FEDERAL UNIVERSITY OF RIO GRANDE DO SUL
INSTITUTE OF INFORMATICS
COMPUTER ENGINEERING COURSE

MARINA SILVA MIRANDA

A Study on the Acceleration of Search Heuristics in Programmable Logic

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engeneering

Advisor: Prof. Dr. Gabriel Luca Nazar
Coadvisor: Prof. Dr. Levi Lelis

Porto Alegre
December 2016

"Let us make our future now,

and let us make our dreams tomorrow’s reality. "

— Malala Yousafzai

ACKNOWLEDGMENT

First, I'd like to thanks my advisor, professor Gabriel Nazar for having an immense patience with me and helping me out whenever it was possible. I know I'm not the ideal advisee, far from it. Really, thanks for everything, and continue being a great professor.

Some years ago when I got in the university I didn't know anything about computation/informatics, I didn't even know what was a programming language, everything worked with magic. And now, in this project, I created a program in C and a hardware description in VHDL, both of which I integrated in a hybrid platform and it works! For that and way more, I have to thank all my professors from the university, practically everything I know one of you taught at one point or another. Keep being great professors, because you really can make a difference.

And now, last but not least, I'd like to thank my parents that always believed that having education is one of the most valuable things in the world.

ABSTRACT

Search algorithms are used in several applications nowadays; they are used for finding the best path from a place to another in a city or between two nodes in a complex graph in a data base system. There is a growing interest in accelerating applications, such as these, in Field Programmable Gate Arrays (FPGAs). An FPGA is an integrated circuit whose logic cells can be configured by the user, at any time. Lately these devices integrate processors with the programmable logic, an example of this is the ZYNQ device from Xilinx with an ARM processor embedded. This work presents a hybrid design of the best path search algorithm A*, in which most of the algorithm was implemented to run in a processor and a part was selected to be implemented in hardware (VHDL). The selected part was the priority queue, which is based on a min binary heap, that the A* algorithm utilizes.The ZEDBOARD (AVNET/Digilent) with a ZYNQ inside was used to develop the proposed solution, which can improve the performance of a min binary heap implemented purely in software by 68.8%.

Keywords: FPGA. programmable logic. processor. ZYNQ. priority queue. A* algorithm. heuristics.

Um Estudo Sobre a Aceleração de Heurísticas de Busca em Lógica Programável

RESUMO

Algoritmos de busca estão sendo utilizados em diversas aplicações atualmente; são usados para encontrar o melhor caminho entre um lugar e outro em uma cidade ou entre dois nós em um grafo complexo em um sistema de banco de dados. Existe uma tendência cada vez maior de acelerar aplicações, como estas, em Field Programmable Gate Arrays (FPGAs). Um FPGA é um circuito integrado no qual as células lógicas podem ser reconfiguradas pelo usuário/desenvolvedor a qualquer momento. Recentemente, estes dispositivos integram processadores com lógica programável, um exemplo disso é o ZYNQ da Xilinx com um processador ARM embarcado. Este trabalho apresenta um design híbrido do algoritmo de busca de melhor caminho A*, no qual a maior parte do algoritmo é implementada para rodar em um processador e a uma parte foi selecionada para implementação em hardware. A parte implementada em hardware (VHDL) é a lista de prioridades que o A* utiliza, a qual é baseada na estrutura de um *heap* binário. A placa ZEDBOARD (AVNET/Digilent) com um ZYNQ dentro foi escolhida para desenvolver a solução proposta, que melhora a performance de um heap binário (lista de prioridades) implementada puramente em software em 68.8%.

Palavras-chave: FPGA, lógica programável, processador, ZYNQ, fila de prioridades, algoritmo A*, heurística.

LIST OF ABBREVIATIONS AND ACRONYMS

AXI	Advanced Extensible Interface
BRAM	Block Random Access Memory
BSP	Board Support Package
FF	Flip-Flop
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HW	Hardware
IP	Intellectual Property
LUT	Lookup Table
PL	Programmable Logic
PS	Programmable System
SDK	Software Development Kit
SoC	System-on-a-Chip
SW	Software
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WNS	Worst Negative Slack

LIST OF FIGURES

CONTENTS

1 INTRODUCTION

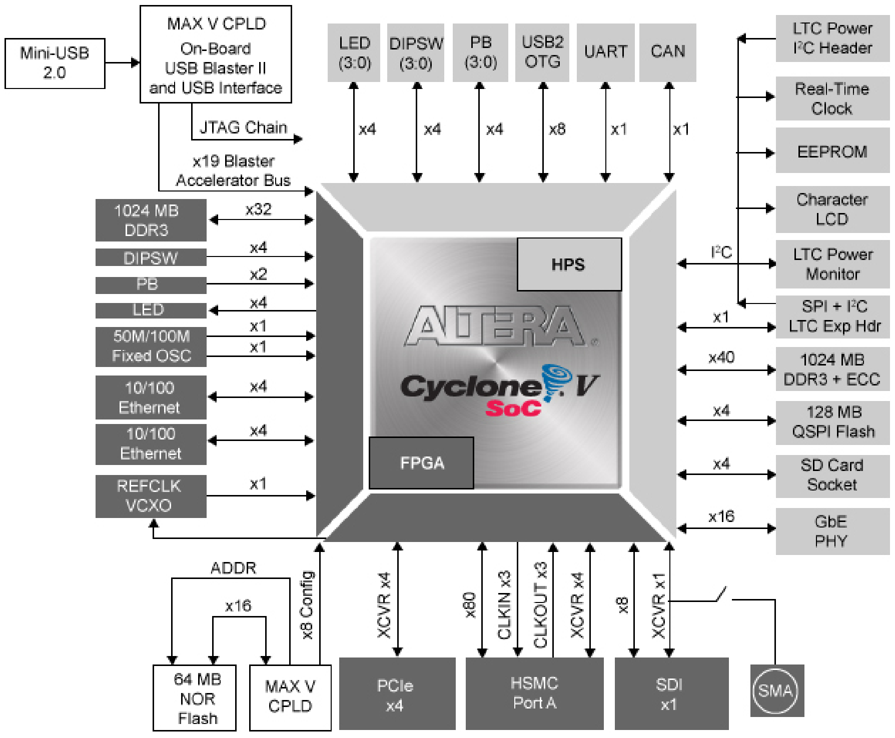
It is becoming more popular to find reconfigurable platforms based on Field Programmable Gate Arrays (FPGAs) integrated with processors. The top integrated circuit manufacturers are already investing in this new approach, one example being the Zynq devices from Xilinx, which packs in the same integrated circuit ARM processors fully interconnected (via AXI bus - ARM AMBA bus) with an FPGA. Another example is the new Stratix 10 from Altera, which packs in the same chip the programmable logic and a quad-core ARM Cortex-A53 fully interconnected (via AXI bus also), which promises better performance and power efficiency (BTDI, 2013).

These devices give designers more flexibility when devising a solution, since part of an application can easily run in software and the other part can be implemented in hardware. The co-design of an 802.11a transceiver system using Zynq SoC Drozdenko et al. (2016) and the Implementation of ECG Encryption and Identification on the Zynq SoC (ALI et al., 2016) are some examples of the utilization of these hybrid architectures.

This project's goal is to accelerate a best path search algorithm in a hybrid platform, such as the Zynq device from Xilinx. The A* (A star) search algorithm was primarily chosen to be accelerated, that is, it will be mostly implemented in software with some of the most time consuming parts implemented in hardware (FPGA). This algorithm consists in a node-to-node search based on costs, in which each time a node is visited all its neighbours are added to a list of most promising nodes. And so it goes on for every node in this list until the goal node is found. The structure chosen to be accelerated was the list of potential nodes to be visited, which basically is a priority queue.

The best path search algorithms are frequently used in embedded systems and data centers. Some examples of the use of this algorithm in embedded systems are: path planning for mobile robots (DUCHON et al., 2014), search and rescue of people in danger in a perfect maze (LIU; GONG, 2011) and path planning for Unmanned Aerial Vehicles (UAVs) (TSENG et al., 2014). The best path search algorithms are also used in data centers to find the distance between two users (VIEIRA et al., 2007), to analyze the most influential nodes (people) in social networks (KEMPE; KLEINBERG; TARDOS, 2003) or to find relations between web pages (close web pages have related contents, enabling page suggestions) (UKKONEN et al., 2008).

Figure 1.1: Schematic of the Cyclone V SX SoC Development Board.



Source: Altera's website.

Microsoft has released a paper describing the use of reconfigurable fabrics (FPGAs) for the acceleration of large-scale services in data centers (PUTNAM et al., 2014). This would be a valid application for the project that is being proposed here, since best path search algorithms (such as A*) are used in data centers and FPGAs are used to accelerate data processing. Hence, the proposed solution has applications in both embedded systems and large-scale data centers. It might improve not only the performance of these applications, but also the system's energy efficiency, which is another big concern in embedded systems and data centers.

This work is organized as follows. Chapter 2 presents the theoretical references. The proposed solution is presented in chapter 3, and the results are presented in chapter 4. The results are discussed in chapter 5. Finally, chapter 6 concludes the work.

2 THEORETICAL REFERENCE

2.1 Related Work

Related work on the acceleration of shortest path search algorithms is summarized here. Tommiska and Skytta (2001) present the Dijkstra algorithm implementation in FPGA, with the intent to use this solution for network routing. The architecture used is composed of a memory device (ROM) to store the nodes/adjacency list, a prefetching block to get the potential next node, a comparator block that selects the best path from one node to another and a RAM memory to store the shortest path so far (visited nodes). A version of this algorithm was also implemented in C for comparison, and from the results obtained, the execution time for the FPGA solution grew linearly according to the number of nodes, whereas for the C solution the execution time had quadratic growth.

The paper on the Floyd-Warshall algorithm (BONDHUGULA et al., 2006) discusses the parallelization of this search algorithm in an FPGA, the optimization of the algorithm's parameters and compares the results obtained running an application in the FPGA with the results obtained running the application on a CPU. Even though the Floyd-Warshall algorithm computes all-pairs shortest path solutions for the problem (different from A*), it is important to notice that, in this paper, all the solution was implemented in FPGA, which is not what is intended in this project (A*) proposal, although the methodology used has some similar aspects to the one proposed in this project.

Priya, R. and Sridharan (2006) present the implementation of the revised simplex method for linear programming, a single-pair shortest path algorithm, in FPGA. Again, the entire solution is implemented in hardware, where parallelism is used to speed-up even more the timing. A possible issue with this solution is that in the paper, the solution has been tested for small graphs, and, according to the results encountered, a lot of the area in the FPGA has been used. This leads to the conclusion that this solution is not scalable, but it is a very fast one.

From the paper written by (BONDHUGULA et al., 2006) on the Floyd-Warshall implementation in FPGA, we will utilize the idea of testing the performance of the hybrid A* solution with the performance of a version of the A* search algorithm implemented in C running in a general-purpose processor. Moreover, from (PRIYA; R.; SRIDHARAN, 2006) it is possible to conclude that the implementation of a search algorithm in hardware seems to accelerate a lot the search for a path, but it might not be worth it if the area utilized on the chip is that huge for such a little quantity of nodes and arcs.

Idris et al. (2009) propose the implementation of the A* algorithm entirely in FPGA, which was done by assuming that the nodes are organized in a perfect grid. This assumption, in a way, facilitates the implementation of the solution, given that there is no need to create/keep an adjacency list, but it also limits the applications in which this project can be used. Assuming a grid also facilitates on the calculation of the heuristic, since the best way to calculate the distance is already known (Manhattan distance) and it will result in the exact cost from one node to the other.

Akiba, Iwata and Yoshida (2013) present the implementation of another shortest-path algorithm, the breadth-first search, commonly known as one of the most simple search algorithms. The authors demonstrate that an acceleration of this method can be done by dividing/parallelizing in different cores the searches and pruning (least promising paths of the graph are ignored) for search space reduction.

The best path implementation proposed by Akiba, Iwata and Yoshida (2013) obtained a very small average query time for 1000000 random pairs of vertices, but the time spent on preprocessing (calculating beforehand the distance between some vertices of the graph) the datasets is very high, making this solution very time consuming.

Related work was also found on the implementation of priority queues. (RöNNGREN; AYANU, 1997) presents a comparative study on priority queue algorithms. The authors obtained the following results: for queue sizes smaller than 1000 entries, the Splay Tree, the Skew Heap and Henricksen's algorithms show good access times, whereas for lager queues, the Calendar Queue and the Lazy Queue have better access times.

(BLOOM; G.; NARAHARI, 2012) proposes a priority queue as a hardware data structure, which is implementing the data structure using hardware mechanisms to improve the overall performance. A shift-register based priority queue that supports very large queues was proposed.

2.2 A* Algorithm

The A* is a well known form of best-first search algorithm (HART; NILSSON; RAPHAEL, 1968). The cost from one node to another is evaluated and the most promising node is selected. The cost function is different from other algorithms, such as the Dijkstra algorithm. In A*, the next node n to be expanded is the one that minimizes the total cost ($f(n)$), which is the sum of cost from the source to n ($g(n)$) and an estimation of the cost from n to the goal ($h(n)$), as shown in (2.2.1). Given that $h(n)$ is an admissible heuristic, as will be discussed in section 2.2.1, the A* algorithm is complete and optimal, that is, it will always find the best path from the start node to the goal.

$$f(n) = g(n) + h(n) \tag{2.1}$$

2.2.1 Heuristics

The heuristics is what differentiates the A* from uniform-cost search algorithms. It is a 'guess' of the real distance/cost between a node and the goal node. Since it is not actually the distance between the nodes, depending on the heuristics calculation the algorithm converges more rapidly to the best path.

The chosen heuristics function has to respect one rule to guarantee the optimality of the algorithm it has to be admissible. The heuristics function can also be consistent, which is important to avoid duplicate node expansion.

- *Admissibility:*
  The cost to reach the goal is never overestimated. That is, for all nodes in the graph, the heuristic function is never greater than the actual cost from the current node to the goal node.
  This characteristic allows pruning (eliminating possibilities from consideration) branches/sub-trees without missing the best path.
- *Consistency:*
  According to Russel and Norvig (2014), a heuristic $h(n)$ is consistent if, for every node n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting n' plus the estimated cost of reaching the goal from n'.

All consistent heuristics are admissible, but the contrary is not always true, although it is quite difficult to achieve, in practice, a heuristic function that it is admissible but not consistent.

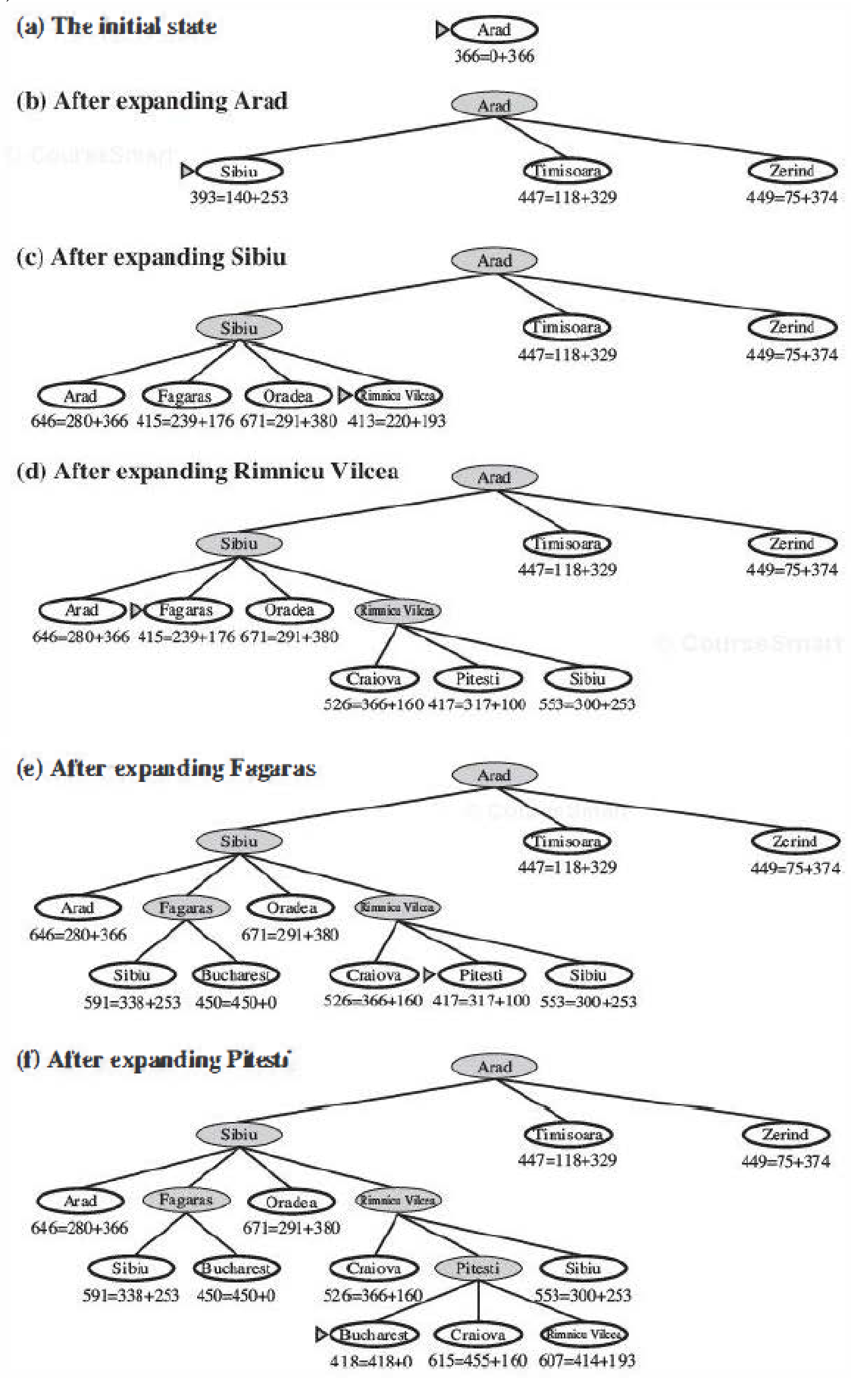
2.2.2 Algorithm Flow

The program flow of the A* algorithm should be:

1. Assume that the start node is the current node;
2. Find the current node neighbors by checking the adjacency list;
3. Calculate the heuristic ($h(n)$) and the total cost ($f(n)$) for each neighbor node;
4. Add neighbors to the priority queue with calculated cost (f(n)). The priority queue is a queue in which all the entries are ordered priority-wise. By using this type of queue there will be no need for searching the next node to be visited. A node will always be inserted in the correct position, keeping the queue ordered. In the case of the A* algorithm, the considered priority will be the cost function ($f(n)$), ordered from minimum (top of the priority queue) to maximum values;
5. Mark the current node as opened/visited. A list should be used to make sure no repeated nodes will be revisited;
6. Save the current node in a visited node list. This will allow the algorithm to form a path from the start node to the goal node;
7. Make the first node in the priority queue the current note for evaluation. This will be done by doing a fetch operation in the queue;
8. Repeat 2 to 7 until the goal node has been found;
9. Use visited node list to form a path from the goal node to the start node;
10. Return best path.

Figure 2.1, presents an example of the A* algorithm. It searches the best path between the node `Arad` to the node `Bucharest`. In a) the algorithm usumes that the initial node is the current node . b) Expands `Arad`, by adding it to the visited list, and puts its neighbours into the priority queue. c) Gets most promesing node from the queue, `Sibiu`, adds it to the visited list and puts its neighbours into the priority queue. d) Gets most promesing node from the queue, `Rimnicu Vilcea`, adds it to the visited list and puts its neighbours into the priority queue. e) Gets most promesing node from the queue, `Fagaras`, adds it to the visited list and puts its neighbours into the priority queue. f) Gets most promesing node from the queue, `Pitesti`, adds it to the visited list and puts its neighbours into the priority queue. Then gets the most promesing node of the priority queue, `Bucharest`, and stops because it is the goal node.

Figure 2.1: An example of the A* algorithm flow. Under the nodes is shown $f(n) = g(n) + h(n)$.



Source: (RUSSEL; NORVIG, 2014)

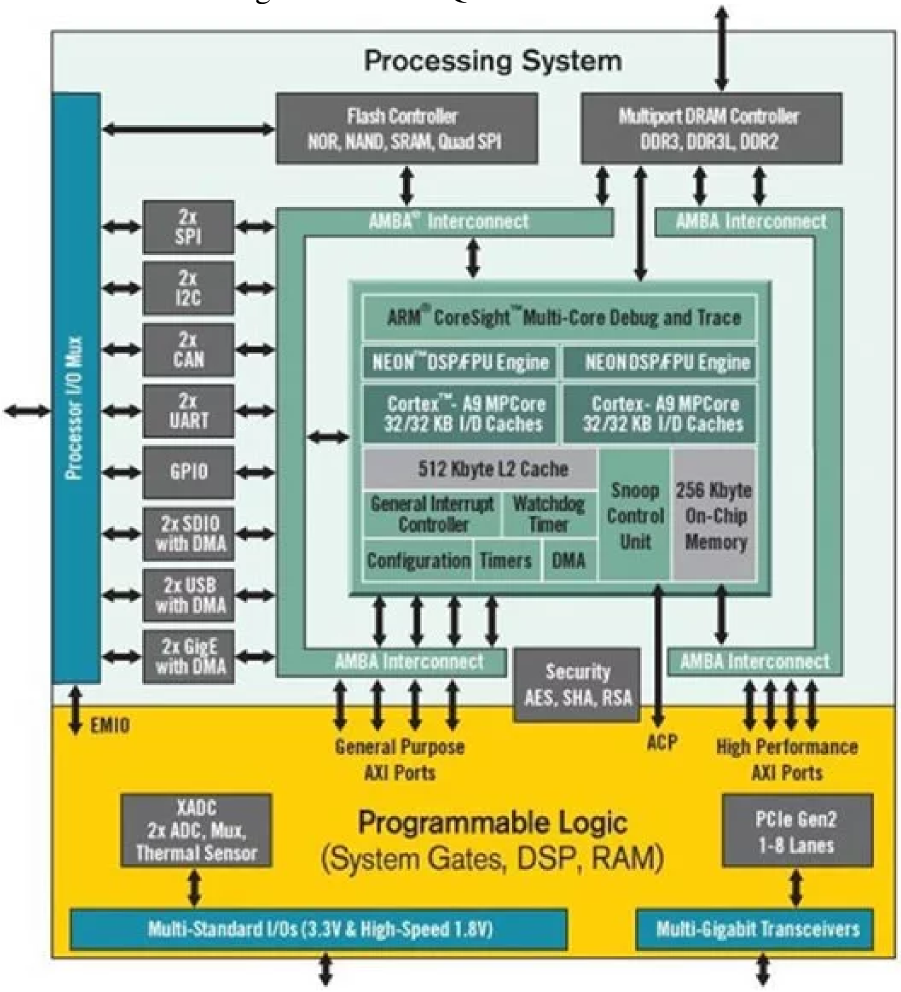
Note that the Dijkstra algorithm is a special case of A*. Both algorithms have the same behavior when the heuristic for all the nodes in the graph is equal to zero (which is an admissible and consistent heuristic).

2.3 ZEDBOARD

Nowadays there are several options for developers in need of hybrid (software + hardware) solutions. Altera and Xilinx have IPs for soft processor cores, such as the Microblaze (Xilinx) and the Nios (Altera). The problem with soft cores is that they occupy a lot of FPGA space and the performance is not so good because the clock frequency is not very high, generally around 125 MHz. Apart from soft cores, these two programmable logic device suppliers are integrating hard processors in their's FPGAs, examples of this are the ZYNQ device from Xilinx and the Cyclone V from Altera. For this project, the ZedBoard (AVNET/Digilent) was chosen, it contains a ZYNQ-7000 FPGA.

An overview of the ZYNQ-7000 is presented in figure 2.2.

Figure 2.2: ZYNQ-7000 Overview.


Processing System
Processor I/O Mux
Flash Controller NOR, NAND, SRAM, Quad SPI
Multiport DRAM Controller DDR3, DDR3L, DDR2
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
AMBA® Interconnect
AMBA Interconnect
ARM® CoreSight™ Multi-Core Debug and Trace
NEON™ DSP/FPU Engine
NEON DSP/FPU Engine
Cortex™- A9 MPCore 32/32 KB I/D Caches
Cortex- A9 MPCore 32/32 KB I/D Caches
512 Kbyte L2 Cache
General Interrupt Controller
Watchdog Timer
Configuration
Timers
DMA
Snoop Control Unit
256 Kbyte On-Chip Memory
AMBA Interconnect
Security AES, SHA, RSA
AMBA Interconnect
EMIO
General Purpose AXI Ports
ACP
High Performance AXI Ports
XADC 2x ADC, Mux, Thermal Sensor
Programmable Logic (System Gates, DSP, RAM)
PCIe Gen2 1-8 Lanes
Multi-Standard I/Os (3.3V & High-Speed 1.8V)
Multi-Gigabit Transceivers

Source: Xilinx's website.

The ZYNQ-7000 has a dual-core ARM Cortex-A9 processor integrated with programmable logic, which is exactly what it is needed to implement the hybrid A* solution. There are also several peripherals connected to the ZYNQ-7000 and available to the programmable logic part, such as UARTs or I2Cs, that will not be used in this project.

The speedgrade of the ZYNQ inside the ZEDBOARD is -1, and, according to Xilinx's documentation, for this speedgrade the clock frequency of the ARM processors is 667 MHz. A 64-bit counter implemented in the programmable logic runs at a maximum of 277 MHz according to the Xilinx documentation.

2.3.1 AXI Interface

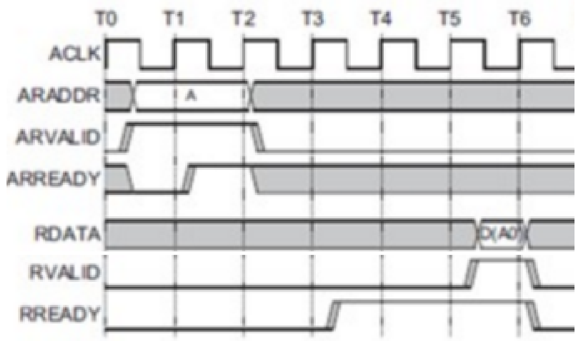
As shown is figure 2.2, the processor cores communicate with the programmable logic through the AXI interface, which is an internal bus created by ARM AMBA and specified in the document AMBA® AXI^{TM} and ACE^{TM} Protocol Specification. The AXI interface is based on a valid/ready handshake, first the master sends a VALID signal with DATA or ADDRESS and the slave responds with READY. The slave then operates (read/write on a register) and sends a VALID signal with the RESPONSE (which can be OK or an Error Indication) for which the master replies with a READY signal.

The AXI interface is divided in 5 channels:

- *Write Address Channel*: The master asserts VALID and sends the ADDRESS to the slave, and the slave stores it and sends the READY signal;
- *Write Data Channel*: The master asserts VALID and sends the DATA to the slave, and the slave writes it to a register (specified by the address sent previously) and sends the READY signal;
- *Write Response Channel*: The slave asserts VALID and sends a RESPONSE, and the master replies by sending the READY signal;
- *Read Address Channel*: The master asserts VALID and sends the ADDRESS to the slave, and the slave stores it and sends the READY signal;
- *Read Data Channel*:The slave asserts VALID and sends a RESPONSE with the VALUE read from the specified register, and the master replies by sending the READY signal;

Figure 2.3 presents an example of the ready/valid handshake.

Figure 2.3: Ready/Valid Handshake.



Source: AMBA® AXI^{TM} and ACE^{TM} Protocol Specification.

The AXI interface supports 32-bit and 64-bit data buses. It also supports wait states. There is no timeout implemented in the AMBA interconnect and the AXI master, that is, once the master starts an operation (read or write) by asserting the VALID signal, the slave can wait several cycles before asserting the READY signal.

3 PROPOSED SOLUTION

Two implementations of the A* algorithm were made. The first is the algorithm entirely in software. With this version it was possible to identify one of the most costly parts of the algorithm, the priority queue operations/management. The queue operations are so costly because each time a node is needed there is a memory operation, so the ordering and re-ordering take several memory operations.

The second implementation of the algorithm was the hybrid implementation, in which the same software developed before was used but with the priority queue implemented in hardware. The A* solution is used to compare with the hybrid solution for performance measure purposes.

The hybrid implementation of the A* algorithm was divided into three parts (each described in one of the following subsections). The first part is the software, that will do most of the operations. The second part is the connection between the programmable logic (PL) and the programmable system (PS). The last part is the VHDL priority queue.

3.1 A* Software

Before starting the search for a best path between two nodes, the program loads into its environment the nodes graph from the files stored in the DRAM of the ZEDBOARD.

The program loads into an adjacency list the graph described in the '.gr' and '.co' files (3.1.1). After creating the adjacency list, the software program follows the A* search flow presented in 2.2, that is, it selects the current node and calculates the cost to the goal node (by adding the cost to the node and the heuristics) and starts an AXI write transaction (sending the resulting cost value + node identification) to the PL. Then the software inserts the current node in a path list and starts an AXI read transaction to get the next current node.

The software always waits for the AXI slave response, which takes only the AXI transaction cycles, before executing the next operations.

Note that the software was implemented in C with all the priority queue operations (push/pop of the queue) translated to Vivado's SDK BSP AXI read/write operations (Xil.in32 and Xil.out32).

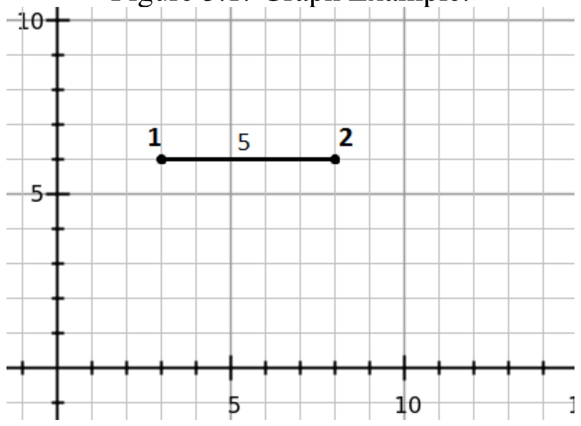
3.1.1 Challenge 9 format

The standard Challenge 9 format is a way to describe graphs used in the 9th Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) Implementation Challenge - Shortest Paths. The graph is described in two files:

- '.gr' - Lists all nodes, arcs and arc weights;
- '.co' - Lists all nodes coordinates.

Each line of the file starts with a letter, 'c' for comment, 'p' for problem line (with number of nodes and the number of arcs), 'a' for an arc descriptor line and 'v' for a coordinate line. Each file line has one command followed by a new line.

Figure 3.1: Graph Example.



Source: the author.

For instance to describe an arc between nodes 1 and 2 with weight 5 the command line is :

a 1 2 5

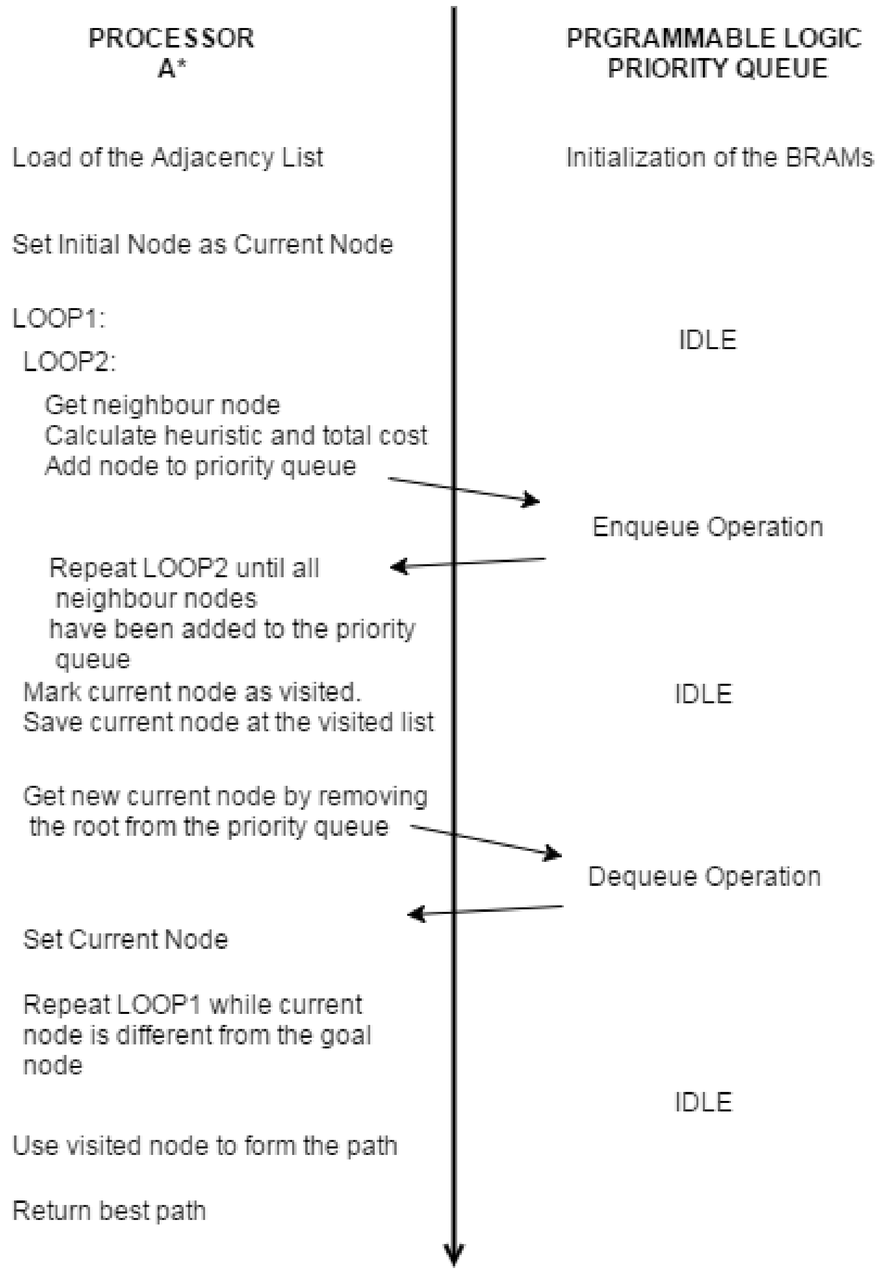
To describe the coordinate of each of this nodes the command lines are:

v 1 3 6
v 2 8 6

3.1.2 SW - HW Interaction Flow

The diagram in figure 3.2 presents the flow of the entire solution.

Figure 3.2: Processor SW interaction with HW priority queue.

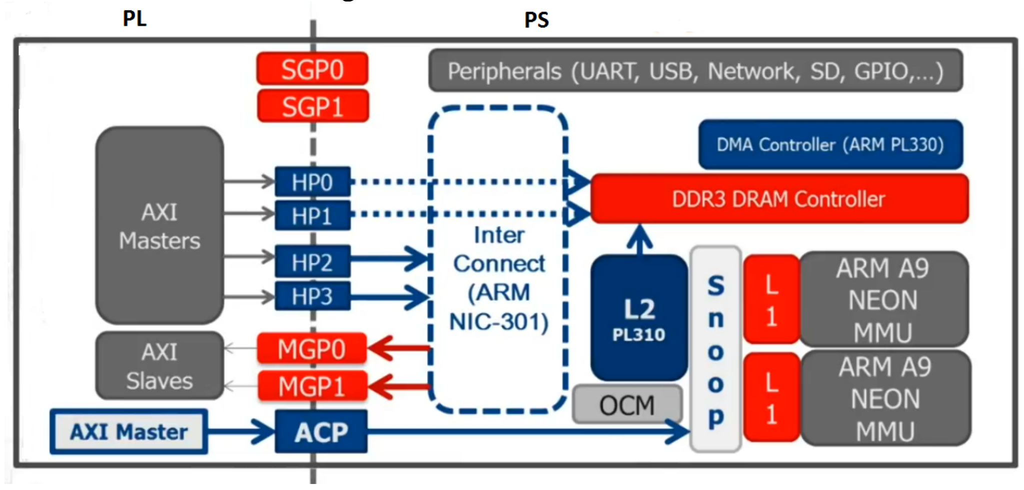

PROCESSOR
A*
PRGRAMMABLE LOGIC
PRIORITY QUEUE
Load of the Adjacency List
Initialization of the BRAMs
Set Initial Node as Current Node
LOOP1:
IDLE
LOOP2:
Get neighbour node
Calculate heuristic and total cost
Add node to priority queue
Enqueue Operation
Repeat LOOP2 until all neighbour nodes have been added to the priority queue
Mark current node as visited.
Save current node at the visited list
IDLE
Get new current node by removing the root from the priority queue
Dequeue Operation
Set Current Node
Repeat LOOP1 while current node is different from the goal node
IDLE
Use visited node to form the path
Return best path

Source: the author.

3.2 PS-PL Connection

Figure 3.3 presents an overview of the connections between the programmable system (PS) and the programmable logic (PL).

Figure 3.3: PS-PL connections.



Source: Microelectronic Systems Design Research Group website

As shown is figure 3.3 there are 4 ways of passing data between the PS and the PL:

- Slave General Purpose Ports - SGP0 and SGP1;
- High Performance Slave Ports - HP0, HP1, HP2 and HP3;
- Master General Purpose Ports - MGP0 and MGP1;
- Accelerator Coherency Port - ACP.

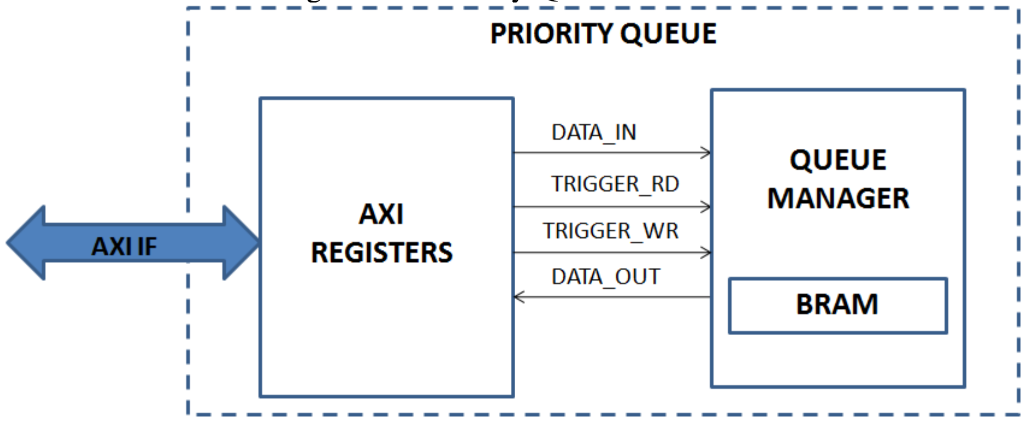
Since the strategy is to run the program in the PS and use the priority queue functionality in the PL, the PS is the AXI Master that requests queue operations to the PL which is the AXI Slave. Because of that, the port used is the MGP0, which is a master for the PL and a slave to the PS.

According to ZYNQ's technical references manual, the PS can access the PL via MGP0 port by communicating through the memory mapped address range `4000_0000 to 7FFF_FFFF`.

3.3 VHDL Priority Queue

The priority queue architecture is based on the binary min heap presented by Kumar et al. (2014). An overview of the block implemented is presented in figure 3.4.

Figure 3.4: Priority Queue Overview



Source: the author.

The binary heap is a complete binary tree (all levels of the tree, except the last one are fully filled) consistent with the Heap property, the key (estimated cost) stored in each element id is less than or equal to the keys in each node's children. The binary heap supports two operations, bubble-up and bubble-down. The bubble-up adds a new element to the heap, it consist of three steps :

1. Add the element to the bottom level of the heap;
2. Compare the added element with its parent; if they are in the correct order, stop;
3. If not, swap the element with its parent and return to the previous step.

The bubble-down removes the root of the heap, it consist of three steps :

1. Replace the root of the heap with the last element on the last level;
2. Compare the new root with its children; if they are in the correct order, stop;
3. If not, swap the element with one of its children and return to the previous step.

3.3.1 AXI Registers

The AXI Registers block contains an AXI slave interface with 2 registers:

- Address **[0x00]** - Data Read (32-bits): Contains the identification of the root of the binary heap, the next data to be requested by the PS. This allows the fastest possible AXI response. This is a read-only register.
- Address **[0x04]** - Data Written (32-bits): Composed of the estimated cost (16 bits - which is the size of a 'short int' in C) and the node identification (16 bits). This is a read/write register.

Whenever there is an AXI write on the Data Written register, the Trigger_WR signal is asserted. Respectively, when there is an AXI read operation, the Trigger_RD signal is asserted. These triggers plus the data in the registers are sent to the Queue Manager block.

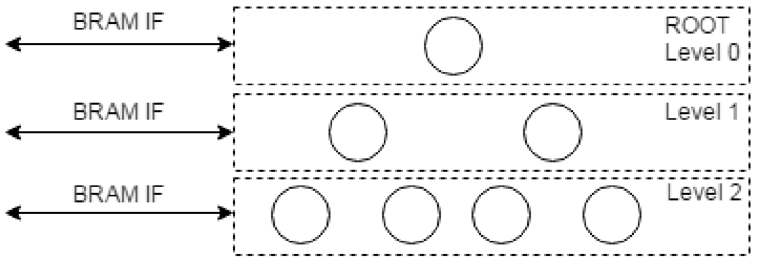
3.3.2 BRAMs

There is one BRAM block for each binary heap level, allowing parallel access to data during an Enqueue operation. The BRAMs are fairly simple to operate, they are dual-port and composed of 8 signals:

- Read Clock: Clock used to read from the BRAM;
- Write Clock: Clock used to write to the BRAM;
- Write Data;
- Write Address;
- Write Enable;
- Read Data;
- Read Address;
- Read Enable;

The BRAM answers 1 cycle after the rising edge of the enables.

Figure 3.5: Binary Heap in a BRAM structure. Each level of the heap is stored in one BRAM



Source: the author.

The architecture of the heap is saved in a BRAM by level, that is, each level of the binary heap is stored in a BRAM. Figure 3.5 presents an overview of the distribution of the heap nodes in the BRAMs structure.

3.3.3 Queue Manager

The Queue Manager is responsible for handling all the binary heap operations, that is, the dequeue (bubble-down) and the enqueue (bubble-up). This block also keeps updated the value of the Data Read register with the root of the heap. The Queue Manager is divided in 3 modules (the Finite State Machine (FSM), the Get Parents and the Get Children) and Block Random Access Memories (BRAMS).

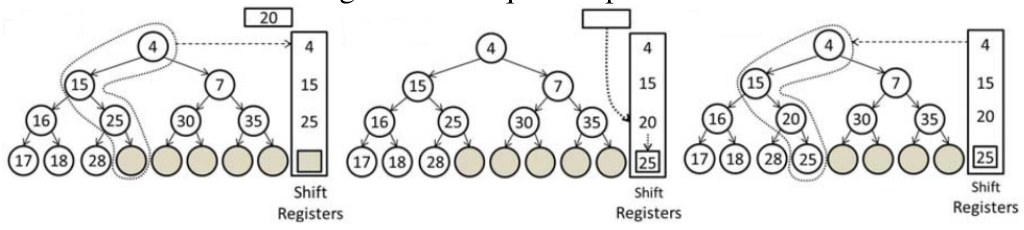
To explain these three modules, we will need to understand the priority queue key operations, the dequeue and the enqueue.

Enqueue

The basic Enqueue operation is: a path from the first vacant leaf to the root is calculated, the new node to be inserted is compared to all the nodes in this path and re-ordered if necessary (if the new node has a lower estimated cost than one of its parents they change places) and the nodes are re-inserted in the heap.

An example of the enqueue is shown in figure 3.6.

Figure 3.6: Enqueue Operation

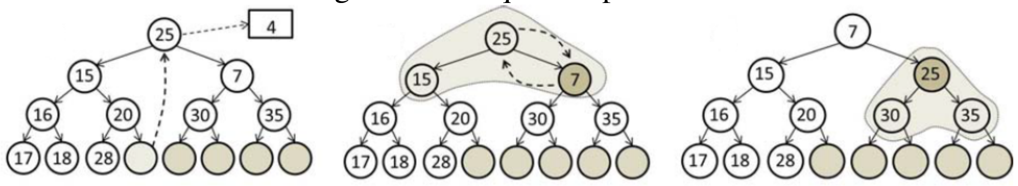


Source: (KUMAR et al., 2014)

Dequeue

When a Dequeue operation is triggered, the last leaf node is inserted in the root position and a re-order operation is triggered. The re-order operation is done by comparing and swapping the new root to its children. Obviously it will cause modifications in the heap, but in the worst case log(n) swap operations will occur.

Figure 3.7: Dequeue Operation



Source: (KUMAR et al., 2014)

3.3.3.1 Get Parent Module

This module calculates the index of the parents of the first vacant spot in the heap (k), based on the following function:

$$k^{th} = i/2^k$$

In hardware, a fast and easy way of implementing this function is to shift right k times the value of i.

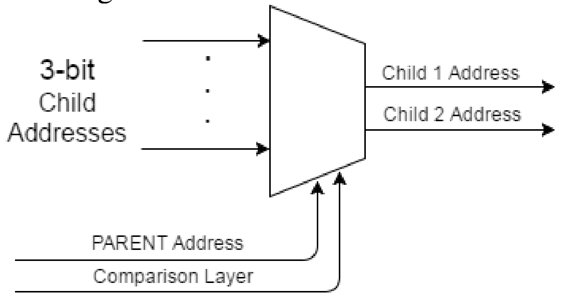
The indexes are calculated in parallel to the rest of the computations, they depend on the current SIZE of the queue. The SIZE is updated every time a transaction (incremented if there was a node added to the queue, decremented if a node was removed) finishes. One cycle after the SIZE register is updated, new parent values are made available, which is before they are needed by the enqueue operation.

3.3.3.2 Get Children Module

This module is composed of two multiplexers that select a PARENT's children address based on the PARENT's address and the LAYER of comparison. The LAYER is a signal that indicates in which phase of the Dequeue re-order the FSM is. For instance if LAYER = 0 the children of the ROOT are the nodes in BRAM Level 1, and so on. The LAYER signal selects in which BRAM the children are stored, and the PARENT address selects which of the nodes in the BRAM are his children. An example of the comparison LAYERs is shown in figure 3.9.

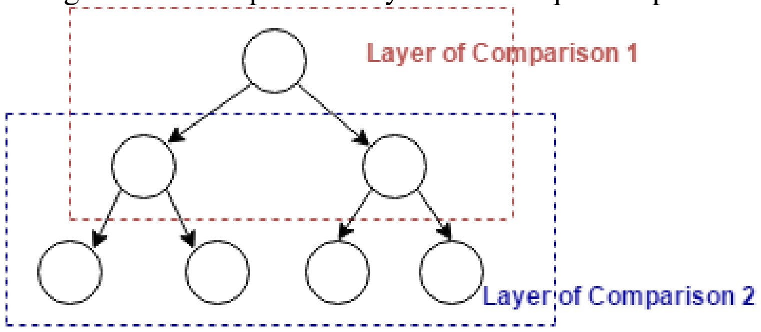
Figure 3.8 presents the architecture of this module.

Figure 3.8: Get Children Module.



Source: the author.

Figure 3.9: Comparison Layers on a Dequeue Operation.



Source: the author.

3.3.3.3 FSM Module

Figure 3.10 shows the fluxogram of the FSM.

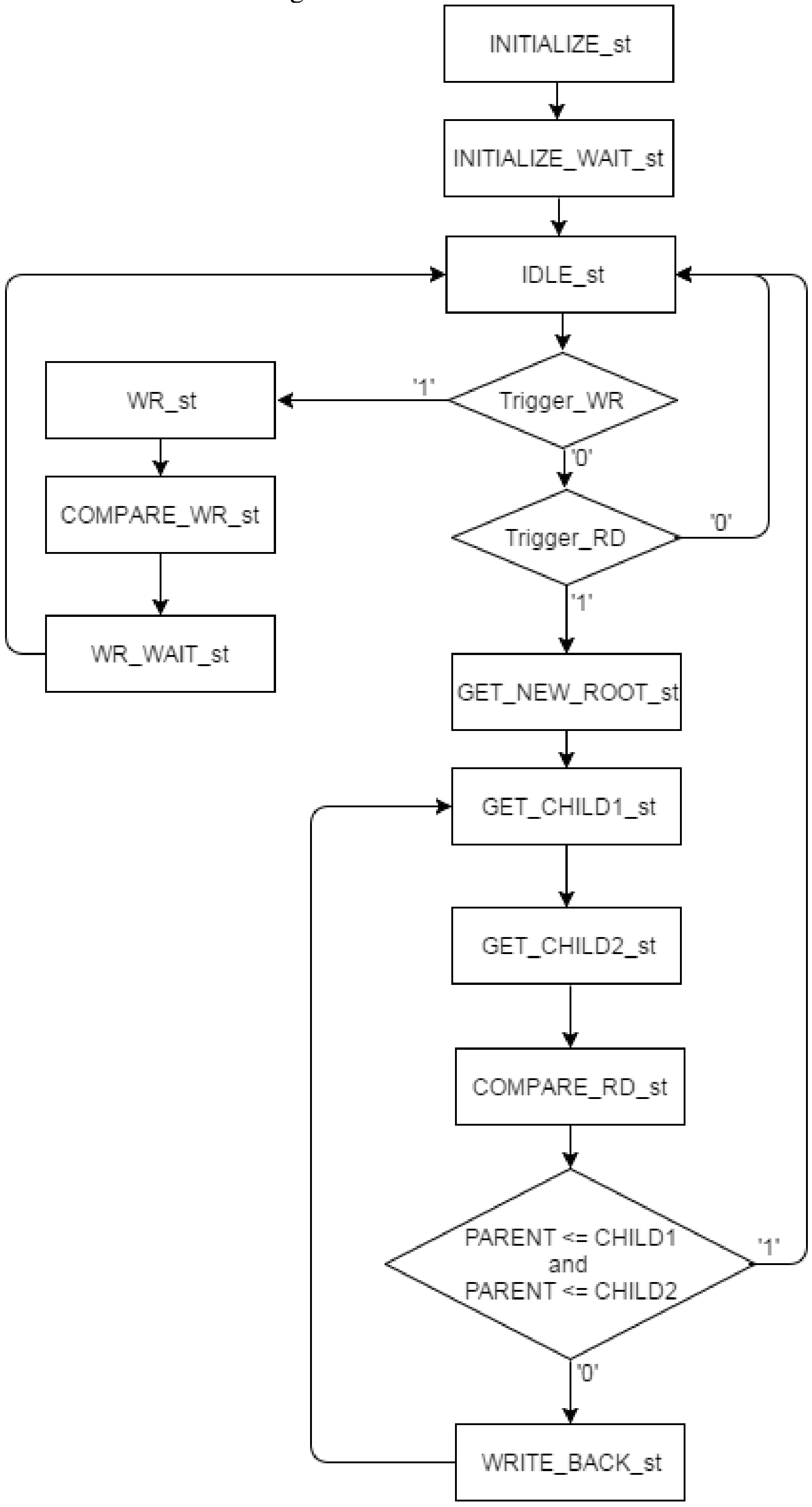
After the system is reset, 0xFFFFFFFF (the highest estimated cost possible)is written to all BRAM positions, this is handled at the INITIALIZE_st and INITIALIZE_WAIT_st. The INITIALIZE_st state sets each BRAM initial address and asserted the write enables, and the INITIALIZE_WAIT_st state keeps the enables asserted and increments the BRAM addresses so that all positions are covered. The write enables are de-asserted after the initialization of the values of the BRAMs is complete. This is done in order to be easier to check which positions are available and to end the compare/swap operation in the dequeue operation (in state COMPARE_RD_st).

After the initialization, the system goes to the IDLE_st until an operation is triggered. Since an insert operation in the queue cannot happen at the same time a removal is occurring, the FSM handles both operations to make sure the queue will always be ordered before a new operation.

If Trigger_WR is asserted, the Enqueue operation begins. At the WR_st all parents of the current vacant node are retrieved from the BRAMs (the address of the parents was already calculated by the Get Parents module). Then at the COMPARE_WR_st the node to be inserted (N1) is compared to its parents, there are three possible outcomes from these comparisons:

- If N1 has lower estimated cost than all of its parents, N1 becomes the new root and all the parents are shifted one level down;
- If N1 has lower estimated cost than some of its parents, it is inserted in the right spot and the parents with higher estimated cost are shifted one level down;
- If N1 has higher estimated cost than all the other parents, it will be inserted in the vacant spot and no other changes to the queue will happen.

Figure 3.10: FSM Flow.



Source: the author.

After all nodes' spots have been correctly defined, this data is written back to the BRAMs at the state WR_WAIT_st, the register SIZE is incremented (thus changing the output values of the GET PARENT module for the next queue insertion) and the FSM returns to the IDLE_st.

If Trigger_RD is asserted, the Dequeue operation begins. At the state GET_NEW_ROOT_st, the element in the SIZE + 1 position is retrieved from the BRAMs and stored in the register ROOT. 0xFFFFFFFF is written in the position SIZE + 1.

After a new root is defined, this value is loaded to the PARENT register and the CHILD1 and CHILD2 of this node are retrieved from the BRAMs at the GET CHILD states. This has to be done in two states because the children are in the same BRAM. Three comparisons happen at the COMPARE_RD_st:

- If the PARENT has lower or equal estimated cost than CHILD1 and CHILD2, the dequeue operation is over. SIZE is decremented and the FSM returns to the IDLE_st;

- If CHILD1 has lower estimated cost than PARENT and CHILD2, there is a swap between PARENT and CHILD1 values. New PARENT and CHILD1 values are written back to the BRAMs in the WRITE_BACK_st state, and CHILD1 value is loaded to the PARENT register for the next iteration of the dequeue. The LAYER of comparison is incremented (thus triggering an update on the CHILD1/CHILD2 addresses);

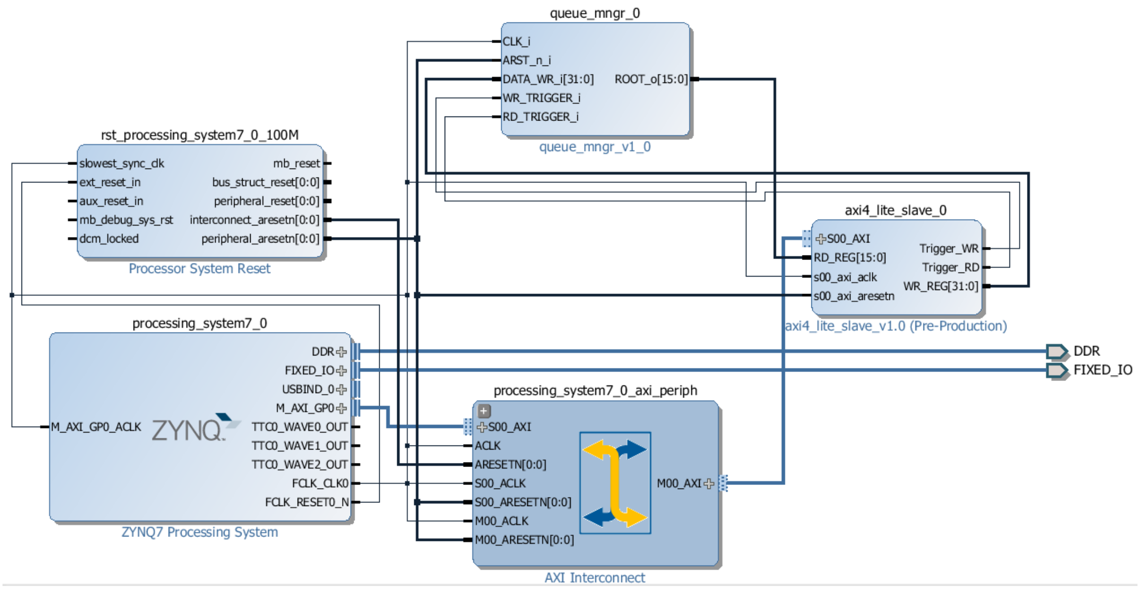
- If CHILD2 has lower estimated cost than PARENT and CHILD1, there is a swap between PARENT and CHILD2 values. New PARENT and CHILD2 values are written back to the BRAMs in the WRITE_BACK_st state, and CHILD2 value is loaded to the PARENT register for the next iteration of the dequeue. The LAYER of comparison is incremented (thus triggering an update on the CHILD1/CHILD2 addresses).

GET_CHILD1, GET_CHILD2, COMPARE_RD_st and WRITE_BACK_st are repeated until no more swaps happen.

3.3.4 Vivado's Hybrid Solution Top Level Block Diagram

The top level block diagram of the implemented solution in Vivado 2014.2 is presented in figure 3.11.

Figure 3.11: Vivado's Hybrid Solution Top Level Block Diagram.



Source: the author.

The top level block diagram is divided in 5 blocks:

- processing_system7_0: It is Xilinx's ZYNQ processing system IP. The processor block generates the FPGA clock (FCLK_CLK0). As shown in section 3.2, the PL is connected to the programmable logic through the M_AXI_GP0 port.
- processing_system7_0_axi_periph: It is Xilinx's AXI Interconnect IP. This block is automatically added when one of the processor's AXI interfaces is used.
- rst_processing_system7_0_100M: This block generates the system reset. It is also automatically added when one of the processor's AXI interfaces is used.
- axi4_lite_slave_0: This is the AXI Registers block as described in subsection 3.3.1.
- queue_mngr_0: This is the queue manager block as described in subsection 3.3.3. The BRAMs are instantiated inside this block.

3.4 Test Environment

The goal of this project is to accelerate the A* algorithm by implementing some of its functionalities in hardware. The software only solution was simulated entirely in the ZEDBOARD processors with a clock frequency of 660 MHz. The hybrid implementation was simulated also in the ZEDBOARD, with a clock frequency of 100 MHz for the programmable logic.

The programs that generate files according to the Challenge 9 format do not generate the '.co' files. Since heuristics admissibility depends on the coordinates (the heuristic function should never be greater than the actual cost from the current node to the goal node).

The graphs where generated using a program called GTgraph (BADER; MADDURI, 2006). This program generates '.gr' files according to the Challenge 9 format, the size of the graph is configurable. The coordinates file was generated manually, with a maximum of 15 arcs, which results in a queue with a maximum of 4 levels. Despite this limitation, the results show that when comparing both designs there is a performance gain in the hybrid solution.

The algorithm's running time is kept by a hardware counter with a 10 ns resolution, that is enabled via AXI transaction when the algorithm starts and disabled when the algorithm ends. This was done because the C 'time' librarie do not have nanosecond/millisecond precision, which is where there is a difference in the solutions' times. The run time as well as the nodes of the final path to the goal were obtained with the help of Xilinx's Microprocessor Debugger (XMD).

The min binary heap in software and the min binary heap in hardware went through three tests:

- Full Depth Insertion: Insert data until all levels of the queue are full;
- Emptying A Full Queue;
- Full A* implementation.

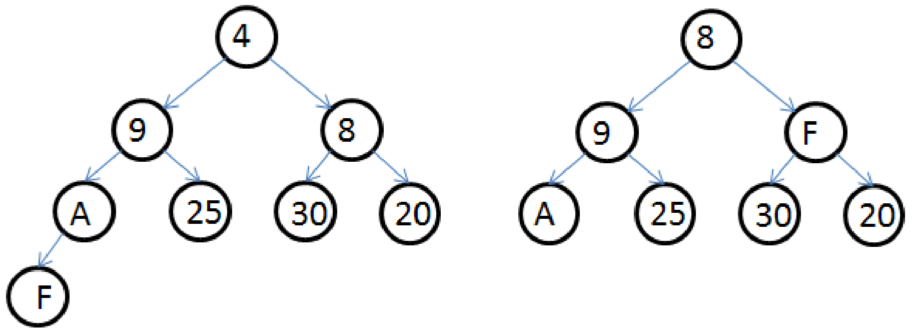
All tests were handled by one blind operator. Each test ran 100 times with random input data.

4 RESULTS

4.1 VHDL Priority Queue HW Simulation

While creating the VHDL implementation of the priority queue, several simulations were ran to guarantee the integrity of the operations and that the min binary heap is always correctly ordered. From the final simulation, it was possible to get a notion of how long the enqueue and dequeue operation take. For instance, the average time to insert a node in the queue is 4 clock cycles, to remove the root it takes 1 cycle and to re-order the queue after a removal can take from 10 to 26 cycles depending on the value of the last leaf and the depth of the queue.

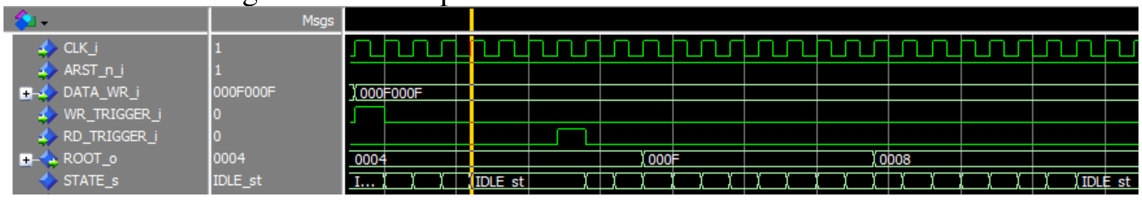
Figure 4.1: Example Graph For Final Simulation Waveform. First F is inserted, then 4 is removed and the heap re-ordered



Source: the author.

Figure 4.2 presents the waveform of the final simulation of the queue manager in the simulator ModelSim. In the beginning of the wave, 4 is being inserted in the queue. After this operation a node removal happens, the node with priority 4 is removed and after some cycles the root is updated with the node with priority 8.

Figure 4.2: Excerpt Of The Final Simulation Waveform

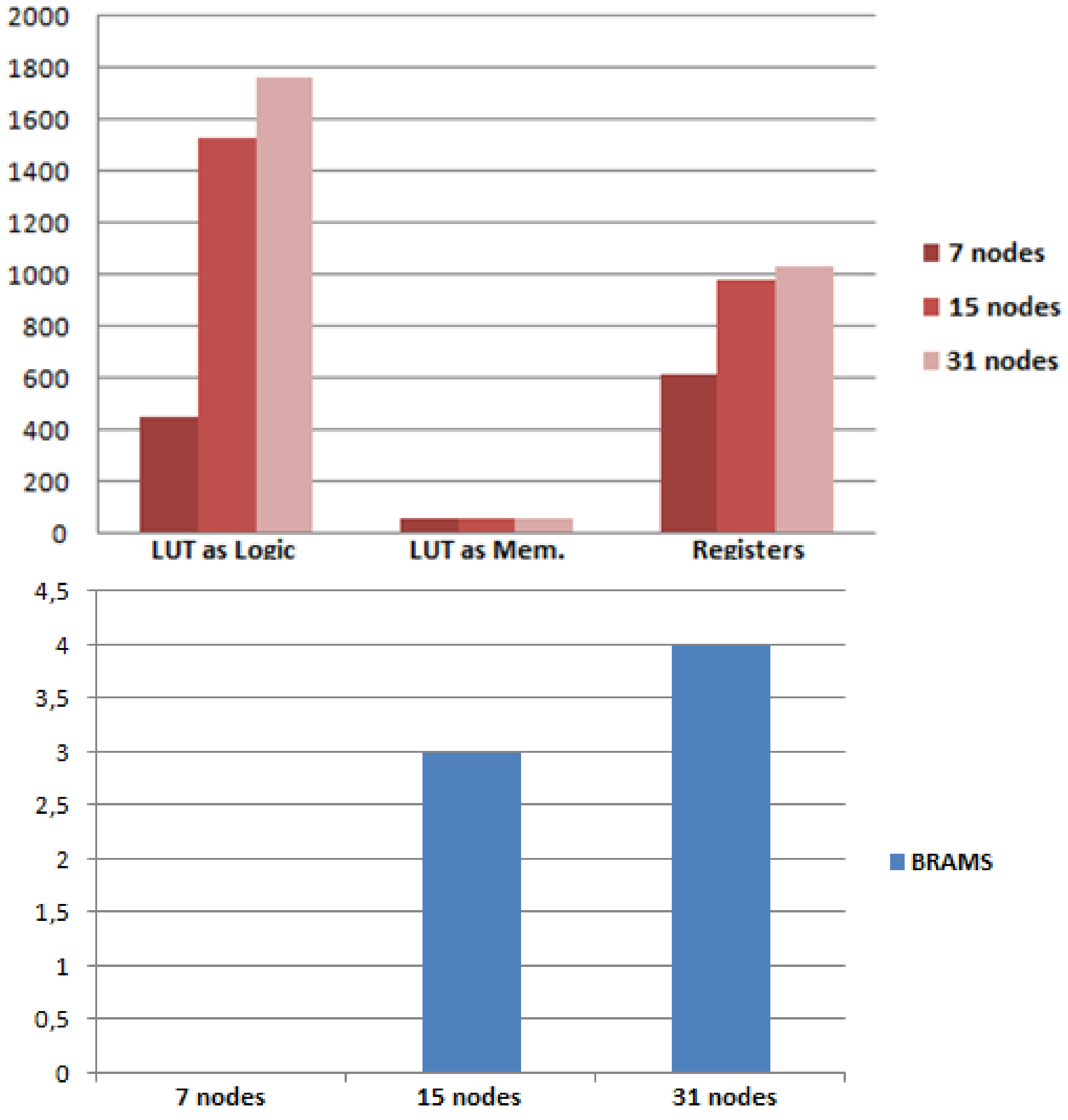


Source: the author.

4.2 VHDL Priority Queue Synthesis/Implementation

After successfully simulating, the design was synthesized and implemented using Vivado 2014.2. Figure 4.3 shows the resource utilization for three different sizes of the priority queue implementation.

Figure 4.3: VHDL Priority Queue Resource Utilization on the ZYNQ SoC.



Source: the author.

Analyzing the timing report for the implemented solution it is possible to estimate the maximum clock frequency based on the Worst Negative Slack (WNS) of the clock signal in the Intra-Clock Paths section. The WNS of the design is 1.506 ns, this means that the minimum clock period is 8.494 ns, hence the maximum clock frequency that the programmable logic will operate correctly is approximately 117.73 MHz.

Vivado also generates reports on power after routing, which provides information about the usage of voltage and other power resources for the entire design. According to this report, the slice logic spends 0.006 W, the Block RAMs spend 0.013 W and the PS7 spends 1.308 W. The static power of the design is 0.147 W. Considering that this report is done after the routing, when there is no program loaded in the processor, there is no way to know how some signals will behave (their switching rates and such), so it is safe to assume that the power consumed by the components is just a power estimate.

4.3 Full Depth Insertion

The first test made was to compare the insertion time of the bare metal software heap and the hybrid solution. 15 random nodes, the maximum capacity of the heap, were inserted 100 times and the run time was measured. The hybrid solution had an average of 1.449 us and the software solution had an average of 3.827 us. Therefore, the hybrid solution is 62.13% faster for insertion.

4.4 Emptying A Full Queue

Similarly to the procedure in section 4.3, 15 random nodes were inserted in the heap and then removed from the heap. After running this experiment 100 times, the average run time for the hybrid solution was 2.03 us and the average run time for the software only solution was 15.67 us. Hence, the hybrid solution is 87.04% faster.

4.5 Solutions With The A*

For the comparison of the two solutions' running time for a whole A* best-path search, the running time counter is only set to count only after all the data initialization (such as the adjacency list initialization) is done. The reason for this is that this part of the program is equal in both solutions.

The heuristics function is Euclidean distance between two nodes. For the generated testing graphs, this is an admissible heuristic. For a 15-arc graph, starting on random points and with random goals, the average running time measured for the hybrid solution was approximately 5.41 us and the running time measured for the software only solution was 17.36 us.

5 DISCUSSION

From the VHDL Priority Queue HW Simulation it is possible to conclude that the re-ordering of the queue is by far the most time consuming operation. This is so, due to the queue architecture presented in section 3.3. Each level of the queue is represented by a BRAM, which is terrible when the FSM needs to read the children of a node (they are in the same BRAM). And there is the delay to read the data from the BRAM. So to get the children of node the following steps occur in the FSM (each represent 1 clock cycle):

1. Put address of the first child + asserts read enable;
2. wait for the BRAM to return the value;
3. Put address of the second child + save first child value;
4. wait for the BRAM to return the value;
5. Save second child value + de-asserts read enable.

The full depth insertion test result makes sense, considering that each time the software handles a variable (during the heap enqueue and specially the dequeue) a memory operation occurs, and in the hybrid solution when adding a node, all nodes in the path (all parents of the first vacant spot + the new node to be added) are inserted in parallel to the queue. These results show that, even though the software operates at a higher clock frequency, there are so much data handling that the memory delay compensates it, thus making the hybrid solution faster.

The removal has a larger running time in both solutions, because they both have to retrieve the parent's children (in the same BRAM/memory) and compare and swap until all the nodes are in the right position again. This can be very time consuming depending on the value of the selected leaf node. If it has a very high cost when compared to the other nodes in the heap, it will take more cycles for it to reach its right position at the bottom.

While comparing the entire A* algorithm implementation using both solutions, there was a increase in performance of 68.8%. The improvement is more similar to the insertion in the queue running time than to the removal, that is because all neighbour nodes are inserted in the heap, but not all nodes are removed. Moreover, some parts of the code remained in software, reducing the total acceleration. One node is removed at each A* iteration until the goal node has been found, which can be way before the queue is empty.

Differently from (TOMMISKA; SKYTTA, 2001), this project's selected algorithm for best-path finding is the A* algorithm, which expands fewer nodes with a good heuristic. The architecture proposed by (TOMMISKA; SKYTTA, 2001) is similar to the one proposed in this project, except for the partitioned implementation in hardware and software, which gives the solution more flexibility (as explained before). Another difference is that a priority queue was not used in the implementation of the Dijkstra algorithm. According to the authors, not using it was a drawback considering the runtime, because the runtime of a priority queue is O(E lg N^2) (where E is the number of arcs and N is the number of nodes) and the runtime without it is O(N^2). So the use of priority queues increases the performance of this type of search algorithms (the Dijkstra algorithm is a case of the A* algorithm).

Comparing the proposed solution with the one proposed by Idris et al. (2009), implementing the A* algorithm partially in software gives it greater flexibility, any graph structure will be accepted and stored. Admitting different types of graphs/applications also implies in having different methods for calculating the heuristic.

6 CONCLUSION

The goal of accelerating the A* bare metal software by splitting its functionalities into software and hardware was successfully achieved. The comparison of the results of a min binary heap implemented entirely in software compared to one implemented in hardware demonstrated that the hardware solution is 62.13% faster for insertion and 87.04% faster for removal of an entire 15-node queue.

The next step on the implementation of this solution is to create a mechanism capable of handling the content in the BRAMs and in the DRAM of the ZEDBOARD. That way when there are more nodes than space in the BRAMs, this new mechanism will expand even more the capacity of the queue putting/getting data in/from the DRAM. This mechanism could be implemented in hardware or in software (as suggested by Kumar et al. (2014)). Additionally, a possible next source of acceleration is to transfer the cost and heuristics functions calculations to the hardware.

REFERENCES

AKIBA, T.; IWATA, Y.; YOSHIDA, Y. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. **SIGMOD'13 Proceedings of the 2013 SIGMOD International Conference on Management Data**, p. 349–360, 2013.

ALI, A. A. S. et al. Heterogeneous implementation of ecg encryption and identification on the zynq soc. **2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)**, May 2016.

BADER, D. A.; MADDURI, K. Gtgraph: A synthetic graph generator suite. (paper). 2006. Available from Internet: <http://www.cse.psu.edu/~kxm85/software/GTgraph/gen.pdf>.

BLOOM, G.; G., P.; NARAHARI, B. Shared hardware data structures for hard real-time systems. **EMSOFT'12 Proceedings of the Tenth ACM International Conference on Embedded Software**, p. 133–142, Oct 2012.

BONDHUGULA, U. et al. Parallel fpga-based all-pairs shortest-paths in a directed graph. **20th InternationalParallel and Distributed Processing Symposium**, Apr 2006.

BTDI. **Altera's Next-Generation FPGAs: Advanced Process Lithographies Lead to Performance, Power Consumption Efficiencies**. 2013. Available at: http://www.bdti.com/InsideDSP/2013/07/11/Altera Accessed: October 2016.

DROZDENKO, B. et al. High-level hardware-software co-design of an 802.11a transceiver system using zynq soc. **2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)**, April 2016.

DUCHON, F. et al. Path planning with modified a star algorithm for a mobile robot. In: **Modelling of Mechanical and Mechatronic Systems**. [S.l.]: Procedia Engineering, 2014. p. 59–69.

HART, P.; NILSSON, N.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. **IEEE Transactions on Systems Science and Cybernetics**, p. 100–107, July 1968.

IDRIS, M. Y. I. et al. High-speed shortest path co-processor design. **Third Asia International Conference on Modelling & Simulation**, May 2009.

KEMPE, D.; KLEINBERG, J.; TARDOS, E. Maximizing the spread of influence through a social network. **SIGKDD 2003 Washington**, 2003.

KUMAR, N. et al. Hardware-software architecture for priority queue management in real-time and embedded systems. **Int. J. Embedded Systems**, v. 6, p. 319–334, 2014.

LIU, X.; GONG, D. **A comparative study of A-star algorithms for search and rescue in a perfect maze**. 2011.

PRIYA, T. K.; R., K. P.; SRIDHARAN, K. A hardware-efficient scheme and fpga realization for computation of single pair shortest path for a mobile automaton. In: **Microprocessors and Microsystems**. [S.l.]: Elsevier, 2006. p. 413–424.

PUTNAM, A. et al. A reconfigurable fabric for accelerating large-scale datacenter services. **2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)**, June 2014.

RöNNGREN, R.; AYANU, R. A comparative study of parallel and sequential priority queue algorithms. **ACM Transactions on Modeling and Computer Simulation (TOMACS)**, v. 7, p. 157–209, Apr 1997.

RUSSEL, S.; NORVIG, P. **Artificial Intelligence - A Modern Approach**. 3rd. ed. Essex-United Kingdom: Pearson Education Limited, 2014.

TOMMISKA, M.; SKYTTA, J. Dijkstra's shortest path routing algorithm in reconfigurable hardware. **Field-Programmable Logic and Applications - 11th International Conference**, 2001.

TSENG, F. H. et al. A star search algorithm for civil uav path planning with 3g communication. **Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP)**, Aug 2014.

UKKONEN, A. et al. Searching the wikipedia with contextual information. **CIKM '08 Proceedings of the 17th ACM conference on Information and knowledge management**, p. 1352–1352, 2008.

VIEIRA, M. V. et al. Efficient search ranking in social networks. **CIKM '07 Proceedings of the sixteenth ACM conference on Conference on information and knowledge management**, p. 563–572, 2007.