Radboud Universiteit

A Neuromorphic Implementation of a Distributed Shortest Path Algorithm

Course Project Neuromorphic Engineering
Radboud University Nijmegen

Author 1:
Thijs Luttikholt
t.luttikholt@student.ru.nl
Radboud University Nijmegen
Master AI - Intelligent Technology

Author 2:
Arne Diehl
arne.diehl@donders.ru.nl
Radboud University Nijmegen
Master AI - Cognitive Computing

January 4, 2022

Contents

Abstract

In recent years, brain-inspired neuromorphic computing architecture has gained popularity. This paper describes the implementation of a shortest path search algorithm on such a neuromorphic architecture via an Arduino device. A parallelized algorithm was designed and implemented by making use of, Arduino IDE, Digital and Quartus. Test results of the implementation show that the algorithm is effective in finding the cost of the shortest path between any two vertices in a pre-defined graph. Due to the various components in this project being generic, our intermediate results can be used in other circuits or projects.

keywords: neuromorphic architecture, shortest-path search, paralellized algorithm

Introduction

Programming on traditional von Neumann hardware constitutes the majority of all programming activity at this point in time. Programming languages such as Java, Python, Haskell, C# and Javascript, while following different paradigms all fundamentally presuppose the same type of computational formalism known as the Turing machine. Consequently most software written to date is conceived with these traditional models of computing in mind. This approach in combination with a steady pace of innovation in the area of hardware manufacturing has changed many professional fields and significantly improved productivity through means such as automation (Harris et al., 2009). However, despite this massive success of the traditional computing paradigm, there may be a limit to the growth in computing power that has continued through much of the past decades. According to computer scientists such as (Shalf, 2020) and (Kuhn & Padua, 2020), the pace of performance improvement for traditional hardware is about to slow down dramatically in the coming years. In anticipation of this slowdown, new types of architectures have started to gather attention. One of these architectures is the brain-inspired neuromorphic architecture.

As the name suggests, neuromorphic architecture is hardware that takes inspiration from the structure of the brain. This can take many shapes, but there are commonalities between the approaches that are currently labeled as being "neuromorphic". By parallelising computation and co-locating memory and computation, many of the neuromorphic architectures benefit from low energy consumption, fault-tolerant operations and are inherently suited for distributed and parallel processing (Nawrocki, Voyles, & Shaheen, 2016). Due to these benefits, the use of neuromorphic architectures is interesting for fields such as machine learning and other computationally intensive fields. Given the advantages in distributed processing, this hardware may even allow for algorithms that reduce the time complexity of common operations in these fields.

However, using this new architecture will require programmers to change their way of thinking about algorithms. In order to make use of the distributed processing, developers will need to conceive of algorithms in such a way that sub-processes can be performed in parallel rather than in sequence. An additional goal of this project was therefore to familiarize ourselves with this principle.

In this project, we design and implement an algorithm for shortest path search using a neuromorphic approach. The shortest path problem is a well-known problem in the field of artificial intelligence (AI). For solving this problem, several algorithms have been designed over the years. Examples of such algorithms are Dijkstra's algorithm (Dijkstra, 1959), the Bellman-Ford

algorithm (Bellman, 1958) and the Floyd-Warshall algorithm (Floyd, 1962). The goal of a shortest path search algorithm can be described as finding the lowest cost path between two vertices in a graph. Such a graph consists of vertices and edges, where edges have a cost value and are used to connect vertices. These edges can be either directional or non-directional, in the latter case one can travel in both directions along the edge. Whereas in the former case, the edge can only be traversed in one given direction.

This paper introduces an algorithm that was designed to work in parallel, in order to make use of the capabilities of a neuromorphic device. We will describe the algorithm, the implementation of the algorithm, the conducted tests, and discuss our implementation. Finally, we will describe possible improvements to the algorithm and give an indication as to how these improvements might be implemented.

Algorithm

In order to make use of the qualities of a neuromorphic device, the classical serial graph search approach needed to be rethought to fit a parallel process. Thus, we started by determining how to solve the problem of shortest path search in a parallel way. In this section, the implemented algorithm will be described in detail.

We first need to introduce several definitions which are needed for the algorithm description. We define the input to the algorithm to consist of a graph $\mathbf{G}$, an initial vertex **vs** and a terminal vertex **vt**. Secondly, we will define the output of the model to consist of a value representing the lowest path cost to get from **vs** to **vt**.

Given an input graph $\mathbf{G}$, consisting of vertices $\mathbf{V}$ and edges $\mathbf{E}$, we will define $\mathbf{e_{ij}}$ to denote the edge that goes from vertex i to vertex j. The cost value of this edge will be defined as $\mathbf{w_{ij}}$. For a vertex i, the set of vertices that can be reached by a direct edge from i will be defined as $\mathbf{N_i}$.

Next, we will define three functions to be used in the formal algorithm description. The first of these functions is $send(sender, receiver, message)$. This function is used to indicate that a sender vertex sends a message to a receiver vertex. For this function, the sender is a single vertex, while the receiver can either be a single vertex or a list of vertices. We will further work under the assumption that the message is a tuple of two elements, the first of which is a numeric value which represents the accumulated weight of the path while the second element is a list which represents the current path.

The second function is $receive(vertex)$, this function returns the lowest value the vertex in question has thus far received. In the case that a vertex has not yet received any message, this function will return a *none* statement comparable to that in the python programming language.

The third function is $rtrn(tuple, idx)$ which takes a tuple and an integer as input and returns the value at the specified place in the tuple.

It is important to note that we assume communication to be synchronized. We will refer to this communication as communication rounds, where theoretically all nodes can partake in such a round of communication with synchronized usage of the *send* function.

Using the above definitions, we define the

algorithm as follows:

1. $\forall j \in N_{\mathbf{vt}} : send(\mathbf{vt}, j, \mathbf{w}_{\mathbf{vt}j})$
2. $\forall r \in \{v \in V | receive(v) \neq none\} :$
   $\forall j \in N_r :$
   $t_1 \leftarrow rtrn(receive(r), 0) + w_{rj},$
   $t_2 \leftarrow rtrn(receive(r), 1) + r,$
   $send(r, j, (t_1, t_2))$
3. Repeat step 2 for $|V| - 1$ steps
4. $receive(\mathbf{vt})$

In words, this algorithm operates as follows: in step 1, the initial vertex sends a message to each of its neighbouring vertices. This message will contain a value equal to the cost of the edge between the initial vertex and each individual neighbour. In step 2, each vertex that has already received a message will send a message to their neighbours. This message will contain a value of the edge cost, added to the value of the lowest-value message the vertex in question had received at that point. In addition to the accumulated path weight, every message will also contain the current path. Step 2 will be repeated for an amount of times equal to the amount of vertices minus one. The function $receive(\mathbf{vt})$ then returns the final result.

The time complexity of this algorithm is $\mathcal{O}(|\mathbf{V}|)$, where $|\mathbf{V}|$ is the size of the set of vertices in the input graph. Note however, that this algorithm expects the input graph to be within a distributed network to begin with.

Implementation

For the implementation of the algorithm, Digital (Neemann, 2016) was used to create and test the circuit. Subsequently, the circuit was exported to verilog and incorporated in a Quartus project. The project connects the circuit with a JTAG component (HerrNamenlos123, 2020). These combined components were exported to a bitstream and then reversed (Diehl, 2021). The reversed bitstreams are compatible with the MKR Vidor 4000 by Arduino. This Arduino board comes equipped with the Cyclone 10 neuromorphic chip developed by Intel.

To utilize the developed circuits, we wrote a short program for the Arduino IDE, which takes user input and sends it to the Arduino board, which in turn sends the input to the FPGA, which then performs the algorithm and returns an answer to the Arduino, which finally sends the answer to the host computer.

In the remainder of this section, we will describe the implementation of the algorithm in Digital, as well as the implementation of additional parts in Quartus and the Arduino IDE. For this description, we will divide the implementation along its hierarchical structure, beginning with the lowest level circuits.

Base of the model

In this section, the base parts of the model will be described. This includes the inter-vertex connections, as well as the vertex module itself.

Incoming edges

In order to represent the incoming edges of a node, a generic Digital module was created. This module makes use of a parameter to set the amount of incoming edges for a node. Using this value, a specific circuit is created, which has as primary function to choose the lowest incoming path cost. The way this is performed is by pairwise comparison of path values and passing on the lower value to the next comparison. The path list is passed along as well. The reason for this reduction is to limit the amount of inputs as early as possible. In our model, only the lowest input will be kept in memory, and has a chance of being part of the

eventual output. Therefore, we can ignore any other inputs in the wires component, since they are not required for the later calculations.

In the image below, we see the generic circuit that is created by setting the input parameter to a value of 5:

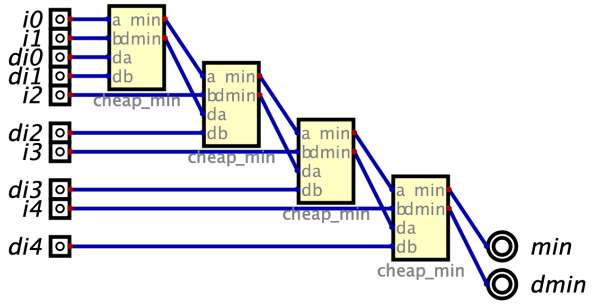

Figure 1: The wires_in component

As can be seen in figure 1, this component makes use of a second component called the cheap_min. This second component was created for this project as well. It takes four inputs, and outputs the lowest value out of the two inputs, as well as the current path list. The image below shows the implementation of this component:

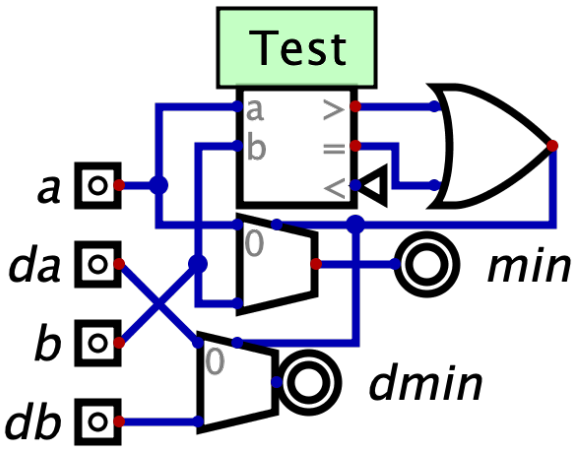

Figure 2: The cheap_min component

As can be seen, the two inputs are fed into a comparator, followed by an OR-gate. The result of which is 1 in the case that input a is higher than or equal to input b, otherwise it is 0. This result is used to indicate to a multiplexer which one of the inputs should be chosen. This value is subsequently fed to the output and a multiplexer. This multiplexer is making sure that the correct path list is passed to the output.

Basis of the vertex module

Next, we will describe the expected functionality of the base vertex module. The module should take one input. In the case that this input is lower or equal than the value currently stored in memory, the memory is overwritten to be equal to the input value. Lastly, at any time, the module should output the value that is stored in memory at that time. At the same time, when the weight memory is updated, the path memory saves the incoming path list as well.

In the image below, the Digital implementation of this module is shown:

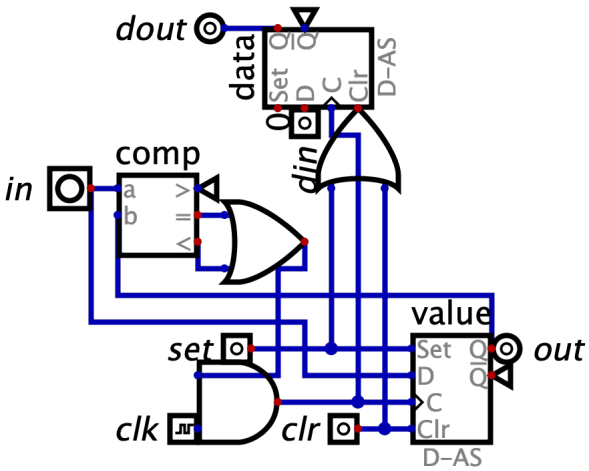

Figure 3: The decreasing_memory component

In the image, we see that the input is fed to both a comparator and an asynchronous D flip-flop. In the comparator, only the second and third output is used, and fed into an or gate. This gate returns 1 if the input is lower or equal compared to the stored

value, and 0 otherwise. The output of this or gate and a clock signal is fed into an and gate. This modulated clock signal is connected to the weight flip-flop. At a positive edge of the clock signal, the memorized value is overwritten by the input at that time. Given the logic for the and gate, only when the incoming signal is sufficiently low will the flip-flop receive a rising edge on the clock input. Similarly, this circuit also stores the incoming path list in memory, whenever the incoming weight value is sufficiently small.

Outgoing edges

Next, we describe the part that represents the outgoing edges of a vertex. For this component, the intended functionality will be shortly described. The component should take two inputs, and should output N+1 separate values, corresponding to the N vertices the vertex is connected to in addition to the path list. The component was made generic, and makes use of two parameters that determine the exact generic circuit. The two parameters are integer values indicating the amount of outgoing edges, and a list of weight values, with a length equal to the amount of outgoing edges.

In figure 4, the generated circuit is shown in the case of three inputs, with corresponding weights of 1,2 and 4 respectively.

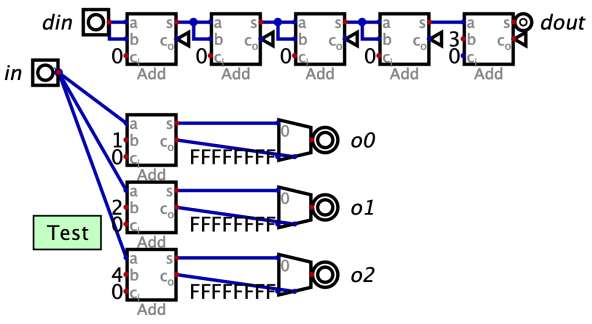

Figure 4: The wires_out component

As can be seen, the component takes one input for the path weight update (t_1 in the algorithm) and one input for the path list update (t_1 in the algorithm). The path weight is increased by the edge weight for every edge that we will send the message towards. The output of this add operation is multiplexed in such a way that if an overflow happens, the highest possible 32 bit number ($2^{32}-1$) is returned, otherwise the output of the add operation is returned. This is to safeguard against overflown numbers distorting the results. This implementation means however that if the result is $2^{32}-1$, then we cannot deduce anything about the shortest path, because we have reached the numeric limit of this hardware implementation. Only smaller results can therefore be actual outcomes. The path list update happens by shifting the "din" 32bit input to the left by 4 bits and then adding the index of the current node to the number as well. This way, we can understand the output as a concatenation of 4 bit node indices that represent the path. These two unconnected components are in the same submodule as they are both needed for the output of a node module.

Combining the previous components

The three main components that were just described, namely the input wires, vertex basis, and the output wires were combined to form the node_module that forms the basis for the creation of a full graph. This combination was once again implemented as a generic circuit in Digital, using two parameters. The first parameter is the amount of neighbourhood vertices, which sets the amount of both input and output wires, meaning that this module assumes that every edge is bidirectional. The second parameter is the list of edge weights, which is used for the output wires as described earlier. In figure 5 below, we can see the circuit that is created when the vertex is supposed to have neighbourhood size of 2.

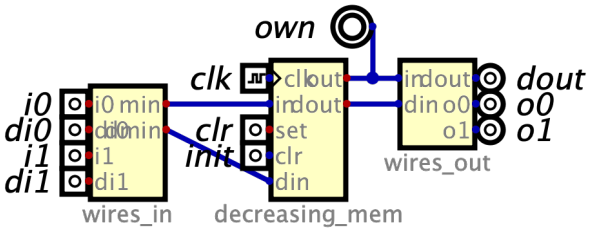

Figure 5: The node_module component

In figure 5, we can see the components that were described earlier linked together in one component. These are the input wires, the vertex basis, and the output wires. The input called "init" is a binary value, representing whether the current vertex is chosen as the initial vertex. If this input is high, the memory will be set to zero, since the distance from a vertex to itself should be zero. The input "clr" is used to "clear" the internal memory, which means that it will set its weight memory to the maximum value and its path list memory to the lowest value. The output called "own" is an additional output, which corresponds to the stored memory value. This is used to read out the final value of the component at the end of the total shortest path length determination process. All other inputs, and outputs aside from the clock signal are needed for node to node communication.

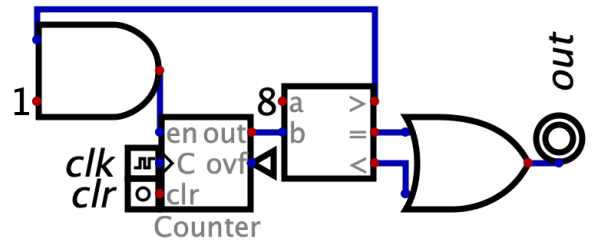

Figure 6: The final_countup component.

Combining Vertices to Make a Graph

The description of the node_module above facilitates the implementation of step 1 and step 2 of our algorithm. To realise step 3 and 4 of the algorithm however, we will have to add a counter component and a component that extracts the shortest path length, as well as the actual shortest path. Furthermore, in order to fully implement this algorithm, we also need to formalise how the input and the output are handled.

Countup

In figure 6 the final_countup component schematic is visualised. This component takes no input currently, but is generated by changing the input on the comparator circuit to the required value. In this case the countup circuit will count up to the value we set at the comparator circuit, hence why we refer to this value as "countup time". In the context of our algorithm, this value is set to the amount of vertices in the graph. In words the final_countup component uses a counter circuit to count from 0 to the countup time. Once the counter circuit has reached the countup time, the counter circuit stops counting and the single bit output becomes high.

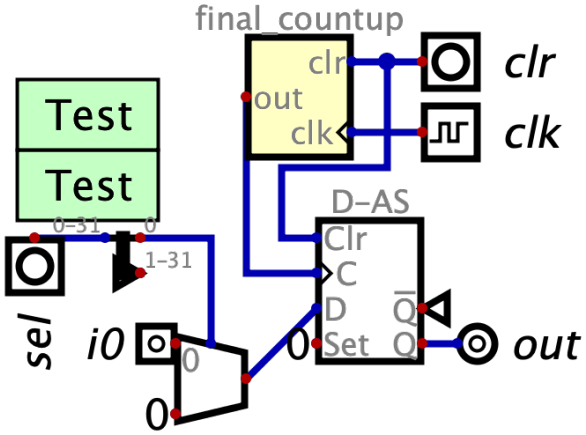

Figure 7: The Digital schematic of the extract_path component.

Extract shortest path

In figure 7 we can see the schematic of the extract_path component. This component is used to extract the path length from the terminal vertex. The component is written to be generic such that it can be instanti-

ated to handle variable amounts of inputs. Essentially, this component waits until every vertex has received every possible message and then outputs the value present in the selected (terminal) vertex. If the implementation of the base model was correct, then this value should indeed be the length of the shortest path. The value that the final_countup component counts towards, is the amount of incoming signals minus one. This component is used for both the extraction of the shortest path itself and its weight.

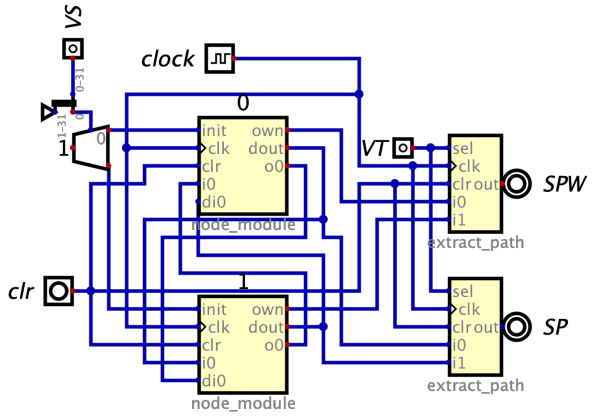

Figure 8: The Digital schematic of an example graph implementation.

Graph Implementation

Figure 8 contains the implementation of a simple graph with two vertices and a single edge with weight 4. The single bit input labelled "clr" is used to suppress the clock until the circuit is supposed to run. The input "VT" is used to designate **vt**, the terminal vertex, whereas "VS" is used to designate the initial vertex **vs**. The extract_path component sets the final_countup component to the correct amount of time needed for a graph with two vertices, which is equal to the amount of nodes in the graph. After the path search has completed, 32 bit output SP displays the shortest path, while SPW displays the total weight of the shortest path.

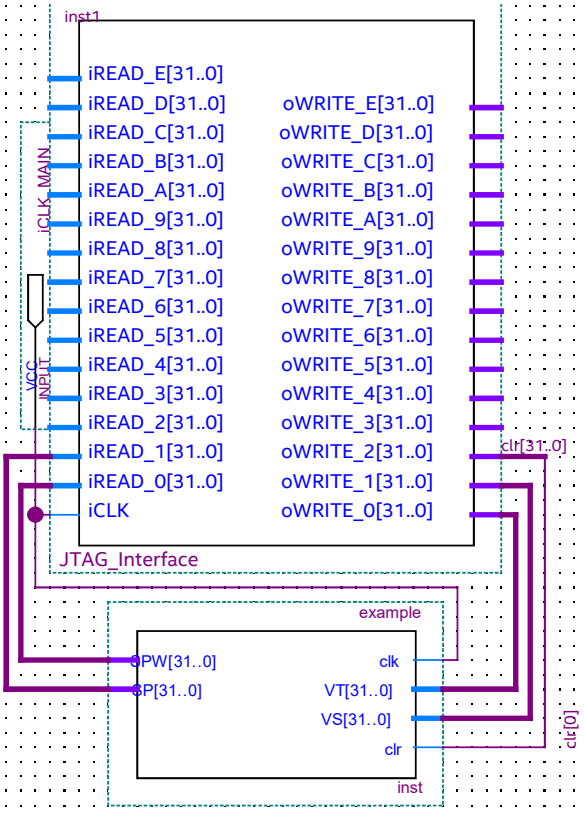

Figure 9: The Quartus block design schematic of an example graph implementation.

JTAG connection

The connection to the JTAG project (HerrNamenlos123, 2020) is described in figure 9. That figure shows how the example graph from figure 8 and figure 11 is being connected to the main JTAG component. The clock component called "iCLK MAIN" is connected to the clock "wCLK120" which is a 120MHz clock. Furthermore, all inputs and outputs are connected to the JTAG component. These can be read by the cpu on the board and subsequently sent to the host computer that the board is connected to. The Arduino code that performs this communication can be summed up as follows:

```
writeJTAG(2, clear=1);
writeJTAG(1, initial_vertex);
writeJTAG(0, terminal_vertex);
writeJTAG(2, clear=0);
delay(1);
```

```
shortestPathLength = readJTAG(0);
shortestPathList = readJTAG(1);
```

The complete code can be found in our github repository (Luttikholt & Diehl, 2021).

Testing

Every component has been rigorously tested to perform as expected. The test cases presented here are handpicked to show correctness without sacrificing readability. The extended test results can be found in our github repository (Luttikholt & Diehl, 2021). The test cases were written and performed in Digital, as they had to work within the context of Digital's non-standard generic components. Only the test cases of non-generic components such as the example graph implementation can be exported as a verilog testbench. The test cases themselves can also be found in our github repository (Luttikholt & Diehl, 2021), as they are part of the .dig files in which the circuits are stored.

Table 1: Timing results in final_countup.

clk	clr	out	Correct
0	1	1	✓
1	0	0	✓
2	0	0	✓
3	0	0	✓
4	0	1	✓
5	1	0	✓
6	0	0	✓
7	0	0	✓
8	0	0	✓
9	0	1	✓

final_countup

To test the behaviour of the final_countup component (see figure 6), a test has been written in which the correct timing and reset behaviour can be observed. The results are shown in table 1. The component was generated with a counter value of 4.

Table 2: Timing & correctness results in extract_path.

clk	clr	sel	i0	i1	out	Correct
0	1	0	0	0	0	✓
1	0	0	0	0	0	✓
2	0	0	0	0	0	✓
5	0	0	1	0	1	✓
8	0	1	4	8	8	✓

extract_path

Test results for the extract_path component (see 7) tests can be found in table 2. Note that the table shows one timing result as well as some additional correctness results. The component was generated with two inputs.

Table 3: Correctness results in wires_out.

in	o0	o1	o2	din	dout	C.
0	1	2	4	0	4	✓
10	11	12	14	1	20	✓

wires_out

The correctness tests of the component called "wires_out" (see figure 4) can be found in table 3. The component was generated with 3 outputs and weights 1, 2 and 4.

Table 4: Correctness results in decreasing_mem.

clk	clr	set	in	out	di	do.	C.
0	1	0	10	0	15	0	✓
1	0	0	10	0	15	0	✓
2	0	1	10	$2^{32}-1$	15	0	✓
3	0	0	10	10	15	15	✓
4	0	0	50	10	77	15	✓
5	0	1	50	$2^{32}-1$	77	0	✓
6	0	0	50	50	77	77	✓
7	0	0	10	10	15	15	✓

decreasing_mem

The sequential correctness results for the component called "decreasing_mem" (see figure 3) can be found in table 4.

Table 5: Correctness results in cheap_min.

a	b	a	b	out	dout	Correct
0	0	0	0	0	0	✓
0	2	0	0	0	0	✓
2	0	0	0	0	0	✓
2	2	5	7	2	7	✓

cheap_min

Correctness test results of the component called "cheap_min" (see figure 2) can be found in table 5.

Table 6: Correctness results in wires_in.

i0	i1	di0	di1	min	dmin	C.
5	5	5	5	5	5	✓
5	3	3	5	3	5	✓
0	0	0	0	0	0	✓

wires_in

The correctness results of the component called "wires_in" (see figure 1) can be found in table 6. The component was generated with 5 inputs.

node_module

The correctness results of the component called "node_module" (see figure 5) can be found in table 9 in the appendix. The circuit was generated with two inputs, two outputs and with weights 1 and 2 for the outgoing edges.

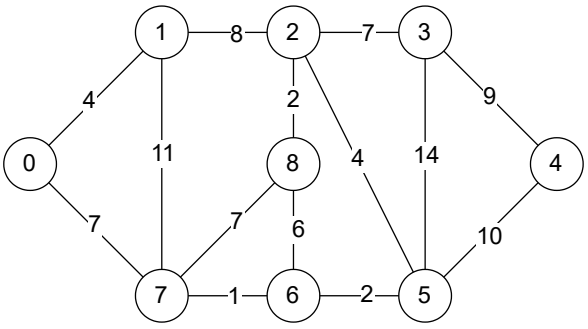

Figure 10: The example graph that was used to test our implementation.

Example Graph

The example graph is an implementation of the graph in figure 10. The implementation itself can be seen in figure 11. The important behaviour of this component is to return the minimum path and its length between the designated vertices as seen in the graph. This requires the component to output a zero for 8 clock cycles and subsequently return the shortest path. This is because the algorithm requires exactly $|V|$ clock cycles where V is the set of vertices in the graph. Thus to show the correct timing of the graph, the test results in table 7. Test results that show the correctness of the output itself can be found in table 8. Note that the clock signal is increasing by 10, as it takes one clock cycle to reset the circuit and 9 cycles to produce the result.

Table 7: Timing results in example graph.

clock	clr	**vs**	**vt**	spw	sp	C.
0	1	0	1	0	0	✓
1	0	0	1	0	0	✓
2	0	0	1	0	0	✓
3	0	0	1	0	0	✓
4	0	0	1	0	0	✓
5	0	0	1	0	0	✓
6	0	0	1	0	0	✓
7	0	0	1	0	0	✓
8	0	0	1	0	0	✓
9	0	0	1	4	1	✓

Table 8: Outputs of example graph

clock	clr	**vs**	**vt**	spw	sp	C.
19	0	0	2	12	18	✓
29	0	0	3	19	291	✓
39	0	0	4	21	30292	✓
49	0	0	5	11	1893	✓
59	0	0	6	9	118	✓
69	0	0	7	8	7	✓
79	0	0	8	14	296	✓
89	0	1	0	4	16	✓

Discussion

The rigorous testing of all the created components showed that we have correctly created circuits that together implement a distributed algorithm calculating the minimum path in graphs.

Our implementation has some limitations that are worth mentioning. It is for example not possible to dynamically change the graph that the algorithm will make use of. For each graph, a circuit consisting of multiple sub components has to be created, which is a time consuming task. In future work, this may be alleviated by either creating generic graph components in Digital, or by using large fully connected graphs which can be pruned at run-time. In doing so, the current generic sub components of the graph could be reused.
Another limitation is that the edge weights are limited to integer values in the range of 0 to $2^32 - 1$. This implies that we cannot represent floating point edge weights and are not able to handle edge weights outside of the specified range. These limitations are due to the choices we made in the types of circuits that we use and could be improved in future work by increasing the bit size of most components and implementing some kind of quantization scheme for floating point values. These limitations do not pose insurmountable challenges, but fall outside the scope of this project. Our implementation succeeds in performing our algorithm in parallel, despite these limitations.

Another challenge encountered during the creation of our implementation was to get the timing of the communication rounds correct. It's very hard to make sure that all components communicate at the exact same time and it seems that if we would scale up our solution, these timing issues would pose a threat to the correctness of the results. In our example graph however, this did not pose a significant problem.

Conclusion

During this project, we have implemented a shortest-path search algorithm on a neuromorphic device. Although limitations to the implementation still exist, the testing results show that the program is effective in fulfilling its task. The correct output is provided for the shortest path between any two vertices in the pre-defined graph. We can conclude based on this study that the Arduino device can be used to perform a shortest path search algorithm.

References

Bellman, R. (1958). On a routing problem. *Quarterly of applied mathematics*, *16*(1), 87–90.

Diehl, A. (2021, October 18). *Quarev telegram bot.* Retrieved from `https://github.com/adiehl96/quarevTelegramBot`

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, *1*(1), 269–271.

Floyd, R. W. (1962). Algorithm 97: shortest path. *Communications of the ACM*, *5*(6), 345.

Harris, F. C., Lee, G. K., Rubin, S. H., Ting, T. C., Gaston, B., & Hu, G.

(2009). Impact of computing on the world economy: A position paper. In W. Li (Ed.), *Proceedings of the ISCA 24th international conference on computers and their applications, CATA 2009, april 8-10, 2009, holiday inn downtown-superdome, new orleans, louisiana, USA* (pp. 271–277). ISCA.

HerrNamenlos123. (2020). *Jtag_interface.* Retrieved from `https://github.com/HerrNamenlos123/JTAG_Interface/`

Kuhn, R., & Padua, D. (2020).

Luttikholt, T., & Diehl, A. (2021). *Neuromorphic engineering.* GitHub. Retrieved from `https://github.com/adiehl96/NeuromorphicEngineering` doi: 10.5281/zenodo.5818863

Nawrocki, R. A., Voyles, R. M., & Shaheen, S. E. (2016). A mini review of neuromorphic architectures and implementations. *IEEE Transactions on Electron Devices*, *63*(10), 3819–3829.

Neemann, H. (2016, March 11). *Digital.* Retrieved from `https://github.com/hneemann/Digital`

Shalf, J. (2020). The future of computing beyond moores law. *Philosophical Transactions of the Royal Society A Mathematical Physical and Engineering Sciences*, *378*(2166), 20190061. Retrieved from `https://app.dimensions.ai/details/publication/pub.1124197943` (https://escholarship.org/content/qt25b4s3dp/qt2 doi: 10.1098/rsta.2019.0061

Test results of node_module

Table 9: Correctness results in node_module.

clk	clr	init	i0	i1	di0	di1	own	o0	o1	dout	Correct
0	1	0	10	10	66	77	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	3	✓
1	0	0	10	10	66	77	10	11	12	1235	✓
2	0	0	10	5	66	77	5	6	7	1235	✓
3	1	0	10	5	66	77	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	3	✓
4	0	0	50	50	66	77	50	51	52	1235	✓
5	0	0	50	3	66	77	3	4	5	1235	✓
6	0	1	50	3	66	77	0	1	2	3	✓
7	1	0	50	3	66	77	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	3	✓
8	0	0	50	3	66	77	3	4	5	1059	✓

Example Graph Implementation

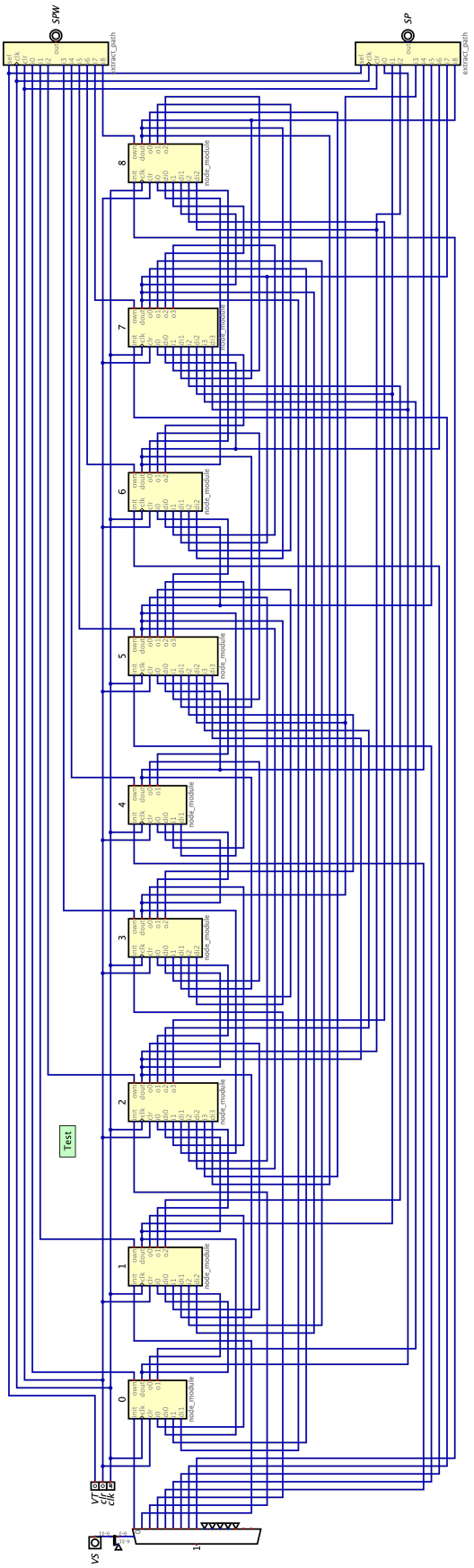

Figure 11: The implementation of the example graph.