Proceedings of the 2007 IEEE Symposium on Artificial Life (CI-ALife 2007)

An Unbounded Parallel Binary Tree Adder for use on a Cellular Platform

James Weston, and Peter Lee
Embedded Systems Group
Department of Electronics, University of Kent
Canterbury, Kent, CT2 7NT, UK
Phone: +44 1227 824210, E-Mail: jlw25@kent.ac.uk

Abstract – **Cellular automata are by definition highly parallel structures and are therefore capable of giving rise to massively parallel systems. The highly parallel nature of the cellular automata framework permits the creation of a multitude of structures, endowed with the flexibility to perform vast amounts of calculations concurrently. This flexibility and parallelism is also now present in a number of hardware platforms allowing for the adaptation of automata models into hardware. Presented herein is a binary tree adder implemented in cellular automata, able to perform substantial numbers of additions simultaneously. The number of calculations performed is only limited by the automata size. The binary tree adder is also more simplistic in terms of both states (25 used in total) and structure, than has been published before. Due to advances in hardware technology, it is a very realistic ambition for the future to be able to represent the tree adder structure on a cellular platform such as, an FPGA, allowing for such advantages as, increased robustness which is an area regarded as vital for developing the future of electronics hardware.**

I. Introduction

This paper is principally concerned with demonstrating that arithmetic in cellular automata can be performed efficiently, and simplistically, giving rise to the possibility of implementing highly parallel systems. The desire and intention is to show that it is now possible to transfer these properties from a cellular automata model into different forms of cellular hardware.

Past research into arithmetic within cellular automata will be discussed and will lead into a detailed account of the development and implementation of a binary tree adder. The binary tree adder presented here is a highly parallel structure capable of performing binary addition in a highly parallel manner. It has been developed with hardware in mind as it is the eventual aim to place the binary tree adder onto an FPGA. Field programmable gate arrays offer the flexibility to be able to implement cellular structures like the binary tree adder as they have the inherent ability to locally link cells of a simplistic nature, leading to the advent of vast parallelism within the structure.

The worlds of cellular automata and electronics hardware have until recently been rather disparate domains. A technique that has brought the two areas closer together is the concept of cellular computing [1]. Cellular computing provides the possibility of enabling cellular automata structures to be embedded within electronics hardware. Field programmable gate arrays and the cell matrix [2] are two such platforms within which cellular automata can be implemented.

Cellular computing including cellular automata offers the opportunity to create truly parallel systems capable of, in cellular automata extensive computation and within electronics, numbers of computations only limited by the number of cells that can be placed onto a specific platform. Cellular computing provides such advantages as vast parallelism, simplicity of the cell, and locality, easing the burden of communication within a system as well as providing the possibility of increasing fault tolerance within a system.

The concept of cellular computing offers the potential to answer one of the eternal questions surrounding the area of electronics hardware. How can truly robust systems be designed and built? This question is very important for the future development of hardware, as designs become more complex the need for greater robustness is ever present. The answer to this question of how to increase the robustness of hardware is through the use of techniques such as self repair [3] which can be implemented using techniques derived from automata theory.

The field of cellular automata has been in existence for a number of decades since its inception by John Von Neumann [4] in the 1940s. During this time cellular automata has grown in its popularity as a useful modelling and theoretical tool. Much work has been completed using cellular automata from such diverse scientific fields as biology and physics as well as from numerous other more mathematical fields such as computing and engineering. It is this diversity that is responsible for the continued and renewed interest in cellular automata to this day.

Von Neumann was the first to entertain the idea of self replication in cellular automata with his self replicating automaton. He concluded that computation universality and universal construction need to be present to enable a self replicating machine. Codd [5] then developed upon this introducing the periodic emitter (a basic timing element in his system) and the data path which allowed him to simplify Von Neumann's ideas.

Langton [6] then developed these ideas into the first truly simplistic self replicating loop and even simulated his findings, demonstrating his theory that universal construction is not a necessary condition for self replication. Byl [7] then simplified Langton's ideas even further and managed to create self replicating structures that are substantially less complex than Langton's original.

The first signs of computation in cellular automata related to self replicating loops came from Tempesti [8]. Tempesti realised that although the self replicating loops were able to self replicate they were not capable of performing tasks with any computational significance, moreover they could be likened to a software or biological virus capable exclusively of self replication. Tempesti introduced the ability to perform independent constructional and computational tasks alongside replication. In addition to this the loops also continue to live after replication, rather than dieing like in the previous work from Langton.

The remainder of this paper will describe arithmetic in cellular automata, as well as the details of the binary tree adder, from its first design phase through to the finished model of the binary tree adder. The results shown in this paper prove the possibility of implementing a truly parallel structure in cellular automata, capable of being transferred onto a cellular platform with the eventual aim of improving robustness within electronics hardware through making use of self repairing technologies.

II. Arithmetic In Cellular Automata

One of the next steps in the developmental history of cellular automata was the introduction of arithmetic operations into a cellular automata space. Performing arithmetic operations within cellular automata is not a trivial process and can take any of a number of structural forms, depending on the proposed application. For example, perhaps a digital filter is being created using binary number representations or addition is being performed using hexadecimal number representations.

The main reason to try to perform arithmetic operations in cellular automata is so that a theory can be obtained that can be transferred to a form of cellular hardware. The theory in cellular automata allows for certain factors to be monitored like synchronisation and communication within the system. Performing the arithmetic in a cellular manner also allows for later properties like self repair to be implemented within the system through making use of the cellular architecture of such structures.

The pre-eminent starting point regarding the research into arithmetic in cellular automata presented here, is the advent of the particle machine by Squier and Steiglitz [9]. The machine they proposed, depicted in Fig.1, is capable of addition, subtraction and multiplication as well as combinations of these enabling it to perform for example, finite impulse response digital filtering.

The theory they proposed is based around the collisions of particles within a tube like structure which they termed the particle machine. Within this particle machine there are two addends consisting of four types of particle (both left and right moving zeros and ones). The addends meet least significant bit (LSB) first which makes the encoding of the carry bit easier.

At the cell where the addend particles collide, a fifth type of particle is placed, known as the processor. This processor is

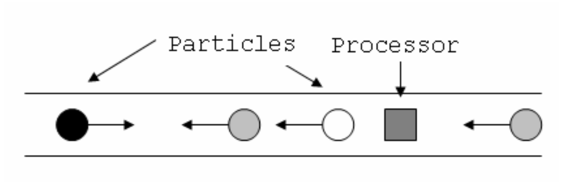

Fig. 1. The particle machine.

actually one of two particles used to encode the carry bit resulting from the collision. Collisions may occur when particles are travelling in different directions or in the same direction at different speeds. Fig. 1, shows the general model of the particle machine with its particles, and processor states. Also note that the particle machine is open ended for insertion of particles at either end.

One of the advantages of such a parallel system is the homogeneity of the implementation. The composition from like parts means that only the hardware supporting a particle set and their interactions needs to be implemented. This in turn yields decreased design time as each cell is identical and more simplistic hardware designs become a reality.

Squier and Steiglitz state that they are not concerned with retrieving the results and say that the computation is complete when the result appears somewhere in the array or that it could come out by using some form of tapping along the array. This is something that needed to be altered in the binary tree adder as the result needed to be clearly defined so that it could be used again later by another processor as part of the tree structure.

Their paper also details another method for use within the particle machine whereby the addends are stored on top of each other in data atoms which are defined to be stationary. This has obvious advantages in terms of optimising space usage, but is not feasible in a system where multiple processors and interactions may be required. The result needs to be more mobile and well defined so that it can be used within another processor cell, this could occur by using a tapping method, allowing the resultant to move out of the structure.

They conclude that once a substrate and its particles are defined, specifying streams of particles to inject for particular configurations is a programming task, not a hardware design task and hence the design time has been drastically decreased. This also provides added flexibility to the hardware that would otherwise not be obtained as the particles needed are merely injected into the system as and when they are needed.

Petraglio [10] then took the particle machine theory and put it into practice within the structure of self replicating loops. Due to using the self replicating loops the theory of the particle machine was altered to fit around the loops. Petraglio implemented both addition and multiplication and managed to perform combinations of both of these.

The major difference between the two works is that Petraglio uses one automaton to compute the end result of a single collision between two binary data bits, whereas Squier uses processor cells to compute the entire addition. As can be seen from Fig. 2, the starting point for addition consists of the

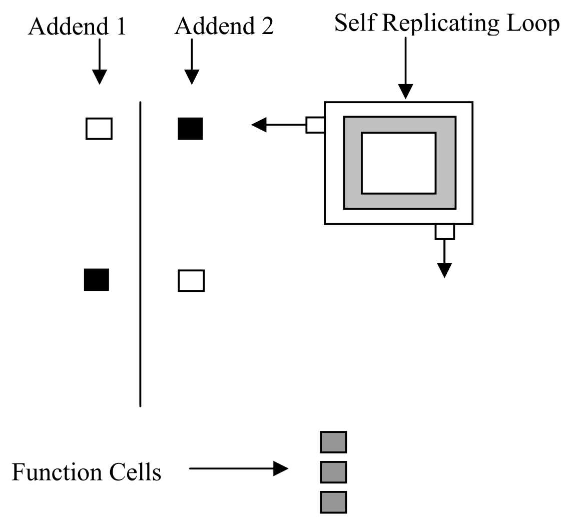

Fig. 2. Petraglio's self replicating loop adder.

two addends used to create the sum, a self replicating loop and a special function cell. The self replicating loop is made up of an internal inert loop which is the sheath and an active external program loop which contains the program to be executed as well as the information for self replication.

Petraglio also managed to perform multiplication through a slightly altered method and combinations of both operations for example, equation (1) (where 'A', and 'a' are binary numbers).

$$(A + B) * (a + b) \qquad (1)$$

The finished automaton is made up of over 30 states and is easily simulated. The method used takes the ideas and concepts gained from the particle machine and develops a new method for arithmetic in cellular automata. They proved that self replicating loops can be used for complex mathematical operations. This however is not the most efficient way to perform these operations within the cellular automata framework.

The main issue with Squier and Steiglitz, and to an extent Petraglio's work is that they are more parallel components rather than complete parallel systems. What is meant by this is that they are built for a single addition and not a complete series of additions. In the case of Squier and Steiglitz there is a single particle machine which produces a result and this machine is then incapable of interacting with another particle machine through the transfer of resultants between machines. Petraglio's work also details how the self replicating loops can be started again to perform another addition, but again this is not a series of interacting additions it is separate additions, that cannot interact with each other.

Presented in the next section is the binary tree adder, a system with the ability to produce a resultant and then feed this into another processor and hence create a massively parallel system, through components capable of interacting with each other. Petraglio details how to make multiplication and addition work together but it takes a lot of area and also a lot of time, and so is quite inefficient due to the usage of self replicating loops.

The fundamental objective considered when developing the binary tree adder was the need for the adder to be a system of interacting components rather than just a lone component capable of only a single addition. This process involves gaining a re-usable resultant and hence a processor capable of interpreting this resultant. This objective fosters the idea behind a truly parallel system rather than a parallel component.

Another fundamentally important task considered was the simplification of the system so as to make an easier transference into hardware and also to create the most efficient adder possible. The other substantial aspect was to try and decrease the computation time compared with Petraglio and hence make the adder more efficient in terms of time.

III. The Binary Tree Adder

As mentioned earlier there were a number of aims when starting out to produce the finished version of the binary tree adder. The essential issues thought of when designing the system were simplicity, efficiency, and parallelism.

The binary tree adder has developed upon the ideas of the particle machine, starting as a simple system, undergoing a number of design changes to eventually produce a highly parallel binary tree adder capable of efficiently performing large numbers of interacting computations.

A. Initial Stages of Development

The initial form of the adder functions in a similar manner to that of the particle machine. It is composed of a single data path with bi directional flow, a static processor cell, and a sheath, this is shown in Fig. 3, stage A. The adder functions by allowing the states representing binary ones and zeros to flow into the processing cell (state 3 at stage A). When the binary number representations collide at the processor, the processor cell's state is updated and it is then ready to output the resultant in a downward direction.

When designing the initial adder one of the main points to consider was to keep the concept of the particle machine as this is a highly efficient system. This was achieved through using the single data path with bi-directional flow. The data path has bi-directional flow to allow two addends to flow into the processor. The first addend (highlighted to the left of the processor in Fig. 3, stage A) moves to the right into the processor cell and the second addend (highlighted to the right of the processor in Fig. 3, stage A) moves to the left into the processor cell.

The main development over the particle machine is that a resultant comes out of the processor, as shown by Fig. 3, stage C, where the resultant has been highlighted. This is unlike the particle machine, where the result remains inside the structure of the machine. As can be seen from Fig. 3, the process takes 12 time steps to perform a 3-bit addition, including any carry bit calculation, which is highly efficient.

```
                                      STAGE A
10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10
14  12  14  17  14  2   14  1   14  2   14  11  14  3   9   11  9   2   9   1   9   2   9   17  9   12  9
10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10
                                      Time = 0
```

```
                                      STAGE B
10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10
14  14  14  14  14  14  14  14  14  14  12  14  17  7   17  9   12  9   9   9   9   9   9   9   9   9   9
10  10  10  10  10  10  10  10  10  10  10  10  10  1   10  10  10  10  10  10  10  10  10  10  10  10  10
                                                15  2   15
                                                    1
                                      Time = 9
```

```
                                      STAGE C
10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10  10
14  14  14  14  14  14  14  14  14  14  14  14  14  3   9   9   9   9   9   9   9   9   9   9   9   9   9
10  10  10  10  10  10  10  10  10  10  10  10  10  2   10  10  10  10  10  10  10  10  10  10  10  10  10
                                                10  1   10
                                                10  2   10
                                                10  1   10
                                                    10
                                      Time = 12
```

Fig. 3. Three stages of the initial adder.
(The numbers represent the states in the cellular automata space)

Some basic information about the states / system necessary to understand the functioning of the adder is as follows:

1) All collisions take place at the processing cell.
2) The processor cell only changes state and never location, it is completely static in this sense.
3) State 10 is a sheath state to keep the data path.
4) State 1 represents a binary zero, state 2 represents a binary one.
5) State 14 enables data flow to the right, state 9 enables data flow to the left.
6) State 15 allows the resultant to move downwards in the cellular automata space.
7) State 11 is the start bit, state 12 is the stop bit.
8) State 17 is the carry bit.

As can be seen from Fig. 3, two, 3-bit numbers are input least significant bit first to gain an output. Table one shows all possible collision permutations within the system at the processor cell (where CB = carry bit).

TABLE 1
THE PROCESSOR CELL INTERACTIONS

Calculation	Resultant
1 → Carry 0 ← 1	Output 0 Carry 1
1 → Carry 1 ← 1	Output 1 Carry 1
0 → Carry 0 ← 0	Output 0 Carry 0
0 → Carry 1 ← 0	Output 1 Carry 0
0 → Carry 0 ← 1	Output 1 Carry 0
0 → Carry 1 ← 1	Output 0 Carry 1
CB → Carry 0 ← CB	Output 0 Carry 0
CB → Carry 1 ← CB	Output 1 Carry 0
CB → Carry 0 ← 0	Output 0 Carry 0
CB → Carry 1 ← 0	Output 1 Carry 0
CB → Carry 0 ← 1	Output 1 Carry 0
CB → Carry 1 ← 1	Output 0 Carry 1

B. Intermediate Stages of Development (Simplification)

At this stage it was decided to optimise the simplicity of the adder structure, so as to allow for easier transference into hardware later on, this can be seen in Fig. 4. The processor cell was made a stand alone cell with a sheath (state 20) and the inputs were taken away from the processor. This was to allow other inputs to enter the same processor at later times during the life cycle of the system. The sheath was also removed from the inputs to decrease structural complexity, however this increased the number of transition rules. This in turn led to the introduction of state 21 in the input number. This extra input state tells the processor to output a downward mover state (state 22) to enable the resultant of the addition to propagate down through the automata space.

Fig. 4, shows 3 stages of the binary tree adder after intermediate development. The figure shows a 3-bit

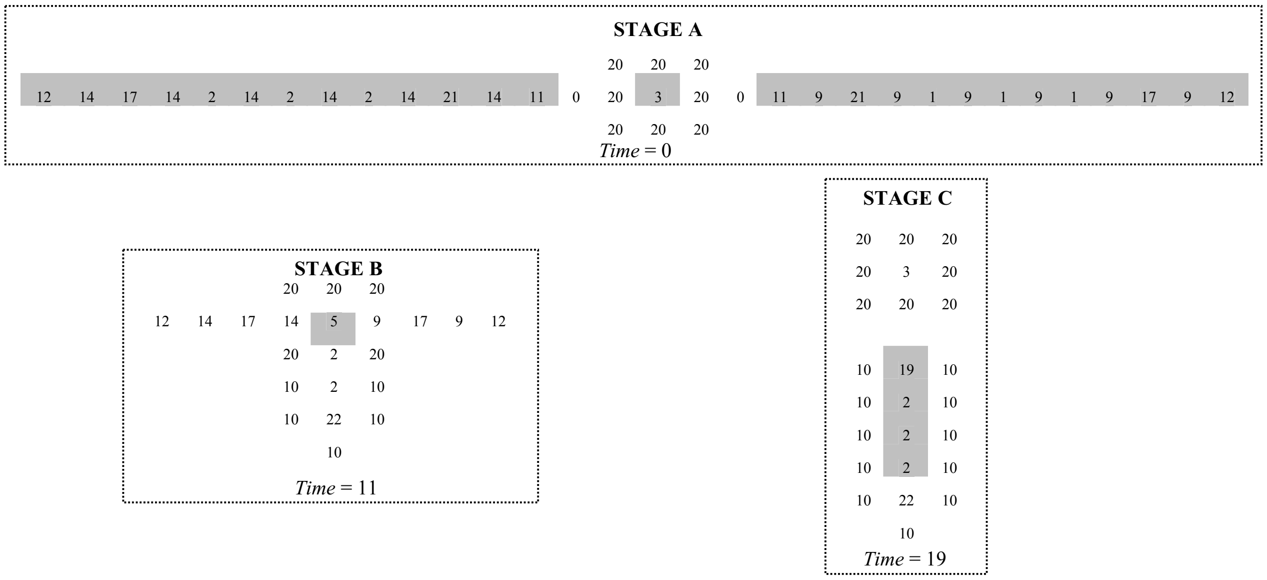

Fig. 4. The intermediate adder implementation, an example of a 3-bit binary addition.

calculation that takes 19 time steps to complete. At stage A the two inputs are highlighted to the left and right of the highlighted processor cell. The left input represents three ones and the right input represents three zeros. These two numbers then flow into the processor and produce the resultant, which is highlighted at stage C. The resultant is three ones and a nothing number state (state 19). The nothing number state is used when there is no carry bit to be output by the processor.

To be able to perform a series of additions using the same type of processor cell it was realised that the resultant summation of an addition needed modification. This is so that it can be used as input into another processor, the process behind this is shown in Fig. 6, and involved altering the output in the following way:

1) Adding extra states at the beginning and end so that the start bit, downward mover bit, carry bit, and stop bit are all present.
2) Removal of the downward mover state (state 22).
3) The internal mover states (9 or 14 depending on the direction required) needed to be added in between the other states.
4) Changing the direction of the resultant, to either left or right moving.

Performing the modification operation would lead to the resultant in Fig. 4, stage C to be converted as shown in Fig. 5. State 14 in the modified resultant could equally be state 9 depending on the direction in which the resultant needs to be turned. As can be seen, the resultant of three twos is maintained and the modified result is suitably ready to enter into another processor cell as an input.

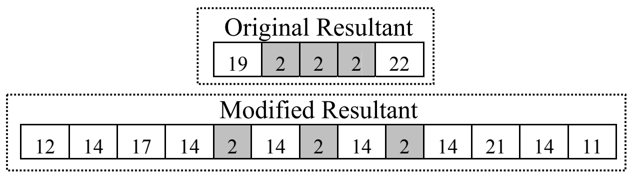

Fig. 5. An example of resultant modification.

C. Resultant Modification (Turning and Concatenation)

This section of development was important to allow resultants to be processed into a form able to be recognised by another identical processor. As mentioned previously this involved 4 stages. The entire process of concatenation of states and turning of the resultant is shown in Fig. 6.

The concatenation of states to the front of the resultant is shown in stages A, B, and C of Fig. 6. This was achieved by using the holder cells (state 25). They have been termed the holder cells as they simply hold a cell in place ready for it to be concatenated with the front of the resultant. It can be seen from stages A, B, and C that the states 21 and 11 are added to the front of the resultant.

The turning and addition of the internal movement states is achieved by using the wall like structure shown in stage C. The wall referred to is formed by the row of states (state 24, another state is used for left turning) highlighted at stage C. These are used to turn the resultant and add the necessary state in between depending on whether the resultant needs to be turned to the left (state 9) or the right (state 14), stages C and D show the resultant being turned to the right.

The addition of the last three states to the end of the resultant is achieved simply by passing the resultant over the three states at the end of the wall, as shown by stage E where one by one they are concatenated to the end of the resultant.

On completion of the outlined processes the resultant is then in a suitable form to be able to be processed by another processor cell and hence there is the underlying ability to create a parallel system of adders interacting with each other.

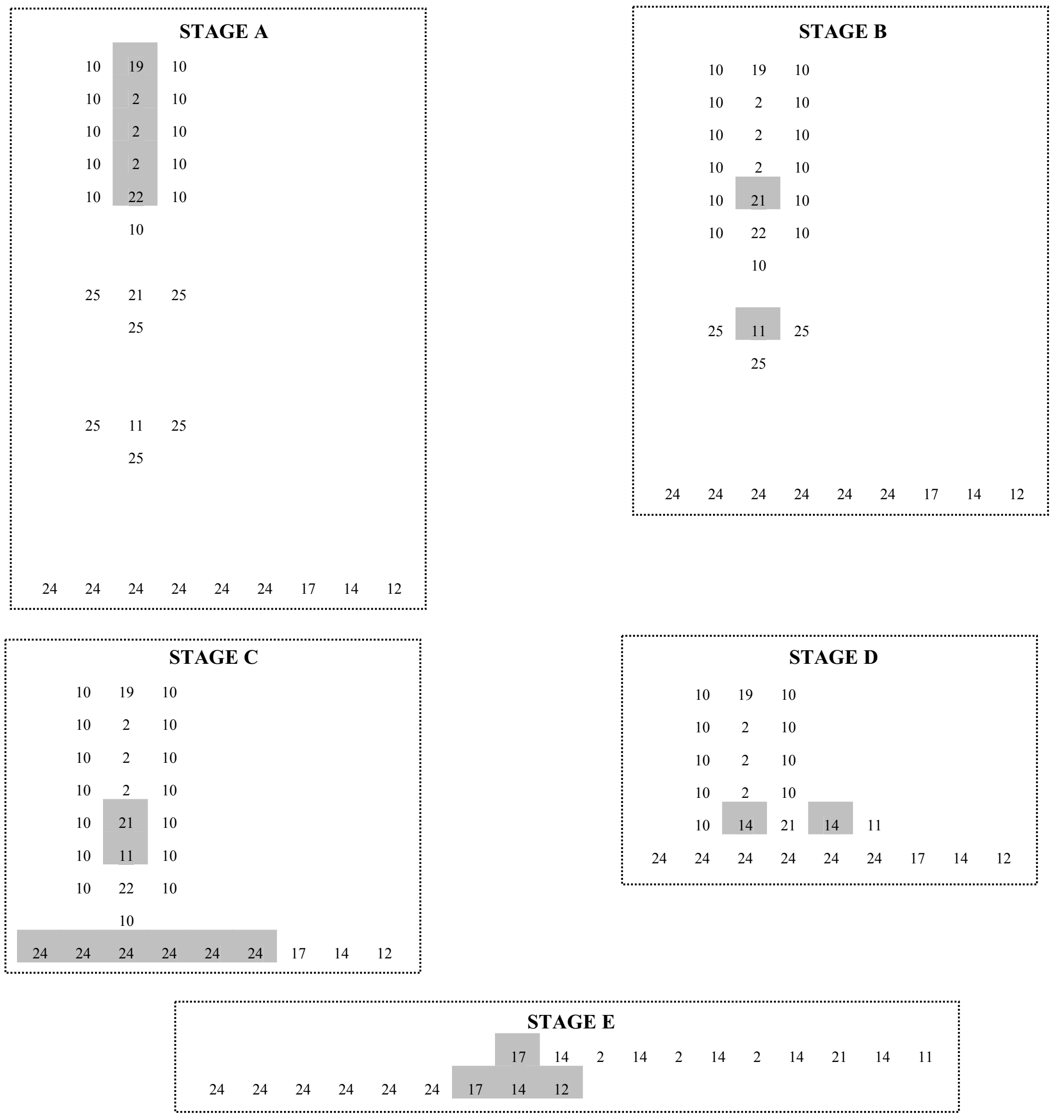

Fig. 6. The concatenation of front, rear, and directional states and turning of the resultant.

D. Final Version of the Binary Tree Adder

The final version of the adder[1] is a completely functional binary adder which has been termed the binary tree adder. It is possible to add any binary number of the same or differing lengths and also has a carry bit implemented for full functionality. A representation of the finished tree adder structure is shown in Fig. 7.

In Fig. 7, the processors are represented by squares so in the example shown there are three processors. The two at the top of the figure take in two inputs each and both produce a resultant. The resultants are then altered into the same format as the original inputs, through the concatenation and turning mechanism described earlier. Having completed this, the two resultants then flow into the final lower processor to be summed together.

The example in Fig. 7, is a simple example which can be extended. Theoretically a suitable parallel adder system could be conceived for any calculation required by using this process. This could involve any number of processors and inputs depending on the calculation to be performed and hence is a massively parallel adder system.

IV. Hardware Implementation

The structure of an FPGA is not dissimilar to the cellular framework of a cellular automata space and therefore provides an ideal platform within which to implement the binary tree adder. It has numerous advantages including the ability to locally connect cells and some similar properties (the cellular structure) allowing for the implementation of such a system.

[1] A video of an implementation of the binary tree adder can be viewed on the web at: http://www.ee.kent.ac.uk/research/theme_project.aspx?pid=62

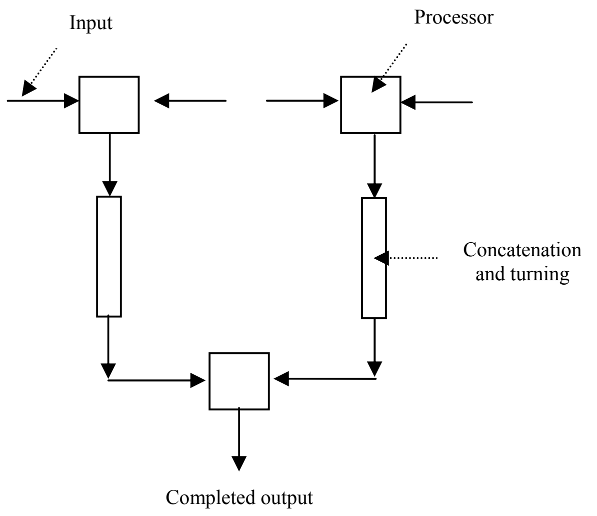

Fig. 7. A representation of the completed binary tree adder.

An FPGA has a number of advantages for implementing this type of hardware, for example, the availability of memory to store the transition rules in each cell. FPGA technology also brings with it a number of other benefits including the ability to reconfigure the system whenever may be required, and even the capability to partially reconfigure the FPGA, allowing different operations to be performed at the same time.

The initial idea for the design of the hardware is shown in Fig. 8. The figure shows the planned hardware design at the first stages of development, where an FPGA is used and made up entirely of cells equivalent to the cells in a cellular automata space. Fig. 8, shows, data cells (D), processor cells (P), turning cells (T), and resultant cells (R). These are used as simplified representations of the 25 data states that make up any implementation of a binary tree adder as presented here.

Each cell is likely to be composed of a couple of CLBs (configurable logic blocks) and should have the same capabilities as a cell in cellular automata. For example, each cell should know all of the transition rules and should be updatable in discrete time steps. Each cell will also need the ability to be in any one of a finite number of states, and have the flexibility to be locally connected with all of its neighbours.

The hardware design envisaged for the FPGA is based around splitting the FPGA in to different segments, performing completely separate operations that are possibly linked if the need arises. Fig. 8, shows an FPGA configured to perform three separate addition operations. Once each of these are performed a next batch of calculations may be loaded (through full FPGA reconfiguration). In theory new sections could be configured after each section is completed or on demand. A new section may be loaded (through partial reconfiguration) depending on factors such as the number of cells required by a calculation and the number of cells actually available on the hardware.

Having set up this concept of cellular automata within a cellular platform it is hoped that the next progression will be

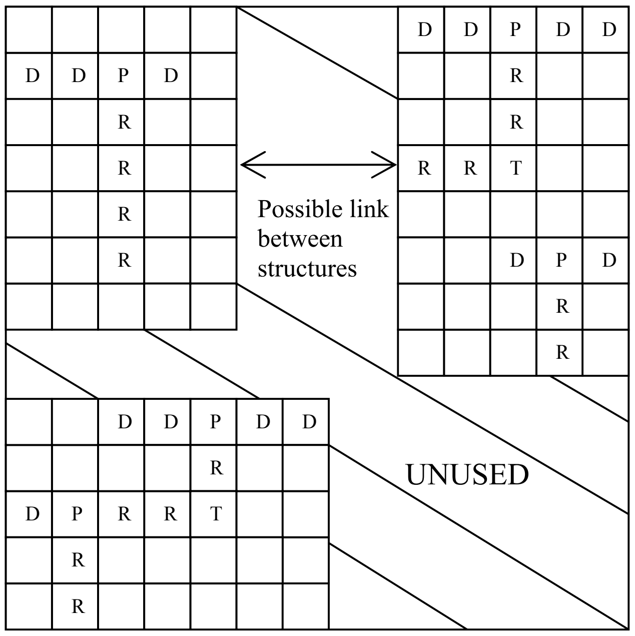

Fig. 8. An idea for the hardware implementation.

to introduce self repairing technologies to the system. This may be in the form of remembering the state of each cell at a previous time step so that if a cell malfunctions the system can be reconfigured to the previous time step, swapping the faulty cell for a spare operational cell.

Introducing self repairing technologies for the gain of increased robustness also leads into the field of fault tolerance for maximising robustness. The ambition with regard to this is to give rise to the possibility of performing the same additions simultaneously to check the validity of the resultants and hence the idea of detecting faults within the calculations.

These hardware design options bring about many interesting properties that may be present within the system including self healing and fault tolerance to increase robustness as well as the ease of design as all the individual cells are made up of the same components. The properties also bring a number of advantages including decreased design time and all of the advantages associated with vast parallelism.

V. Analysis And Comparison Of Results

Having created an implementation of a binary tree adder it is interesting to look back and compare this quantitatively with other models. This can be broken into a number of participant areas including, cost, in terms of area and the total number of time steps to completion, and complexity, in terms of the total number of states and the total number of transition rules. These are by no means the only way to judge a models complexity or cost and are merely a representative way of performing such a task where in fact other factors could be considered to be used for comparison and analysis.

The binary tree adder produces the result of a 3-bit addition using a maximum space (including input and output) of 29 cells by 10 cells. This is comparable indirectly with other models such as Langton's loop [6], which starts using a space of 10 cells by 10 cells and then replicates using ever greater amounts of space. Comparing the binary tree adder more directly to a model closer in purpose to itself shows the binary tree adder in a good light. Compared with [10] which details a method of binary addition using self replicating loops, a binary tree adder implementation is far more space efficient than using a self replicating loop. The area taken by any model is a necessary factor for consideration in theory if the model is to be transferred onto hardware. This is due to the inherent physical space constraints on all hardware.

In terms of the total number of states and the total number of transition rules the binary tree adder compares favourably to other models showing that it is not too complex. The binary tree adder uses 25 states (not including the quiescent state) and under 900 transition rules which is comparable with [8]. This model uses over 600 transition rules for self replication without any of the more complex operations being performed which would require more transition rules and more states. Comparing the binary tree adder to its closest model would again see the binary tree adder compared with [10] which uses over 30 states compared to the binary tree adders 25. The 30 states does however allow the multiplication operation to be performed which is not trivial. This is another important aspect to consider for transference of a model to hardware as there is always limited memory available on which to store the set of transition rules and the states associated with these.

One of the main areas of success of the binary tree adder is its time to completion of an operation (cycle time). As explained earlier a 3-bit addition is performed in merely 19 time steps which is highly efficient when it is considered that for a self replicating loop to fully replicate it takes over 100 time steps in both [6] and [8]. This means that the cycle time of [10] the most closely comparable model to the binary tree adder must have a cycle time of well over 250 time steps for a 3-bit calculation. This is due to the loop replication time.

This analysis and comparison shows that the binary tree adder compares favourably and is of significance when considering the implementation of cellular automata models within hardware. It shows the models suitability to hardware in comparison to other previous models including [9], the theory upon which the binary tree adder is based.

VI. Conclusion

The binary tree adder has proved to be an intriguing theory within cellular automata and also when referred to in the context of implementing the theory within a form of cellular hardware. The success with the adder stems from its ability of being a vastly parallel, highly simplistic, and efficient (in terms of time and space) structure.

The simplicity of the structure will allow for ease of transference onto hardware and also enables the efficient calculation of the addition operation compared with previous models.

The development of a multiplier in a similar context to the binary tree adder should allow for more complex arithmetic to be performed including operations like digital filtering. Another area for further development may be that of adding some form of communication within the system as at the moment the operations and movements must start at specific times in order for the operation to be performed correctly. This would add to the structural complexity but would increase the systems intelligence when performing such operations.

This implementation of a binary tree adder has been created using only synchronous cellular automata. It may be interesting to investigate the type of system presented here within asynchronous cellular automata. Asynchronous systems do not have some of the specific timing constraints of synchronous systems as outlined in [11] which details some of the advantages of implementing asynchronous cellular arrays within hardware.

The binary tree adder is also an interesting theory when it is thought about with regard to hardware as it introduces a whole number of interesting properties to systems including reduced design time, self repairing technology and fault tolerance. These properties help to increase the robustness of new hardware which is a property of great importance for the future, as we see the complexity of systems increasing.

References

[1] M. Sipper, "The emergence of cellular computing," *Computer*, vol. 32, pp. 18-+, 1999.

[2] L. J. K. Durbeck and N. J. Macias, "The Cell Matrix: an architecture for nanocomputing," *Nanotechnology*, vol. 12, pp. 217-230, 2001.

[3] G. Tempesti, D. Mange, and A. Stauffer, "Self-replicating and self-repairing multicellular automata," *Artificial Life*, vol. 4, pp. 259-282, 1998.

[4] J. Von Neumann, *Theory of Self-Reproducing Automata*. Urbana, IL: University of Illinios press, 1966.

[5] E.F.Codd, Cellular Automata. Academic Press, New York, 1968.

[6] C. G. Langton, "Self-Reproduction in Cellular Automata," *Physica D*, vol. 10, pp. 135-144, 1984.

[7] J. Byl, "Self-Reproduction in Small Cellular Automata," *Physica D*, vol. 34, pp. 295-299, 1989.

[8] G. Tempesti, "A new self-reproducing cellular automaton capable of construction and computation," in *Advances in Artificial Life*, vol. 929, *Lecture Notes in Artificial Intelligence*, 1995, pp. 555-563.

[9] R. K. Squier and K. Steiglitz, "Programmable parallel arithmetic in cellular automata using a particle model," *Complex systems*, vol. 8, 1994.

[10] E. Petraglio, J. M. Henry, and G. Tempesti, "Arithmetic operations on self-replicating cellular automata," in *Advances in Artificial Life, Proceedings*, vol. 1674, *Lecture Notes in Artificial Intelligence*, 1999, pp. 447-456.

[11] F. Peper, J. Lee, F. Abo, T. Isokawa, S. Adachi, N. Matsui, and S. Mashiko, "Fault-tolerance in nanocomputers: A cellular array approach," *IEEE Transactions on Nanotechnology*, vol. 3, pp. 187-201, 2004.