Implementation of an FPGA testbed as an educational experiment for students of informatics

Lukas Stasytis
Faculty of Informatics
Kaunas Technology University
Student g. 50, Kaunas 51368
e-mail: lukas.stasytis@ktu.edu

***Abstract*—The following paper describes a hardware implementation of a simulation platform on an FPGA board to provide a test bed for machine learning experimentation. The educational value that such a project can have for a student in putting theoretical programming, digital logic and computer architecture principles to practice.**

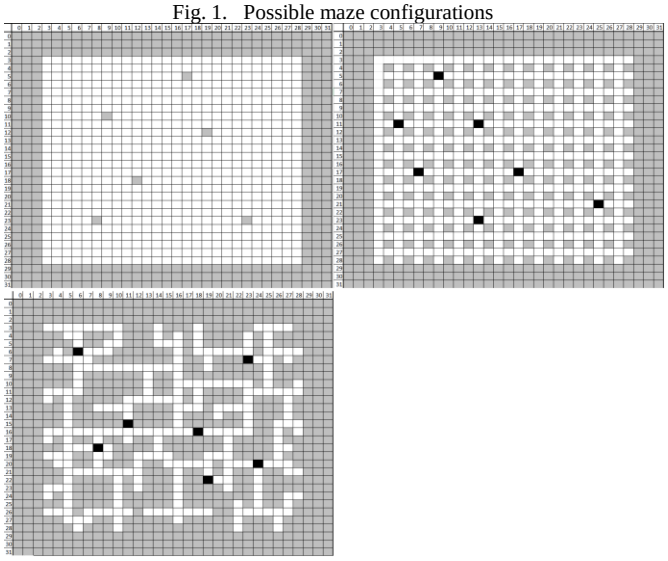
I. Introduction

The author of this paper, while undertaking the 2nd semester of a Computer Science program, took up the following project as a means to put studied theoretical Digital Logic principles to practical use beyond the typical course lab work and to better understand how hardware is designed and implemented. The goal of the project is to implement a two dimensional simulation platform using the VHDL programming language on to an FPGA (Field Programming Gate Array) board. Machine learning implementations, like neural networks, could then be attached to said implementation to run high speed experiments. As such, this project is meant to lead up to an artificial intelligence research project. A key focus of this paper is to show how taking up such a project was extremely valuable in revising old and learning new computer science principles for the author.

II. Implementation overview

As a whole, the implementation simulates a two dimensional bit maze with '1' bit-wide entities that can traverse the maze taking turns in sequence. The maze itself can have a set configuration to either simulate a complex maze, a flat field or a simple checker-board pattern. The entities are divided into two groups: the player and a list of guards. The goals of these entities can vary, but the primary goal of the player is to reach a designated tile on the maze while the goal of the guards is to catch the player. Figure 1 shows some of the possible maze configurations that the implementation covers.

Fig. 1. Possible maze configurations



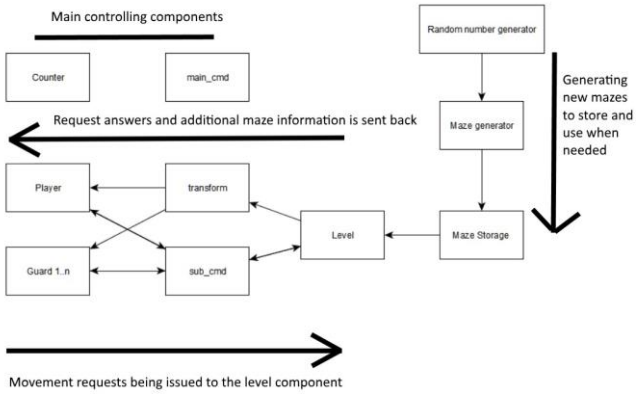
The black squares represent entities and the goal while the grey squares represent walls. A surrounding wall is generated around the structure, for reasons that will be discussed in the level component implementation overview. The general process run-down of the implementation is as follows:

(a) An external input command is given to start the simulation after choosing a specific mode in which it will run.
(b) The initial maze is generated by taking into account the chosen mode.
(c) Each entity is given a random starting position and initial maze information to help make the initial decision.
(d) The simulation starts and over the course of a series of counter clocks, each entity makes a decision of which direction to either move to or turn to.
(e) The entity decisions are sent to the component holding the maze's information to check if the decision is legal.
(f) The newly generated maze information for the next decision is sent back to each entity.
(g) A turn summary is generated and sent to the player and, if chosen to, system output to store.

(h) Based on the results of the turn, a new turn begins or the simulation is reset.

Figure 2 shows the entire structure of the implementation with all its components.

Fig. 2. The base structure of the platform



Going from right to left, the ten components are as follows:

(a) Random number generator
(b) Maze generator
(c) Maze storage
(d) Level
(e) Main command
(f) Transform
(g) Sub command
(h) Counter
(i) Player
(j) Guard (multiple)

The following is a short run down of each component and how they function in the platform as a whole before going deeper into the structure of each component and development principles:

(a) The main command component controls all the other components in the system. The system inputs go directly to this component and based on those, the component sends out a new state signal for each other component to react to. This state signal defines what each component should be doing - being on standby, preparing for a simulation or running a simulation.
(b) Once the simulation request is made, the maze generator starts generating the initial maze with the help of a random number generation component.
(c) The newly generated maze is sent to the maze storage component and from there, to the level component. The maze generator continues to store new maps in the storage component afterwards.
(d) Initial maze information is sent to each entity from the level component via the sub command component, which is in charge of routing entity requests and maze controller (level) answers.
(e) With the initial configuration of each entity set, the actual simulation starts. Each entity sends out a request for a new move.
(f) The counter dictates which entity request will be handled first, allowing for a parallel and iterative request handling. This is done by limiting each component's functions to specific counter ticks.
(g) The level component, after checking each request - making sure the move is legal and that no collisions are present, sends an answer back to the entities and modifies the maze array to reflect on a passed or rejected movement request. At the same time, a small field view of the maze is generated for the entity to use when making the next move. This information is sent to a transform component.
(h) The transform component modifies the maze information that goes to the entities. Specifically, removes entity view vision behind walls.
(i) After each entities request has been handled and all the answers have been delivered, a turn summary is generated inside the level component and sent to the player and the system output/storage. Based on this summary, the simulation can continue or be reset due to player loss or victory.

III. Component Overview

A. State machines and the main command unit

At the highest level, the platform acts as a state machine. The main command component, which connects to each other component of the platform, has a three bit-wide variable which determines which state the platform is in. Each other component has a case statement at the very top, checking which state the platform is in to decide on a specific task that it will be performing. Figure 3 shows a bit dictionary of the state signal.

Fig. 3. Bit dictionary of the state signal

0	0	0	Standby
0	0	1	Generating field
0	1	0	Uploading field
0	1	1	Generating entities
1	0	0	Simulation running on mode 1
1	0	1	Simulation running on mode 2
1	1	0	Simulation running on mode 3
1	1	1	Simulation running on mode 4

By using such a state configuration, modularity can be achieved within the simulation to allow for different types of simulations to be run without needing to recompile and re-upload the hardware configuration. There exist two more main state machine implementations within the platform, namely the use of a counter to decide what state the turn is at and a state machine for the guard artificial intelligence, albeit only for testing purposes. Both will be explored in their respective sections.

B. Maze generator

The current primary maze generation module uses a recursive backtracking generation algorithm implemented iteratively using a stack of 14 bits the size of the maze path. 7 for the x and y coordinates of each tile that has already been traversed. Over a few hundred series of clocks, the component generates a full maze of a given size and then sends the maze one line at a time to the maze storage component. The generator itself is fed with a random number each clock cycle from the random number generation component and uses said number to decide in which direction to continue digging paths. All the variables are stored in registers and the maze's size goes up to 128*128 tiles wide, but could be adjusted with only a few edits in the code.

A list of algorithms were looked at to determine which one would best suit a hardware implementation. Some of the algorithms looked at were:

(a) Kruskal's algorithm - very easy and quick implementation, but produces a lot of dead-ends in the maze, which can be very difficult for an AI to overcome.
(b) Recursive division algorithm - requires very large stacks to implement iteratively, would likely consume too much space on an FPGA.

The recursive backtracking algorithm was the final choice because of its simple iterative implementation that requires a single, easy to read stack and highly branching paths, which could potentially allow for a more fluid machine learning process. An iterative approach was picked, because there is no value to be gained in generating mazes over a single clock cycle, given the comparably long simulation times that require hundreds of clock cycles to complete. Additionally, a recursive method is very resource heavy and an FPGA board of any variant is heavily limited in space.

C. Random number generator

A simple LFSR - Linear Feedback Shift Register was used to create a pseudo-random number generator with minimal effort. The generator starts generating numbers from the moment an input command by the user is sent and reset every time the input is given again. Given how the FPGA of choice (Altera's DE0 Development and Education board) runs at 50 MHz clock speeds, a human input should already add enough randomness to the seed to allow for fairly random mazes to be generated. Given the fact that different components of the FPGA board can be accessed, some of which can hold highly random data information, alternative, more random and automated methods of generating seeds might be considered in the future. Given that the random number generator is separated from the actual maze generator, modularity is easy to accomplish.

D. Maze storage

The generated mazes are stored in a four-layer wide stack of mazes, each consisting of 'map width' x 'map width' size array of bits, where each one represents a wall. By saving mazes in a stack, rather than generating only a single maze and uploading that, we are able to instantly swap from an older maze configuration to a new one once a maze has been cleared. Given how a single maze can take a few hundred clock cycles to build and a single turn takes ten cycles, we can see in practice that a map buffer can easily be achieved and used to not have to wait in between the player beating mazes. The maze stack also allows for pre-generated mazes to be input into the system from external memory.

E. Level component

The level component has four functions:

(a) Storing the current simulation maze array with all necessary information of each tile
(b) Generating the starting positions of each entity and sending them out.
(c) Checking for the legality of incoming movement requests and making changes to the maze array appropriately. That is, making an entity position change reflect in the actual array that holds the maze for collision checking.
(d) Generating maze information for each entity when handling a movement request

The level component is the heart of the platform that practically holds the entire simulation within itself. Entity information and stored mazes for future simulation cycles are the only external pieces of information that the level component does not hold within itself. The component's basic structure is based on a series of if statements that determine if the entity making a request is a player or a guard, if it is requesting to turn their vision or to make a movement and if the entity has other entities in its field of view to count damage checks. This is followed by a collision check by testing if the tile that the entity wants to move to is already filled with an entity or not. If a collision occurs, the entity's position is simply left unmodified. A core design principle within the level component is that the request is handled by simply modifying the incoming request signal and sending the modified signal back to the entity that made the request. Figure 4 illustrates this signal.

Fig. 4. A movement request signal breakdown

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a						b							c			d	e	

In the table, going from left to right, we have:

(a) '7' bits to store the x axis position of the entity.
(b) '7' bits to store the y axis position of the entity.
(c) '3' bits saving the unique id of the entity.
(d) '1' bit telling if the entity is requesting to move or to turn
(e) '2' bits telling the direction in which the entity would like to move or turn its vision.

Once a request arrives at the level component, the id tag is checked to determine if the requesting entity is a player or the guard, then it is checked if the entity is requesting a move or a turn and based on that information the current position of the entity and the requested movement direction are added together and checked for validity. Once validity has been confirmed, the position coordinates of the entity may or

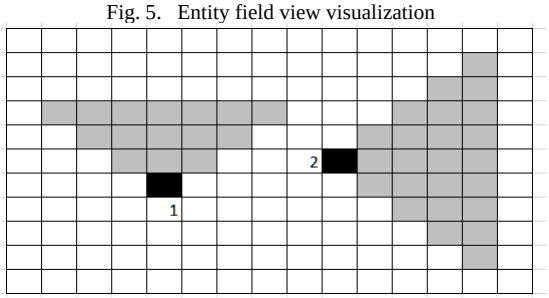
may not be changed. All this information is then sent back to the entity in an answer signal that is generated using the same format as the request signal. Furthermore, a field view is generated based on the direction the entity is facing and sent to the transformation module.

The generation itself is achieved by simply multiplying the entity coordinates by a number ranging from '-4' to '4', excluding '0', which depends on the direction the entity is facing. This way, every required maze array tile for a field view can be obtained with minimal if statements. This does, however, make any further mathematical operations in the same clock cycle impossible on the generated map view.

Lastly, a '3' bit-wide wall is present at the edges of the maze array to eliminate the need to worry about out-of-index errors. In other words, rather than creating many positional checks to make sure that the field view does not look past the maze array, the array is simply expanded further and filled with wall tiles. In doing so some register space is sacrificed for an easier algorithm.

Figure 5 shows an example of what the field view looks like.

Fig. 5. Entity field view visualization



Number 1 marks a guard entity facing northwards while number 2 - a player entity facing eastwards.

The player entity is given one extra line of vision to give it an advantage over the guards. The information is sent in three or four data packets, each holding the tile information of each specific tile in that line. The maze arrays have '4' bits assigned to each tile, marking the wall, player entity, guard entity and objective variables.

F. Sub command unit

The sub command unit, located between the level and the entity components, acts as a router to make sure that the level component is getting a single request each clock cycle and that the entities are all receiving their answers. This unit is not necessary, given that the entities could just be directly connected to the level component, however the separation allows to vastly simplify the level component, which is already much more complex than the rest of the platform and so losing one clock cycle per turn to achieve this result is deemed a worthwhile trade. Listing 1 shows the code of the sub command component.

Listing 1. Sub command component

```
begin
 if(rising edge(clk)) then
  if state(2) = '1' or state = "011" then
   case counter is
         when "0001" =>
          tolevel <= sp0;(1)
         when "0010" =>
          tolevel <= sg01;
         when "0011" =>
          to level <= sg02;
          to entities <= from level;(2
         ) when "0100" =>
          to level <= sg03;
          to entities <= fromlevel
         ; when "0101" =>
          to level <= sg04;
          to entities <= fromlevel
         ; when "0110" =>
          to level <= sg05;
          to entities <= fromlevel
         ; when "0111" =>
          to entities <= fromlevel
         ; when "1000" =>
          to entities <= from level;(3
         ) when others =>
   end case;
  end if;
 end if;
end process main;
```

The following is a list explanation of the three noted lines:

(1) A signal 'sp0' (from the player entity) is being sent forward to the level component using the 'to level' signal.
(2) Later in the turn cycle, the component is sending forward a request and also sending back an answer of a previous request in parallel. In this case - sending forward 'sg02' (second guard entity) and receiving 'sp0' (the pre-last forwarded request's answer)
(3) At the end of the turn cycle, all that is left is for the component to handle the remaining answer signals being generated by the level component.

An important distinction is that the incoming entity requests come from specific input signals, while the answer signal to the entities goes into the same "to entities" signal, this is because only the sub command component is inputting an answer signal, while all six entities are inputting the request signals. So in the return trip, each entity can simply feed from the same bus and check if the answer travelling matches their identification number, while this is not possible while making the requests, because the signals would clash. A master-slave communication protocol could be used to make the entity input also come from a single signal. This is not done for simplicity sake, since the "slave" count is very limited in the current implementation, leaving the protocol as a potential

future modification if deemed necessary.

G. Field view transform

The transform component is given the information that an entity is meant to obtain from the level component and then makes some changes to account for set rules. Specifically, the current iteration of the implementation has a single bit dedicated to saying that the tile is a wall. Naturally, an entity should not be able to see what is behind a wall. Thus, a filter is ran that removes all information that should not be visible to the entity, because of walls obstructing vision. This is achieved by a series of if statements. Each tile in the first few lines of the cone is checked for holding a wall value and if it does - each tile behind it is made null, mimicking a lack of vision.

The reasoning behind this component is that, in the level component, a clock cycle is already used up when making a mathematical transformation on the entity's facing direction and coordinates to determine which tiles to grab without needing to input too many if statements. As such, the transformation component is separated from the level component to make the scrambling transformation independently. This also allows to easily modify the component to change how the transformation is done.

H. Entities

A few practices present in object-oriented programming are put into use. Namely, how all the entity components are similar to objects in how they store functions and all entity variables and are treated the same by other hardware components within the system. However, unlike in true object-oriented programming, the object count is fixed and requires code readjustments to change, which does cause a scalability issue. Each entity has a separate hardware component in which all information about the entity is stored. The entities are each given a unique identification tag at the start of the simulation by a direct input command. Additionally, each component is prepared to be connected to a separate component that would decide which move the entity would make. This is how machine learning would be implemented into the platform. By attaching, for example, a neural network to each entity. After receiving an answer to a turn, each entity would send this answer with additional information from past turns to the neural network, upon which, a move can be decided. The answer would then be generated in a single clock cycle and sent back to the entity to then generate a request to the level component. Lastly, the player entity is different from the guard entities, because at the end of each turn, it also receives a direct turn summary from the level component. Using this summary, the entity checks if the simulation should be reset because of the objective being achieved or the player having lost.

I. The counter and platform parallelism

Figure 6 shows the main conveyor of the simulation.

Fig. 6. Platform simulation conveyer

E_SC	Entity sends a request to sub_cmd
SC_L	sub_cmd sends it to level
L_T	after handling the request, level sends an answer to the cmd and a turn data packet to the transform
T_E	transform scrambles the new information of the turn and sends it out for one of the entities to pick up
SC_E	sub_cmd sends the answer forward for one of the entites to pick up
L_E	level sends turn end results directly to the player

counter	0	1	2	3	4	5	6	7	8	9
player	E_SC	SC_L	L_SC+T	SC+T_E						L_E
guard1		E_SC	SC_L	L_SC+T	SC+T_E					
guard2			E_SC	SC_L	L_SC+T	SC+T_E				
guard3				E_SC	SC_L	L_SC+T	SC+T_E			
guard4					E_SC	SC_L	L_SC+T	SC+T_E		
guard5						E_SC	SC_L	L_SC+T	SC+T_E	

The way the conveyor works is that by implementing a counter component that counts up to a specific number of choice and then resets, we are able to fully control what each component is doing at each clock cycle while the simulation is running. The conveyor greatly decreases the amount of clock cycles each turn requires to be cycled by giving each entity a piece of information to work with. For example, in the counter = '4' cycle, going from up to down, it can be seen that:

(a) The sub command and transformation components are sending out the 1st guard's request answer to be picked up by said guard.
(b) The 2nd guard's answer is being handled and sent out to the sub command and transformation components.
(c) The 3rd guard's request is being sent to the level component from the sub command.
(d) The 4th guard's request is being sent from the entity to the sub command component.

Additionally, at the end of the conveyor, the level component generates a turn summary and sends that to the player entity. If a component that implements machine learning is added to the platform, it would be using up two additional clocks (entity sends a request, the component sends back an answer) after each entity has received its field view for that turn.

J. Discussion

The development process of the overviewed implementation has proven to be an extremely educational experience for the author with each part of the system acting as a tool to revise and learn new computer science principles. The following is a list of some of the lessons that were learned in the development process:

(a) State machines.
(b) Hardware time dependency - at the hardware level, a mathematical calculation can only be correctly made if it is known that the variables that make up the formula have already been calculated. Given the nature of clock cycles, this had to be carefully looked at.
(c) Random number generation - a look at how random numbers are generated at the hardware level is required. Different algorithms have to be looked at to decide which one is most fitting for the job.
(d) Random number generator seed randomness - different approaches can be considered while designing a hardware chip and how there exists a possibility to implement a hardware random number generator by using physical

processes of the chip, such as thermal noise and clock drifts, which could achieve much greater degrees of randomness than software implementations.

(e) Signal handling, component structure and optimization - the level component requires efficient handling of the movement requests to send out an answer in the same clock cycle without creating variable dependencies, but at the same time keeping the implementation of a manageable size. For example, the transformation component was separated from the level component entirely to allow for field view generation to be done very efficiently by using a single entity's coordinates and direction to apply a mathematical transformation and obtain a field view.

(f) Routers - a simple router had to be implemented within the platform. Signal handling and time planning were necessary.

(g) An event - condition - action (ECA) system is put into practice with the answer signal handling.

(h) Object oriented programming principles are practiced, namely by separating entities into separate components and treating them much like objects while developing the platform. The answer and request handling is done by state machines that work very much like object methods. Such an implementation gives an interesting look into the differences between software and hardware design.

(i) Artificial intelligence is looked at by implementing a state machine to the guard entities for testing purposes.

(j) Hardware parallelism implementation using a conveyor

(k) Counter implementation and usage in hardware

K. Future work

After careful review, it has become apparent that the design is highly inefficient when fully implemented in hardware. The current design takes up over 12000 LUTs or 80% of the entire matrix capacity of the FPGA board used for testing the system. As such, the entire platform will be reimplemented in software while creating a real-time interface between a computer and the FPGA board and implementing only the machine learning components within the FPGA board. The current plan is to use a RS-232 communication protocol and send out entity field data to the FPGA and take back only the answer from the board. If the RS-232 standard turns out to be too much of a bottleneck in the system, more sophisticated ways of interfacing the devices will be looked into.

IV. Conclusion

In this paper, the implementation of an experimental test bed platform as a personal educational project was covered with all the informatics principles and mazes that were touched upon while developing. Over the course of the implementation and after reviewing the final design, it has become clear that having the entire design implemented on an FPGA board is highly impractical, because the greatest bottleneck is in the machine learning process, not the platform simulation. Limiting the hardware implementation to only the machine learning part and moving the rest of the system to software would be a more cost efficient approach and will be the next focus of the research project. The implementation does still show a way to simply implement an entity simulation within hardware and acted as an excellent tool for the author to further his education.

Acknowledgement

I would like to express my deepest thanks to my mentor Kazimieras Bagdonas for not only guiding me through every step of the way towards realizing this project, but also helping me find my direction and develop a deep passion for learning.

References

[1] E. E. Almeida, J. E. Luntz, D. M. Tilbury, "Event-condition-action systems for reconfigurable logic control," *IEEE Transac- tions on Automation Science and Engineering*, vol. 4(2), pp.167–181, 2007.

[2] K. L. Dittrich, S. Gatziu, A. Geppert, "The active database management system manifesto: A rulebase of adbms features," In *International Workshop on Rules in Database Systems*, pp. 1–17. Springer, 1995.

[3] S. S. Huang, A. Hormati, D. F. Bacon, R. Rabbah, "Liquid metal: Object-oriented programming across the hardware/software boundary," In *European Conference on Object-Oriented Programming*, pp. 76–103. Springer, 2008.

[4] F. Islam, M. Ali, B. Y. Majlis, "FPGA implementation of an LFSR based pseudorandom pattern generator for mems testing," *International Journal of Computer Applications*, vol. 75(11), pp. 30-34, 2013.

[5] B. Jun, P. Kocher, "The intel random number generator, "*Cryptography Research Inc. white paper*, 1999.

[6] M. Smiroldo, "Method and apparatus for managing a number of time slots during which plural bidding devices can request communication access to a central device," August 5, 1997. US Patent 5,654,968.