish the independent parts of a recursive function and place them in a pipeline.

One key point to their implementation is the use of run-time reconfiguration. Because of the area consuming nature of their proposal, the pipeline is subject to the area constraint that is imposed by the natural size of the FPGA. This means that if the recursion is sufficiently deep, the pipeline will not suffice to cover all of the recursion depth,due to the lack of space in the FPGA. Thus, they employed the run-time reconfiguration concept in order to switch in and out of the FPGA parts of the pipeline, according to their usefulness to the processing. This, in turn, imposed new restrictions to the design of the pipeline; each pipeline stage has to be spatially equal to the smallest reconfiguration unit needed by the FPGA.

Also, the reconfiguration time is much bigger than what the FPGA can afford to wait for. The solution to this problem was to develop a branch prediction algorithm in order to pre-process which parts of the pipelining will be needed and pre-fetch them before they are needed.

Ferenzis and Grindy also proposed in their study to translate, when possible, recursion to iteration to gain iterations benefits.

The result achieved by their method is extremely high speed gain; e.g. for the Quicksort algorithm with 8,000 samples, their design needed a little under 200,000 cycles in order to complete, whereas with a stack-based solution it needed a little above 1,200,000 cycles (however, the cycle time is not in consideration in this work). [11]

2.1.4 Spyridon Ninos & Apostolos Dollas

A recent work on the subject, is Ninos Spyridon's study Modeling Recursion Data Structures for FPGA based implementation. In his study, he points the disadvantages of previous proposals and presents his model for recursion's implementation on hardware. Ninos Spyridon proposes a transition from a processing oriented design to a data oriented design. Moreover he spots that all presented solutions so far use the stack solely for data storing and that the general concept was to decide among current state and conditionals, which will be the next state. Additionally he proposes an alternative approach. In his approach the decision, on what will be the next state, is made from the data stored in the stack, when there is a return from a recursion call and not from current state and conditionals. According to Ninos this way there are three benefits:

1. Simplification of logic and hardware needed for transitions between the calls.

2. An even more compact design, since most of algorithms data is kept in stack.

3. Minimization of algorithm's data transactions, since on one action (push or pop) both data for processing and next state gets available.

Ninos data oriented recursion solves many of the problems that are present to the other proposals. First of all the area consumed is minimal. Given that today, block memories exist in every modern FPGA, by using them as the base for to implement stacks, no major area overhead is imposed.

Experimenting on Knight's Tour problem and binary tree search he achieved minimal area consumption, that allowed him to use small FPGAs for his designs. He also gained speed, with the use of block memories, for reading and writing in one cycle. [12]

2.1.5 Gathered Knowledge and Arising Problems From Existing Models

Even though previous researchers proposed solutions and achieved a very good level of overall performance, they managed it in some cost and trade-offs. The main and most important conclusion is that, on hardware designing, instead of using a general implementation model, we should choose the way to implement recursion based on the problem we want to solve and its characteristics. Generality can lead to non optimal designs. Examining each problem separately and its features is more probable to reach to optimal solutions.

Tsutomu Moruyama, Masaaki Takagi and Tsutomu Hoshino's approach put the seperate algorithm steps in a pipeline with a control unit to check the different states of the pipeline. If the pipelined processing is favored, the control unit will allow it , or else it runs a semi-sequential processing. This way speed is increased but in weight of area. According to the authors the division and parallelism they inserted provide speedup that equals the depth of the pipeline. So this solution is more usefull for big recursions where speed can be gained.

Valery Sklyarov's proposal with the use of HGS simplifies the design for many algorithms described by recursion, and introduces the software clarity of algorithm design to hardware. There are various drawbacks in his solution: the use of three stacks suggests greater area consumption than a single stack solution and insert complexity to control them that can lower whole design speed. As mentioned previously generality can lead to non optimal solutions.

George Ferizis and Hosam El Gindy proposed a model that examines primarily if recursion can be treated as iteration to benefit, as in software. Whether transforming recursion to iteration or not, they proposed to divide the problem call (or loop) and pipeline it to parallelize the sub-problems.

Ninos Spyridon approach is based on simplification of solution to gain speed and area. He proposed a one stack model which contains contexts for the next calls. This way, after a pop, the design gets the next state and the data needed to continue. This way calls can be treated separately with sole commitment to follow the order provided from stack (Last-In-First-Out).

Conclusions Epigrammatic :

1. Design based to problem first and then to General methods.
2. Simplicity for design provides less area consumption.
3. Treat recursion as Iteration if possible to gain speed.
4. Parallelize the design and the sub-problems

5. Use data on stack to decide next state.

6. Use of fast Memories, like Block Memories.

7. If the algorithm allows it, treat calls as seperate procedures.

2.2 Quicksort Algorithm

Quicksort is a well-known sorting algorithm developed by **C. A. R. Hoare** that, on average, makes **O(nlogn)** (big O notation) comparisons to sort n items. However, in the worst case, it makes O(n2) comparisons. Typically, Quicksort is significantly faster in practice than other O (nlogn) algorithms, because its inner loop can be efficiently implemented on most architectures, and in most real-world data it is possible to make design choices that minimize the probability of requiring quadratic time. Quicksort is a **comparison sort** and in efficient implementations, it is not a stable sort. [1]

2.2.1 Recursive Quicksort vs Iterative Quicksort

Quicksort algorithm since firstly developed was described as a recursive algorithm, but it is also possible to modify it to describe it as a loop. In simple pseudo code, the algorithm might be expressed as this:

```
function Quicksort ( list array)
 list less
 list greater
 variable pivot

 if length(array) <= 1 then
  return array

 pivot = select and remove a pivot value from array

 for each element in array
  if element <= pivot then
   append  to less
  else
   append  to greater

 return concatenate( Quicksort(less), pivot, Quicksort(greater))
```

Algorithm Analysis: Quicksort sorts by employing a *divide and conquer* strategy to divide a list into two sub-lists until we reach a single element list. **Quicksort can be represented as a binary tree**. The algorithm can also be described as the recursively execution of three steps.

These steps are:

1. **Pick an element, called a pivot, from the list.** Pivot can be chosen from the list in various ways, randomly or from a fixed position. A fixed position may be the first, the last or the middle element. Each case provides to the algorithm implementation with different advantages and disadvantages. Using a fixed position for pivot selection we avoid the use of a random number generator function which is time consuming. On the other hand random pivot select is like mixing the sub-list on each call. This mixing can be useful for avoiding situations that may lead the program to crash (e.g. on sorting dataset with multiple element appearances) and also in some cases can make the algorithm faster.

2. **Reorder the list,** so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

3. **Recursively sort the sub-list of lesser elements and the sub-list of greater elements.**

Notice that the algorithm only examines elements by comparing them to other elements. This makes quicksort a comparison sort. This version is also a stable sort (considering that the "for each" method retrieves elements in original order, and the pivot selected is the last among those of equal value).

The correctness of the partition algorithm is based on the following two arguments:

- On each iteration or call, all the elements processed so far are in the desired position: before the pivot if less than or equal to the pivot's value, after the pivot otherwise (loop invariant).

- Each iteration or call, leaves one fewer element to be processed (loop variant).

The correctness of the overall algorithm follows from inductive reasoning: for zero or one element, the algorithm leaves the data unchanged; for a larger data

set it produces the concatenation of two parts, elements less than or equal to the pivot and elements greater than it, themselves sorted by the recursive hypothesis.

The disadvantage of the simple version above is that it requires (n) extra storage space, which is as bad as mergesort. The additional memory allocations required can also drastically impact speed and cache performance in practical implementations. There is a more complex version which uses an **in-place** partition algorithm and can achieve the complete sort using O(nlogn) space use on average (for the call stack):

```
function partition(array, left, right, pivotIndex)
 variable pivot = array[pivotIndex]
 swap array[pivotIndex] and array[right]
 storeIndex = left
 for i from left to right     1
  if array[i]     pivotValue then
   swap array[i] and array[storeIndex]
   storeIndex = storeIndex + 1
   swap array[storeIndex] and array[right]
 return storeIndex
```

This is the **in-place** partition algorithm. It partitions the portion of the array between indexes left and right, inclusively, by moving all elements less than or equal to a[pivotIndex] to the beginning of the sub-array, leaving all the greater elements following them. In the process it also finds the final position for the pivot element, which it returns. It temporarily moves the pivot element to the end of the sub-array, so that it doesn't get in the way. Because it only uses exchanges, the final list has the same elements as the original list. Notice that an element may be exchanged multiple times before reaching its final place.

This form of the partition algorithm is not the original form; multiple variations can be found in various textbooks, such as versions not having the storeIn-

dex. However, this form is probably the easiest to understand.

Once we have this, writing quicksort itself is easy:

```
procedure quicksort(array, left, right)

 if right > left then

   select a pivot index (e.g. pivotIndex = left)

   pivotNewIndex := partition(array, left, right, pivotIndex)

   quicksort(array, left, pivotNewIndex - 1)

   quicksort(array, pivotNewIndex + 1, right)
```

However, since partition reorders elements within a partition, this version of quicksort is not a stable sort.

Chapter 3

Quicksort Implementation

3.1 Memory Model Definition

Memory space allocations and deallocations, in software and hardware as well, are major time consuming functions and may lead to stall the processing until data become available. By reducing allocations and deallocations, with the use of an **in-place** partitioning technique, time and space can be gained. With an in-place partitioning of the dataset there will be no extra memory space allocated for the dataset or its children.

Quicksort receives an unsorted array, a sequence of n unsorted elements, and produces another array, which is a sequence of n sorted elements. The main goal of this section is to define the way this array or a sub-array will be presented so the software or the hardware will be able to locate it and execute the quicksort's call procedure on it. The use of in-place partitioning, where the first allocated space is the only space needed, we can use pairs of indexes to declare a sub-array.

i.e. The whole dataset, consisted of n elements, will be placed on memory slots addressed from 0 to n-1.

Quicksort examines this array and finds that there are x elements belonging to the lesser array and the rest to the greater array. To present this two sub-arrays, two pairs of indexes are needed, 0 to x-1 slot for the lesser and x to n-1 for the greater sub-array.

After we have defined the way the array and its parts will be presented, we need to define how the array's allocated space will be located in memories. In software implementations this is an easy thing to do, with the use of a pointer that point the array's position in memory. On the other hand on hardware we can not use pointers or call a space allocation function to reserve space, so we define an initial position to be defined as the beginning of the array. This position may

be statically or dynamically defined before processing starts! Supposing we will use a statically defined begin position and assuming it is the slot addressed by 0, the only thing left to do is to find the end of the array.

Software languages support various mechanisms to find the end of an array, but in hardware we have to implement some to locate the end of the dataset. In order to do so, we set the first element to keep the index of the last element. By this simple solution we gain generality and implement a generic design that can receive any size of input dataset with only one address given. By the input address we can calculate the length of the dataset and create hardware and software that only needs an address, pointing the begin of the array, to execute quicksort.

This may not be necessary in software, as mentioned above, but in hardware although it is easy to find the begin of the dataset it is not so easy to locate the end. Defining as the begin of the array a specific memory point, lets say memory slot 0, and storing there the index of the last slot used for the last element, we have the information needed to start quicksort execution.

i.e.

Memory Address	0	1	2	3	4	5	6
Data Stored	6	$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$	$\mathbf{x_4}$	$\mathbf{x_5}$	$\mathbf{x_6}$

This way we know that the dataset is stored between slots 1 and 6, with minimum of memory slots, only one, consumed for initialization.

Using In-place partitioning as mentioned in 2.2.1 and this memory model, the only input for a call should be two indexes showing the beginning and the end of the part of the dataset for examination. On a quicksort's call return, two pairs of indices, the most, will be its output to locate its children. These two pairs will be future inputs for the quicksort. The one will be processed straight away and the other will be pushed in the stack, to be processed later.

i.e. Quicksort gets as input, indexes 2 and 6, this means that the set of elements between 2 and 6 should be processed (values at memory slots 2,3,4,5 and 6). Lets say that the whole dataset, where sorting takes place, is this:

Memory Address	1	2	3	4	5	6	7	8
Data Stored	1	3	5	3	2	2	8	6

At this point it doesn't matter what happens to the rest of the array but only what happens to the slots between the indexes. Based on quicksort algorithm, we know that all elements before the front index are sorted and all elements after

rear index are unsorted. Looking at figure 3.3, that shows how recursion tree expands, is understandable why this happens.

When quicksort completes a session, executed with middle element as pivot (3), the dataset will be changed to:

Memory Address	1	2	3	4	5	6	7	8
Data Stored	1	2	2	3	5	3	8	6

and two pairs of indexes will be the output. These two pairs will be 2-3 and 4-6. The slots 2 and 3 contain elements smaller than pivot, and the slots 4 to 6 contain elements greater than pivot.

After this call the produced two children need to be handled, one child can be processed from quicksort and the other need to be pushed to stack.

With the storage model defined, we may proceed now to the use of other techniques to optimize our software and hardware solutions.

3.2 Quicksort Algorithm

Based on the original Hoare's Quicksort algorithm and implementing it again, there are some decisions to make before continuing.

Choices which need to be made

1. Choose which will be then base comparison for dividing the array.
2. Choose how the pivot element will be selected.

Base Comparison Base comparison will divide the given array to its children sub-arrays. Which comparison would be the base to divide the array? The original algorithm chooses a pivot element and then separate the array into three parts: one that contains the lesser than pivot elements, one that contains the pivot and one that contains the greater than pivot elements.

Considering that each element is not necessarily unique, i.e. multiple element appearances, the algorithm version with partitioning at three is non optimal, because it is difficult to control multiple pivot appearances. The original algorithm merges the three partitions after a call has returned, something we want to avoid in order to make an algorithm that merges automatically. By using a condition that divides the input array just into two sub-arrays and an in-place partitioning design, we can avoid having to merge the arrays after calls return, see 2.2.1.

By using **in-place** partitioning, the storing of the sub-arrays takes place at the same, firstly allocated, space with swaps and their merging after calls return is automated. Swapping elements too often may be time consuming but it is a lot faster than waiting for space allocation and deallocation.

In this way quicksort's recursive calls, can go to the end of the algorithm and our algorithm can be considered as **tail recursion**. Having as the last things to do the two recursive calls we gain tail recursion's benefits, see 1.1.

To decide which will be the base comparison, two options appear:

1. Less and equal to pivot to the lesser child sub-array.
2. Greater or equal to the greater child sub-array.

We will use as base comparison the second option, so in case the element is smaller than the pivot it goes to the left child and when it is greater or equal to pivot it goes to the right child. This decision, we will explain later this decision.

Selecting the pivot element The pivot element can be chosen with two ways:

1. From a specified position, i.e. always the middle element of the array,
2. Dynamically (randomly), i.e. different place for each call,

Each choice has its advantages and disadvantages. Choosing to randomly the pivot for each call, can provide in most cases faster sorting, but in the other hand, running a random number generator function consumes a lot of time. So we move to the other option, a specified position of the array for the pivot. This position should be easy to calculate, like the beginning of the array (first element), the end (last element) or the middle element.

Quicksort Algorithm Description

1. The quicksort takes as input two indexes, called **current_start** and **current_end**, and the array for processing is located between them.
2. Copies the input indexes to temporary variables or registers (**temp_start** and **temp_end**).
3. Selects and copies the pivot element to a global variable or a register.
4. Starts reading from the beginning of the array to examine the array's elements.

5. if the element being examined is lesser than the pivot is stored at the slot indexed by temp_start and then temp_start is increased by one and the data in that slot is set for examination. Otherwise if the element under examination is greater or equal to pivot, it will be stored into the slot indexed by temp_end, after the element in that position has been read to be the next element for examination. Then the temp_end is decreased by one.

6. Continues performing these steps until the two temp indexes meet each other. The meet point of the indexes is the partitioning point and two pairs of indexes can be calculated to continue.

7. Calls quicksort for left child pair of indexes.

8. Calls quicksort for right child pair of indexes.

9. Return

At this point, where a description of the algorithm is available, we can explain why we have chosen greater or equal comparison instead of lesser than equal. If we had chosen lesser or equal condition and by using this algorithm is possible to reach an infinite recursion,

i.e. having to sort the values 2,8,5 with the lesser or equal going to the lesser sub-array condition we will get the same array for sorting forever. Because all values are less than or equal to the pivot (8), there are all going to be stored at the same position. 2 ¡ 8 go to position 1, 8 = 8 goto position 2 and 5 ¡8 goto position 3 again and again.

On the other hand, with grater or equal going to the greater sub-array condition, the array can be sorted. 2 ¡ 8 go to position 1, 8 = 8 go to position 3 and 5 ¡8 go to position 2, split array at two non-empty sub-arrays one contain 2,5 and one containing 8.

3.3 Recursion's Call Tree Pruning

Quicksort calls itself at most two times, one for each of its two children, until it reaches a leaf. A leaf is considered a call on an one or no element array and returns immediately. The evolution of the algorithm through the recursion's calls can be represented as a tree with two children per node the most (see figure 3.1).

With some extra logic, the algorithm, can avoid calls that have no effect on the result and gain time and space by reducing calls. In example there is no

need to get into an array which is a leaf. Furthermore examining tree nodes that are full of the same one element has no meaning and may lead the algorithm to infinite recursion.

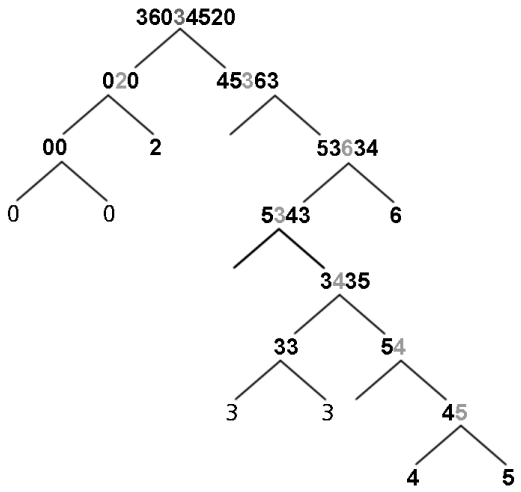

Figure 3.1: Recursion Tree example

By adding some extra functionality to the design we can identify if a sub-array is full of equal to each other elements, or is empty, or has only one element. All we need is a flag[1] for each child to signal the status of the child when a call returns. This flag for the greater or equal to pivot array will be called **all_pivot** and in its initial state it is raised. If one of the elements which are not less than the pivot is not equal to pivot the flag is lowed. So when the element examination ends and the all_pivot flag is low, it means that at least one element is non-equal to pivot and is in greater or equal to pivot sub-array. Else if the all_pivot flag is still raised it means either the array is empty or it contains multiple appearances of elements equal to the pivot, so no more sorting is needed.

Correspondingly for the less than pivot sub-array we will have a flag called **all_same** with initial state raised. For this sub-array except of a initialized raised flag, a variable is also needed to store the value of the first element examined and is less than pivot. All lesser than pivot elements will be compared with this variable also and if they are equal the flag stays raised or else the flag is lowed. Raised flag means that no more sorting is needed on the sub-array among the lesser to the pivot elements, either because it is empty or single element, either

[1]boolean variable or one bit register, that hold True/False states

because it has multiple appearances of one element.

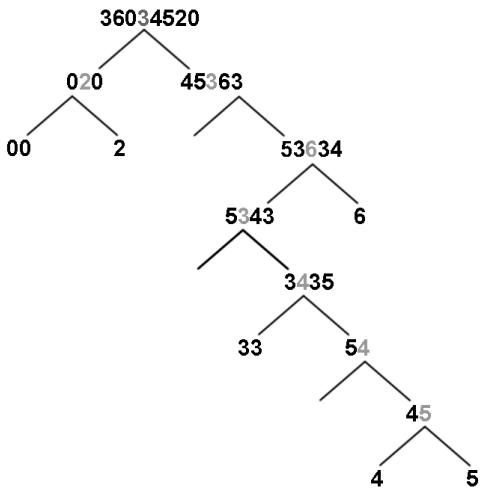

Figure 3.2: Pruned Recursion Tree example (all leafs are also pruned but shown to prove that sorting is done)

3.4 Optimized Software Implementation

Todays computers may be really fast and cheap and able to run a great variety of applications but they are restricted within the limits imposed by memories. A PC can execute many instructions very fast but there is a point that memories cannot provide the data as fast as the execution needs. Due to this memory bottleneck, optimized algorithms usually are designed to avoid frequent memory reading/writing and extra space allocations. This is accomplished with the use of intercalary memory stages, like caches, which are a lot faster. This stage will keep duplicates of memory and we can take advantage of the algorithm's spatial and temporaol locality.

As we already discussed some of the most time consuming, functions in software are memory allocations, so by minimizing memory allocations and deallocations (free), time is gained, as well as space. **In-place** memory use on algorithm designing, provide space and time reduction, by using the same space that firstly was allocated for the dataset. The algorithm should perform element swaps on this space. This way, the size of memory needed according to dataset length is known and the only non static size will be the space for stack. An

upper limit and an average limit for stack space needed can be calculated and this can be useful afterwards on hardware designing. In-place method on a divide and conquer sorting algorithm for using the firstly allocated space, is called **in-place-partitioning**. See figure 3.3.

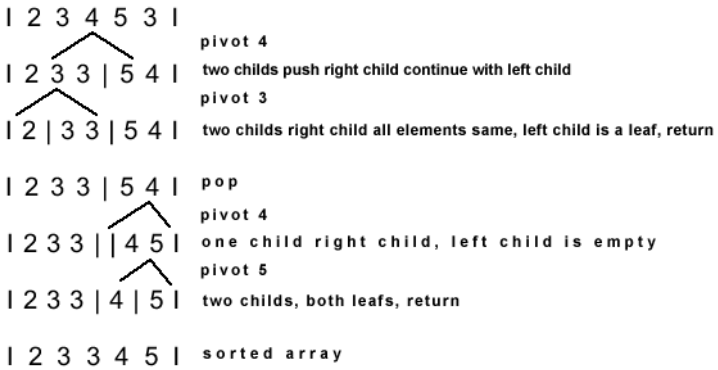

Figure 3.3: Graphical example of in place partitioning

Another useful method for software designing is described in Section 1.1.1, called **tail recursion**. Tail recursion as discussed provides a cheaper in stack space algorithm. This happens because by using tail recursion a lot less pushes to stack are needed.

Quicksort algorithm, since first developed from A.C.R.Hoare, supported multiple element appearances in the dataset (e.g. on list 2443264 there are two appearances of number 2 and three of number 4). In this case extra functionality is needed to support multiple element appearances and to protect the algorithm from getting to infinite recursion. On the other hand, with this extra logic, a heuristic pruning on the recursion tree can be applied and provide a faster and less space consuming algorithm. In these case algorithm will not proceed on a call or on a push of a list (sub-array) full of the same element. This way stack space, execution time and function-call-overhead paying are saved.

An algorithm designed with all the features mentioned should take as input arguments two array indexes, lets say start and end, and the pointer where the whole array begins. First thing to do is to copy the indexes to temporary ones and then choose the pivot element based on these bounds, lets say middle element (choose first or last element to avoid this equations). Middle element would be

the element in $\frac{\mathbf{start}}{\mathbf{2}} + \frac{\mathbf{end}}{\mathbf{2}}$ [2] position. Then a sequence of computations will be done to examine all the elements between start and end index.

This computations are:

1. Read an element (initial state read first element).
2. Compare with pivot and choose which will be the next element to process (where it should taken from the front or the rear part of the array being examined).
3. Store the compared element at the front or at the rear part of the array.

Initializing each call to begin with the element indexed from start, the processing sequence takes the element and compares it with pivot. If the current element processed is smaller than the pivot it will be stored at the front part of the array, else if current element is greater or equal than pivot, it will be stored at the rear part. This comparison's result will affects the advancement of the algorithm. At first it decides where the compared element should be stored, at the front or at the rear part of the under examination sub-array. Based on this comparison's result the algorithm also decide where the next element for comparison will be taken from.

When the compared element is smaller than the pivot, it is stored at the position pointed from start index and then the start index increases by one to show the next element at the front. If the compared element is greater or equal to the pivot, is stored at the position pointed from the end index, but before, the element at that position get swapped to a temporary variable for further processing afterwards. Then the end index is decreased by one.

This sequence of computations is executed repeatedly until the two indexes, start and end, meet each other (start increases and end decreases). At that point the examination of this sub-array is can be considered as completed and the meet point is where the in-place-partitioning will take place. The algorithm continues by calling quicksort for left child sub-array, but first we check if the sub-array has length greater than 1 and that contains elements which are not all the same. Afterwards it does the same check for the right child sub-array and call quicksort for the right child sub-array.

When the two calls on the sub-arrays returns there is no need to merge them or do anything more because all the operations take place on the firstly allocated space for the whole dataset.

[2] Finding the middle element $\mathbf{middle} = \mathbf{start} + \frac{\mathbf{end}-\mathbf{start}}{\mathbf{2}} = \mathbf{start} + \frac{\mathbf{end}}{\mathbf{2}} - \frac{\mathbf{start}}{\mathbf{2}} = \frac{\mathbf{start}}{\mathbf{2}} + \frac{\mathbf{end}}{\mathbf{2}}$

```
int swap = 0;
int temp = 0;

int pivot = 0;

int all_same  = 0;
int same = 0;

int quicksort(int start, int end, int * elements){

        pivot = elements[array->start/2 + array->end/2];

        int temp_start = start;
        int temp_end = end;

        all_same = 1;
        int all_pivot = 1;

        temp = elements[start];

        while (temp_end >= temp_start){

                if( temp  >= pivot
                        if( temp != pivot ) all_pivot = 0;

                        swap = temp;
                        temp = elements[temp_end];
                        elements[temp_end] = swap;
                        temp_end--;

                }else{
                        if(temp_start == start) same = temp;
                        if(temp != same) all_same = 0;

                        elements[temp_start] = temp;
                        temp = elements[temp_start];
                        temp_start++;

                }
        }
```

```
        if( !all_same) quicksort(start, temp_end, elements);
        if( !all_pivot) quicksort(temp_start, end, elements);

        return 0;

}
```

3.5 Hardware Implementation Models

3.5.1 General

Dedicated hardware designs are expected to be faster than software designs executed on a general purpose computer to solve the same problem.

By creating a device that has to accomplish a specific job, hardware will be designed to be optimal for this job. On the other hand a general purpose computer, as a PC, has to support many types of input/output and to accomplish a number of jobs and in many cases simultaneously. A design that handles many different problems is improbable to be optimal for all.

Analytically, a general purpose computer contains a CPU unit that can execute a specific instruction set. Software applications have to be translated to a sequence of commands (instructions), the CPU can understand and execute to solve a problem. Using these predefined instructions to synthesize the problem's solution leads to a slower design than a dedicated hardware solution,

i.e. if a PC, that uses 32 bit words, has to inverse a 64 bit vector, it has to execute at least two instructions[3]. On the other hand on hardware a 64 bit vector can be implemented as well as an 64 bit inverter to have the inversion accomplished in only one cycle. By completing the job, in half time, speed is gained, but if a 128 bit vector needs, to be inversed changes have to be made at the design[4].

Although a general purpose computer can execute these instructions very fast[5], a lot faster than a FPGA, it is also aggravated by managing various Input/output (I/O), peripherals and other system resources. All these system

[3]Supposes a Computer with pipelined stages and throughput one instruction/cycle is used.

[4]This example may look silly but it presents that hardware may be faster but it is not as flexible as a PC.

[5]a PC clock, today, has at least 10 times faster clock than a common FPGA

resources need to be controlled and function without conflicts, this is accomplished by adding more instructions between the solution's instruction sequence. These extra instructions consume time and increase the application's completion time. Apart from system resources used from the application, there may be others non-used that also need to be managed.

At this part the need of an Operating System (OS) appears, which handles all the system resources. So when an application is being executed on a PC, apart from the applications instructions, the OS commits instructions to manage the resources of the system or executes another application simultaneously. So there is time loss, because of the interfering instructions of the OS or other active applications.

On the other hand designing dedicated hardware gives the opportunity to specify and describe in low level what our hardware should do and does not have to support more portability or functionality than the needed. In simple words, ASICs or FPGAs are profitable for one job, CPUs for various jobs.

All these reasons why the amount of time for the completion of an application is greater than the time needed from dedicated hardware to accomplish the same job. Furthermore, most of the PCs do not support any parallelization, one instruction per cycle the most, but hardware designs can implement parallelization of the simpler sub-problems the problem can be divided to.

Dedicated hardware use may be a lot faster than software but programmable logic or chip production is analogically more expensive than buying a PC, and doesn't have its flexibility to run different applications. So on hardware designing the reduce of cost should be one of the main goals, considering the given benefits. Apart from the cost, another matter to consider is the convenience of each solution to be applied.

3.5.2 Quicksort on Hardware

The basic concern should be to make the solution as simple as possible and easy to implement. Based on the original Hoare's algorithm and on the software implementation of section 3.4, we will describe a solution on hardware and try to take advantage of all the parallelization we can import. Our goal is to gain speedup against software and existing hardware solutions, but also space, so we can discuss later other possible expansions.

Except of a discrete problem, the need for sorting mostly appears as a sub-problem in other problems. Taking in consideration this, we will treat quicksort as a part of a problem, in order to make some economy on resources but still gain speed against software. Assuming that, beside quicksort, on the same FPGA, the rest of the problem has to be implemented, we will design it as small as possible. Then we will prove that after a certain point of parallelization there

is no substantial gain worth of FPGA space consumption. For example taking an FPGA and trying to fully utilize it only for a sorting algorithm, as we will prove later on this study, its not beneficial. Thats because today a PC can sort via quicksort big datasets extremely fast, so even if we gain great speed-up, the solution cost will set a stopper.

Datapath overview Here we will try to briefly present all the needed parts to implement quicksort on hardware. First of all we need space to store and retrieve the data, which will be sorted. We will assume the dataset is some type of RAM, we will refer to it as main storage. Using data directly from a **RAM** will make read/write operations extremely expensive in time, and this can prove a design non-profitable. So we need to use intercalary memory stages. By placing a fast memory stage (i.e. cache) on the FPGA and a mechanism to load and unload it, we can gain time by taking advantage of spatial and time locality. By starting to load it before the element processing starts, we may have available an element per cycle to process.

The main storage connects to the memory manager, that handles the connection with **block memories**[6], but also is the interface between memory and processing.

Having an interface between memory and the rest of the design we will place a component that executes a quicksort call, and from this point it will be called **QS** for simplicity.

i.e. QS component takes as input indexes 2 and 6, this means that the set of elements between 2 and 6 should be processed (values at memory slots 2,3,4,5 and 6). Lets say that the whole dataset, where sorting takes place, is this:

Memory Address	1	2	3	4	5	6	7	8
Data Stored	1	3	5	3	2	2	8	6

At this point it doesn't matter what happens to the rest of the array but only what happens to the slots between the indexes. We know that all elements before the front index are sorted and after the rear index are unsorted. Looking at quicksort algorithm and at figure 3.3, showing how recursion tree expands, is understandable why this happens. When QS component completes a session, executed with middle element as pivot (3), the dataset will be changed to:

Memory Address	1	2	3	4	5	6	7	8
Data Stored	1	2	2	3	5	3	8	6

[6]Block memories are contained in every modern FPGA an can provide one cycle reads/writes.

and two pairs of indexes will be the output. These two pairs will be 2-3 and 4-6 and QS component will be idle. The slots 2 and 3 contain elements smaller than pivot, and the slots 4 to 6 contain elements greater than pivot.

For this call two children need to be handled, as well as in software, assuming there is only one QS component running, one child can be processed from the idle QS and the other need to be stored. For storing QS contexts[7] we need a stack, and this stack will be implemented on the FPGA to gain speed and have data from pop available in one cycle.

All these components need to be controlled, so a control unit will handle quicksort algorithm and take care for all components functional. Handling quicksort algorithm means that this unit should make sure that, data are present when needed, and in case they are not to halt the process. The control unit also handles the push/pop functions on the stack and the growth of quicksort call tree.

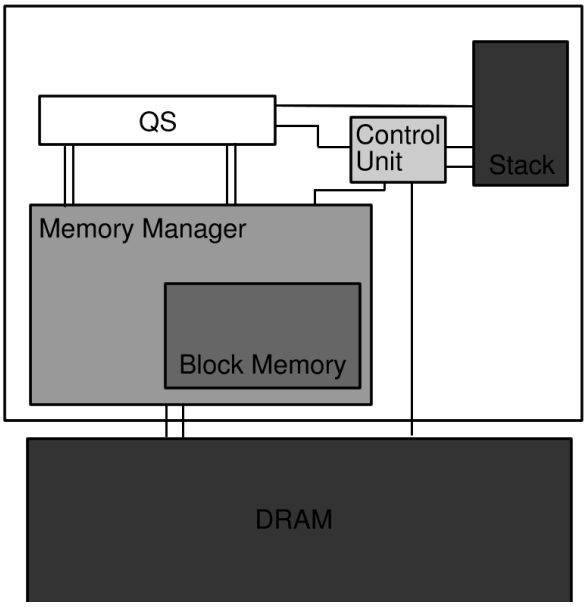

Figure 3.4: Datapath Overview

Before analyzing each component described, we will talk about the aforementioned techniques used on our software and will be implemented on our hardware.

[7]This pair of indexes present the context of a call, which will be analytically explained later.

Taking advantage of common design solutions for software and hardware we reach to the obvious and expected conclusions, that we should:

1. Use an in-place construction to avoid memory allocation and complicated index use and addressing.

2. Describe the algorithm as tail recursion to decrease the stack size needed and simplify the whole procedure[8].

3. Import heuristic ways to avoid recursion calls when they are not needed[9]. This way we can prune the quicksort call tree.

In-place partitioning and use of memory As well as in software we can gain speed if we reduce memory allocations. The whole unsorted dataset of n elements will be in a Block Memory contained on the FPGA (how data can be loaded to block Memory will be discussed later). The dataset will be stored as a sequence of elements and the only special feature needed is the first element, at memory slot indexed by 0, to contain the index of the last element. This way we may calculate the length of the dataset and the first context (input) for the QS component. The first context is indexed between memory slot 1 and the memory slot address in memory slot 0 (the first memory slot). So memory slot 0 is reserved only to provide information at the control unit and does not get sorted.

i.e.

Memory Address	0	1	2	3	4	5	6
Data Stored	6	$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$	$\mathbf{x_4}$	$\mathbf{x_5}$	$\mathbf{x_6}$

This model for data storing, allow us to design hardware which is able to work for any size of dataset with no need for extra information, except the dataset. This way our design can be implemented on future's FPGAs, that will contain even larger block memories[10].

[8] Nothing left to be done after a call returns

[9] Sub-array full of the same element or one element sub-array considered as sorted

[10] Today there are FPGAs with up to 10MB of block memory but these FPGAs are still expensive.

Pivot Position Previously we mentioned that pivot can be an element randomly chosen or an element from a predefined position of the sub-array under examination i.e. first, middle or last element. Rejecting the randomly chosen pivot solution, we have to decide which will be the specified position of the pivot.

The best solution should be to use the middle element but when the whole dataset can not be loaded, from the main storage to the Block Memories, prior to sorting. In these cases we will load an unload block memories from main storage in parts. So we should load at list three parts from memory before sorting starts the front the middle and the last part, but if we set as pivot the last element only two parts need to be loaded. Setting as pivot the first element of the array might look a good idea but we will have always a swap to do when we compare the first element with pivot and they are equal so first element goes to the end always. So we will use as pivot the last element of the under examination sub-array, from now on.

3.5.3 The QuickSort Component (QS)

QS we will call the component that handles a quicksort's call. QS component will take as input a pair of indexes, which points to the part of the dataset that has been asked to evaluate and divide. On this sub-array QS component will proceed to read,compare and store each element and when division is completed, it will output two pairs of indexes and the needed control signals **all_same** and **all_pivot**. These pairs will locate the divisions, of the input. The two control signals will inform the control unit for the children's content. If all_same is true (logic 1) then the child need to be examined, else if it is false (logic 0) then there is no need for more proceeding on this child, the same thing the all_pivot signal does but for the right child.

One of the main advantages of dedicated hardware design is that we can exploit parallelization. In order to do so for quicksort we should divide the procedure of QS, and connect the parts to work in a pipeline design. As we have already discussed we can divide a call's procedure at three theoretical stages:

1. **Read Element Stage** This stage asks memory to provide an element stored on a specific address (rd_address) considering comparison's result!

2. **Compare Element Stage** This stage compares the read with the pivot, informs **Read Element Stage** where to read from the next, the front or the rear part of the array and informs the **Store Element Stage** where to store the compared element.

3. **Store Element Stage** This stage, based on the comparison's result, sends the element to be stored at the front or at the rear part of the array.

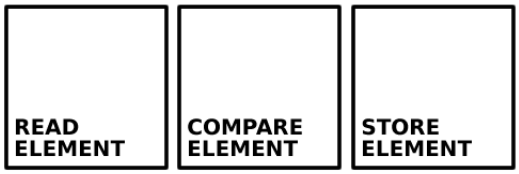

Figure 3.5: Graphical Theoretical Pipeline Stages

QS Component Input/Output

Name	I/O	type	info
current_start	in	bit vector	Where the array begins, memory address.
current_end	in	bit vector	Where the array begins, memory address.
clk	in	bit	Clock
enable	in	bit	Enable signal to turn on the component.
rd_address	out	bit vector	Address to read the next element for examination.
rd_data	in	bit vector	Data read. The next element for examination.
rd_enable	out	bit	Read enable signal.
wr_address	out	bit vector	Address to write the examined element.
wr_data	out	bit vector	Examined element to be stored.
wr_enable	out	bit	Write Enable signal.
lchild_start	out	bit vector	Left child start index.
lchild_end	out	bit vector	Left child end index.
rchild_start	out	bit vector	Right child start index.
rchild_end	out	bit vector	Right child end index.
all_same	out	bit	if all elements on lchild are equal.
all_pivot	out	bit	if all elements on rchild are equal.

Although this three stages may describe a call's procedure completely, the time consumed by memory for a read has also to be considered. With the use of fast block memories, to draw the data from, the read will take at list one cycle, so we can be sure that there is no data loss. Possible data loss can lead to storing a different dataset of the one we supposed to sort or even to putting the algorithm in a infinite sorting. To avoid all these the address to be read should be decided

and asked for memory and then wait one more cycle to get the data so we can be sure the data are read correctly.

i.e. Reading should be evolve in time in three cycles, starting from cycle x:

1. On cycle x the rd_address is set to the address of slot y.
2. On cycle x+1 the read of slot y take place.
3. On cycle x+2 the data from slot y is read and available with no data loss. At this cycle data can be compared.

At this point we should define the kind of Block Memory we will implement to connect the QS component with and see how the stages will be modified. The use of a dual port block memory gives to the design the feature to read and write at the same time in different or even in the same memory slots. By implementing a dual port block memory the previously described stages will be modified and replaced. First of all, because of the one cycle that is needed for the read, a stage that will be the connection with memories has to be added between **Read Element Stage** and **Compare Element Stage**. That leads to the following stages:

1. **Read Address Generator Stage** This stage decides, based on comparison's result, the address to for the next stage, textbfRead/Write Element Stage, for reading.
2. **Read/Write Element Stage** This stage consumes 1 cycle waiting for data to be read correctly. This stage is actually the Block Memories and provide the data to the next stage. It also transfers all its input to the next stage, **Compare Element Stage**, and at the same time based on **Store Address Generator Stage** signal writes the compared element.
3. **Compare Element Stage** This stage gets data from **Read/Write Element Stage** compares them with pivot, informs **Read Address Generator Stage** whether to read from the front or the rear part of the array and informs the **Store Address Generator Stage** where to set the wr_address to store the compared element.
4. **Store Address Generator Stage** This stage is a virtual stage that connects **Compare Element Stage** with memory, and is presented for the clarity of the design.

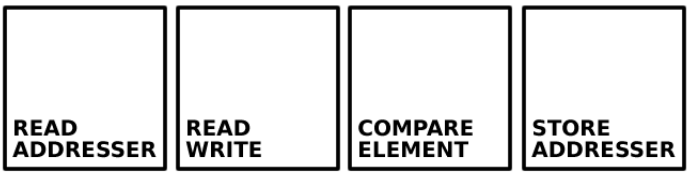

Figure 3.6: Graphical Practical Pipeline Stages

The software solution executes a sequence of commands serially and need to initialize the examination element (*temp* variable in code) to be the first element of the sub-array and then start the loop for the elements examination. Right after this element gets compared with pivot the next element for examination can be chosen. So forecasting the first element to get compared we can execute the algorithm.

On the other hand on our pipelined design a read, a comparison and a storage of an element take place at the same time and between the read and the comparison a cycle is consumed from memory for the data to get available. With forecasting only the first element to be examined there is a trap that can lead to miss-examining the dataset. By the time the first element has been compared and the comparison's result is available the second element should have been asked from memory to be read. So decisions, based on the first elements comparison, can be made for the third element and not the second. Trying to decide based on the first element comparison about where to read from the second can lead to stalling the processing, on every comparison that sends the reader to read from opposite part (i.e. if compared element was at the front part and is sent to be stored to the rear). This will cause great loss of time and there also occurs the need to control these stalls and reinitialize some stages, which will consume more area.

The solution to avoid the stalls, have an unstoppable processing of elements and gain the full advantages of our pipeline is to initialize QS component's first two cycles. By initializing QS component to read the first two elements from front part of the array, we gain unstoppable processing. The comparison of the first element will decide which will be the third element to be read, the comparison of the second element which will be the fourth element to be read and so on.

One problem arises from this solution, when a sequence of elements like this 3,7,3 is given for sorting, the QS component will return 3,7,3 again, as the right child and an empty array as left child. No mater how many time this array will be given as input to QS it will return the same thing and this will drive the design to an infinite loop and eventually a crash of the system.

By adding an extra comparison between the current_start address and the

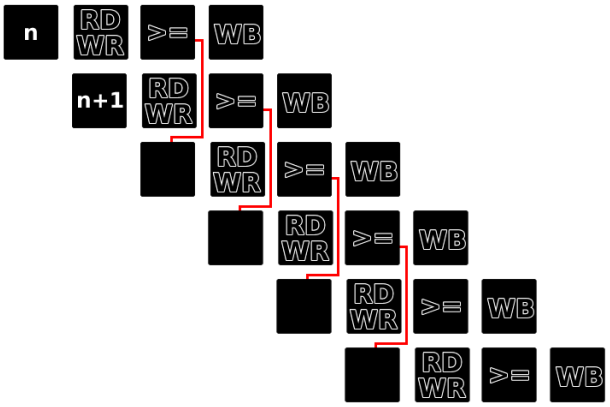

Figure 3.7: Pipeline Stages in Time Connectivity

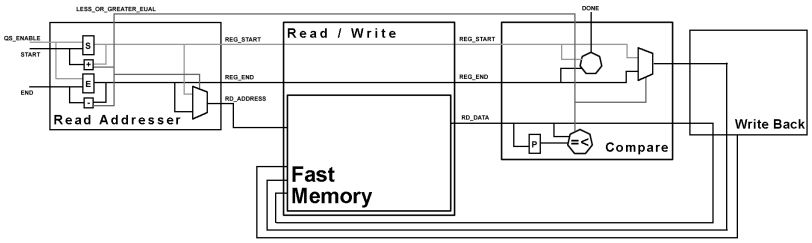

Figure 3.8: QS Design

temp_start address the component will be able to decode if the element, being compared, is the first element of the sub-array: if it is the first element of the array and it is also equal to pivot (based on the result of the base comparison) instead of storing this element at the rear part of the array, it will be stored at the front part of the array. This way the sub-array will be mixed and the elements will be reordered and part-sorted. Reordering the sub-array the problem is solved and the array can be divided correctly.

i.e. 3,7,3 reorders to 3,3,7.

Memory Slot	1	2	3
Data	3	7	3

1. Element 3 from slot 1 is compared with pivot and stored in slot 1 (front part) instead of slot 3 because it is the first element and it is equal to pivot. Also set the third element for comparison to be one element from the start.

2. Element 7 from slot 2 is compared with pivot and stored in slot 3 (rear part) because it is greater than pivot.

3. Element 3 from slot 3 is compared with pivot and stored in slot 2 (rear part) because it is greater than pivot.

4. Two children produced slot 1 as left child and slots 2,3 as right child.

1	Split	2	3
3	—	7	3

Implementing all these modifications we have a component that execute the procedure of a quicksort call with no drawbacks. This component can also be used with multiple instances, as we will describe later, for theoretical experiments.

3.5.4 The Stack Component

Describing the algorithm as tail recursion and using in-place partitioning method, a lot of advantages appears and the implementation of our stack get considerable gains. Tail recursion provides algorithms with recursive calls us the last thing to do. In-place partitioning on quicksort let as have an automated merge, because the children of a sub-array are independent. This means that the algorithm after call returns has nothing left to do, so there is no need to keep in stack any sub-array already examined. This reduces the area needed for stack and also the the push/pops.

i.e. The dataset

Memory Address	1	2	3	4	5	6
Data Stored	2	3	5	3	8	5

with pivot the last element (5 from slot 6) will be divided in to two sub-arrays,

1	2	3	Split	4	5	6
2	3	3	—	8	5	5

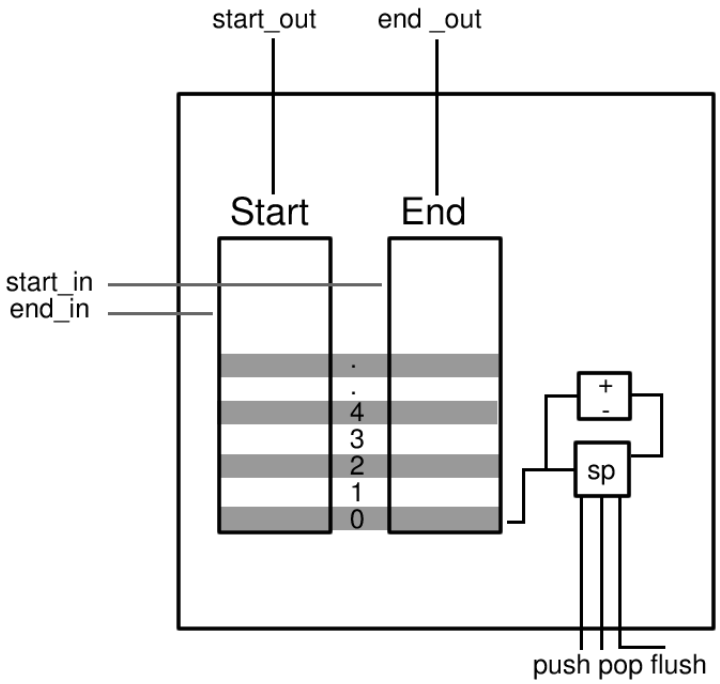

Figure 3.9: Stack Component

The two children sub-arrays are located on slots 1-3 and 4-6. The parent array is located at 1-6 and is examined without the need to keep track for this call anymore. The two pairs of indexes[11] are the only information needed from now on. The left child's pair will be sent as input to the idle QS to get examined and the right child's pair will be pushed to the stack. So we may store a very small amount of information for each call. With our design, we have to store information only for the future calls without loosing any information.

[11]Quicksort algorithm divides the array in independent sub-arrays with no common slots used.

Stack Component Input/Output

Name	I/O	type	info
flush	in	bit	Signal, for initializing the stack.
push	in	bit	Signal, for push to stack.
pop	in	bit	Signal, for pop from stack.
clk	in	bit	Clock
start_in	in	bit vector	Start of array, index to be pushed.
end_in	in	bit vector	End of array, index to be pushed.
start_out	out	bit vector	Start of array, index from pop.
end_out	out	bit vector	End of array, index from pop.
full	out	bit	Signal, raised when stack is full (sp=max_slot.
empty	out	bit	Signal, raised when stack is empty (sp=0).

The only information that needs to be kept (pushed), for each future call, is the pair of indexes that locates the partition of the sub-array the call has to examine. The stack on each push stores two bit vectors that contain the addresses for the start and the end of the sub-array, (see figure 3.9). Furthermore this implementation offers the benefit to decide the following algorithm's states only from the data stored in stack.

3.5.5 The Memory Manager Component

General On FPGA's Block memories a fast intercalary memory stage, between processing and main storage, can be constructed to provide fast reading and writing. Even when we have this fast stage we need to load it with data from somewhere, the source of the dataset may be chosen from a variety of hardware. Today FPGAs come on boards that provide connections with Hard Disk Drives (SATA, IDE), Compact Flash Cards, RAMs (SDRAM, DRAMS) etc[12]. In our experiments we will use a SDRAM Model and all assumption will be based on a Xilinx's Multi-Port Memory Controller[13], that can read from memory in **Burst-Mode**[14] 128 words in 64 blocks of 64 bits[15].

Having this MPMC as base we will calculate the time consumed for loading

[12]All these input assumptions are based on the connections provided from most of the Xilinx's FPGA boards.

[13]MPMC v4.08a

[14]Burst-mode is a generic computing term referring to any situation in which a device is transmitting data repeatedly without waiting for input from another device or waiting for an internal process to terminate before continuing the transfer of data.

[15]Equals to 2 x 32 bit. 32 bit is the common size of a PC word

the Block Memories from the SDRAM. At this point we will define that we will use 32 bit words as most of todays PC use so we can have a reliable comparison between software and hardware.

To perform enough experiments and examine a variety of possible implementations of quicksort algorithm, in theoretical and practical base, we will try to approach the memory issue, which adds the main bottleneck to the design, for two main cases.

Cases based on available area for fast memory construction:

1. The Block Memory area is enough to construct a fast memory where the whole dataset can be loaded.

2. The Block Memory area is not enough to construct a fast memory where the whole dataset can be loaded.

Memory Manager Component Input/Output Table

Name	I/O	type	info
enable	in	bit	Control Signal to activate Memory Manager.
initialize	in	bit	Control Signal to initialize to start reading from slot 0.
clk	in	bit	Clock
rd_address	in	bit vector	Address to read the next element for examination.
rd_data	out	bit vector	Data read. The next element for examination.
rd_enable	in	bit	Read enable signal.
rd_hit	out	bit	Control Signal to inform that the address exists on block memory.
wr_address	in	bit vector	Address to write the examined element.
wr_data	in	bit vector	Examined element to be stored.
wr_enable	in	bit	Write Enable signal.
read	out	bit	Signal to enable read from main storage.
address	out	bit vector	Address to read/write on main storage.
write	out	bit	Signal to enable write to main storage.
data_in	in	bit vector	Data from main storage to Block Memory.
data_out	out	bit vector	Data from Block Memory to main storage.
rdy	in	bit	Signal from main storage to inform Memory Manager that data are coming and are ready for writing to block memories.
current_start	in	bit vector	Current array, being processed, start.
current_end	in	bit vector	Current array, being processed, end.

Use of large Block Memories Block Memories, contained on most of today's modern FPGAs are large enough[16] to load on them large datasets. By using one of those FPGAs and based on the fact that our logic design will be extremely small we can load the biggest part of the dataset on the block memories and then start the element processing and continue loading until the whole dataset is on the FPGA. By parallelizing the element processing with the Block memory loading, we gain some time from loading. On the other hand this parallelization will add extra logic that will consume area on the FPGA and making it less profitable in time to worth the area, since we want to make our design as small as possible (see 3.5.2).

[16] Up to 10MB.

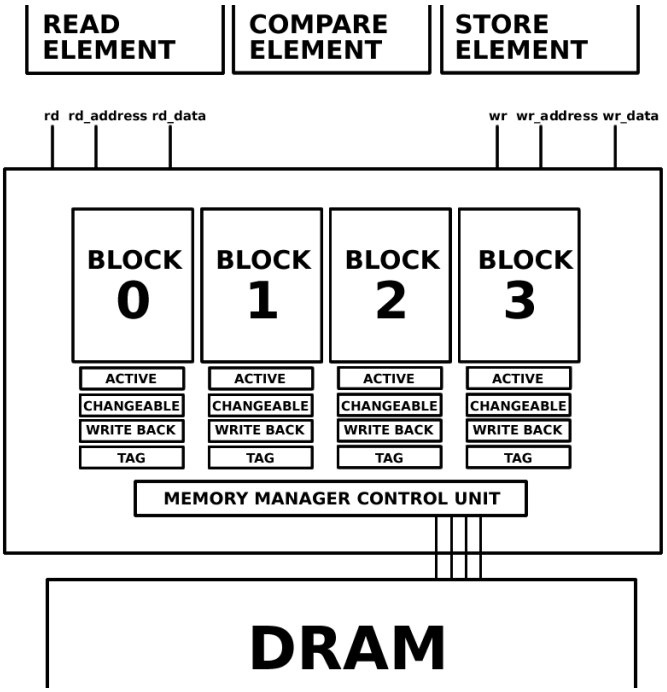

Figure 3.10: Memory Manager

Loading first the Block Memory, then starting element processing is easy to implement. The block memory will be loaded, repeatedly and serially, with 128 words every 32+128=160 cycles (initialization time + reading time) until the slot indexed with the value in first slot (see 3.1) is loaded.

Choosing to parallelize the memory, although the gain in time is not worthy of the area consumption, we have to load the biggest part of the dataset (over 50%) before starting the element processing, because element processing will be faster than the loading. Furthermore this loading can not be serial from the begin to the end of the array, since in in-place partitioning both the front and the rear part of the array are used. The need to load interchangeably from the front and the rear part appears. Reading in burst-mode first from the front part is easy to load interchangeably because after the first load we can define the end of the array and calculate the addresses to be read without extra time loss except the initialization cycles for each burst-mode read.

Use of small Block Memories Small block memories exist in many FPGAs, mostly old ones, but we should examine this case as well because, as previously discussed, we might implement quicksort as a part of a problem which may also need Block memory space. The use of small block memories lead us to

continuously read and write back parts of the dataset. All these transfers will increase the time dramatically because while the algorithm execution evolves parts of the dataset have to be transfered many times.

For this case the memory manager has to work continuously and in parallel with the rest of the design, by preloading some parts and switching afterwards the parts in block memories with the needed parts from main storage and trying to have these parts already available. Based on the MPMC we should read in burst mode the parts. But in this case we need to add logic to the Memory Manager Component to decide which part to read, based on the sub-array of the dataset that is being processed,

i.e. in this example we will refer to the memory in blocks, 0-127, 128-256 and so on:

Block Address	0	128	256	384	512
Block	0-127	128-255	256-383	384-511	512-639

If the FPGA we use can store 2 blocks only, we should load before processing starts the first block to get the address of the last block and load it too before starting the element processing. Afterwards one of these blocks will be written again and elements from blocks in main storage will be needed. Afterwards we must write back this block and replace it with the next from memory. If it is a block from the rear part of the dataset with it's previous part and if it is from the front part with it's next part.

These block swaps will consume time but in the real design we will have more than two blocks available and the memory manager can replace the written parts while it provides the processing with elements from other parts of block memories.

Flags Each part in memory will be, as long as the MPMC can provide in a burst-mode read and the block memory space will be separated in equal size parts that will have three flags to describe each part state. The first flag type is **active** and shows if this part is ready for reading and storing elements on it. The second flag type is **write_back** which shows that this part have been modified completely and is ready to go back in main storage if needed. The last flag type is **changeable** and shows that this part is either never used, either it has never been modified, (so there is no need to be written back, as it's duplicate exist in main storage).

While element processing evolves and asks for elements, the Memory Manager that has to provide them and also make some calculations to change the flags state.

1. If in one part both first and last element are written the write_back flag must be raised.

2. If a part has been written back to main storage, the write_back flag should be lowered and the changeable flag raised.

3. The active flag should be raised if this part of the block memory has been loaded at list once.

Part Identification When parts in block are loaded the Memory Manager need a way to identify their content and what part of the main memory they duplicate. To do so each part in block memory has to write a tag based on the starting address of that part in main storage.

Part in Block Memory and Identification Information

Tag	Active	changeable	Write_Back	Element x	Element x+1	...	Element x+n

The available area in Block memory will be separated in two: one half will be used to load duplications of the font part of the dataset and the other half to load duplications of the rear part of the dataset.

The Memory Manager component, in this case, will take two indices, from current registers, and start loading to block memories alternatively parts from the front and the rear part of the dataset until the available area is full and there is no part signaled for write back. Then the Memory Manager will wait until one of the loaded parts is set for write back. This part will be replaced and when the replacement is over the Memory Manage will wait until another part's write_back flag is raised.

This way we have Memory Manager work in parallel with Element Processing most of the time, except in some cases where the dataset is too long and the use of small Block Memories will not supply a profitable enough solution worthy of the cost of an FPGA.

3.5.6 The Control Unit

Control Unit is the component which is responsible for all other components to work together without conflicts, in order to avoid possible data loss. and assures that sorting will be accomplished and that it will be reliable. This component is a FSM[17] which controls Memory Manager component, the Stack and Element Processing components at the same time. Finally it checks the algorithm evolution and manages decisions to proceed on a recursion's call.

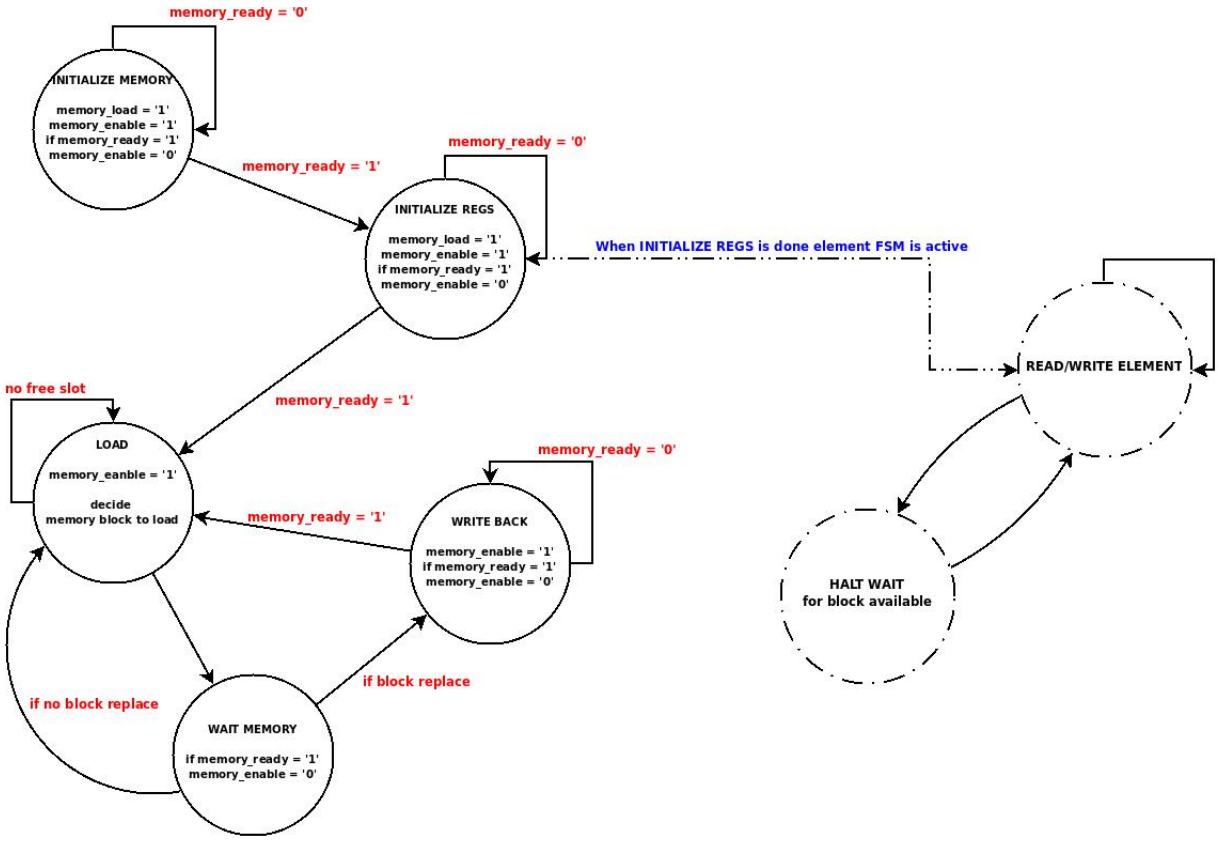

Figure 3.11: Control Unit F.S.M.

[17] **F**inite **S**tate **M**achine

Control Unit Component Input/Output Table

Name	I/O	type	info
start	in	bit	Signal to Enable and start the Design.
clk	in	bit	Clock
mm_enable	out	bit	Signal to enable Memory Manager.
rd_hit	in	bit	Signal to check Memory Manager's and QS status.
Current_start	out	bit vector	Input for Current Reg Pair of indexes
Current_end	out	bit vector	That going to be examined
push	out	bit	Signal to control stack.
pop	out	bit	Signal to control stack.
flush	out	bit	Signal to control stack.
start_out	out	bit vector	Pair of indexes to
end_out	out	bit vector	be pushed to stack
start_in	in	bit vector	Pair of indexes
end_in	in	bit vector	popped from stack
qs_enable	out	bit	Signal to enable QS component.
all_same	in	bit	Signal from QS, to inform about left child's content.
all_pivot	in	bit	Signal from QS, to inform about right child's content.
lchild_start	in	bit vector	Pair of indexes to
lchild_end	in	bit vector	locate left child
rchild_start	in	bit vector	Pair of indexes to
rchild_end	in	bit vector	locate right child
idle	in	bit	Signal from QS about it's status (active/idle)
finish	out	bit	Signal enabled when sorting is completed.

Control Unit will firstly initialize the MPMC and the Memory manager to start loading the Block Memories. While the first part is being loaded the stack will be flushed to get the stack pinter initialized to 0[18]. When the first element from the first part is ready to get stored in Block Memories, it will also be stored at current_end register and the current_start register will be initialized from Control Unit to address 1. By this the current registers will be initialized to contain the pair of indexes that locates the whole dataset.

With current registers initialized the control unit will enable the QS component, which will get the current registers as input and initialize the temp registers. Afterwards it will ask Memory manager to provide the element from slot indexed by current_end to set it as pivot (last element of the array being processed). This

[18]sp=0 <=> Empty Stack

Element will not available in block memory until the load of the rear part ends. Memory Manager will signal, by raising rd_hit, that it is ready to provide that data to Element Processing.

Control Unit waits until Element Processing is over and then checks the QS component output. If QS gives as output two pairs, the one (left child) will be sent for processing and the other will be pushed to stack or, if multiple instances of QS component are available, to another QS[19] for processing. If QS component outputs only one pair, cause one of the children is either empty, full or the same element and doesn't need further examination, it will send this pair for processing and do nothing else. If QS component does not output any children then QS goes idle and the control Unit asks stack, if not empty, to pop a unexamined pair for processing. If a pop is declared but the stack is empty that means the sorting is accomplished and algorithm is complete and control unit starts Block memory unload procedure.

3.5.7 Recapitulating

After describing the basic functionality of each component of our design's datapath we can present a more detailed graphical overview of it.

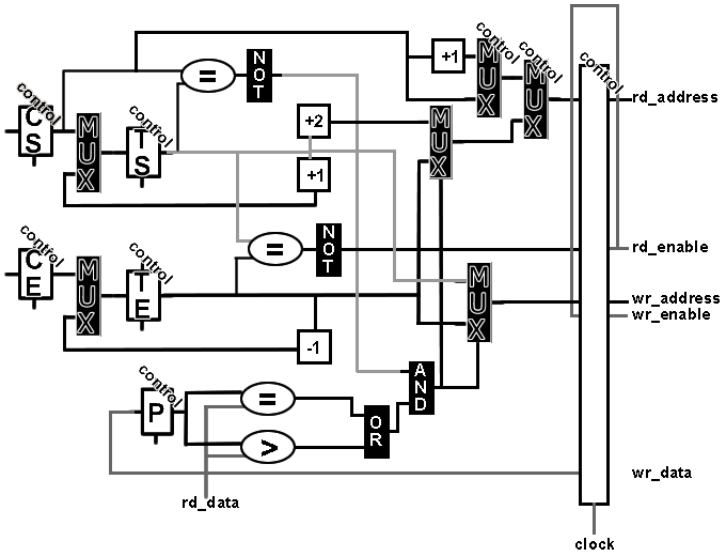

Figure 3.12: QS Circuit in Simple Parts

[19]On multiple QS design each component has an id 0,1,2,etc and the lowest id gets higher priority. Check from 0 to n which QS is idle.

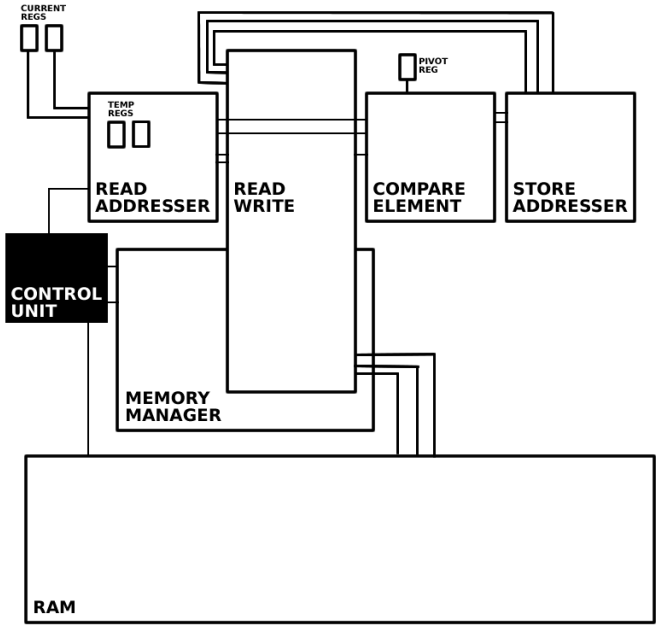

Figure 3.13: General Datapath

In this diagram the use of block memory as a stage in pipiline can be comprehensible. The connection of the Block Memories with the Memory manager at the same time is accomplished by not loading the parts of memories being used for processing element at the given time.

We may also describe the base circuit of QS component for element iterative examination and its pipelined stages in simple logical and arithmetical parts.

Chapter 4

Experiments

4.1 General

After the research on previous implementations of recursive algorithms on hardware and the presentation of our quicksort implementations, in this chapter, we will show the experimental results and map the advantages and the disadvantages of each solution.

Experiments

1. Aplied Experiments: Proof that our implementations work!
   (a) Practical Experiments on our software implementations.
   (b) Post Place and Route Simulation for the design that loads the whole dataset before element processing starts.
2. Functional Simulations: Proposals for future implementation and consideration. Conclusions and proofs for functional models.
   (a) Experiments with the download-able designs and comparison with the functional simulation results to proof they match.
   (b) Multiple QS instances experimental design and the functional prove that after a certain point quicksort's calls parallelization does not has major benefits. Examine what level of call parallelization has the major gain in overall performance

4.2 Hardware and Software Equipment

For all experiments performed, on both software and hardware, we used words that had a size of 32 bit, which is the most usual word length. On all of our hardware designs the word length can be modified (longer or sorter) from generic variables in the VHDL code. In our designs we used datasets with integers that were randomly generated from C, python and bash scripts. Using the syntax of xilinx's .coe files we created input files for software and hardware designs, in order to make tests with exactly the same datasets.

For all the hardware experiments presented on this study we used a Xilinx Virtex2P XC2VP30 FPGA. All the experiments on software were executed on a PC and with an Intell Pentium Core Duo 2.6 GHz CPU and 2GB dual port DDR2 RAM. All software experiments were executed on an operating system free of burden.

In order to test quicksort we created and used scripts in c, python and bash language to create random input datasets in .coe format based on xilinx's coe syntax but also for checking the sorted output. We check a sorted output for being sorted correctly but compare it with the input to assure that the output contains the same elements with pivot.

4.3 Practical Experiments

4.3.1 Practical Experiments on Software Implementation

The operating system, where all the software experiments were executed on, was Ubuntu v8.04 and the sorting time was counted with shell's time function and with the use of C language's library time.h. We present real time but and the user execution time. Comparing the real and the user (see figure 4.1) time proves that the computer the experiment took place on was free of other burden.

Software With the Dataset Loaded From File

Dataset	Average Sorting Time	
Elements	Real	User
100	0.001s	0.00s
1000	0.006s	0.004s
10000	0.050s	0.043s
100000	0.49s	0.040s

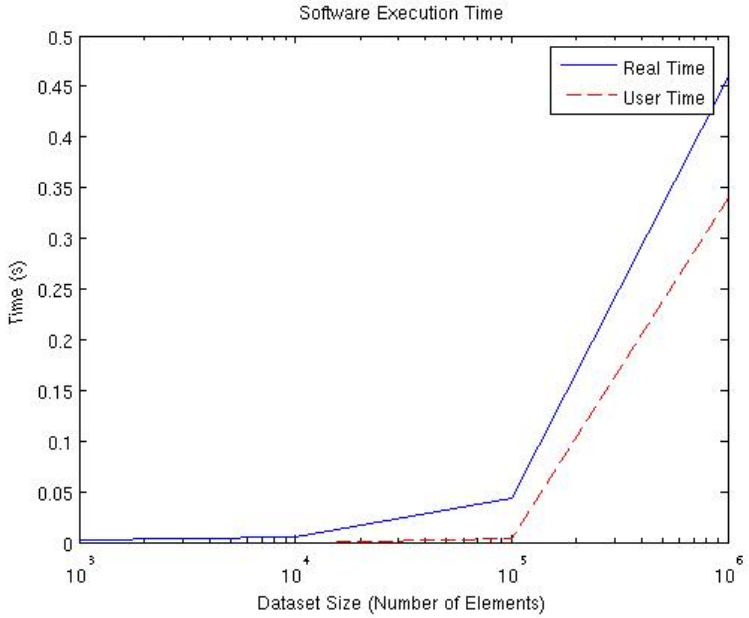

Figure 4.1: Real and User Software Time

Software With the Dataset Created In RAM

Dataset	Average Sorting Time	
Elements	Real	User
100	0.001s	0.00s
1000	0.002s	0.00s
10000	0.005s	0.003s
100000	0.043s	0.34s

Software With Random Pivot selection

Dataset	Average Sorting Time		
Elements	Real	User	System
100	0.001s	0.00s	0.0004
1000	0.001s	0.004s	0.00
10000	0.004s	0.004s	0.00
100000	0.033s	0.32s	0.004

4.3.2 Hardware Design Area Utilization

Based on Xilinx's ISE implementation reports, we present the FPGA's area utilization for the complete design implementation with the Memory Manager that loads the whole dataset but also for only the QS component to prove their small size.

FPGA Resources Consumption for the Complete Design

Logic Utilization	**Used**	**Available**	**Utilization**
Slice Flip Flops	311	27,392	1%
4 input LUTs	504	27,392	1%
Logic Distribution			
Occupied Slices	346	13,696	2%
Related Logic Slices	346	346	100%
Unrelated Logic Slices	0	346	0%
Total of 4 input LUTs	505	27,392	1%
Used as logic	504		
Used as Route-Thru	1		
Bonded IOBs	35	556	6%
RAMB16s	4	136	2%
BUFGMUXs	1	16	6%

4.3.3 Post Place and Route Simulations

In order for our experimental results to be accurate, we made post place and route simulations for some designs and for two datasets of different size. Before post place and route simulations for the whole design we did for the QS component which reached frequency of 255MHz and consume based on Xilinx ISE 4% of the FPGA area(this means it is extremely small). Based on our design that uses large Block Memories and loads the dataset on them before the element processing starts, the whole design accomplished in post-route simulation the frequency of 185 MHz. The sorting results for 10000 and 100000 was 0.00009s and 0.008s. The place and route simulation results match the theoretical results for sorting since the data are stored in Block Memories.

[1]Lookup Table (LUT) is a data structure, usually an array or associative array, often used to replace a runtime computation with a simpler array indexing operation.

[2]Input/Output Block (IOB) register packing for Xilinx Virtex devices.

FPGA Resources Consumption for the QS component

Logic Utilization	**Used**	**Available**	**Utilization**
Slice Flip Flops	171	27,392	1%
4 input LUTs	280	27,392	1%
Logic Distribution			
Occupied Slices	200	13,696	2%
Related Logic Slices	200	200	100%
Unrelated Logic Slices	0	200	0%
Total of 4 input LUTs	280	27,392	1%
Bonded IOBs	128	556	23%
BUFGMUXs	1	16	6%

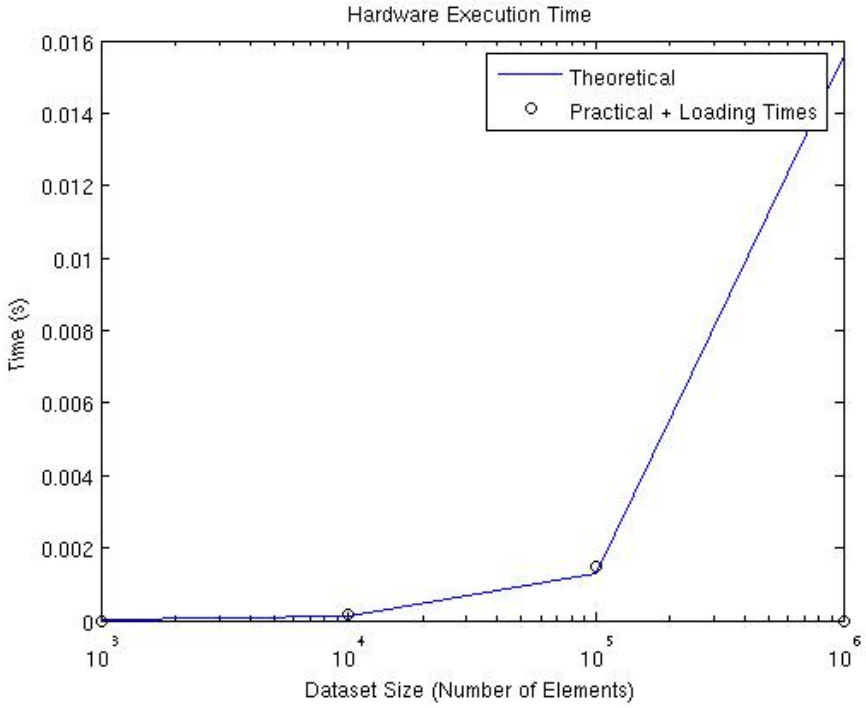

Figure 4.2: Functional vs Post Place and Route

4.3.4 Practical Experiments Conclusions

The first conclusions from this study comes from the practical experiments on hardware and software implementations we executed. Software solution in a common today's PC showed that quicksort can be extremely fast, but on the other hand our hardware implementation proved that dedicated hardware design can do it even faster. Although our design gives great speedup, 20x, it may not be enough to justify the cost of an FPGA or the manufacturing of ASIC's. As we discussed in section 3.5.2 Quicksort usually occurs as a sub-algorithm in bigger algorithms that need fast sorting and not only as a stand alone problem. All placed implementations consumed a maximum of 4% of the area of a Virtex 2 Pro which leaves most of the space free to place the circuit to solve other parts of the algorithm and make our solution useful. The area utilization tables shows that our design can fit in smaller FPGAs as well.

4.4 Functional Simulation Experiments

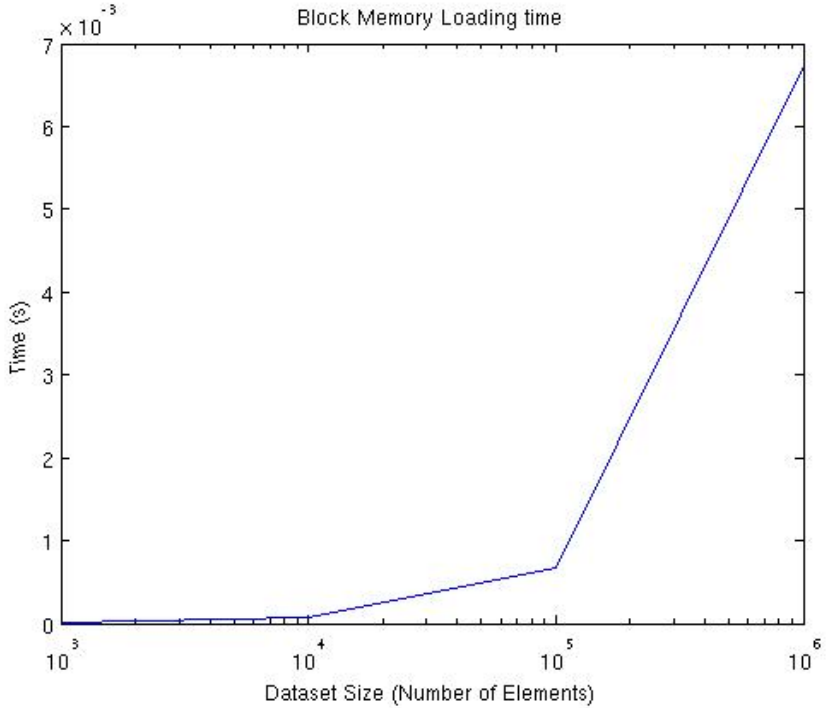

Figure 4.3: Dataset Loading Time

Loading Times

Dataset Elements	Loading Time
100	8.64 * 10^{-7}s
1000	6.054 * 10^{-6}s
10000	6.74 * 10^{-5}s
100000	6.75 * 10^{-4}s

Hardware Sorting Time

Dataset's Elements	Average Sorting Time
100	7.58 * 10^{-6}s
1000	9.74 * 10^{-5}s
10000	1.28 * 10^{-3}s
100000	1.56 * 10^{-2}s

Although theoretical experiments took place first, we will present them after having the practical experiments presented as proposals for further studying. Testing solutions in simulation may not be so accurate as post and place simulation but it is an easier way to examine some design ideas.

At first we tested all our design with a SDRAM model based on Xilinx's MPMC v4.08a but also we performed some tests on implementation based on the fact that the dataset is loaded on the Block Memories.

All the results matched the post n place results but in order to compare them with software we must calculate the needed time to load the dataset on the block memories.

By adding the loading times in the sorting times we have the time needed to compare with software. As we already discussed on 4.3.4 we gain speed against software but not enough to justify the cost of an FPGA. On the other hand our design provides a really small, in area consumption, design that makes it profitable for using it as a subproblem.

This whole study is basically a study on how we may divide and parallelize an algorithm in parts, which operate together and simultaneously, to solve a problem. We divided and parallelized the procedure of Quicksort call and gained time from these. Moreover we parallelized the loading of an intercalary memory with the element processing and gain some more time.

Whats left to parallelize?

Quicksort algorithm's sub-calls are independent so we may parallelize the calls,

i.e as we discussed using in-place partitioning and tail-recursion method we gain an algorithm that does quicksort's merge automatically. After a QS examines this array:

Memory Slot	1	2	3	4	5
Data	1	7	2	5	3

It will output two pairs of indexes that present this two sub-arrays:

1	2	Split	3	4	5
1	2	—	5	3	7

These two sub-arrays have no common slot to refer to so based on our in-place, tail-recursive, quicksort version we may examine, if we have more than one QS, at the same time. Furthermore each time that a pair of indexes has to be pushed on the stack , in a design with multiple QS instances it may be sent for processing to an idle QS. In case there is no idle QS then the pair would be pushed to stack. Our QS component is extremely small so if we have a Block Memory with multiple input/output ports we may connect some QS component to operate at the same time so they will never ask to read or write the same memory slot.

4.4.1 Functional Testing Multiple QS Design

We tested this design with functional simulations for a variety of datasets 100, 1000, 10000, 100000 and for design that used different number of instances. Some experiments have no practical meaning and even if the Block Memories obtain the needed connectivity it would be worthless to download these designs. On the other hand these experiments help us reach to conclusion about the level of call parallelization that its gain is profitable.

4.4.2 Time vs. QS Instances Figures and Tables

From this table and the commensurate figures we can see that after a certain point of call parallelization no major profit is gained.

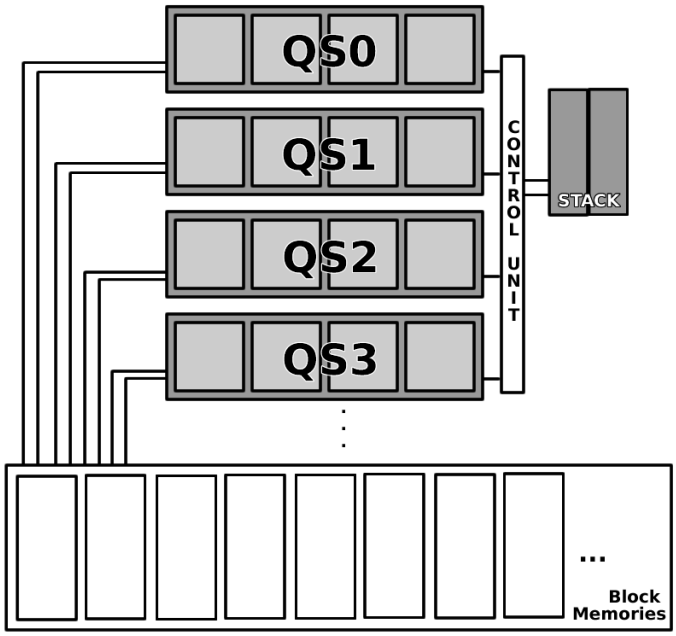

Figure 4.4: Theoretical Multiple QS Design

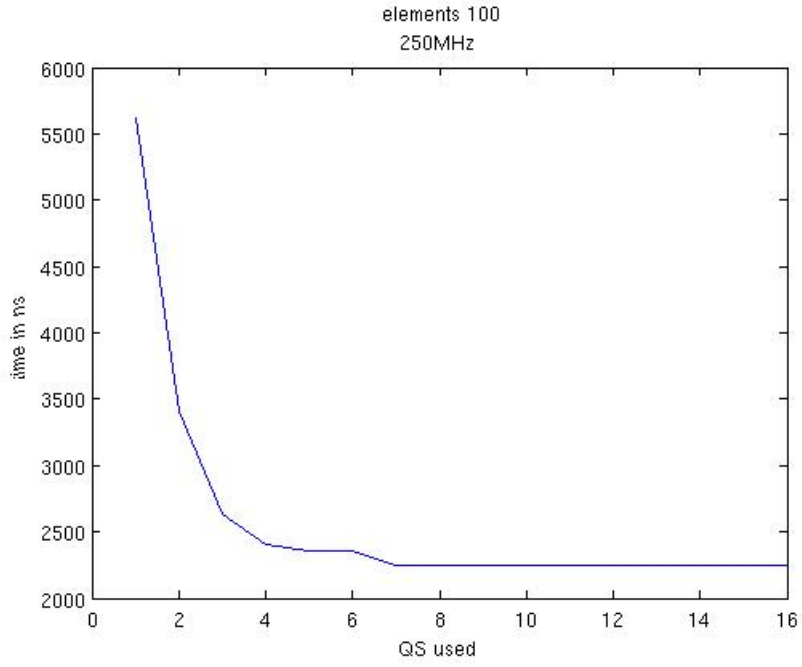

Figure 4.5: Multiple QS Instances for 100 Elements

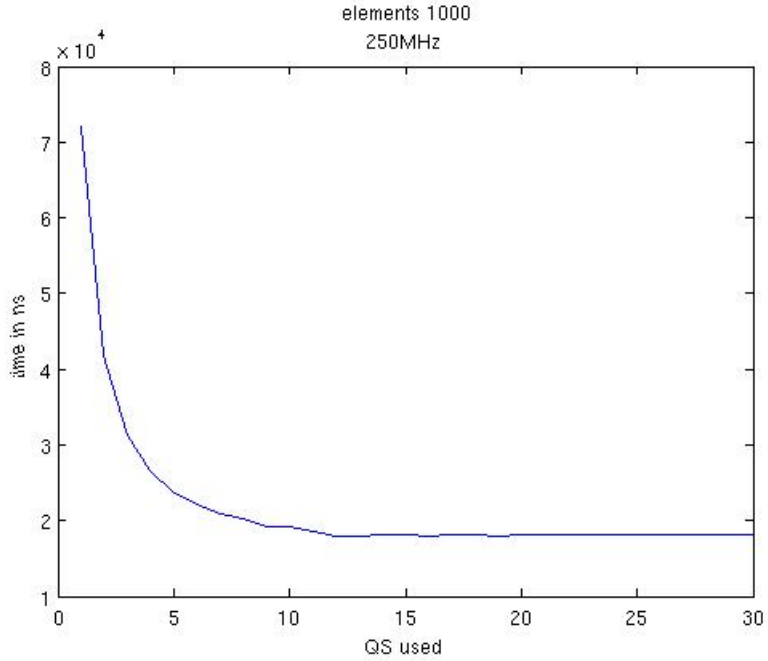

Figure 4.6: Multiple QS Instances for 1000 Elements

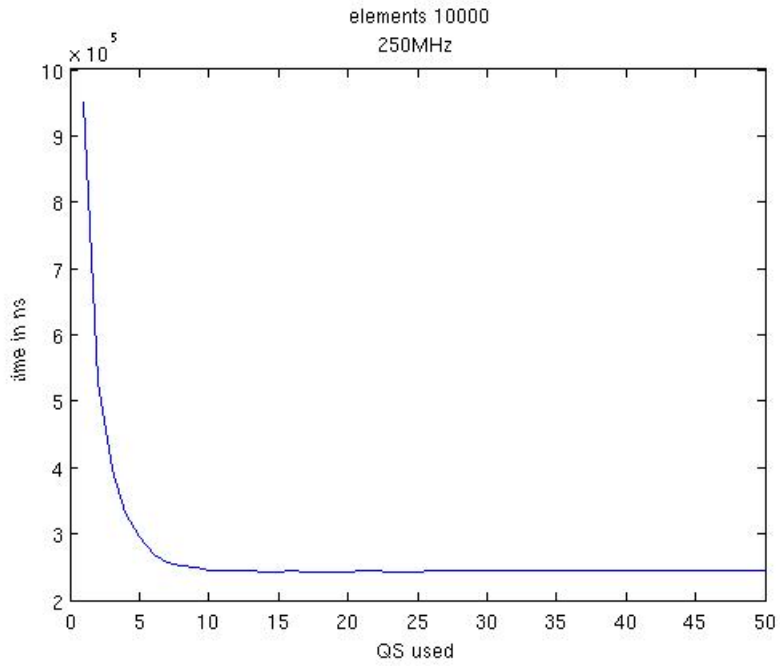

Figure 4.7: Multiple QS Instances for 10000 Elements

Multiple QS Design Sorting Time

QS	Sorting Time (ns)			
Instances	100	1000	10000	100000
1	5616	72120	949712	11554964
2	3404	41796	530532	6297268
3	2636	31312	396048	4497012
4	2400	26496	332408	3644184
5	2360	23652	294672	3181668
6	2360	22084	269920	2887332
7	2244	20836	256252	2716048
8	2244	20180	251316	2668712
9	2244	19156	249060	2564708
10	2244	19192	245296	2534496
11	2244	18496	244776	2494936
12	2244	18004	244160	2490204
13	2244	17952	243648	2476224
14	2244	18176	243472	2463340
15	2244	18060	243432	2459768
16	2244	18000	243584	2449568
17	2244	18180	243500	2444800
18	2244	18160	243496	2440172
19	2244	17960	243456	2437696
20	2244	18132	243432	2434060
21	2244	18132	243556	2432460
22	2244	18132	243564	2429052
23	2244	18132	243472	2426868
24	2244	18132	243460	2430532
25	2244	18132	243504	2428516

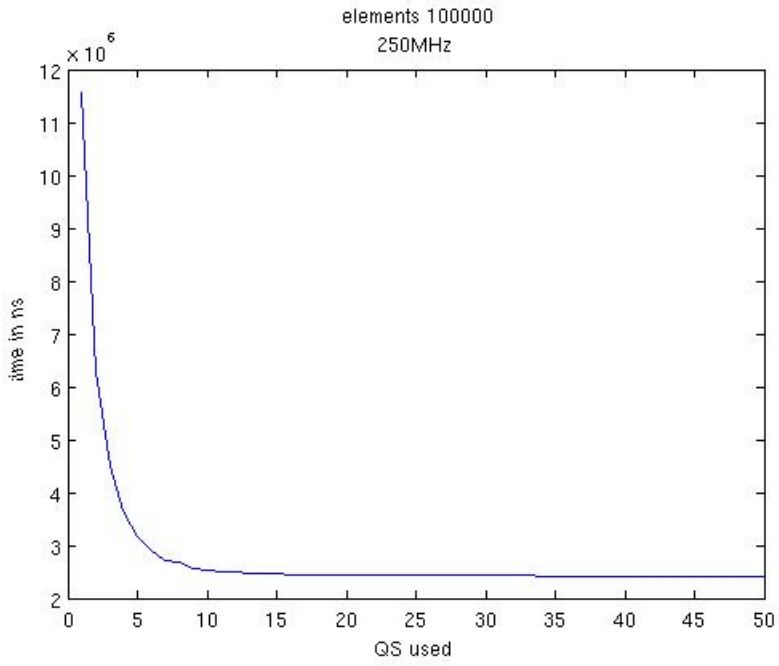

Figure 4.8: Multiple QS Instances for 100000 Elements

4.4.3 Functional Simulation's Conclusions

With multiple QS instances we could gain great speed up against any software implementation, but we have to set a limitation to the call parallelization level. This limitation occurred because:

1. There is no FPGA that can fit in it 30 QS Instances and their control unit.
2. There is no Block Memory with all these inputs/outputs and if there was it would be extremely slow.
3. From the functinal simulation's Figures 4.5,4.6,4.7,4.8 we can understand that there is no major gain after a certain point of parallelization. A good level of parallelization is around four instances. From one to four instances the curve shows that there is great gain in time for any size of dataset.

A Block Memory to provide four sets of input/output ports to connect four QS component can be designed with the interfering of another intercalary memory stage between Block Memory and Processing, but this is an idea for further researching.

4.5 Quicksort Worst Case

The worst case of a sorting algorithm, which reorders a list in order to put the elements in a numerical increasing order, such as Quicksort, is all the elements to be different and to be in completely reversed order,

i.e. a list that contains the numbers 1,2,3,4,5,6 has as worst case:

6	5	4	3	2	1

Executing our Algorithm on Hardware that:

- Has as base comparison: elements smaller than pivot goes to left child list and the rest to the right child list.
- Select as the pivot element the last element from the list.
- Use in-Place partitioning and perform swaps on the firstly allocated space.

we will prove, by an example, that the worst case scenario does not need any extra space for stack for any size of dataset.

1. Input dataset is | 6 | 5 | 4 | 3 | 2 | 1 |
2. For pivot 1 the input is divided at an empty left child and at this right child, | 4 | 3 | 2 | 1 | 5 | 6 |. Only one child need to be processed so it will be the next for examination.
3. For pivot 6 the input is divided at | 4 | 3 | 2 | 1 | 5 | — | 6 |. The right child does not need to be processed (one element list[1] so quicksort is called for left child list only.
4. For pivot 5 the input | 4 | 3 | 2 | 1 | 5 | will be divided at | 4 | 3 | 2 | 1 | — | 5 | the right child does not to be processed (one element list) so quicksort is called for left child list only.
5. For pivot 1 the input | 4 | 3 | 2 | 1 | will be divided at an empty left child and at this right child, | 2 | 1 | 3 | 4 |. Only one child need to be processed so it will be the next for examination.
6. For pivot 4 the input | 2 | 1 | 3 | 4 | is divided at | 2 | 1 | 3 | — | 4 | the right child does not need to be processed (one element list) so quicksort is called for left child list only.

[1] One element List is considered Sorted.

7. For pivot 3 the input | 2 | 1 | 3 | is divided at | 2 | 1 | — | 3 | the right child does not need to be processed (one element list) so quicksort is called for left child list only.

8. For pivot 1 the input is divided at an empty left child and at this right child. | 1 | 2 |. Only one child need to be processed so it will be the next for examination.

9. For pivot 2 the input | 1 | 2 | is divided at | 1 | — | 2 |. We have two sub-lists that do not need further sorting (one element lists).

10. Dataset is sorted | 1 | 2 | 3 | 4 | 5 | 6 |

In order to sort the worst case input dataset there is actually no push needed. Based to our approach ans implementation of the algorithm although it will take more time it will not ask for any push on the stack.

4.6 Verification of the Design

In order to verify our experimental results and that the design is actually working properly we performed checks on the outputs to verify:

- That there is no data loss or any conflicts that may have changed the input dataset.
- That the output is sorted.

In order to do so we wrote the output on files and compared it with the input to check that it contains the same elements. Afterwards using python scripts we verified if the output is sorted. By writing files we can verify the integrity of our design from functional simulations and also the software designs, where writing to files is possible. So after testing the output of various sizes inputs on functional simulations we had to verify that our design work properly and in Post Place and Route. In order to verify the post place and route simulations we performed simulations that after the sorting was complete it started reading and output the contents of the memories. This way we check through simulation output that it works correctly.

To verify that all parts work properly we tested them with seperate test-benches and performed Xilinx's Implamentation to verify that all parts, except memory models, are downloadable.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

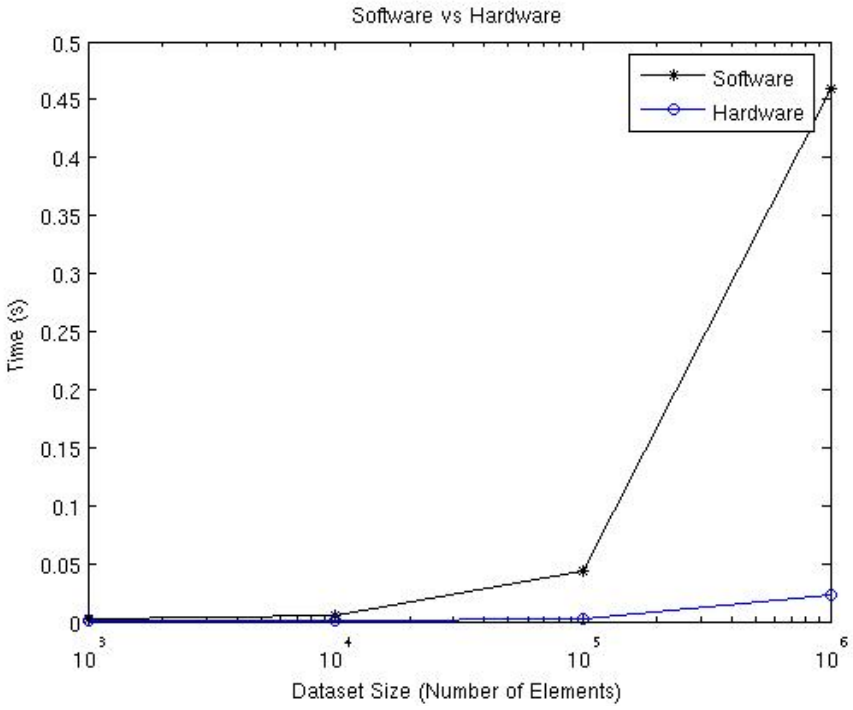

Figure 5.1: Software vs Hardware

After the research done on previous implementers of recursion structures we understood that there is no point to develop a general method for implementing

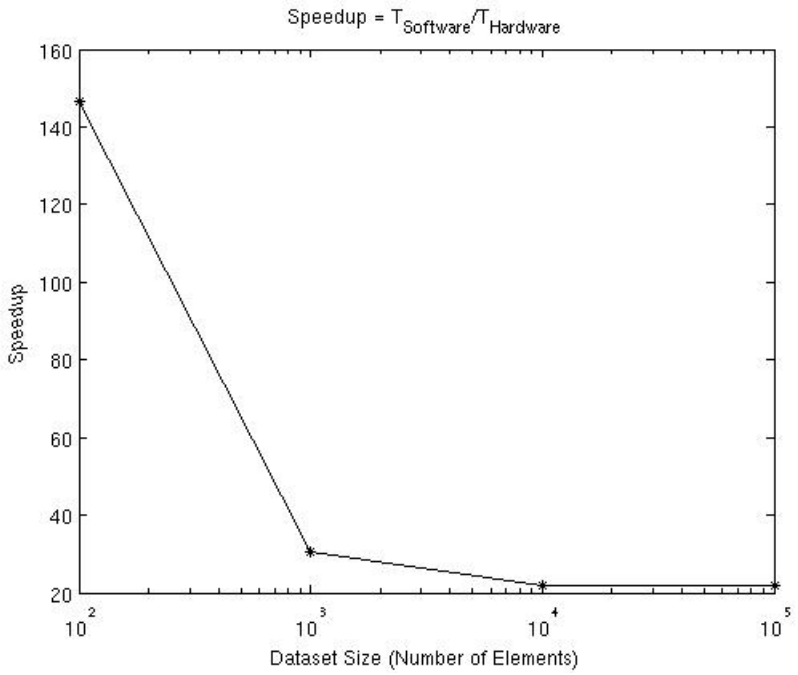

Figure 5.2: Speedup - Single QS

recursion on hardware, because there is a possibility to loose the advantages we may gain from an algorithm's characteristics. So we ended to the conclusion that when we design hardware, that has to solve a specific problem, we should design based on those problem's features and not in general methods.

George Ferinzis and Hosam El Grindy proposed to transform recursion, if possible, in an iteration to gain, like in software, the less call overhead payment. In our quicksort implementation we transformed the quicksort algorithm to perform in-place partitioning and make the calls at the end (tail recursion). In this way we gained an automated merging of the children and a lot less transactions with the stack. By this transformation our quicksort does not need to keep track of the made calls or return to them. It just has to keep information about calls occurred but the have not been made yet. For example on a call on an array that outputs two children that needs to be examined the left will be processed straight after the parent array but the right child has to be stored for later processing.

Next thing we did, was to insert heuristic methods to prune the recursion tree and by adding some more logic to our designs to avoid some pointless calls.

Ninos Spyridon and Apostolos Dollas proposed a structure that lets the design decide the next states from the data stored in stack. This allows to have a simpler design and with one pop to have available all the needed data to make a call. In our design we established a minimum data context to get pushed in stack for

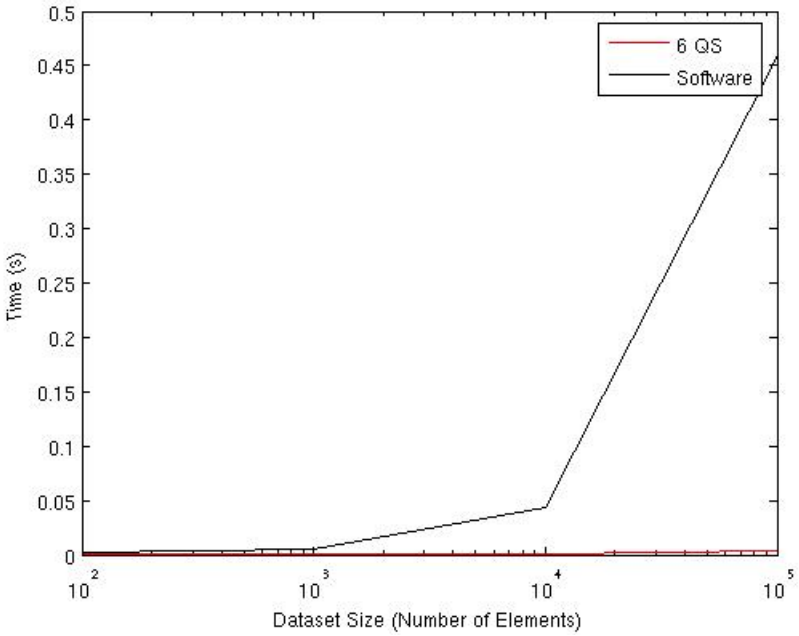

Figure 5.3: Software vs Hardware 6 QS

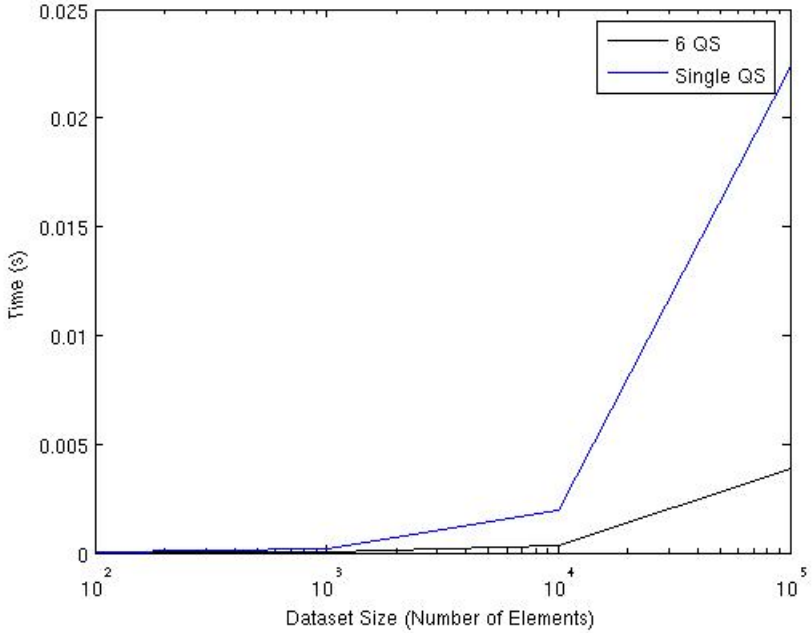

Figure 5.4: Single QS vs 6 QS

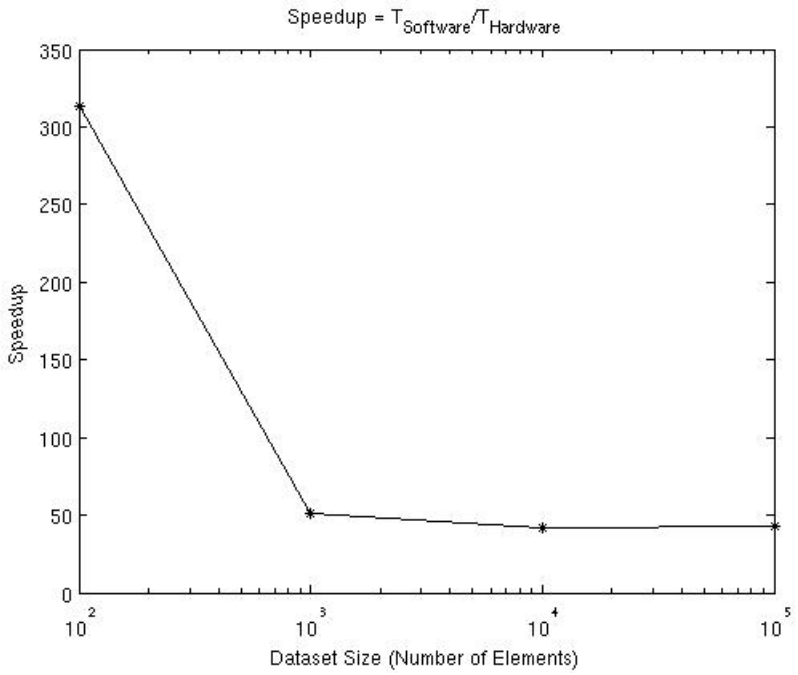

Figure 5.5: Speedup - 6 QS

each future call. When a call context is popped it will provide all the needed information to proceed on a call.

Previous implementers, that used parallelization of the sub problems they divided their designs, which is obviously the best way to gain speed! The algorithm we implemented (quicksort) allowed to design in theoretical base and implement parallelization of calls and we proved that this parallelization is not that profitable if we use brute force. This limitation can be usefull for future research on quicksort and also demonstrated that with 4 or 5 QS component running simultaneously gain much more.

The practical experiment showed that there is gain from hardware implementations but the main profit of our design comes from our design's small size, considering FPGA area consumption, to allow implementers use it as a part of bigger problems that need sorting. As a subproblem our design gain speed and can also justify the cost of an FPGA. Moreover our design can be implemented on the smallest FPGAs but usually those comes with very little Block Memory space witch is needed if we want to sort large datasets. On the other hand our design can be implemented as well on bigger and faster FPGAs, which has a lot of Block Memory Space, to sort larger datasets.

If the only thing needed is sorting, via quicksort, and based that PCs today are cheap and can accomplished a variety of jobs apart from sorting, then one

contemporary PC can handle it with minimum cost. On the other hand if the need for sorting is part of a bigger problem we will implement on a FPGA (or ASIC) then the use of our design is profitable, considering the gain and cost analogy.

5.2 Future Work

This study might end here but it leaves results and ideas for further work and studying. At first all the proposed models can be tested and used with different inputs, larger or of different type. Most of the implemented components (all the basic) are ready for use, alone or as parts of bigger problems.

We implemented various models for quicksort algorithms, in order to experiment, and we reached and presented a general model that gives independent calls. Having independent calls we can easily implement parallelization of them and produce even faster hardware, as we proposed in theory the model exist and also a practical model is tested but it needs the design of a memory that can provide input/output for each QS component simultaneously. This memory model can be made with the use of another intercallary state between same material memories (Block Memories) and the element processing. This intercalary memory stage may be divided in parts equal to the QS instances. These parts will have length equal to the number of QS instances multiplied by the used word size. For each instance of QS four parts will be needed two for data from the under examination array front part and two for the rear. A more sophisticated Memory Manager will handle to load each QS instance's memory's parts from the block Memory and reload them again alternately. This way we may have a memory with multiple sets of input/output to connect to it more than one QS components. For this model only our design that loads the dataset to Block Memories before the element processing starts can be used.

During our researching and implementing on hardware, quicksort solutions, we designed a model that uses very little block memory space and performs loads and unloads to it, from the main storage (SDRAM), simultaneously with element processing. This design can be considered complete, except the part that connects the main storage with the FPGA. Our theoretical results proved that a design like this will be a little bit faster than software. This design's main advantage is not the speed gain but it's cost. Having a small design as our's and by using a small FPGA we may have sorting times equal to software's with only some decades of euros. The manipulation of main storage for quicksort simultaneously with element processing consist a subject for further studying.

Another good idea, for optimizing software this time, could be the use of a GPU[25] parallel with a PC's CPU. According to the fact that quicksort can be

transformed to have all the calls occurred from a call independent we may use a GPU to handle each call's procedure. Each pending call can be assign from the GPU to one of the microprocessors it has to handle the quicksort main procedure for each sub-array[1]. A GPU can execute an instruction with different input as many times as the threads it can handle[2]. In a design like this the CPU will handle the decisions either to proceed on a call or not and also to collect the outputs of the executed calls. By the time all the given calls return the CPU will collect their outputs, decide and then give all the occurred calls to the GPU to be processed. This will be done until no more calls are occurred from the GPU's outputs. For this implementation, NVidia's CUDA[26] language can be used and our software implementation with adding an output collector function.

[1]Each call refers to a different sub-array

[2]A common GPU today can handle 400 threads. That means it can run that many times, in one cycle, a instruction and with different input.

List of Figures

Bibliography

[1] Hoare, C. A. R. *Partition: Algorithm 63, Quicksort: Algorithm 64, and Find: Algorithm 65.*, Comm. ACM 4(7), pages 321-322, 1961. Hoare, C. A. R. *Quicksort*, Computer Journal 5 (1): 10-15. (1962). (Reprinted in Hoare and Jones: Essays in computing science), 1989.

[2] M.Foley and C.A.R. Hoare *Proof of a Recursive Program: Quicksort*, Queen's University Belfast, 1971.

[3] Donald E. Knuth, *The Art of Computer Programming, Volume 3, Sorting and Searching*, 2nd Edition, Addison-Wesley Professional, 1998.

[4] K. Bondalapati and V. K. Prasanna, *Mapping loops onto reconfigurable architectures*, R. W. Hartenstein and A. Keevallik, Eds. Springer-Verlag, Berlin, pages 268277, 1998.

[5] Timothy J.Callahan and John Wawrzynek, *Instruction-Level Parallelism for Reconfigurable Computing*, pages.248-257, 1998.

[6] T. Maruyama, M. Takagi and T. Hoshino, *Hardware Implementation Techniques for Recursive Calls and Loops*, Lecture Notes in Computer Science 1673 - The Ninth International Workshop, FPL99, Glasgow, UK, pages 450-455, 1999.

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

[8] V. Sklyarov, *FPGA-based implementation of recursive algorithms, Microprocessors and Microsystems, Special Issue on FPGAS: Applications and Designs*, vol. 28/5-6, pages197-211, 2004.

[9] V. Sklyarov, I. Skliarova, and B. Pimentel, *FPGA-based implementation and comparison of recursive and iterative algorithms*, Proceeding of FPL05, Tampere, Finland, pages 235-240, 2005.

[10] V. Sklyarov, I. Skliarova, *Recursive and Iterative Algorithms for N-ary Search Problems* in IFIP, pages 81-90, 2006.

[11] George Ferizis and Hossam El Gindy *Mapping Recursive Function To Reconfigurable Hardware*, Canberra ACT, University of New South Wales, 2006

[12] Spyridon Ninos and Apostolos Dollas *Modeling Recursion Data Structures For FPGA-Based Implementation*, Technical University of Chania, 2008

[13] R. Sedgewick, *Implementing quicksort programs*, Comm. ACM, 21(10):847-857, 1978.

[14] *http://www.ieeta.pt/ skl/Research/MostImportantResults.html* (reported results up to March 23d, 2008)

[15] *http://en.wikipedia.org/wiki/Sorting_algorithms*

[16] *http://en.wikipedia.org/wiki/Quicksort*

[17] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press, 2000. *http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm*

[18] *http://en.wikipedia.org/wiki/Inplace*

[19] *http://en.wikipedia.org/wiki/Recursion_(computer_science)*

[20] *http://en.wikipedia.org/wiki/Tail_recursion*

[21] *http://en.wikipedia.org/wiki/Iteration*

[22] *http://en.wikipedia.org/wiki/FPGA*

[23] *http://en.wikipedia.org/wiki/ASIC*

[24] *http://www.inf.fhflensburg.de/lang/algorithmen/sortieren/quick/quicken.htm*

[25] *http://en.wikipedia.org/wiki/GPU*

[26] *http://www.nvidia.com/object/cuda_education.html*
http://en.wikipedia.org/wiki/CUDA

[27] *http://www.xilinx.com/*, Various Topics, about ISE 10.1 and Core Generator.

[28] Eric Huss, *http://www.acm.uiuc.edu/webmonkeys/book/c_guide/*, 1997

[29] Peter J. Ashenden, *The VHDL Cookbook*, University of Adelaide South Australia, First Edition, 1990

[30] R. Timothy Edwards, *Magic Tcl Tutorial*, version 7.2