Digital Design, Verification and Functional Safety Of A* Path Planning Algorithm

Submitted by:

Adham Osama Ali
Ahmed Mostafa Abdelrahman
Eslam Mamdouh Mohamed
Mohamed Gamal Tawfik
Usama Imam Kamal

Submitted to:

Dr. Hassan Mostafa

Communications and Electronics Department,

Faculty of Engineering, Cairo University

Abstract

The conventional A* algorithm consumes a lot of time due to its large number of iterations. In every iteration, the memory is accessed for multiple data structures, functions are evaluated then sorted into the queues. This A* accelerator is designed for real time applications. Parallelism is used, a special cache is designed and modifications to the algorithm are introduced to overcome all the bottlenecks and achieve the target times even after inserting safety mechanisms into the design. The design is functionally and formally verified for random and edge scenarios. It's also implemented in Xilinx Virtex-7 to be evaluated. Experiments prove that this implementation achieves 100s times enhancement for low obstacle maps and 50s times for high ones relative to software implementation. The design is suitable for real-time applications both before and after inserting the safety mechanisms.

Acknowledgment

First and foremost, we would like to thank our supervisor, Dr. Hassan Mostafa, for his assistance and dedicated involvement in every step throughout the process providing a continuous full and professional support.

In addition, we would also like to show gratitude to Mentor Graphics Company for their cooperation and support and for providing us with the all the tools needed to accomplish this work.

Finally, we would also like to thank Eng. Islam Ahmed, Eng. Mohamed Ayman, and Eng. Ahmed Khalil, who were always there for any questions and provided us with insightful comments and immense knowledge.

Thesis Outline

List of Figures

List of Tables

This page was intentionally left blank

CHAPTER 01 Introduction

Organization of the thesis

Chapter 1 provides overview of the thesis and an explanation to path finding algorithms. Chapter 2 illustrates the A* algorithm and how it works. Chapter 3 shows how we visualized the algorithm and its important metrics, it also discusses the heuristics that can be used and our choice for our implementation of the A* algorithm. Chapter 4 takes apart every module in our design, explaining its functionality and how it's implemented and how the whole design works together. Chapter 5 displays a problem within the algorithm itself, shows an example of why it occurs, proposes our solution for the problem, how it was implemented and integrated into our design and how it affected our implementation's performance. Chapter 6 is concerned with the Formal Verification of our design, it discusses how every module what verified and what assertions where proven. Chapter 7 illustrates how our design was functionally verified using the UVM environment and how it was established. In chapter 8, we discuss how the design was implemented on the Virtex-7 FPGA. We discuss the initials faults in the floorplanning and how they were overcome. Also, we show the strategies used for the PnR tools and the final results obtained. Chapter 9 talks about how the design was packaged and integrated with other modules to be deployed and tested on the FPGA. Finally, chapter 10 discuss the functional safety of the design talking about the ASIL levels and what safety mechanisms where used using which tools.

Path Planning

Path planning is a computational problem that aims to find a sequence of movement of a certain object to travel between two points in a defined space. The problem has some associated abstract definitions like configuration, free, target, and obstacle spaces [1] but they are beyond the scope of this thesis and our project. Different algorithm can be categorized by their completeness. An algorithm is said to be complete if it can find a solution or report its inexistence in finite time. Another metric is whether the algorithm can find the best possible path or just any possible path. Algorithms that are incomplete and don't find the best paths, e.g. RRT algorithm, are usually not suitable for automotive applications as their performance and outputs are not guaranteed.

To choose a suitable algorithm from all the possible ones more metrics can be used. From the most notable algorithm are dijkstra and A* algorithms. Both are can find the best existing solution for a map but A* is found to be more suitable for our context. The difference lies in that A* has a heuristic function while dijkstra doesn't. The heuristic function makes the algorithm more complex however it acts like a guide to reach the goal in fewer iterations. The inexistence of a

heuristic in dijkstra makes it unguided and it just explores the entire map until it reaches the intended goal without a clear direction in the iterative part of its algorithm. The A* algorithm is explained more deeply in chapter 2 and its heuristic functions in chapter 3.

Functional safety

Functional safety is the part of a system or device that provides automatic protection for operating correctly even in the cases of a failure in a predictable manner. The standard defines functional safety as "the absence of unreasonable risk due to hazards caused by malfunctioning behavior of electrical or electronic systems" [1].Our design is a safety-critical automotive application as it's used in autonomous cars and the path that the vehicle would take is dependent on the decision of this circuit. Therefore, it's very important to recover from any failure in run time to avoid crisis.

The automotive industry, has developed the ISO 26262 Road Vehicles FUSA Standard. The certification of systems that follows these standards ensures the compliance with the important regulations and helps to decrease probability of accidents.

The standard ISO 26262 particularly addresses the automotive development cycle. It is a multi-part standard defining requirements and providing guidelines for achieving functional safety in E/E systems installed in series production passenger cars. This ISO standard is considered a very good guideline for achieving automotive functional safety.

The ISO 26262 standard defined a risk classification system for the functional safety of road vehicles which is Automotive Safety Integrity Level abbreviated as ASIL. There are four ASIL levels set by ISO 26262 which are A, B, C, and D. Level A represents the lowest hazard rate while Level D represents the highest degree of hazard rate.

CHAPTER

02 A* Algorithm

A* Search algorithm is one of the best and popular techniques used in path-finding and graph traversals. It can be seen as an extension of Dijkstra's algorithm. A* achieves better performance by guiding its search so It is really a smart algorithm, which separates it from the other conventional algorithms.

A* is considered an informed search algorithm which is more useful for large search space. Informed search algorithms use the idea of heuristic, so it is also called Heuristic search.

Consider a square grid having many obstacles and we are given a starting node and a goal node as shown in figure 2-1. We want to reach the goal node and navigate around the map's obstacles from the starting node as quickly as possible. Here A* Search Algorithm comes to be a solution.

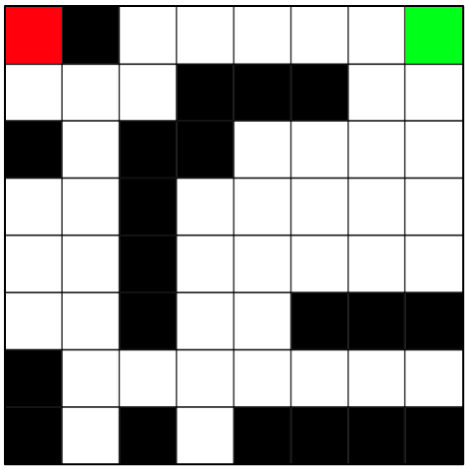


Figure 2-1: Square grid map

Movement through the map is done in steps. What A* does is that at each step it picks the node according to a cost value $f(n)$ which is the sum of two parameters - $g(n)$ and $h(n)$. To add the two parameters, those two need to be at the same scale. $g(n)$ Represents the movement cost of the path to move from the starting node to any a given node on the grid, and $h(n)$ represents the estimated movement cost from that given node on the grid to the goal node.

$h(n)$ Is called the heuristic, which is an important parameter in A* algorithm calculations. It represents a smart guess to move faster to our goal destination. This knowledge helps the algorithm to explore less to the search space and find the goal node more efficiently. There can be many ways to calculate this parameter, which are discussed in the later in section 3.2.

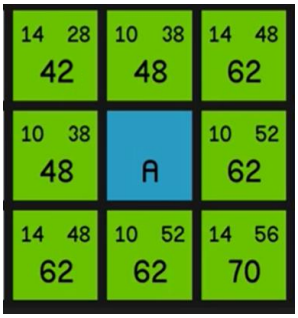


Figure 2-2: Parent node with its Childs having calculated costs

At each iteration, the algorithm starts to pick a specific node on the grid map and it is only allowed to reach one of its eight neighbors. The algorithm computes the $f(n)$ cost as expressed before for all the neighboring nodes as shown for example in figure 2-2 and then enters the node with the lowest total estimated cost. Each neighboring nodes expanded by the algorithm are considered child nodes to the current node being processed and

the algorithm saves the data of the node being processed as a parent node for all eight children if it provides the lowest $g(n)$ for that child node.

The algorithm uses the parent node data to figure out the least path from the start node to the goal node.

The algorithm sorts the grid map nodes into two main lists that help in the operation, an open list which contains all the expanded nodes that are possible to be visited and not obstacles, in addition to the visited list at which the algorithm adds the visited nodes at the end of each iteration. When the lowest cost node is found to be the goal node the algorithm is terminated and this means the heuristic calculations should be zero.

The algorithm uses fixed costs for moving in vertical, horizontal and diagonal directions, these costs used in $g(n)\ and\ h(n)$ calculations. A* algorithm returns the path which occurred first, and it does not search for all remaining paths as its efficiency depends on the quality of used heuristic

CHAPTER 03

Modeling

Visualizing our model

The input to our design or to our high level model of the A* path calculator are some control signals in addition to the map, the start node and the goal node. The map itself is a large array of data structures for each node that describes everything needed in calculations for this node. The outputs of the design or the high level model are some signals like “done” signal which tells us the output is ready, in addition to the main and most important output which is the path.

The path that was found by the algorithm forms an array of addresses, each entry in this array would be the address for a node in the path that was found.

This form of inputs or outputs would be very hard to be traced by human and would take a very large amount of time, it’s only helpful for automated testing where some program can check the validity of the outputs or the inputs.

Due to all these facts, the design and model should be visualized to increase the observability of the algorithm for both digital design and high level model.

Visualizer program

A simple piece of code was developed using Python to visualize the map, the path and the explored nodes by the algorithm. Color codes was used where:

- White squares represent nodes that are not an obstacle or a path node.
- Black squares represent nodes in the map that are an obstacle.
- Green squares represent nodes that were chosen by the algorithm as path nodes.
- Yellow squares represent nodes that were explored by the algorithm, this mean that the algorithm at some point has calculated the cost of these node and found it to have the minimum cost among all other nodes calculated at this point, then moved to one of its children nodes and calculations continue furthermore, but eventually this node was found to be not the best to be included in the path.

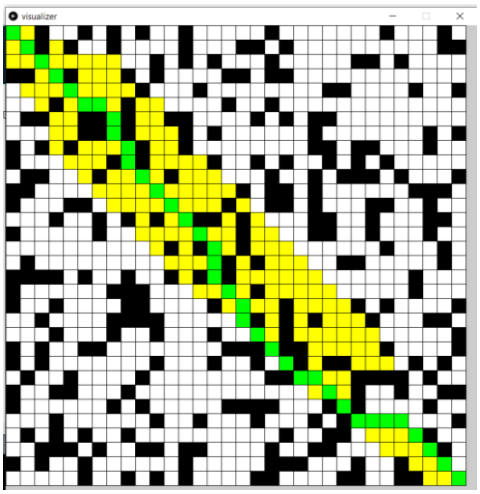


Figure 3-1: Visualizer program

Although visualizing the explored nodes may appear to be redundant, it was super important. Explored nodes quantity has a direct effect on the speed of the path finding by the algorithm, if large number of nodes is explored then the design will take larger time, so reaching the least cost path but at the same time keeping the number of explored nodes a small number would make a great efficient design.

Heuristic choosing

As mentioned in chapter 2, A* algorithm calculates the cost of each node as a sum of two costs, G(n) and H(n), Where G is the accumulative movement cost to go from the start node to current node, and H cost is the heuristic function which is calculated using various ways that will be discussed in this section.

G cost

According to [2], we typically assign G cost of 1 to the vertical or horizontal movement and 1.4 for the diagonal movement, basically, we chose these values as moving horizontally of vertically costs us 1 node or 1 square while moving diagonally costs us a square root of 2 which is approximately 1.4.

we can also multiply these values by a factor of 10 to be (10, 14) or by 5 to be (5,7) instead of (1, 1.4) for simplicity and for speeding up calculations of the algorithm on computers by avoiding using too much floating point calculations. But we must keep in mind that increasing the factor that G(n) is multiplied by will give it higher weight over H(n). The effect of a higher weight for one of the two parameters (G and H) will be explained in the next subsection.

G cost vs H cost

First, if H(n) is equal to 0, then g(n) only is effective and in this case A* algorithm turns into Dijkstra Algorithm which is same for A* in the point that they're searching for least cost path, but the main difference between the two is that Dijkstra explores all map nodes, while A* doesn't always explore all nodes so it's faster. Increasing the factor multiplied by G(n) without increasing the factor multiplied by H(n) will increase the explored nodes by the design, or we can say will slow the design.

On the other hand, if h(n) is relatively higher than g(n), then h(n) is only the effective term, and this turns A* searching algorithm into Greedy Best-First-Search, this can results in a path that doesn't have the least cost.

These facts introduced a tradeoff between accuracy and speed, when deciding on the weights that can be multiplied by each of the terms g(n) and H(n). to overcome this tradeoff and come to an end by the values chosen for our implementation we've decided to choose a suitable heuristic that will be relatively not greatly higher or lower than g(n) and we've decided to choose the weight of g(n) to be either 10 or 5 , so used values for g(n) would be (10, 14) or (5, 7) instead of (1, 1.4) to eliminate floating point calculations. In the next subsections we will discuss the chosen heuristic that will be suitable for the previous decided g(n).

Heuristics for grid maps

According to [3], there's some well-known heuristic functions that can be used such as:

- Manhattan distance
- Diagonal (octile) distance
- Euclidean distance
- Squared Euclidean distance (where Euclidean distance is used without taking the square root of the sum of the squares of horizontal and vertical distances)

Manhattan distance:

When comparing between the previous mentioned heuristic functions, we excluded Manhattan distance as it's more useful when there's only two directions for movement in the map (when diagonal movement is not allowed).

Euclidean distance and squared Euclidean distance:

The most famous and used heuristic function is the Euclidean distance which is calculated by the following equation

$$distance = \sqrt{x^2 + y^2} \qquad (1)$$

, however, using this function will force us to use floating point calculations in our design which will slow down our evaluators. When using squared Euclidean distance it might seem a good choice at first as it removes the need for square root which will make the design less complex and faster, but using this function will turn A* searching algorithm into Greedy Best-First-Search which results in a very fast path finding in some maps but in some other maps it will give us a path that doesn't have the least cost.

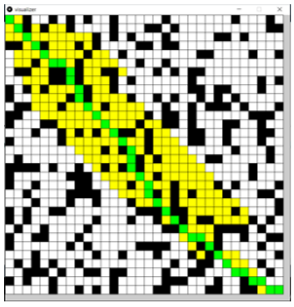
Diagonal distance:

In this function, to compute the distance we can calculate steps you can move if you can't move diagonally, then we subtract the number of steps we have saved by using the diagonal again, this can be calculated using this equation 1. This function is a very good substitute for Euclidean function especially for our design as it doesn't need the calculation of neither a square root function nor floating point calculations.

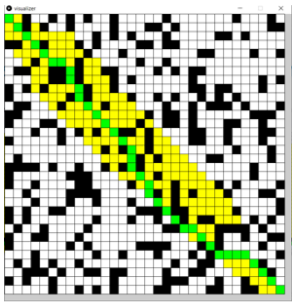
Comparing heuristic functions

Before deciding on replacing the typical Euclidean distance heuristic function by the diagonal distance heuristic function in our design we compared between different functions excluding Manhattan distance which is not suitable for our application.

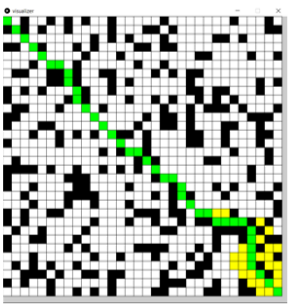
Results for a randomly generated map of each block with probability for each node of being an obstacle equal 30% are shown in figure 3-2 where obstacles nodes are in black color, explored nodes in the map is in yellow color and path nodes is in green color , the three functions gave us the least cost path, but in the favor of number of explored nodes, we can see that Euclidean function explored more nodes than the other two, this implies that this using this function in our design will be the slowest as there's many nodes that have been calculated and stored in design's lists. On the other hand, squared Euclidean distance explored very few nodes in the map, hence it's the fastest as it didn't take much time evaluating nodes and it got the path quickly.



Euclidean distance heuristic used for A algorithm on a 32x32 map*



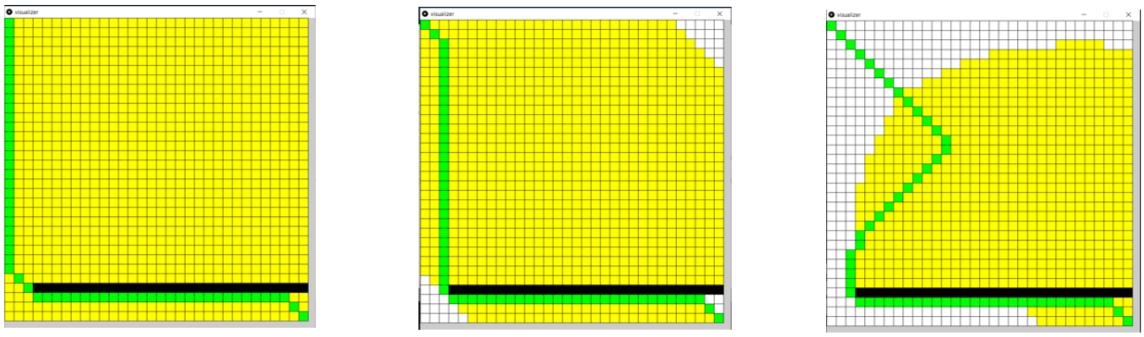
Diagonal distance heuristic used for A algorithm on a 32x32 map*



Squared Euclidean distance heuristic used for A algorithm on a 32x32 map*

Figure 3-2: comparison between outputs using different heuristic functions

After these results only it's very tempting to use squared Euclidean distance to reach the fastest implementation of the algorithm, but this impression have quickly changed after seeing outputs of the algorithm when using same three functions again but for other special maps like the one in figure 3-3 , here we can see that squared Euclidean distance couldn't reach the best path as it didn't explore enough nodes, as the weight of the heuristic function H(n) is so much greater than the G cost G(n), while both in both implementations where we have used Euclidean distance heuristic and diagonal distance heuristic, the best path has been reached. However, the implementation that used diagonal distance has explored less number of nodes and this means that this implementation is the fastest.



Euclidean distance heuristic used for A algorithm on a 32x32 map*

Diagonal distance heuristic used for A algorithm on a 32x32 map*

Squared Euclidean distance heuristic used for A algorithm on a 32x32 map*

Figure 3-3: comparison between outputs for Greedy Best-First-Search map using different heuristic functions

After these results we had to eliminate the squared Euclidean distance heuristic function, and compare between the normal Euclidean distance function and diagonal distance function. Table 3-1 provides the results of this comparison which shows that both heuristic function can give us the best path (the least cost path). The most important observation here is that diagonal distance heuristic function can be very useful especially when the map gets larger as we can explore less nodes and still reach the best path. The results in this table is for randomly generated maps with probability of each node for being an obstacle is 30%.

Table 3-1: comparison between euclidean and diagonal distance heuristic functions

Map size	Euclidean	Diagonal	Best path
16x16	143	132	Yes
32x32	284	204	Yes
127x127	4361	2626	Yes

Chosen heuristic

After previous results and comparisons, it was logical to say that diagonal distance heuristic function is the best fit for our design, as it will make the design faster by exploring less number of nodes and also by eliminating the need of square root or even any floating point calculations.

CHAPTER 04

Digital Design

Design of the A∗ Accelerator

As discussed in section (A* algorithm) each iteration requires to read the eight neighboring nodes data to perform the algorithm operations on their costs and then update the children costs and parent data. This creates a bottleneck during memory access as dual port embedded memory "block ram" is used for data storage. In addition, each iteration requires inserting and sorting the expanded nodes, which creates a new bottleneck as each iteration adds up to eight nodes to the open list. The hardware architecture is designed to tackle the bottleneck problems of the A∗ algorithm.

Data Structure

Data for all nodes in a map with 256×256 resolution is initialized and stored in one block ram. Each memory cell represents a node data structure and is accessed by the node address in the map. Information of each node is represented by the data structure of a total 35 bits shown in figure 4-1. This information includes:

- Parent address used by the algorithm to figure out the least path from the start node to the goal node.
- G cost represents the movement cost of the path to move from the starting node to the specific node on the grid.
- Obstacle/Visited information to help the algorithm take the decision to insert the new expanded nodes or not.

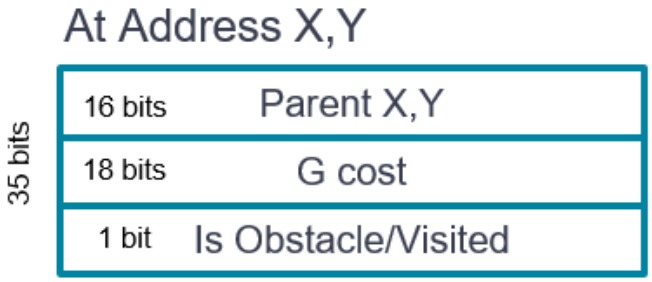


Figure 4-1: Used data structure

Data flow

Our design for A* algorithm is shown in figure 4-2 and its data flow:

- A* algorithm starts its operation from the starting node address until it reaches the goal address. The nodes manager control unit is initialized by the start node address and starts to request the eight child nodes data from memory manager.
- Memory manager is another control unit that only has a specific task, which is the data flow from memory to the algorithm to solve memory access bottleneck. After the data is ready, the algorithm starts to calculate the total estimated cost in eight directions in parallel in the arithmetic unit represented by the eight parallel evaluators.
- Each evaluator passes the expanded node with its cost to the queues to be inserted and sorted in the open list. Each queue has a top block having a node with the least cost to be easily compared by comparator engine block. Comparator engine block solves the problem of splitting the open list to eight queues and starts to compare the eight nodes having the lowest costs to extract the new node and pass the address to the node manager.
- The node manager starts to check if the algorithm reached the goal node or the new node is a visited one, and then starts to request the new child nodes data.

Reaching a new node is divided into three behaviors of the algorithm. First, reaching the goal node, the algorithm in this case is only represented by the path calculator block, which extracts the path from the start node to our goal. Second, reaching a visited node causes the algorithm to request a new one from the open list, and the visited lookup table helps the algorithm to check if a node is visited or not. Third, the new node is not any of the two previous cases, which causes the algorithm to request a new child data.

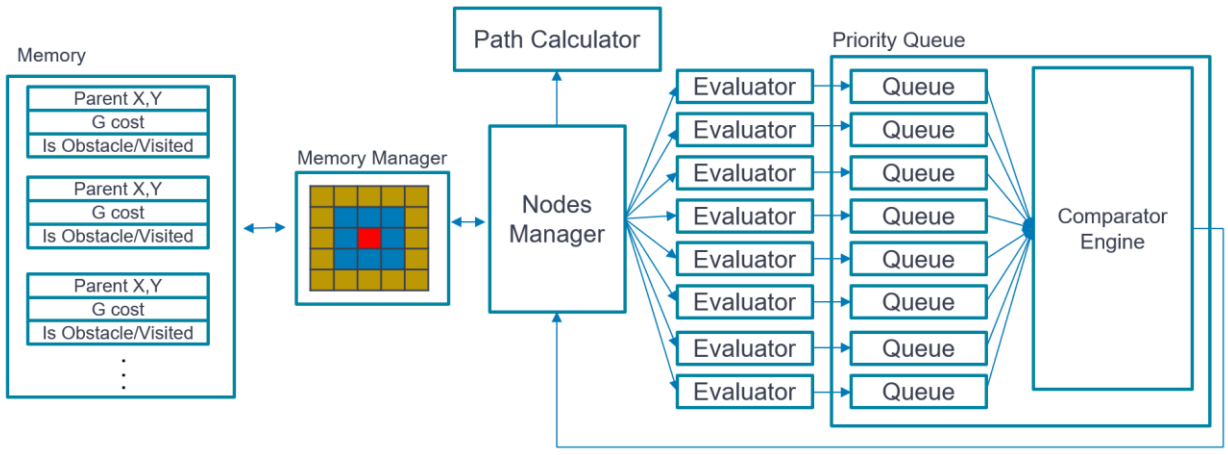


Figure 4-2: The full design

Nodes Manager

The Nodes Manager is considered the brain of the design as it controls the data flow and the enables of all the other blocks. The number of cycles the design spends in one iteration can be known from the number of cycles takes by the Nodes Manager to circle back to one of its states. Figure 4-3 shows its state diagram.

State Diagram

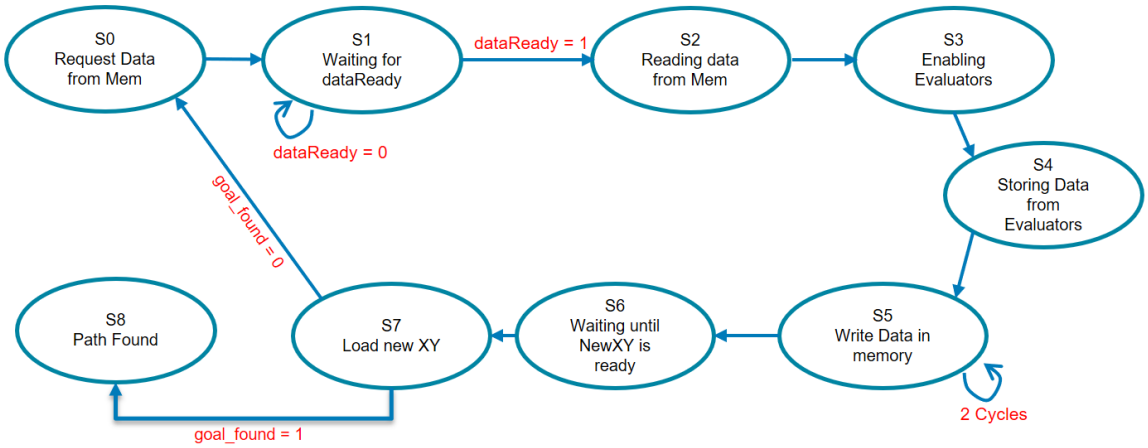


Figure 4-3: Nodes Manager FSM states

Each state corresponds to a step in the data flow discussed in Chapter 2.

(S0) The iteration starts by requesting the data needed for this node from the memory manager.

(S1) The design then waits for the memory manager to signal that the data is ready and The Nodes Manager stays waiting as much as the memory manager wants it to.

(S2-5) The nodes manager then proceeds to store this data in its internal registers as this data will be passed to the rest of the design, edited then passed back to the memory manager to store them in the main memory. The G cost of the child and the current node is passed to each evaluator then the results from the evaluators are stored again in the same registered. What exactly the evaluators do and what data do we get back is discussed later in Chapter 4 Section 5.

(S6) The Nodes manager then goes into an idle state where it waits for the data to propagate through the evaluators, then be sorted in the queues and the comparator engine and for the best node to be ready to be loaded in the current node register.

(S7) It then issues a load signal to that register and then examines the new node if it's the goal node or not. If it's not, then the nodes manager requests the data again and repeats the cycle(S0).

(S8) If the goal is reached then the nodes manager indefinitely enters a "Path Found" state where it disables all the iteration's modules and passes the control to the path calculator

In the case of the memory manager returning the data as soon as the data is requested, every iteration takes 10 cycles to complete. However, this state diagram is edited later to be 11 cycles per iteration after the inclusion of the visited look up table in Chapter 5. The lookup table edit increases the number of cycles per iteration but its overall impact is faster solution times because it decreases other iteration duration to just 2 cycles.

Internal Structure

The internal structure of the Nodes Manager is shown in Figure 4-4.

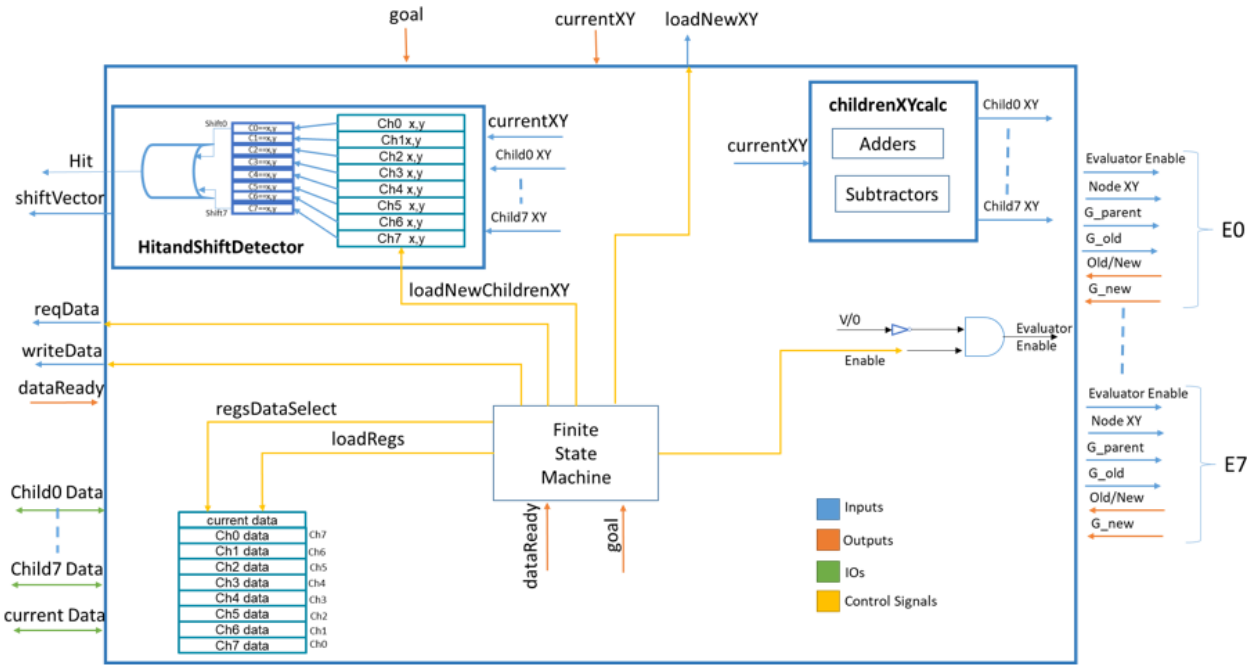


Figure 4-4: Internal Structure of Nodes Manager

In Figure 4-4, the right part is the nodes manager's interface with the evaluators showing the signals that are being sent and received between the 2 modules. Every evaluator needs the XY position of the nodes it's operating on; these values are calculated inside the **childrenXYcalc** by adding or subtracting ones from the X or Y of the current node. Other interface between Nodes manager and evaluator is explained in Chapter 4 Section 5.

The left part is the node's manager interface with the memory manager, the childX data signals are the buses in which the nodes manager and memory manager send the node's data structures back and forth. reqData, writeData, Hit, shiftVector and dataReady control signals which are explained later in Chapter 4 Section 4.

Memory

Each node in the design has 35 bits of data. These data structures have to be stored in a memory of depth equal to the number of nodes present in the map. For our case, the map is a 256×256 square so the memory depth is equal to 65536 memory locations. However, this number can be scaled freely for the design to handle larger or smaller sized of maps. An additional location is added at the end of the memory for when the design tries to access out of bounds nodes, this is explained deeply in Chapter 4 Section 4. This brings the total depth of the memory to number of nodes plus one, in our case 65537. Our design is designed to communicate with a dual port memory since the algorithm is very memory dependent. The design is targeted towards FPGA deployment so using the internal BRAMs is the best option for speed and area but any type of two port memory can suffice. To satisfy the previously mentioned depth and width 65 BRAMs were used to store all the needed data. Figure 4-5 shows the data stored and the memory's interface. When the algorithm is trying to solve the map, this interface is under control of the memory manager alone. This control is handed over to the path calculator when the solution is found and the algorithm is constructing the path.

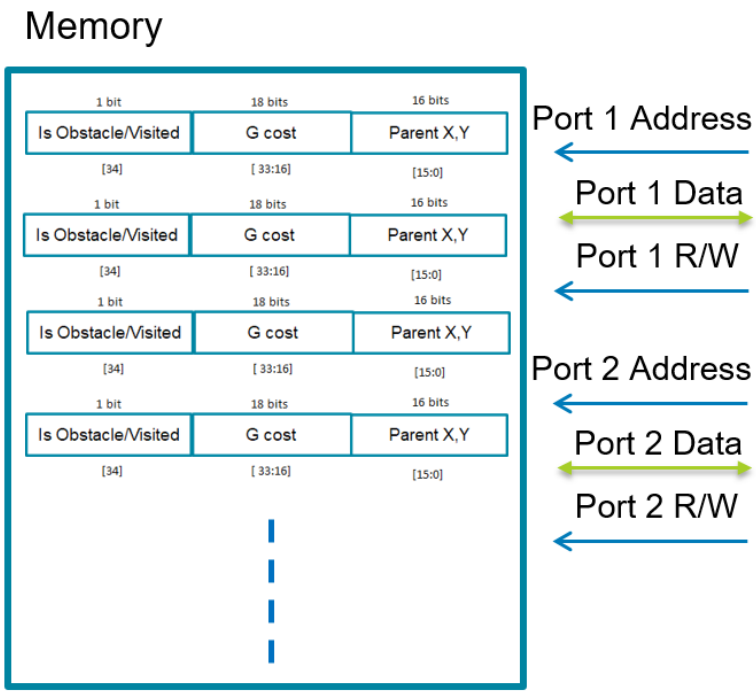


Figure 4-5: Memory's Structure

Memory Manager

The memory manager can be thought of as the cache controller of our A* design. Previous implementations had severe bottlenecks in memory access because the algorithm is very memory intensive; where in every iteration it need a large data structure of the 8 neighboring nodes. The memory manager handles all the memory accesses to read and write and is the middle-man between the Nodes manager and the memory.

Just like in a normal cache, spatial and temporal locality are used to expect the next needed bits of information so when they are requested, they can be immediately delivered without hanging the design too much. Of course, this expectation hits and misses because the locality is only a probability but the memory manager is designed to handle both efficiently and as fast as the memory can handle.

Locality

As discussed in Chapter 4 Section 1, the algorithm needs the full data structure (Visited or Obstacle, Best Acquired G cost, The associated parent with that G cost) for all 8 neighboring nodes for every iteration. After getting the data and computing the F costs of all the nodes, they are compared to all previously computed F costs to choose the least one to be the next explored one.

The cache miss happens when the next chosen node is not one of the previously computed ones and the hit happens when the next chosen node is one of currently neighboring nodes. And because the algorithm contains a heuristic that guides it towards the destination the cache hit happens more often than usually expected from heuristic-free algorithms. This locality indicates that the memory manager should contain the data structures of the current node, its neighboring nodes and their neighboring nodes as well. This is illustrated in Figure 4-6, the red node is the currently investigated node, the blue ones are the neighboring nodes and the yellow nodes are the neighbors' neighbors. The yellow nodes are only present in the memory manager waiting for a cache hit so they can be immediately sent to the algorithm.

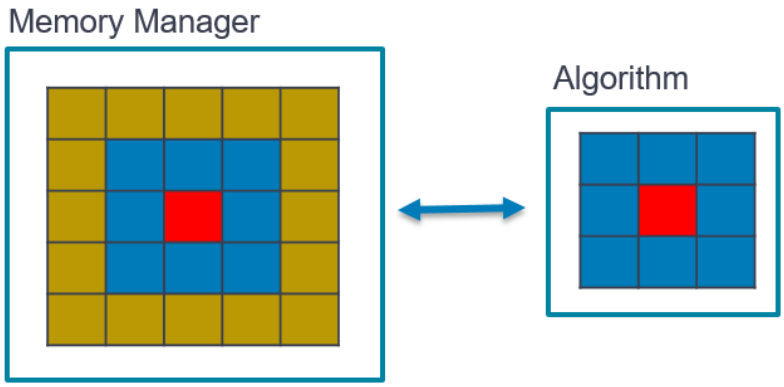


Figure 4-6: Spatial and Temporal Locality used in Memory Manager

Illustrative Example

To better explain the operation of the memory manager and how it sets itself to be ready for the data requests of the algorithm, we can illustrate two examples for hit and miss scenarios.

Hit

Suppose the algorithm just finished its iteration and will request data. The state inside the memory manager is as shown in Figure 4-7(a). The data structures are stored inside the memory manager for the current, neighboring and their neighboring nodes, color coded as explained before.

The memory manager is then signaled that a hit occurred and the algorithm will procced on of the neighboring (blue) nodes, for this example the north eastern one. The request is send to the memory manager as in Figure 4-7(b). And the memory manager starts responding.

Since the algorithm needs all the neighboring nodes of the new node and they are already stored in the memory manager, the needed data is immediately passed to the algorithm. Theses nodes are shown in Figure 4-7(c), colored purple.

While the algorithms processes the received data, the memory manager returns to its original state. It shifts its internal registers in the opposite direction, in this case in the south western direction. Then accesses the memory to fill out the missing data for the new neighboring neighbors of the new node. The needed data are the 2 new sides resulting from the internal registers' shift shown colored green in Figure 4-7(d).

After getting all the needed data from the memory, The memory manager returns to its original state, Figure 4-7(a), and is ready for another data request, hit or miss.

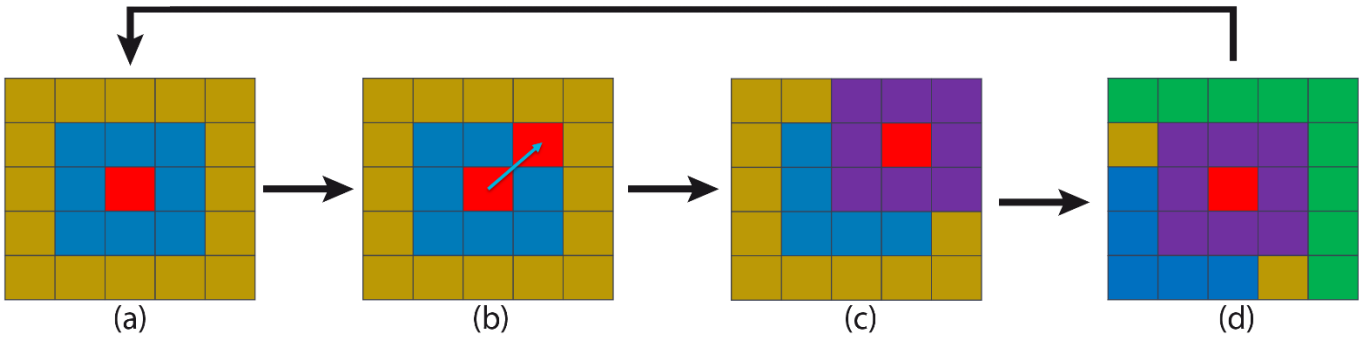


Figure 4-7: Memory Manager internal register states in case of cache hit

Miss

Suppose the algorithm just finished its iteration and is ready to request data. The state inside the memory manager is as shown in Figure 4-8(a). The data structures are stored inside the memory manager for the current, neighboring and their neighboring nodes, color coded as explained before.

The memory manager is signaled that a miss occurred, meaning that the algorithm has moved to a node other than the neighboring nodes to the current node. The request is send to the memory manager as in Figure 4-8(b). And the memory manager starts responding by signaling to the system to halt and wait until it can get the new data from the memory.

The memory manager reads the data structures of eight neighboring nodes and sends them to the algorithm as soon as it fetches them. These nodes are colored blue in Figure 4-8(c).

While the algorithms processes the received data, the memory manager returns to its original state. It reads from the memory to fill out the missing data for the new neighboring neighbors of the new node. The needed data are all the exterior sides shown in green in Figure 4-8(d).

After getting all the needed data from the memory, The memory manager returns to its original state, Figure 4-8(a), and is ready for another data request, hit or miss.

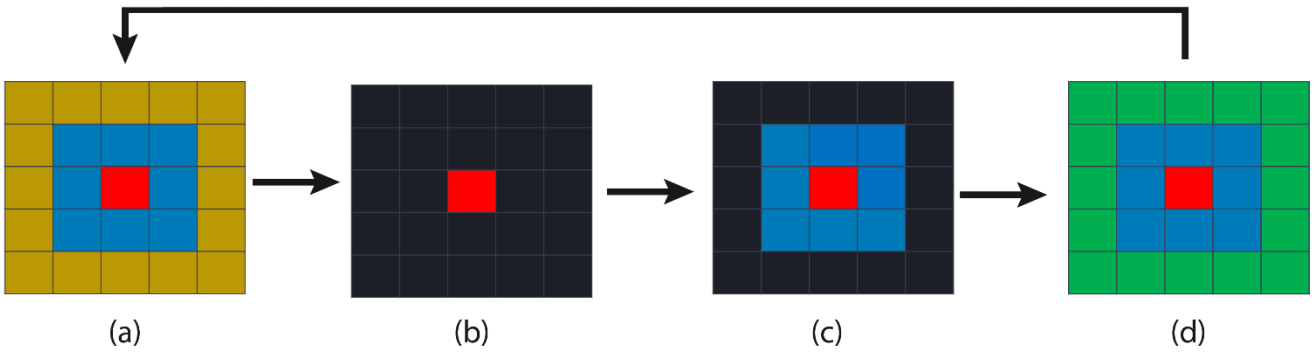


Figure 4-8: Memory Manager internal register states in case of cache miss

Other Functions

The memory manager doesn't only rebuild its state and read from the memory. The algorithm needs to store new information inside the memory in case it computes a G cost for a node better than the value already stored in the memory. This is done by giving the algorithm control of the internal 8 registers, colored blue in Figure 4-8(a), so the algorithm can write the data it wants to store. After the memory manager finishes rebuilding its state, it writes all the new information in the main memory then it waits for another data request.

Edge Nodes

One special case that needed special handling was trying to access neighboring nodes of a node that lies on the edge of the map. Suppose we are at node 00,00. As discussed in the previous section, the address where the data is stored is the XY of the node. So when we try to access the north neighbor of said node, the algorithm should treat this as an obstacle. A fix like that can work easily in a higher model. But in a digital model, getting the address of the north neighbor is done by subtracting 1 from its Y position. Doing this on our 00,00 node yields 00,FF. This will make the memory manager grab the data of an unintended real node and mess up the map's solution.

To fix this in our digital design, we modified the XY calculator inside the memory manager to look out for trying to access an out of bounds node. If this case happens the address for the out of bounds node is calculated as 0x10000. While all the inbounds data are stored from address 0x0000 to 0xFFFF, address position 0x10000 is written as an obstacle node. Now all out of bounds attempts will read and write the same unimportant address without interfering with the important data. Now, any out of bounds access is treated as accessing an obstacle node so the rest of the algorithm doesn't make any computations on this node.

Internal Structure

The internal structure of the memory manager is shown in Figure 4-9. The right part of the interface is how the Memory manager communicates with the Nodes manager. The 9 green busses are how they share information about the current node and its neighbors, whether the memory manager is sending the data to the nodes manager in the start of an iteration or the nodes manager sending the new G costs when they are available to be written into the memory. The Hit/Miss is a signal informing the memory manager what action should it take for this iteration. If the iteration was a hit, the shiftVector has 8 possible values each indicating the direction of the shift. The interaction between the nodes manager and memory manager is timed between the signals reqData and dataReady, as their name indicates one is for the nodes manager to request data and the other to inform the nodes manager that the requested data is ready to be read from the shared buses, in green. Lastly, writeData is a control signal to indicate which module has control of the data bus so conflicts don't occur.

The XY address calculator takes the current XY position and outputs to the finite state machine the addresses of all the 25 nodes so they can be used when needed in accessing the memory. It also handles the edge nodes and trying to access out of bounds nodes like discussed before.

The left part of the interface is between the memory manager and the memory. As discussed before in the previous section, the memory used is a dual port memory allowing the memory manager to read or write two nodes' data in the same cycle.

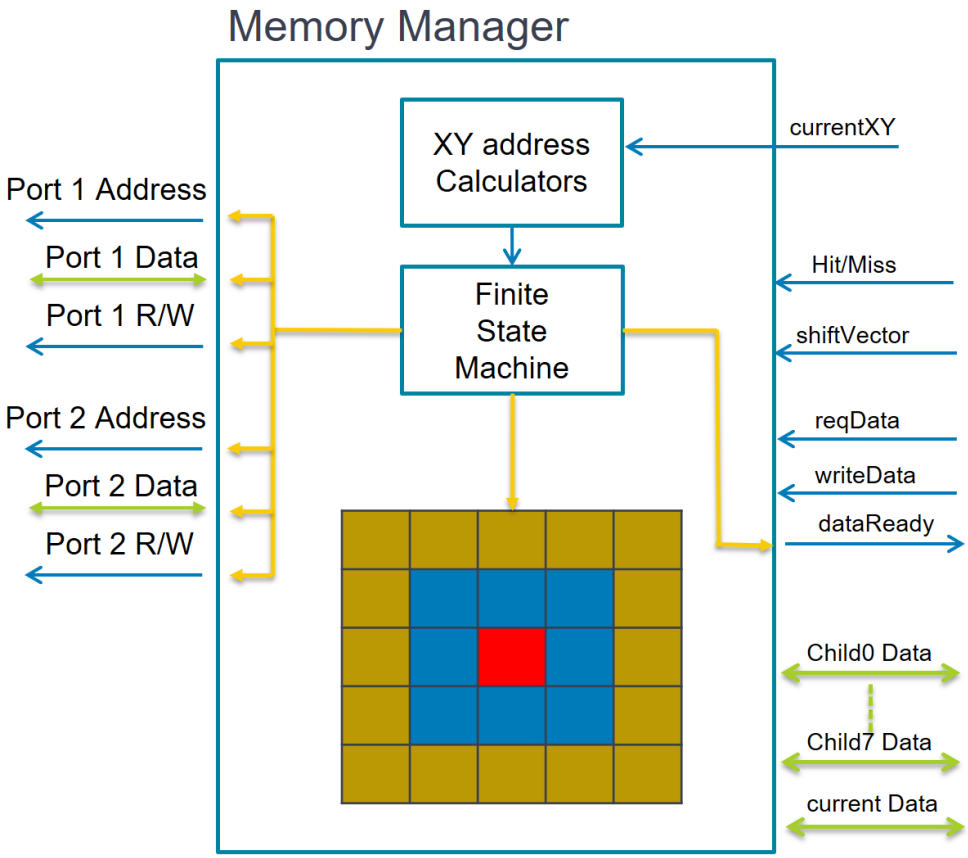


Figure 4-9: Memory Manager internal structure

Memory Access Cost

The whole purpose of designing this cache-like controller was to minimize the time the algorithms needs when it needs data for every iteration. In the case of a hit, the data is already present inside the memory manager so it's sent to the nodes manager immediately in the next cycle. For a miss, the memory manager needs to fetch 8 pieces of data resulting in a 4 cycle halt for the design. These margins allowed the design to not be bottlenecked by the memory side. Also, the memory manager allowed the whole algorithm to be abstracted from the memory with all the addresses, edge and out of bounds hassle. From the algorithm's point of view, it just requests that it needs data for its current node and gets all the data it needs either instantly or after 4 cycle

Evaluators

This block is responsible for calculating G cost, H cost, and hence calculating F cost by evaluating their sum, as described in equations in chapter 3. As for every parent node, we are calculating 8 child nodes, so we decided to use 8 parallel evaluators to calculate costs for every child (direction). In figure 4-10 we can see orange dashed lines which indicate that there is a register within this path in order to break the critical paths here. The use of these registers result in more cycles which are calculated and taken care of while designing and deciding on the number of cycles the memory manager needs for its normal operation.

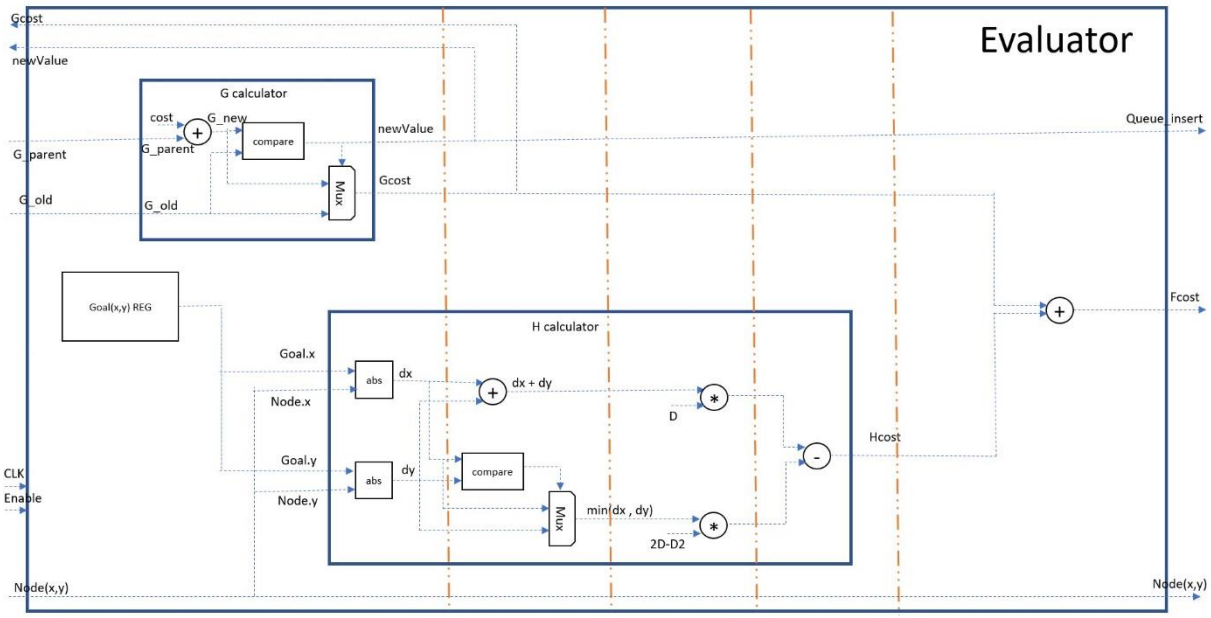


Figure 4-10: Evaluator Internal Structure

Each evaluator takes 5 inputs which are:

1. NodeXY: child node's coordinates (x,y).
2. G_parent: parent's G cost.
3. G_old: child node's G cost which is calculated in an old iteration for a different parent.
4. Enable: for pipelining registers.
5. CLK: for pipelining registers.

Evaluator generates 5 outputs which are:

1. NodeXY: child node's coordinates (x,y) propagated from input through registers.
2. Queue_insert: enable signal for queue block.
3. Fcost: the calculated F cost which is an input to the queue block.
4. Gcost: the calculated G cost that will be stored in memory.
5. newValue: control signal that indicates which has a smaller value, the old G cost or the newly calculated one.

G calculator:

This block calculates G cost for the child node as shown in figure 4-11, so first of all we are adding the movement cost to the parent's original G cost as G cost is an accumulative cost. Movement cost is 5 if we are moving straight, or 7 if we are moving diagonally as mentioned in chapter 3. If the new G cost, which is just calculated, is smaller than the old G cost calculated in a previous iteration for the same child node, then the output will be the new G cost and the newValue signal will be 1. But if the new G cost is larger then, the old G cost will be the output and the newValue signal will be 0.

G calculator takes just one cycle. newValue signal is then will be an input to nodes manager to manage memory insertion and also an input to queue as queue_insert signal to prevent multiple insertions for the same value to the same child node.

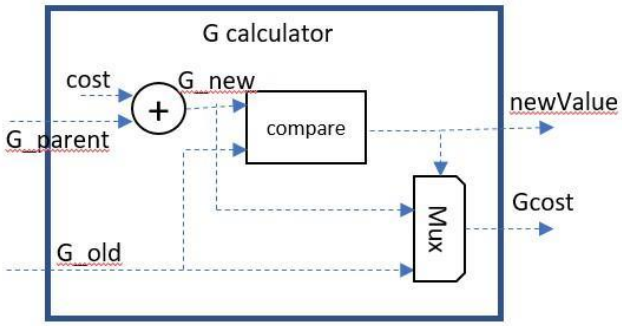


Figure 4-11: G calculator Design

H calculator:

This block calculates H cost as shown in figure 4-12 below according to these equations, which describes the diagonal heuristic function:

$$dx = abs(child\_node.x - goal\_node.x)$$

$$dy = abs(child\_node.y - goal\_node.y)$$

$$H = D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)$$

Where D = 5, D2 = 7, so $(D2 - 2 * D) = -3$

So, $H = 5 * (dx + dy) - 3 * min(dx, dy)$

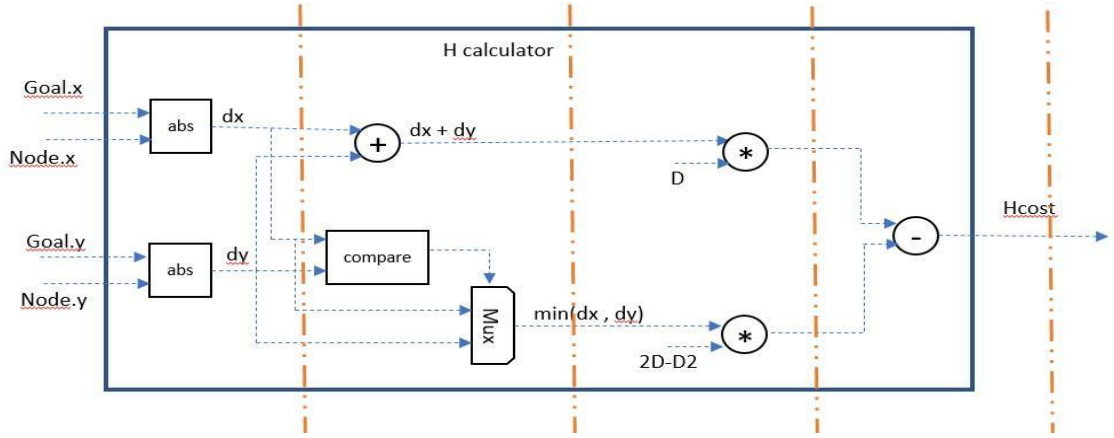


Figure 4-12: H calculator Design

H calculator takes 4 cycles in parallel with G calculator which takes only 1 cycle so outputs from G calculator propagate through pipeline registers. In figure 4-12 there is an abs block that calculate the absolute value. Abs block takes 2 arguments and inverts the Ssecond argument to get ones' complement then adds the two arguments (S=A-B-1). If the final bit from the addition result is one, then the result is negative so invert it again to get the positive value of it (B-A), but if the final bit in the addition result is zero, then the result is already positive so just add one (A-B). Abs block is shown in figure 4-13.

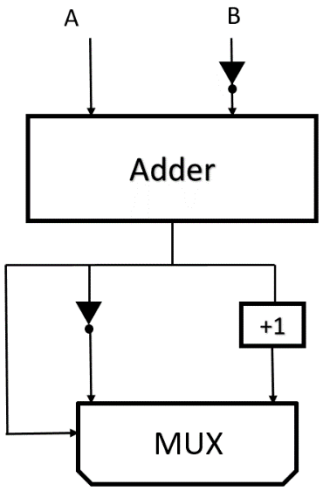


Figure 4-13: Absolute Block

F calculator:

This is the final stage in the evaluator where G cost and H cost are calculated after four cycles and in the fifth cycle, we add them to generate F cost as shown in figure 4-10.

Queues

In our chosen algorithm, after calculating the cost of eight children they are inserted in an open list. The open list is a sorted queue, which sorts the new calculations as the least cost node from all previous iterations will be at the top to be used in the next iteration as the parent node. This sorting operation is creating a bottleneck in software. We tried to overcome this bottleneck by choosing the correct based design and the correct number of needed queues.

Shift Register Based Design

To make a decision on our design, we compared different architectures of building blocks of sorting queues. According to results in [4], the shift register soring queues has the best performance for short length queues. By increasing the length of a single shift register based queue the bus loading problem becomes dominant and degrade its performance roughly. The bus loading problem is that the input faces a high fanout because it's the same input for all blocks. According to our chosen length as we will discuss later the shift register based queue is the most suitable architecture for our design considering low utilization area and best performance.

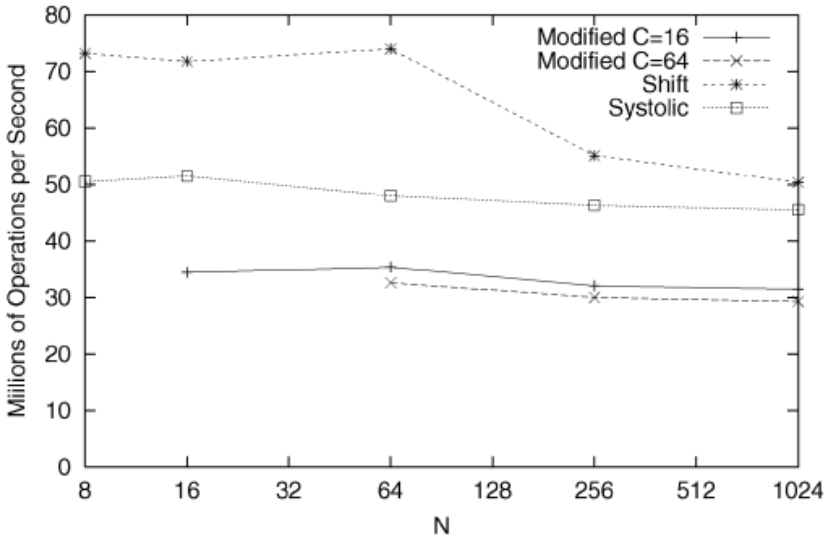


Figure 4-14: Queue architectures comparison

Block Theory of Operation

This block is designed to order set of elements in descending order in order to the least cost node will be available for the design to be used at the top of the queue. As we discussed before our design is shift register based queue. As the input is seen by all blocks and each block compare the input value with the stored value to decide the correct order of the input value.

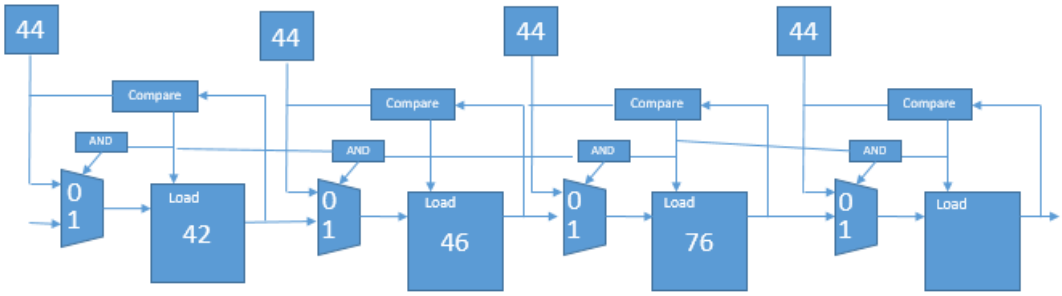


Figure 4-15: Input is same for all blocks.

To understand the theory of operation, consider the example in Figure 4-15. The result in the next cycle will be that the input will be in its correct order and all the higher values will be shifter to lower priority order as shown in Figure 4-16.

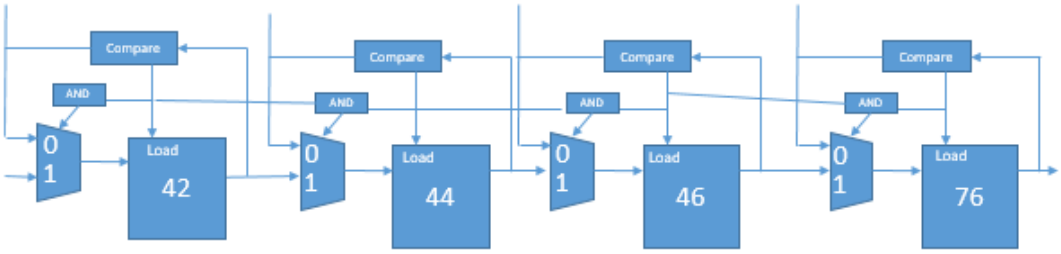


Figure 4-16: Input is being placed in its correct order.

The AND gates between each pair of blocks to get the correct decision of the block by being aware of this block should store the input value or store the value in the higher priority block. This helped us to eliminate an expected critical path of a chain of AND gates between all the blocks across the whole queue. By getting the benefit of that in each new entry to the queue, the queue must be already has sorted elements, therefore we could depend on each two pair of blocks to take the decision correctly.

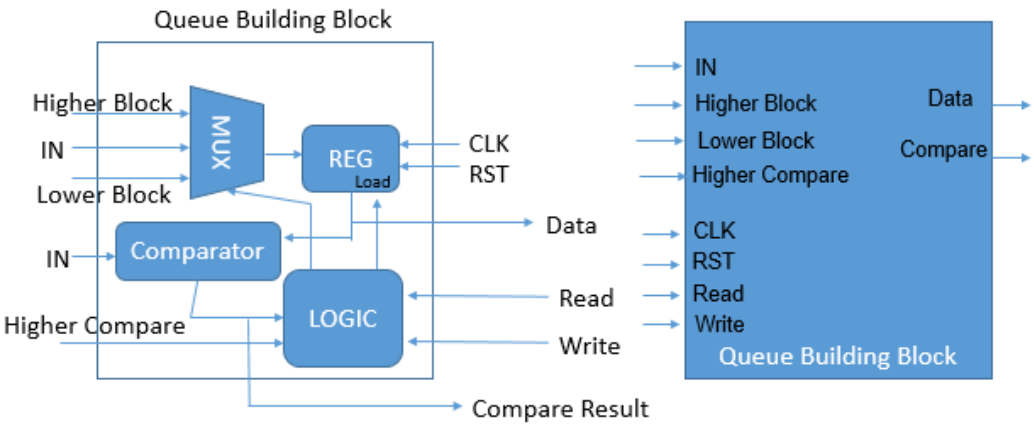


Figure 4-17: Queue Building Block Diagram.

The truth table for the building block in the table below. Each block depends on its own result of comparison and the higher block's comparison result to decide the correct data to be stored.

Table 4-1: Building block truth table.

Write Enable	Read Signal	Compare Result	Higher Comapre	Data
0	0	X	X	Data
1	0	Less than the input	X	Data
1	0	Greater than or equal to the input	Less than the input	Input Data
1	0	Greater than or equal to the input	Greater than or equal to the input	Higher Block Data
0	1	X	X	Lower Block Data

The queue block contains multiple building blocks connected to each other. They exchange values between them to make each value in its correct order. The top building block's stored data is the output of the queue block to the comparator engine as we will discuss later. The bottom building block's stored data is been discarded if the queue overflowed, that's why the decision of the size of the queue is very important to have a correct functionality and avoid discard important data.

Queue Size

In the software there is no limitation on the size of the open list, but this is unfeasible for hardware. Therefore, we need to decide the most suitable size of the queues in terms of low utilization, clock frequency, parallelism degree and the accuracy of the algorithm. In order to decide the correct length, we swept on different lengths for ten thousands map for each different probability of obstacle. The comparison key was the accuracy, which is defined as the algorithm still could find a path with the same G cost when we are using infinite size queue the same as the software. The sweeping had done for two cases: we have four queues in the design or we have 8 queues in the design. The two cases had been tested to decide the most suitable rank of parallelism in terms of the clock frequency, area and the number of cycles in the design.

In order to understand the behavior of the algorithm with different queue size, let's consider the case of the probability of obstacle is 0.1. We could notice that the accuracy is almost the same for long lengths, then it decreases sharply at some point. It's also noticeable that the accuracy decreases faster in the case of 8 queues.

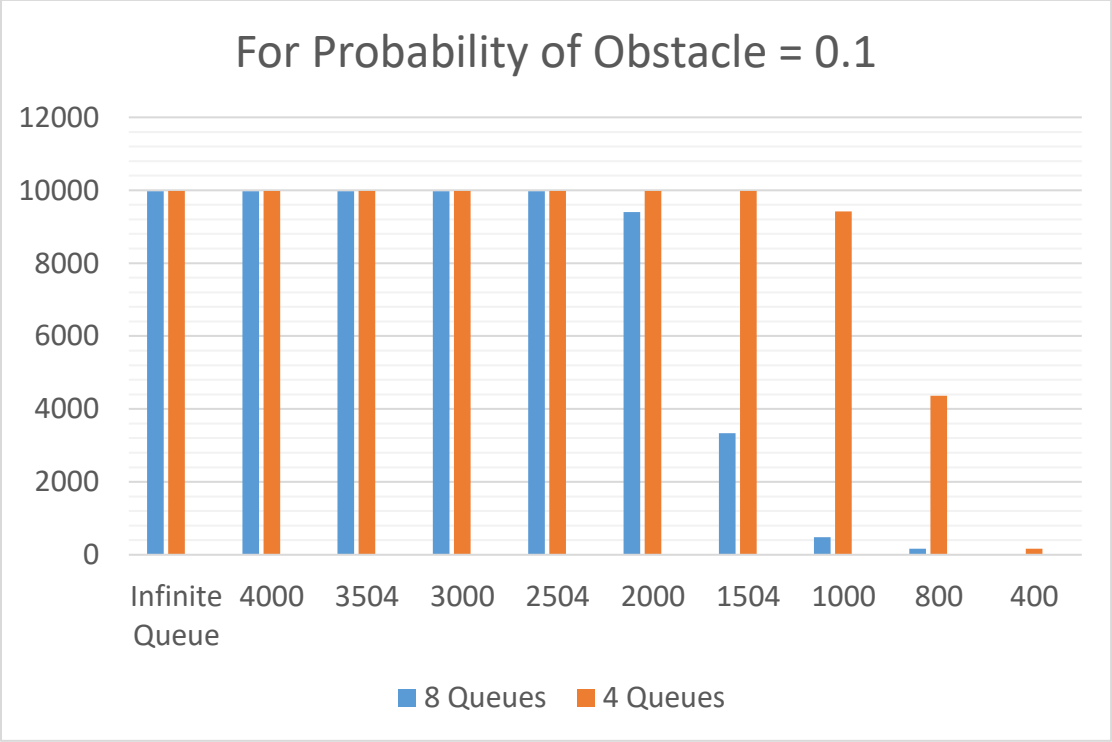


Figure 4-18: Results of sweeping at probability of obstacle is 0.1.

But is there is a feasible size have 100% accuracy? To answer this question let's consider the case of the probability of obstacle is 0.2 with 8 queues. In this case, the accuracy is 100% only with the infinite Queue. This tells us an important fact that limiting the queue size make it impossible to achieve 100% accuracy.

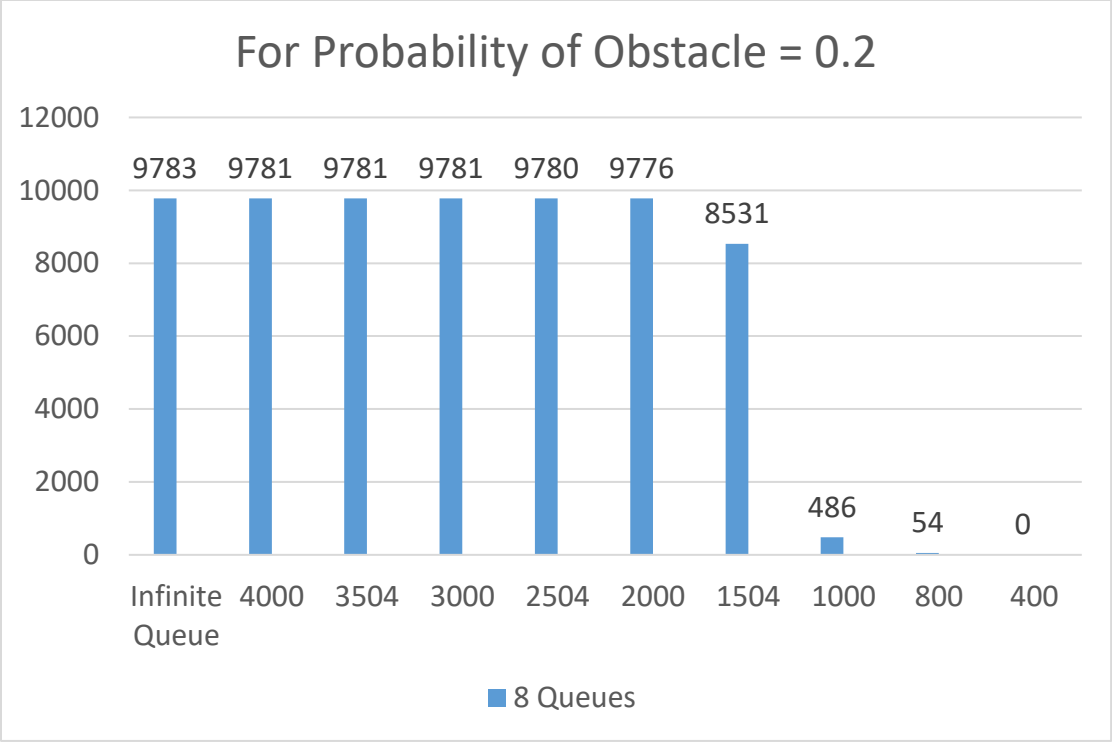


Figure 4-19: Sweeping results with 8 queues at probability of obstacle is 0.2.

This behavior repeats itself with 8 queues and 4 queues. Therefore, our target now to choose the minimum acceptable queue size in terms of accepted accuracy, clock frequency and area.

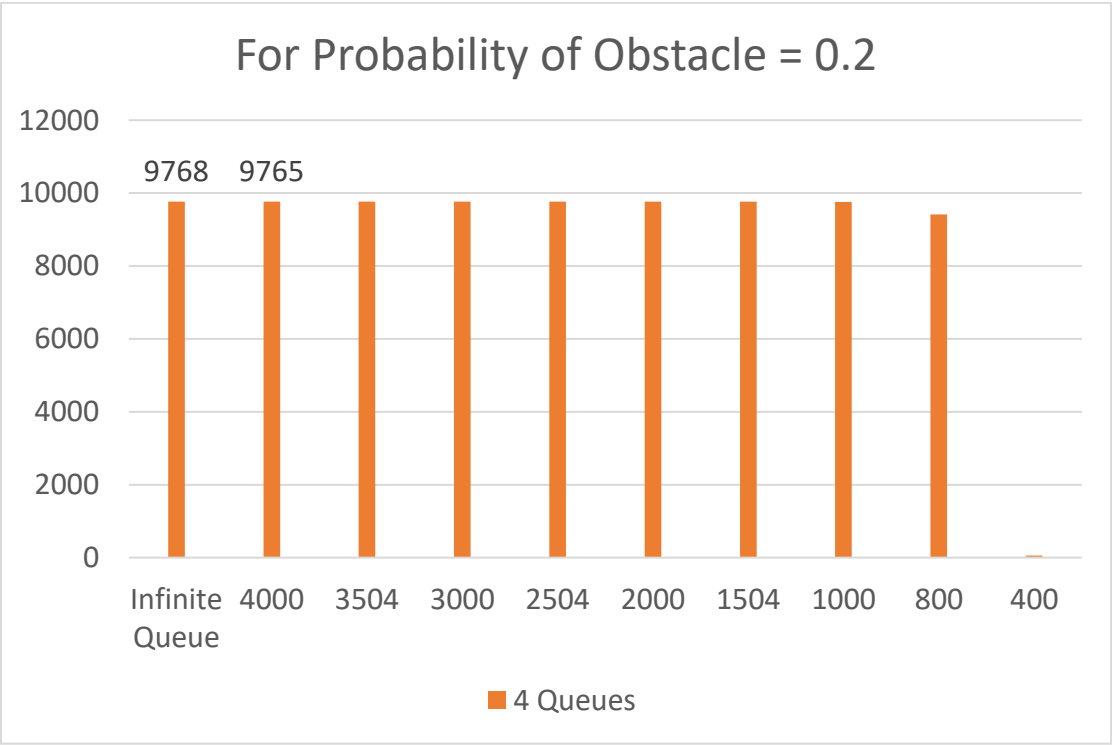


Figure 4-20: Sweeping results with 4 queues at probability of obstacle is 0.2.

It's very important here for understanding the final comparison to mention that the sweeping on queue size had been done on the total size of the eight or four queues not the size of the single queue. The below table contains the final results and describes the minimum size for each probability of obstacle and its accuracy and the fail probability.

Table 4-2: Sweeping Results.

P(obstacle)	4 Queues			8 Queues		
	Minimum Size	Accuracy	Fail	Minimum Size	Accuracy	Fail
0.1	1504	100%	0%	2504	99.99%	0%
0.2	1000	99.8%	0%	2000	99.93%	0.01%
0.3	800	97.11%	0.13%	1504	97.93%	0.044%
0.4	2000	98.8%	0.11%	2000	97.3%	0.126%
0.5	1504	99.86%	0%	1504	99.65%	0.08%

It's also very important to understand the concept of accuracy and fail. The accuracy depends on that the algorithm can find the shortest path. But, if the algorithm couldn't find the shortest path, it doesn't necessarily mean that it wouldn't find path at all but it could find a longer path or if the queue size is very small it wouldn't be able to find the path at all and this is the fail probability. We defined the fail the probability as it is the probability that the algorithm will not be able to find a path at all not even a longer path for a certain map, which has a path between the start and the goal, due to limitation of the queue size.

The final decision had been made according to a comparison between the minimum size in the both cases. The minimum acceptable size in each case had been chosen to be the maximum number between the minimum sizes for the different probabilities of obstacle. For the 8 queues case, the minimum size is 2504, which means that every single queue will have a length of 313 block. For the 4 queues case, the minimum size is 2000 with 500 block in each single queue.

Table 4-3: Final comparison between 4 and 8 queues.

4 Queues with total size is 2000 blocks		8 Queues with total size is 2504 blocks	
Accuracy	**Fail**	**Accuracy**	**Fail**
99.4%	**0.05%**	**99.6%**	**0.02%**

To have final deep sights, the above table contains the accuracy and the fail probability for the two chosen sizes *out of all the swept maps*. It's obvious that the 4 queues case is better for area, but it will have more blocks in the single queue and this will make the effect of the bus loading problem more dominant as we discussed before. Our final decision was to use 8 queues to avoid low clock frequency and reduce number of cycles in the design due to pipelining if we had used the 4 queues.

Routing Optimization

As we have a high number of building blocks in each queue, we thought that the global routing for the RST signal will be very difficult. And in order to reduce routing phase run time and decrease the effort required from the tool to be easier to reach higher frequency. We decided to disconnect the RST signal from all blocks except the top block of each queue. This modification could have a fatal effect on the functionality of the block as now we will have old value exists in the queue out of order. To avoid any conflict, we changed the logic truth table inside the building block to be as seen in table below.

Table 4-4: Truth table after routing optimization.

Write Enable	Read Signal	Compare Result	Higher Compare	Data
0	0	X	X	Data
1	0	Greater than or equal to the input	Less than the input	Input Data
1	0	X	Greater than or equal to the input	Higher Block Data
0	1	X	X	Lower Block Data

After this modification the decision of the stored data became more dependent on the higher block's comparison result rather than its own comparison result. As the block may be storing an old value from old map and it could be out of order. This modification helped us to reduce the routing phase run time by significant value, which helps us to do more runs and reach higher frequency.

Comparator Engine

As discussed in the previous sections, each iteration requires to expand new nodes and to be added to the algorithm's open list represented by Queues in the hardware architecture design. At the end of each iteration, it is required to choose a new node to be the current node to the algorithm and this is done by choosing the least node that satisfies the condition to be the node having the least total cost calculated as discussed in chapter 3.

Since the repeated calculations are done using parallelism and for each node there are eight directions to expand new nodes. Therefore, the algorithm is divided into eight parallel operations, including cost calculation, insertion and sorting the expanded nodes to the open list as shown in figure 4-21.

Comparator engine block is designed as a solution to extract a new node that has the least total cost from the open list in only one cycle. Even if the used parallelism divides the open list into eight parallel queues. This makes the performance the most important design parameter in the design specs rather than area or power.

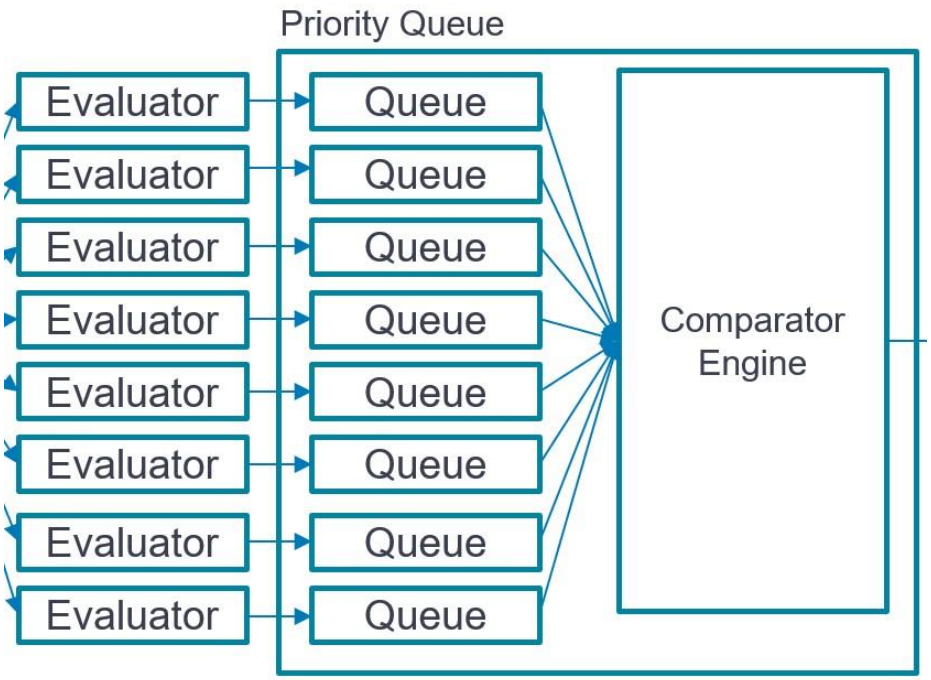


Figure 4-21: parallelized calculations and open list design

After inserting the expanded nodes in the queues, sorting is done to achieve a top block having a node with the least total cost as discussed in chapter 3. The comparator engine needs to read the top block of the eight queues and compare their costs to extract a new node having the least total cost as expected from A* algorithm.

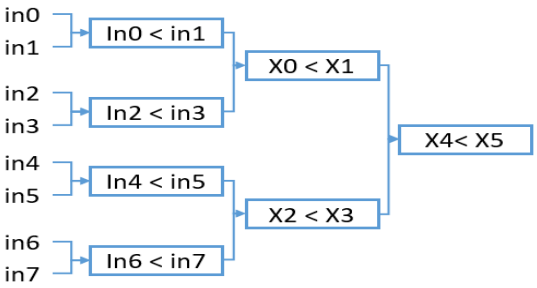


Figure 4-22: Conventional comparator

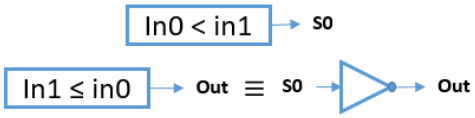


Figure 4-23: Equality optimization

Comparator engine takes the costs of each top to compare them and takes the address of the nodes in the grid map represented by (x,y) to easily out the new node to the algorithm.

The conventional comparator is a basic arithmetic unit that compares the magnitude of binary numbers and for our case the comparison mainly focuses on smaller than operations as shown in the figure 4-22 This technique does not give the best performance as it contains three phases of comparison, also the algorithm needs to extract the input with the least cost not only the information of which number is the smallest. Now it is obvious that the best performance comparator that detects the smallest input in only one phase of comparison and this could be achieved by parallelism of comparison operations and each input is compared to all other inputs in parallel. The figure 4-24 shown illustrates four input comparison operations done in parallel to get the smallest one in only one comparison phase but with a large area compared to the conventional comparator. A priority technique is used to solve the equal inputs case by using smaller than or equal condition for inputs at the bottom of the design. In addition, this technique provides output signals, which easily detects the smallest input and could be used as selection signals to a mux to out the smallest input immediately. This technique is used for A* algorithm case with eight cost inputs compared and use eight node addresses as inputs to out the address having the smallest cost.

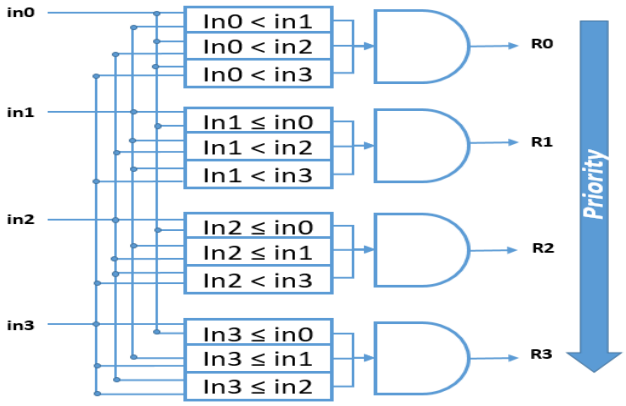


Figure 4-24: parallelized comparison operations

The area plays a main role in parallel comparison operation as it increases gradually by increasing the number of inputs and their bit width. A published paper [5] has provided an optimization solution in area by removing each block with a condition smaller than or equal and replacing them with inverters to signals that provide the opposite condition of the removed block. As illustrated in figure 4-23 the equality is done by inverting the internal signal that shows that the first input is smaller than the second one and this maintains the same condition that the second input is smaller than or equal the first one. By applying this optimization about half of the comparison blocks are replaced by inverters, which is a great solution that provides very high performance with area proportional to the conventional comparator.

Path Extractor

The A* path searching process comes to an end when the current node is the goal node that was specified from the very first moment of starting the process. By then, every node in the memory is either a non-visited node (may be non-visited as the algorithm found it's not useful to visit or it's an obstacle node by nature) or an already visited node. Some of the visited nodes are the ones that were chosen as path nodes, but how can we extract these nodes from the memory that holds the map?

Figure 4-25: Node's Data Structure

According to figure… each entry in the memory that holds all map nodes stores a data structure that represent some useful pieces of information. The most significant bit is '1' when this node is already visited during the path searching process or it was an obstacle node from first.

The next 18 bits represent the G cost of these nodes, or which we can also describe by the accumulative cost to reach this node from the start node until this node.

The least significant 16 bits represent the coordinates the best node that can be a parent to this node, to explain this we can say that every node in a grid map that is divided into squares will have 8 squared nodes around it, and since each node have some G cost which is accumulative, then for any node to have least cost then we must reach it from the one node of the 8 nodes around it which has the least cost.

During the G cost evaluation process that is executed by the evaluator module which is explained in this chapter section 5, and the process of nodes manager module which is explained in section 2, we have learnt that with each iteration, the current node is updated in the memory, where the G cost is updated by the least G cost it can have and the best parent (the one node of the eight nodes around it that has the least G cost).

Keeping in mind all the above mentioned information about the data structure stored in memory for each node, we can back trace the path nodes from the goal node to the start node easily.

Back tracing the path nodes

When the goal is reached the path can be traced by reading from the memory the address of the parent node (the least significant 16 bits of the goal node) , then we can read the parent address 3of that node again, then repeat this process again and again until the parent of the current node is the start node.

Knowing how to extract these nodes, we can extract them and store them in a small memory which can be used to export these nodes when requested. This memory is what going to be explained in the following subsection.

Path memory (LIFO)

choices for implementing the memory that holds the path extracted from the main memory were wide, but the most suitable implementation was a queue which is either a FIFO memory (first in first out) or LIFO memory (first in first out), when output is requested then the path nodes will be ready and out in turns with each clock cycle.

The memory chosen for this purpose was a stack whose operation can be described as LIFO (last in first out), since the path is back traced then the last node that will be stored in this memory will be the start node, so this type of memories was the most suitable where reading the path from it will be each node in turn starting by the start node, ending by the goal node.

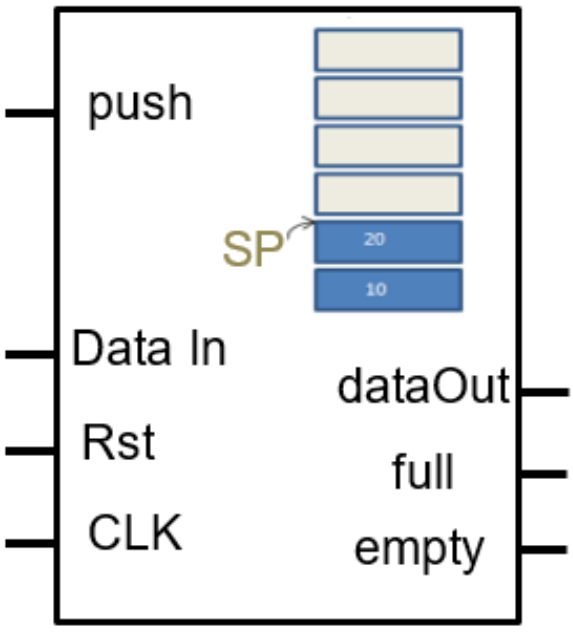


Figure 4-26: LIFO's structure

This module have some input signals and some output flags which are:

- Push: when high, data on the input bus “Data In” is stored on the top of the stack.
- Rst: reset signal which is responsible for resetting of the memory and signals or flags.
- CLK: input clock signal
- DataIn/DataOut: input and output data busses.
- Full: a flag which is high when all memory entries are already filled with data, when this flag is high, input data is discarded.
- Empty: a flag which is high when all the memory entries are empty, when it’s high then we can’t read from this memory as there’s no data to be read.

Full/Empty calculation

The previously mentioned full or empty flags are dependent on a pointer which is called SP (stack pointer). This pointer is initially pointing to the top of the memory (the maximum address) and the empty flag is high at this case. SP is decremented with every push operation and incremented with every pop operation from the stack.

After the path has been found and extracted by the module that called path extractor, then the path memory will be holding the path nodes and as explained, empty and full flags will be low by then, until all nodes are popped from this memory where empty flag would be high by then.

CHAPTER 05 Visited Lookup

In this chapter, we discuss an optimization we had done to the high-level model of the algorithm. This optimization aims to cancel redundant iterations, which being done to a limitation in the software.

Problem definition

While sweeping on the queue size with the high-level model, we have noticed that the number of iterations had been done is much greater that the number of the visited nodes. This note means that in some cases we re-visit an already visited node, which will be a redundant iteration produces non-useful data for our flow.

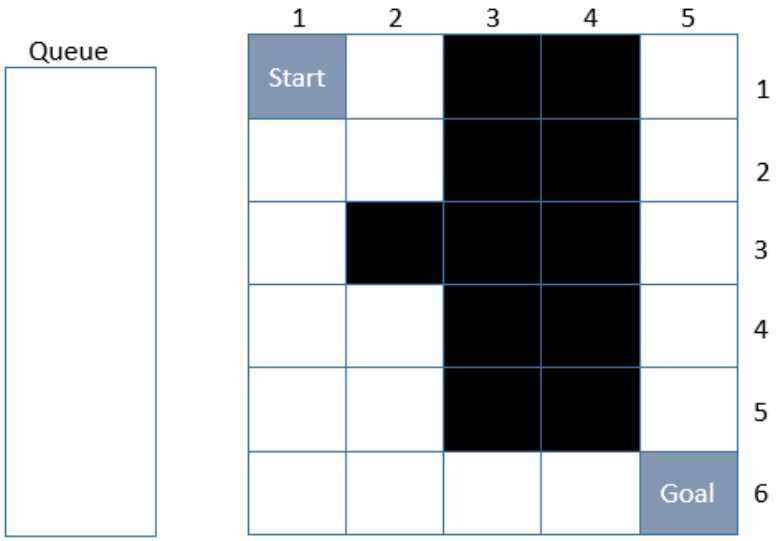


Figure 5-1: Problem Definition example map.

To understand this problem and how it occurs, let's consider this example map in Figure 5.1 and see how the algorithm will operate. Firstly, we will expand around the start node resulting in the values in Figure 5-2.

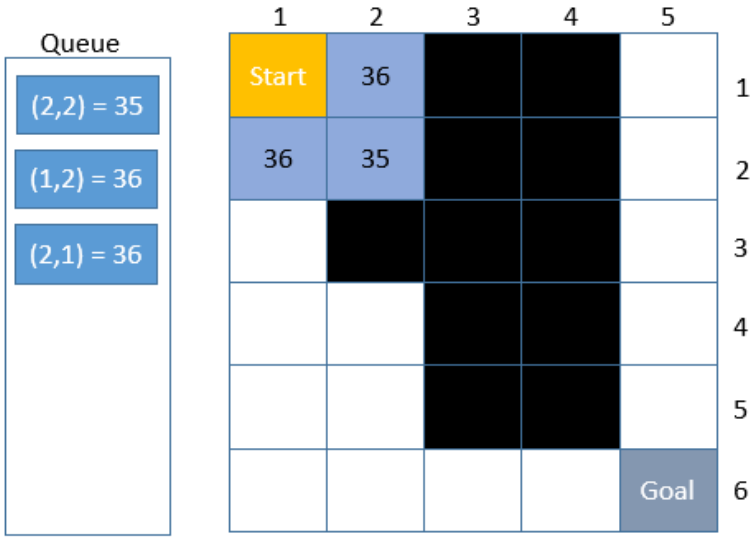


Figure 5-2: Expand Around the Start Node.

It's very important here to focus on the values in the queues and see how the algorithm behave.

Now, the algorithm will move to the node at the top of the queue with the least cost and expand around it and mark the start node as visited node. While expanding around the new node the start node has no need to be evaluated again as it's a visited node and this guarantees for us that we had reached it with the best path already. The obstacle nodes also will be dropped from our calculations as seen in Figure 5-3.

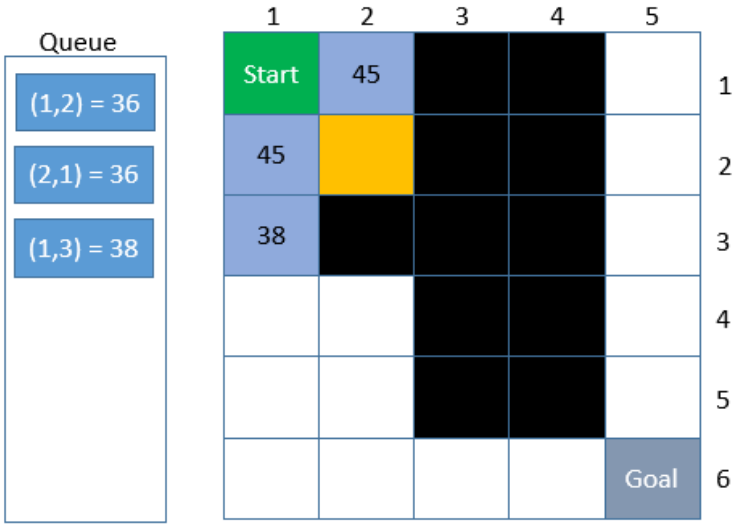


Figure 5-3: The visited node never been re-evaluated.

During this expansion, we will calculate a higher cost for some nodes and this a very important key in the algorithm that those higher costs will never be interested into the queue. As we have a lower cost in the queue for a node that means we have a shorter path for this node than the evaluated one currently. Therefore, there is no need to consider the longer path.

After evaluating and inserting the correct costs into the queue, the algorithm will move to expand around the node at the top of the queue as in Figure 5-4.

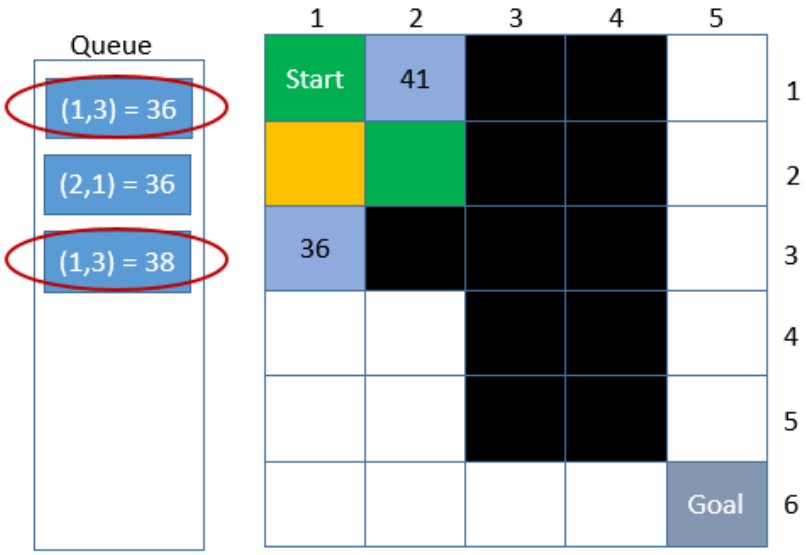


Figure 5-4: The problem source.

We could understand the reason behind the problem from this iteration. The reason is that in this iteration we will calculate a new lower cost for node already exists in the queue. As the new cost is lower than the one exists in the queue, we must consider it and insert the new value into the queue. This creates the situation where we have two values for the same node inside the queue, which will motivate re-visiting this node.

Now, the algorithm will move to the lower cost value of the node as it's the least cost within the queue lifting the other value inside it as seen in Figure 5-5.

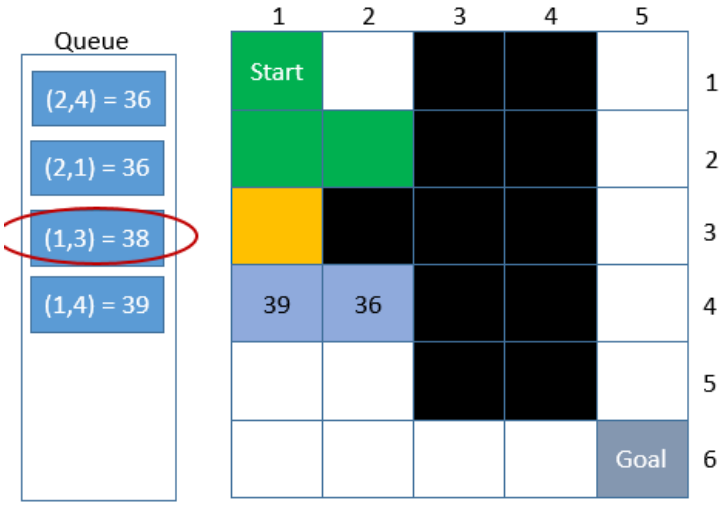


Figure 5-5: Expand around the problem node.

Moving forward steps in the algorithm flow, we will reach the iteration when the higher cost of the problem node be at the top of queue as seen in Figure 5-6.

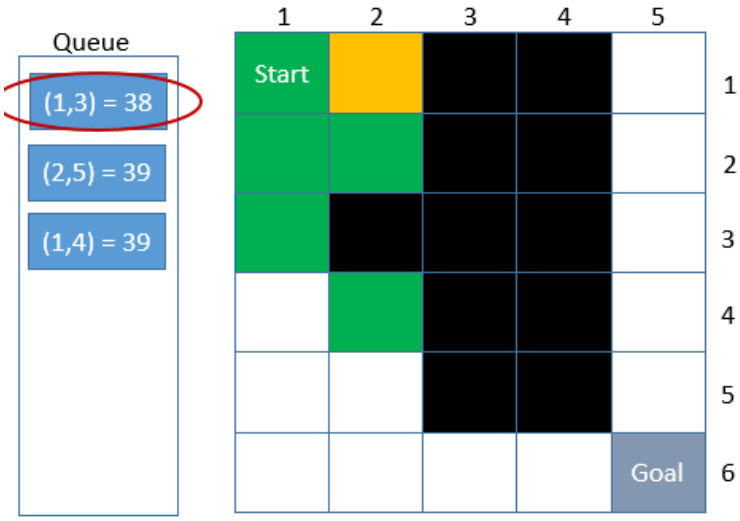


Figure 5-6: Expand around already visited node.

This will create redundant iteration will produce non useful data for the algorithm, therefore performing this iteration will be just a waste of time. As we already visited this node, which means that we had already found a better path for this node.

Problem Solution

To solve this problem and avoid wasting time on redundant iterations. We found that the best solution is to add a look-up table, which records the visited nodes across the map to guarantee that we will never re-visit a visited node.

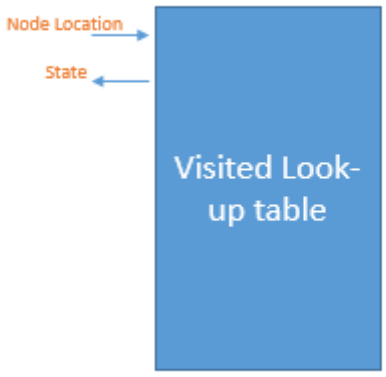


Figure 5-7: Visited look-up table.

We use the built-in BRAMs in the FPGA to build this look-up table. The nodes manager should check the state of the node before begin the iteration to be sure it's not visited and should change the state of the node into a visited node after begin the iteration.

To understand how this will affect the flow of the algorithm. When the algorithm reaches the state in Figure 5-6, the algorithm will firstly check the state of the node and it will found that this node is already visited. This will make our design drops this node and move to the next node in the queue as seen in Figure 5-7.

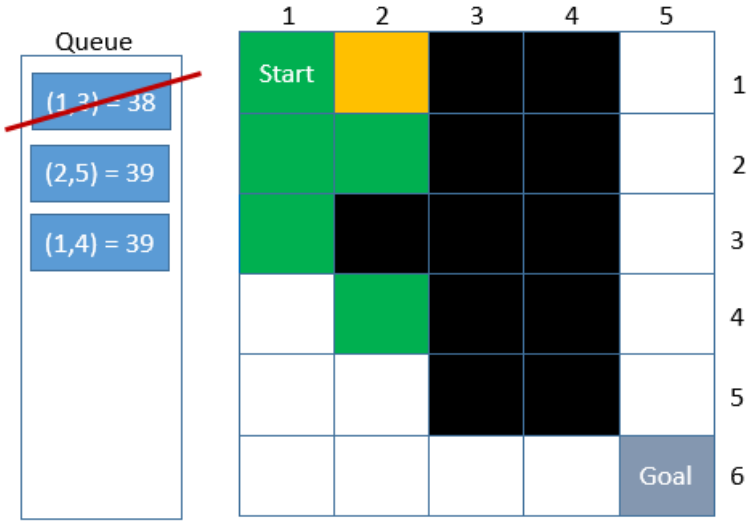


Figure 5-8: The modified algorithm flow.

Design Modifications

While applying this solution to optimize the number of cycles need by the algorithms to find the shortest path, it was necessary to do modifications to our design flow.

High Level Design Modifications

Firstly, we inserted the look up table in the top design as its address port is been connected to the output node for the comparator engine to overcome the BRAM clock latency and its output and input ports connected to the nodes manager to allow it to read the node state and change the node state into visited.

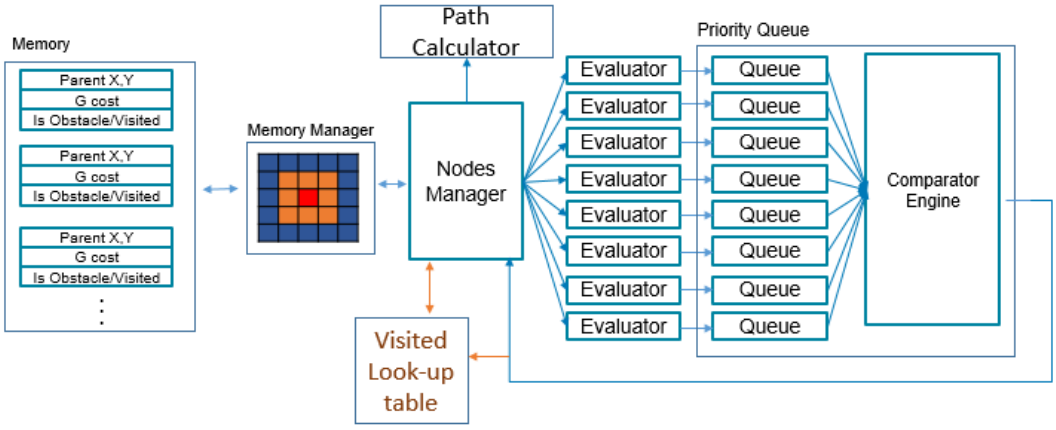


Figure 5-9: Top Deign after inserting the visited look-up table.

Nodes Manager Modifications

This new modification needs to be considered in the Nodes Manager's state discussed before in chapter 4 section 2. The modified state diagram is shown in Figure 5-10. Previously, when loading a new node, it was only checked for being the goal node or not. Now, the node is also checked if its entry in the visited lookup table is a '1' or a '0'. A '0' indicating that this is the first time the node is explored, so the algorithm continues normally and writes the entry as '1'. If this node is explored again later, the nodes manager will read its entry from the lookup table as '1' and will know that is the redundant iteration so all the calculations will be obsolete and a waste of time. In this case, the nodes manager just skips this iteration, requesting a new node to be loaded saving all the cycles that would have been wasted on the repeated one.

Due to the one cycle lag in the lookup table, the total cycles per iteration increased from 10 to 11. However, because redundant iteration happens quite often and every time it happens we now save 10 cycles it results in a decrease of the algorithm's time to solve the map.

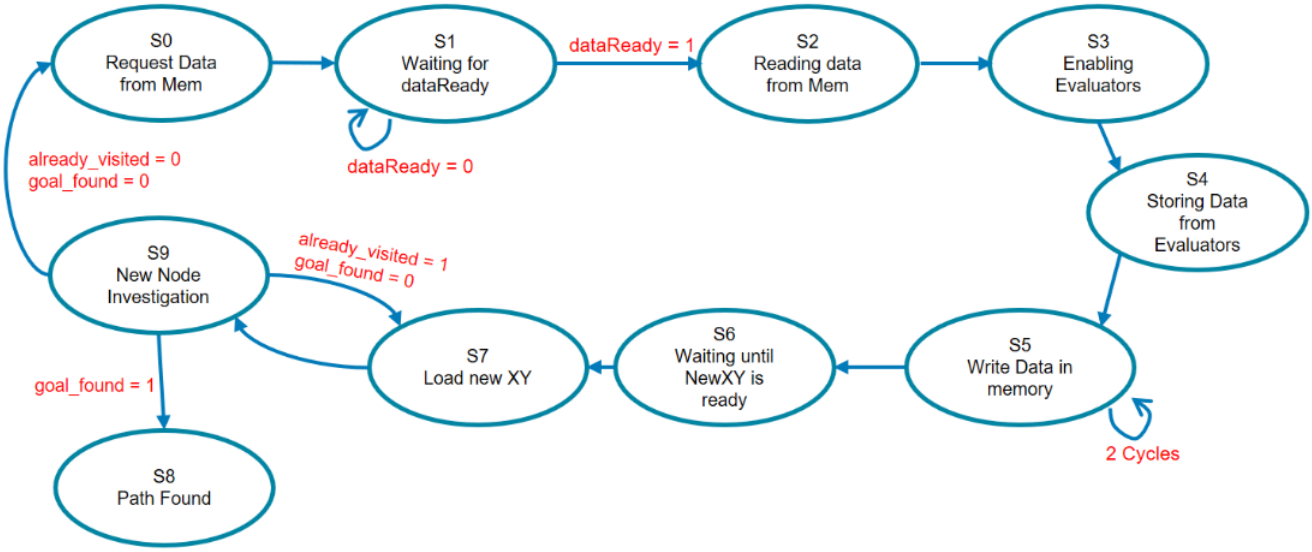


Figure 5-10: Nodes Manager State Diagram After Lookup Table Integration

Optimization Results

This optimization helps us to reduce the total number of cycles needed to find the shortest path. The total number of cycles has been reduced by almost 25% in all the probability of obstacles.

Table 5-1: Optimization results.

Probability of Obstacle	Before Optimization	After Optimization
0.1	52576.9	39580.14
0.2	106173.5	75862.16
0.3	158652.7	111244.5
0.4	206631.7	153106.8
0.5	262883.3	215587.5

CHAPTER

06 Formal Verification

As our design aims to find the shortest path for 256 * 256 map, we have a great number of possible maps and we can't validate all of them using simulation. Formal verification will help us to validate our design's behavior as we could guarantee that the design will operate correctly with any map. Formal verification tools take behaviors and tries to find a counterexample for each behavior searching for bugs. They don't need input stimulus and generate values on the free points using clever techniques to prove or disprove a specific behavior. We used the PropCheck, which is a Mentor Graphics's tool in our formal verification.

Assertions, Assumptions and Coverpoints

We validate our design using assertions and coverpoints. We used the assertions to prove that our design is bug free. We check on important behaviors in our design independently of the map. The coverpoints helped us to cover and check on interested corner cases in our design.

Queue

In the queue block, we checked on the reading, writing, overflow and underflow cases. We have checked that the writing operation is done correctly and the input node is ordered correctly inside the queue. The queue block has a reading operation in which the top value popped out of the queue and all internal values shifted up in only one cycle. We proved that this operation is always done correctly independent of the values of the nodes or the size of queue.

The overflow and underflow behaviors in our design is a very important cases in our design. When the overflow happens and we inserted number of nodes greater than the queue depth the down value should be discarded. In case of the underflow and the queue became empty, the top value should be the maximum value of the cost to avoid any conflict and eliminate this value by the comparator engine.

There is also a very interesting behavior we had covered. When inserting the same cost value as a cost of a node exists inside the queue. In this case, the last inserted node should have a higher priority inside the queue rather than the old node. This helps us to find the path faster. We checked on this behavior using the formal tool and proved it always will be done correctly

The most challenge in formal verification is to limit the state space of the design to avoid state space and memory explosion. As we have a large queue size in our design, we minimize the queue's depth to only 4 blocks to avoid memory explosion. We also used the assumption to define the fact that in our design the reading and writing are mutually exclusive.

Comparator engine

The properties written to describe the behavior of the comparator engine includes two types of properties. First, properties describe the functionality of the design. Second, properties describe the interface of the design.

The comparator engine assertions include properties that describe the functionality, mainly depending on taking inputs and out one of them, which has the smallest total cost so the properties check if any input is the smallest and the module output is the smallest one correctly. In addition, the assertions include properties that check the interface by checking any x propagation at the inputs from the queues and checking that the output signals have only one high signal.

The added cover points check the behavior that all the inputs could be the smallest and the module could correctly cover all these behaviors.

Nodes Manager

The nodes manager is the brain of our design. It controls the flow of our design and send all the control signals to all the blocks inside the design. We used assertions to check on that all the control signals always enabled correctly. We had checked on the enable to the evaluators depend on the node value from manager. We also checked on that the nodes manager enables the path calculator correctly after finding the goal. It was also important to check on the interfaces between the nodes manager and the memory manager to check that the nodes manager send correct control signals.

The visited look-up table is a memory built using FPGA's BRAMs and interfaces directly with nodes manager. All the memories should be black-boxed in formal verification. We used coverpoints to model the outputs from the visited look-up table for both cases if the node is not visited or already visited. And using assertions we checked on that on both cases the nodes manager behave correctly.

Memory manager

The properties written to describe the behavior of the Memory manager includes two types of properties. First, properties describe the functionality of the design. Second, properties describe the interface of the design.

The Memory manager assertions include properties that describe the functionality, mainly depending on checking the FSM states and that for all corner cases (hit, miss, and shift) the FSM enters the correct state. In addition, the assertions include properties that check the interface by checking any x propagation at the inputs from the main memory and checking that the data is ready after two cycles during the hit case and after seven cycles during miss case and properties check that the memory manager writes the new child data after reading the missing parts from memory to its register bank.

The added cover points check that all the corner cases behavior like miss and hit cases are coverable. Also all the shifting directions are reachable by the design.

Path Extractor

The properties written to describe the behavior of the Path calculator includes two types of properties. First, properties describe the functionality of the design. Second, properties describe the interface of the design.

The Path calculator assertions include properties that describe the functionality, mainly depends on checking the FSM states and that it enters the correct states defined in an enumeration and checks that all the states are followed by an expected state. In addition, the assertions include properties that check the interface by checking any x propagation at the inputs from the main memory and checking that the FSM writes in the LIFO in the correct states. After writing the entire path in the LIFO. A property is asserted to maintain the condition that the signal done is always high.

The added cover points check that all states of the design are reachable.

Top Design

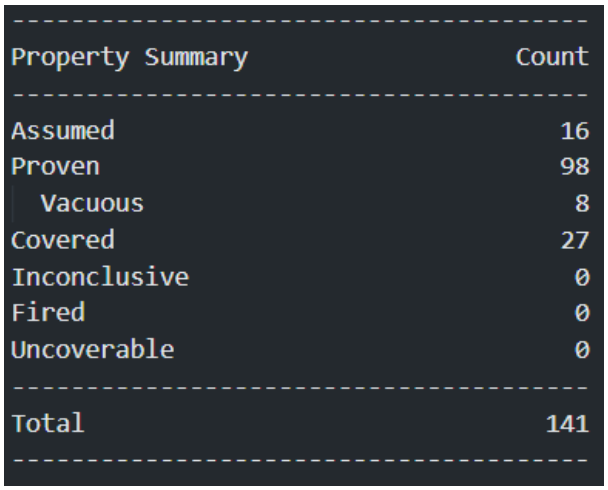
In the top design, we checked for any conflict between any two blocks and the flow of algorithm is always correct. The main memory ports in our design used using multiple blocks, therefore we checked that the correct module at specific time uses them correctly to be sure all the data written or read correctly from the memory.

We also checked on that the control signals flow correctly across the design. As the writing signals from the nodes manager to the queues through the evaluators stages. Also we made sure that the nodes manager reads from the correct queue at the correct time depending on the signals from the comparator engine. We checked on the overall flow of the design from reading the data of children from memory until nodes manager loads new node to begin a new iteration in a correct and exact sequence.

Summary

All the RTL assertions and cover points have been added to the modules together with the high-level assertions added to the top module to provide a great observability of the design by increasing the observers and checkpoints. Formal verification of A* algorithm helps in easily checking the required behaviors of each module and the entire algorithm without the need of any test bench saving a lot of time of creating input stimulus to check all the corner cases.

Propcheck tool report identifies the all used assertions, assumptions and cover points to be checked and the total proven, covered, fired and uncoverable properties as shown in figure 6-1.



Property Summary	Count
Assumed	16
Proven	98
Vacuous	8
Covered	27
Inconclusive	0
Fired	0
Uncoverable	0
Total	141

Figure 6-1: PropCheck property summary

Formal tools

We used Mentor Graphics's formal tools in our functional verification. That increases our insight design knowledge about the design corner cases and verification complexity and helps in decreasing the verification cycle.

- CodeCover: a dynamic verification tool that executes the design and test the covering state of the used test bench by checking that all the states, condition statements and code lines are reachable.
- AutoCheck: a static verification tool that checks many different design issues without executing the RTL.
- PropCheck: a static verification tool that helps in formal model checking verification by Mathematically analyze the space of possible behaviours of the design trying to find a counterexample using clever mathematical techniques.

CHAPTER 07 UVM

Universal Verification Methodology (UVM) is a standard to enable faster development and reuse of verification environments and verification IP (VIP) throughout the industry. The main idea behind UVM is to help companies develop modular, reusable, and scalable testbench structures by providing an API framework that can be deployed across multiple projects. UVM is mainly derived from Open Verification Methodology (OVM) and is supported by multiple EDA vendors like Synopsys, Cadence, and Mentor.

Testing environment:

Our testing environment consists of three main items as shown in figure 7-1 below:

1. Python script which generates a random map with random start and goal points. This script writes start and goal points in a text file that cpp code and UVM environment read it. Then script runs cpp code.
2. Cpp code which models the algorithm. Cpp code reads start and goal points from a text file generated from a python script and when it finishes the algorithm it outputs the Gcost in a text file and the path moved by the algorithm in another text file.
3. UVM Environment which reads start and goal points from a text file generated from a python script and reads Gcost and path from two text files generated by cpp code, then it passes start and goal point to the design and wait for the outputs then compare them with the expected ones coming from cpp code.

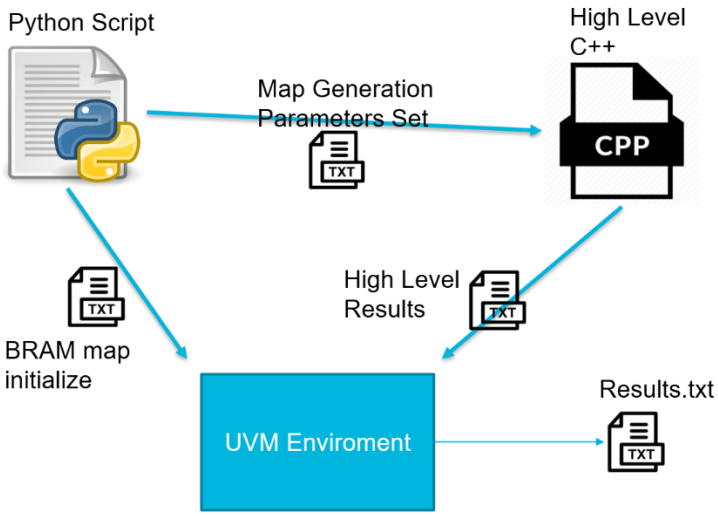


Figure 7-1: Testing Environment

UVM Environment:

Our UVM environment is shown in figure 7-2 below.

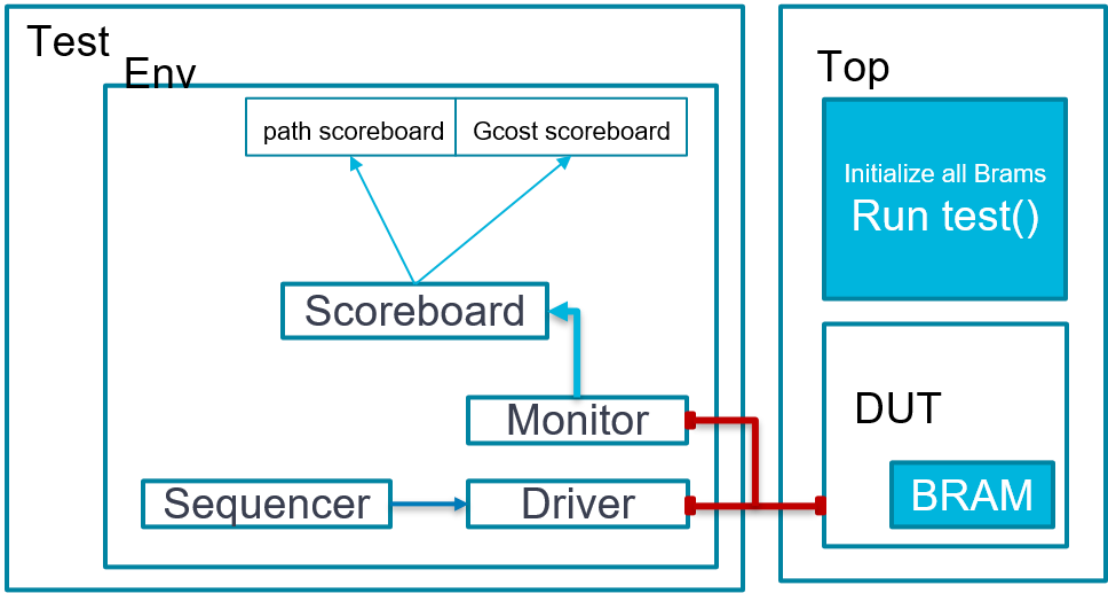


Figure 7-2: UVM Environment

First of all, we modeled the built-in BRAM IP in Xilinx to be able to initialize and write in them outside vivado. Our environment consists of a driver, sequencer, monitor and two scoreboards. We will explain every one of them in the next subsections.

Sequencer:

A sequencer generates data transactions as class objects and send them to the driver for execution. Driver and sequencer are connected together through TLM port by default. Our sequencer takes a sequence that consists of start and goal points and proper input signals for normal operation.

Driver:

A driver is an active entity that has knowledge on how to drive signals to a particular interface of the design. Transaction level objects are obtained from the sequencer and the UVM driver drives them to the design via an interface handle.

Monitor:

A UVM monitor is responsible for capturing signal activity from the design interface and translate it into transaction level data objects that can be sent to other components. Our monitor is capturing G cost and sending it to a scoreboard, which checks the correctness of it, through a TLM analysis port. Then it is capturing the path coming from the design

node by node every cycle and sending it to another scoreboard through another TLM analysis port.

Scoreboard:

A UVM scoreboard is a verification component that contains checkers and verifies the functionality of a design. It usually receives transaction-level objects captured from the interface of a DUT via TLM analysis port. Scoreboard needs a reference model to compare results from the design with it, in our case the reference model is a cpp code that generates its results in text files. We have two scoreboards, one for G cost that receives the design's G cost from a TLM analysis port with monitor and reads the G cost that is coming from the cpp code from a text file, then it compares between them.

The second scoreboard checks the correctness of the path generated by the design with the one generated from the cpp code and stored in a text file. It works exactly the same as the first scoreboard instead it executes every cycle until the path ends. There is a tweak here that different paths can generate the same G cost, thus all these paths are correct, so the design path and the cpp code path could differ and still we have correct results, so this check is not as important as G cost check.

UVM test automation

As explained in figure 7-1, our UVM test can take random or fixed nodes as start and goal nodes and then after these data is derived on the design and the design is done calculating the path, the scoreboard reads the path and the final G cost of the path generated by the C++ high level model and cmopares it against the path and final F cost generated by the design under test.

This test was automated to run a large number of random maps for fixed starting point at (0, 0) and fixed goal node at (255, 255) for all maps, as this configuration is the worst case for these random generated maps.

Design and high-level model results were identical for all the maps generated.

UVM results

We used our UVM environment to give us results about the performance of our design to make a comparison with a paper published last year [5]. The comparison was done for the worst cases in five sets of the probability of obstacles that are 10%, 20%, 30%, 40%, and 50% as shown in table 7-1 below:

Table 7-1 Design timing results

Probability of Obstacles	our Design Time (ms)	Paper design time (MS)
10%	0.198	1.059
20%	0.379	1.087
30%	0.556	1.160
40%	0.765	1.144
50%	1.078	1.088

As we can see in table 7-1 that our design has a better performance in all cases except for a 50% probability of obstacle case that is almost equal to the paper's result.

CHAPTER 08 FPGA Implementation

We were targeting Xilinx Virtex-7 FPGA for implementing our design on it. We had a problem with the default floorplanning, strategies, and fan-out limit at the beginning as it results in a bad performance and a huge negative slack. So to overcome these problems and improve the performance we decided to limit the fan-out, change the default strategies, and change the floorplanning that was done automatically by the tool (vivado) and do it manually as much as possible. We will talk about these changes and their results in the next couple of sections.

Floorplanning:

Floorplanning automatically done by the tool (vivado) was very bad as nets were far from each other although they are connected. We can see in figure 8-1 an example of that bad floorplanning.

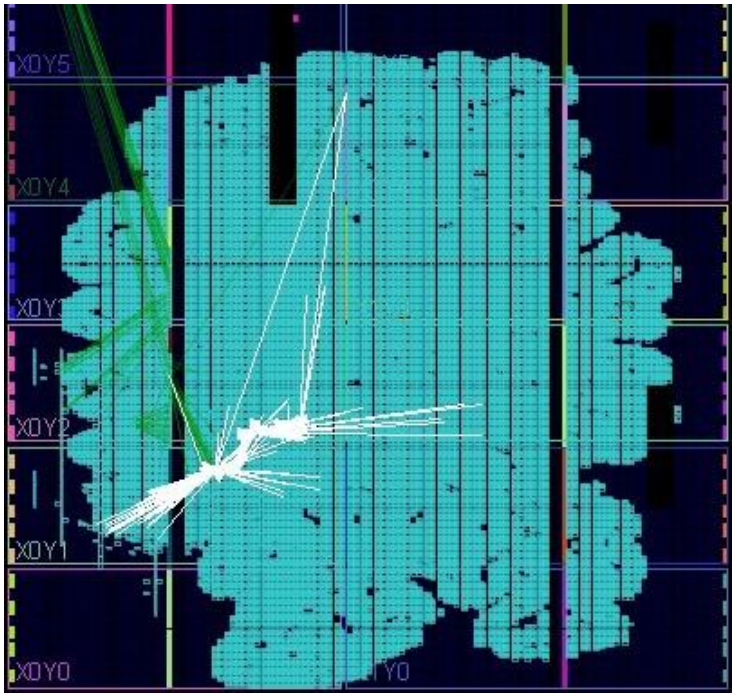


Figure 8-1: Bad Floorplanning

We solved this problem manually as we assigned a Pblock for every block and put it near the Pblocks that connected to it. For example, every evaluator writes in a single queue every time, so we can put them together in a single Pblock as shown in figure 8-2. If we did that, we will have 8 Pblocks that are connected with comparator engine as discussed before. every Pblock is double the area of the block in it to make routing easier.

In the same way, we put the nodes manager, memory manager, and the main memory together in the same Pblock and the comparison engine with the register holding the current node data together in the same Pblock. This manual floorplanning is then put in the constraints to make the tool restricted with it.

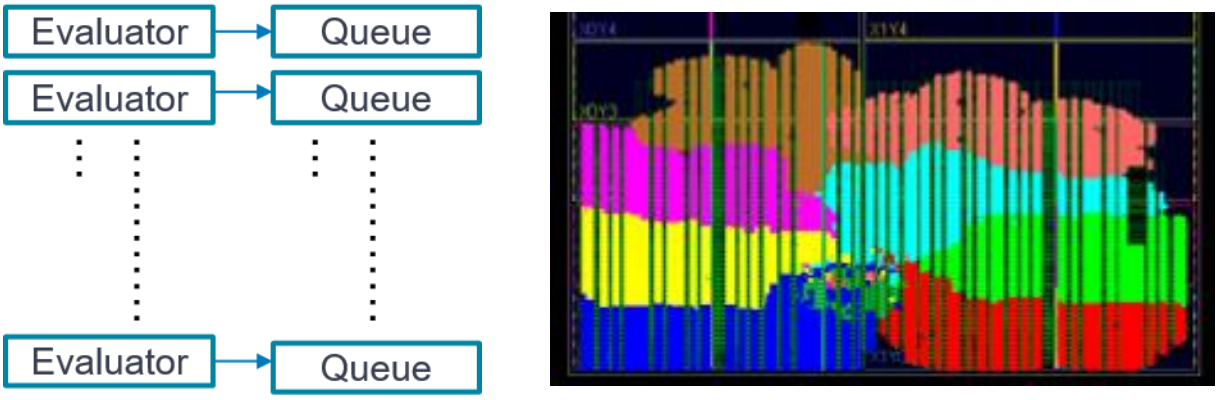


Figure 8-2: Pblocks assignment

Constraints File

We provided a constraints file to the implementation tool contains our floorplanning strategy and false paths. As we discussed before we listed our PBlocks in the constraints file to perform accurate floorplanning. We also listed the false paths in our design, which were mainly inputs and outputs ports in the design. We made all the registered inputs to be a false paths: the RST, the start node and the goal node as they only change once at the beginning of the algorithm and stay fixed for a period of time. The request data for LIFO signal had been excluded from the false paths as it must be synchronized with the design. All the output ports considered as false paths.

Strategies

Using try and error on several runs and with awareness with the corners of our design and how each strategy works, we settled on the strategies listed in the below table.

Table 8-1: Strategies

Synthesis	PerfOptimize
Opt_design	ExploreWithRemap
Place_design	ExtraPostPlacementOpt
Post-Place Phys	ExploreWithHoldFix
Route_Design	NoTimingRelaxation
Post-Route Phys	AgressiveExplore

During implementation runs, we notices that the slack setup violations doesn't exist until we enter the routing phase and the setup violations appear while the tool trying to solve hold violations, therefore we planned our final strategies to do early effort about hold violations.

Timing, Power & Area Results:

The A* accelerator was synthesized, placed, and routed by Xilinx EDA tool Vivado 2019.1. Our design's results after placement and routing are illustrated as shown in table 8-2 below:

Table 8-2: Summarization of area results for our design

Cells	Used	Available	Utilization
Slice registers	100735	866400	11.63%
Slice LUTs	121222	433200	27.98%
Block RAMs	82	1470	5.58%

The maximum frequency of the A* Accelerator is 200MHz. The total on-chip power is 1.569 Watts.

The results from another paper, that is published in the last year, that was targeting Xilinx Kintex-7 XC7K410T are illustrated as shown in table 8-3 below:

Table 8-3: Summarization of area results for other paper

Cells	Used	Available	Utilization
Slice registers	50930	508400	10%
Slice LUTs	134578	254200	52%
Block RAMs	14	795	1.8%

The maximum frequency of their A* Accelerator is 250MHz. There is no information about power.

In the above tables, we can see that our design is better only for slice LUTs. Block RAMs are used in our design for main memory, LIFO, and Visited LookUp table, on the other hand, they are using block RAMs for cache only. They have a better clock frequency than our design but we will see earlier that our design has a better performance overall.

CHAPTER 09 FPGA Deployment

As mentioned before, we are targeting Xilinx Virtex-7 FPGA VC709 kit. This kit has only Eight user LEDs and Five user pushbuttons and a reset switch. This number of I/O pins is not enough for our design, so we used built-in IPs in Vivado that allow us to use as many I/O pins as we need. These IPs are Virtual I/O (VIO), Integrated Logic Analyzer (ILA) and Clocking wizard and connected them with our design as shown in figure 9-1 below.

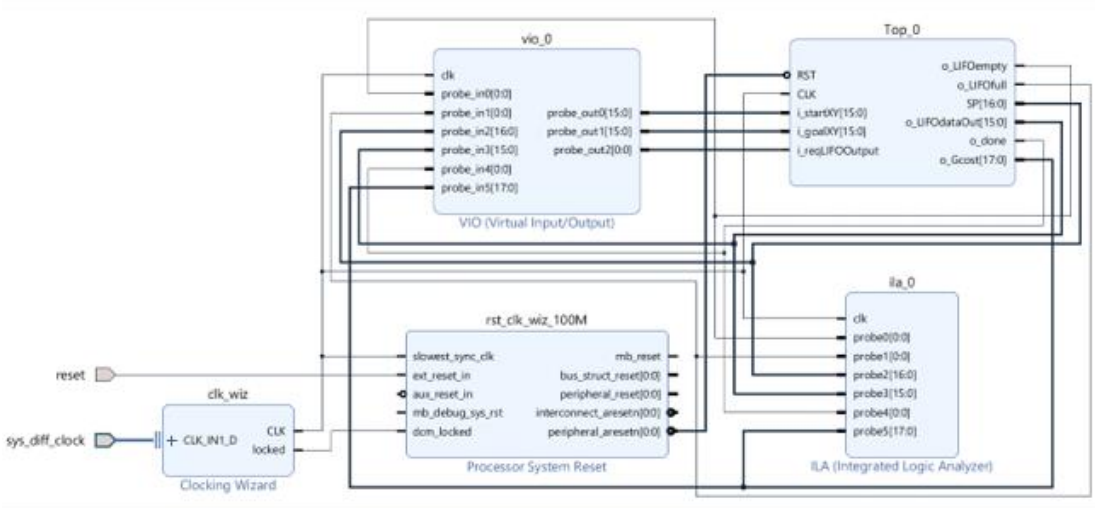


Figure 9-1: FPGA Deployment

VIO:

VIO IP is a customizable core that can both monitor and drive internal FPGA signals in real-time. The number and width of the input and output ports are customizable in size to interface with the FPGA design. VIO's inputs are the design outputs and its outputs are the design inputs.

ILA:

The customizable Integrated Logic Analyzer (ILA) IP core is a logic analyzer core that can be used to monitor the internal signals of a design. ILA is used to visualize the design's outputs in a waveform.

Clocking Wizard:

The Clocking Wizard simplifies the process of configuring the clocking resources in Xilinx FPGAs. Its input is an external differential clock and its output is the operating frequency for the design.

Results:

VIO and ILA IPs work at a maximum frequency of 100MHz, so we made the clocking wizard outputs a clock frequency of 100MHz. The design worked as intended on the FPGA.

- During FPGA deployment, we used a map with probability of obstacles equals to 0.3, start node address equals to 0x0000 and goal address equals to 0xffff.

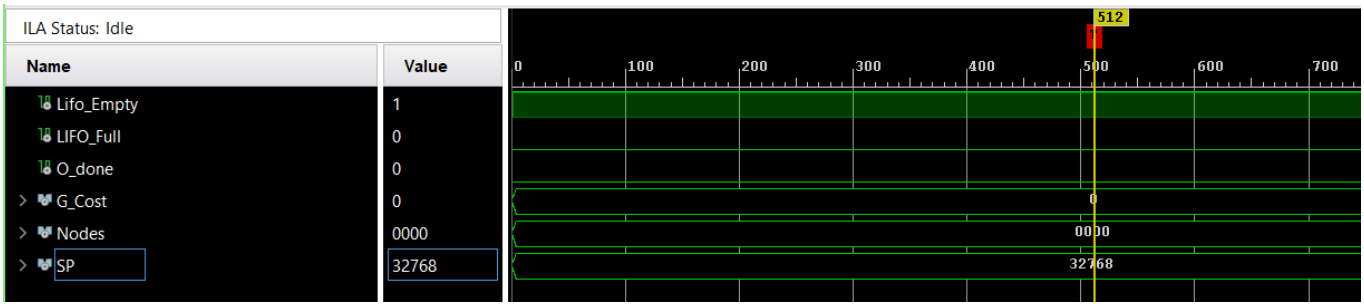


Figure 9-2: output signals during initialization

- The outputs shown in figure 9-2 represents the design state after initialization. With a LIFO being empty, low done signal and G cost equals to zero.

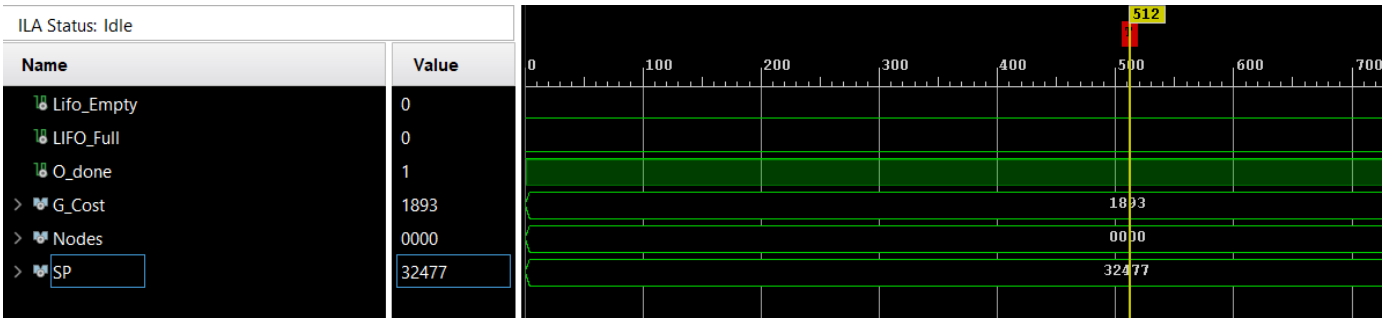


Figure 9-3: Output signals during operation

- When the algorithm reaches the goal node and writes the path in LIFO block. The done signal will be high, LIFO will not be empty and G cost will be calculated as shown in figure 9-3.
- After requesting the path from the LIFO, the LIFO will be empty again and the Last captured Node would be the goal node as shown in 9-4.

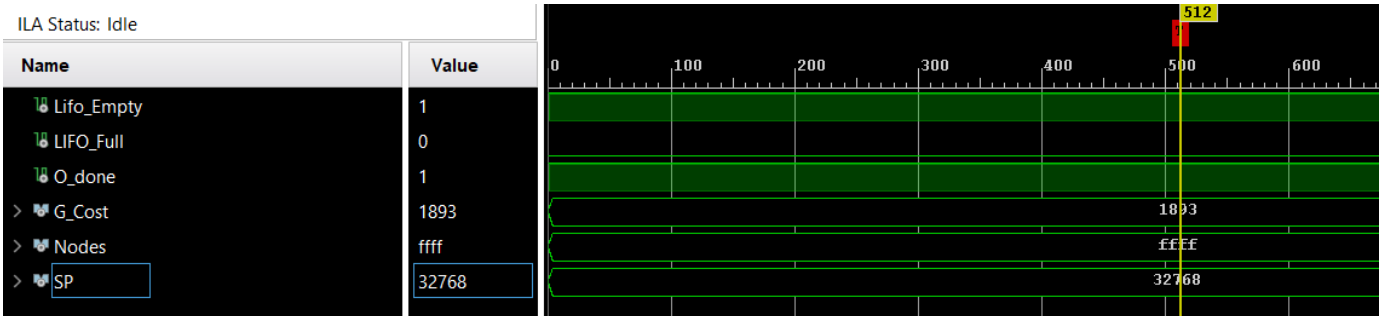


Figure 9-4: Output signals after requesting the Path

CHAPTER 10

Functional Safety

Introduction

Every electronic device has a life time. After this life time the probability of a failure to happen increases gradually. The failure happens due to many phenomena such as electro-migration. Failure may also happen randomly, not only after device's life time.

Applying functional safety mechanisms helps in making the design immune to failures. This is done usually by inserting some safety mechanism like duplication, triplication, parity or memory ECC into the design. Safety mechanisms help the design to recognize a failure and even recover from it. In our design, we applied safety mechanisms to prevent any failure to affect its functionality.

Functional safety mechanisms are being inserted to follow the 26262-ISO standard. This standard defines the metrics to measure the safety level of the design. The Automotive Safety Integrity Level (ASIL) is defined as the safety level of the design according to the injected safety mechanisms and the total area of the design. ASIL measurements depends on the probability of exposure, controllability by driver and severity of failure.

In the functional safety process, we calculate the safety metrics and providing early safety architectural guidance to achieve the safety target. Then, we insert safety mechanisms in the design to cancel the effect of the random errors. Before moving to the final step, we must re-calculate the safety metrics to make sure that after insertions the design still meets the safety target unless that, we should go back again to insert more safety mechanisms or modify them. Finally, the last step is to perform the fault campaigns to prove that the design safeness achieves the target safety levels. This final step is considered the verification of the safety level reached.

Tools

We used mentor graphics' functional safety tools that are still in development phase. They are four tools that depend on each other which are SafetyScope, Annealer, RadioScope, and KalidoScope that described in the next sections.

SafetyScope

Analyzes RTL or the netlist for FIT and DC. FIT stands for Failure In Time that is the measure of the hardware random defect rate for the permanent and transient errors in a design. DC stands for diagnostic coverage that is the measure of the number of permanent and transient faults. SafetyScope is used in the safety exploration process that determines

what safety mechanisms (SMs) the design requires to meet the safety targets (ASIL A|B|C|D).

Our design targets ASIL D, so we decided to use the following SMs:

1 Nodes manager, Memory manager, Path extractor, Comparator engine: The failure of any of these modules would result in a failure in the whole map leading to either failing in founding the path or exporting a wrong path. So since these modules should be more immune to failures, triplication SM was used.

2 Evaluators, Queues: these modules consume the bigger portion of the used resources in the FPGA, so triplication would have caused a problem in placement and routing. In addition, the failure in any of these modules would result in a waste of clock cycles but not a failure in finding the path. So, duplication SM was used.

3 Memories (main memory, LIFO, and visited lookup table): ECC (Error Correction Code) was used, as this SM is suitable for memories in the design.

Until now we calculated the DC value which is now 99% which is equal to the DC value for the ASIL D safety target. We can't calculate the FIT now as we need to make these changes first and this is done by Annealer and RaidoScope, then SafetyScope is used again to calculate the FIT value which must be less than 10 for the ASIL D safety target.

Annealer

Adds functional SMs into the design at the instance level. A control and an observe blocks are instantiated in the design hierarchy that generate sticky error outputs (alarms). Duplication is done by Annealer that creates two instances of the top design and makes appropriate connections for all the outputs and inputs between the first and second instance along with the checker block.

Triplication is done by Annealer like duplication except that triplication has a majority vote that decides which output is correct. This is done by comparing the three outputs and the output that appears at least from two of them is decided to be correct.

RadioScope

RadioScope tool do the same job as Annealer tool but at the FF level.

KalidoScope

The KaleidoScope tool injects faults and manages the fault campaigns that test the circuits of mission-critical systems. The tool injects faults into safety-critical nodes in the design, which are then tested to determine if the SMs detect the faults. It injects and propagates faults by using the safety context in the VCD from RTL simulation. There is four expected fault outcomes from any injection:

1. Alarm Triggered (Detected Fault): Machine state is different from the golden safety context; the fault is propagated to an alarm.
2. Safe fault: Machine state is not different from the golden safety context.
3. Unsafe fault: Machine state is different than the golden and alarm did not fire.
4. Unresolved fault: Machine state is different from the golden. Fault is propagated to a black-box (analog or user-defined).

Results after Functional Safety

After applying functional safety mechanisms to the design, it was synthesized, placed, and routed by Xilinx EDA tool Vivado 2019.1. Summary of resources used are illustrated as shown in table 10-1 below:

Table 10-1: Summarization of area results for our design

Cells	Used before FUSA	Used after FUSA	Available	Utilization before FUSA	Utilization after FUSA
Slice registers	100735	177644	866400	11.63%	20.5%
Slice LUTs	121222	227001	433200	27.98%	52.4%
Block RAMs	82	82	1470	5.58%	5.58%

The maximum frequency of the A* Accelerator after functional safety mechanisms insertion is approximately 145MHz. The total on-chip power is 1.471 Watts.

In the above results, we can observe that after applying FUSA mechanisms, there is a large increase in the resources used on the FPGA, this is due to using duplication or triplication for some modules. We can also observe the degradation in frequency. From the very first beginning of deciding to use FUSA mechanisms on the design we expected the degradation in frequency, , hence the design was optimized on every level so after insertion of FUSA mechanisms we had still some good acceleration on the performance of the path finding process done by the A* algorithm.

Using again the average number of cycles that were calculated using results of UVM test explained in chapter 7, the average time to find and extract the path after inserting safety mechanisms to the design tested on random maps setting start and goal nodes at worst case is given in table 10-2 below:

Table 0-2: Design timing results after FUSA mechanisms insertion

Probability of obstacle	our Design average Time **before** FUSA (ms)	our Design average Time **after** FUSA (ms)
10%	0.198	0.273
20%	0.379	0.523
30%	0.556	0.762
40%	0.765	1.055
50%	1.078	1.487

We can say that there's some trade-off between functional safety and performance or power as using it will always increase the power (for a constant frequency) as the resources used are more, and it will always decrease the frequency of the total design. But this degradation is tolerable and in front of the huge benefits of Functional Safety.

References

[1] "synopsys," [Online]. Available: https://www.synopsys.com/automotive/what-is-asil.html. [Accessed 2021].

[2] P. Lester, "A* pathfinding for beginners," 2005.

[3] Red Blob games, "Heuristics From Amit's thoughts on path finding," [Online]. Available: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html. [Accessed 2021].

[4] J. R. a. K. G. S. Sung-Whan Moon, "Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches," vol. 49, p. 13, 2000.

[5] S.-H. P. D.-W. K. Young-Ho Seo, "High-level hardware design of digital comparator with multiple inputs," 2019.

[6] X. J. ,. a. T. W. Yuzhi Zhou, "FPGA Implementation of A∗ Algorithm for Real-Time," China, 2020.