An Exploration of Array Sorting in Hardware and Software

Anders Mæhlum Halvorsen Rahmat Mozafari Ole Martin Ruud Østlem Tuzkaya

21.09.2020

Abstract

We explored and implemented three different sorting algorithms as software (single-thread C programs) and hardware (VHDL) implementations in FPGAs. Furthermore, we compared the different implementations with regards to efficiency, performance, flexibility, resource usage and code complexity, while particular contrasting hardware and software trade-offs. We found that concurrent sorting algorithms can be parallelized in hardware to achieve higher performance, however hardware implementations require more development effort, have more code complexity and also require more hardware resources. Hence there is a trade-off to be made between multiplexing the algorithm in time or in space, depending on the context of the application.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

The level of concurrency varies among array sorting algorithms. In some cases it is low, e.g. selection sorting, but in other cases, it is higher, e.g. odd-even sorting. As Vasquez' shows in his Hackaday article [1], we can utilize field-programmable gate arrays (FPGAs) as a platform to implement parallelized versions of concurrent sorting algorithms. Further Skliarova [2] shows an example of implementing a concurrent sorting network on an FPGA. In both cases, one can take advantage of the parallel nature of hardware to increase the performance of the algorithms.

Depending on the context of the application, choosing the algorithm and its implementation with care can be crucial to fulfil the given requirements. Different algorithms and implementations can have widely different properties when it comes to performance, efficiency, flexibility, resource usage and code complexity. Understanding these trade-offs is valuable to develop highly effective and well-engineered applications.

We will explore and implement three different sorting algorithms as software (single-thread C programs) and hardware (VHDL) implementations in FPGAs. Furthermore, we will compare the different implementations with regards to efficiency, performance, flexibility, resource usage and code complexity; in particular contrasting hardware and software trade-offs.

Chapter 2

Methods

The tools used in this paper was Vivado 2020.1, Vitis 2020.1 and the Zybo Zynq-7000 board [3]. We used Vivado for the hardware implementations and we programmed the Zynq-7000 board. For the software and IP implementation, we used the Vitis IDE, where we also used the Zynq-7000 board for testing.

For all of our algorithms, we followed the same steps. We started by creating a finite-state machine with datapath (FSMD) architecture of the overall algorithm we were currently building; we did this to get an overview of what components and signals were needed. Based on the FSMD architecture created, we designed the algorithmic state machine with datapath (ASMD) chart. This was done to easily convert the chart into code when implementing the algorithm, while also having a good overview of the states needed.

After finishing making all of the necessary charts, we then started to implement the sorting algorithm in Vivado. To make the code as reusable as possible, we created an individual file for each of our components. We also made the last two algorithms generic, such that the amount of inputs and sizes could be easily adjusted by the user.

To confirm that our sorting algorithms worked as expected, we created a test bench and analysed the outputs.

The software implementation, in contrast to the hardware implementation, was a much more straight forward process. Firstly, we created a generic block diagram with the Zynq processor unit in Vivado. Then after uploading it to Vitis, we simply connect the Zybo board to the computer and started running the software code. For testing, we used the built-in Vitis serial console.

For our first algorithm, we made an effort to create an IP block. This turned out to be an immensely time-consuming process due to a lot of troubleshooting. Seeing that the implementation of the IP would not have had a significant impact on our vision or result for our paper, we chose to exclude it from consideration. However we left the results for the particularly interested in appendix D.

Chapter 3

Results

In this section, we are going to presenting the results we gathered through to applying our methods. Firstly we will give a quick overview comparable results from Vivado, and then we will go into each algorithm and the corresponding software and hardware implementations.

3.1 Overview of implementations

The different implementations were synthesized and simulated using Vivado 2020.1. The most of the results here were gathered through the Vivado interface using the provided analysis tools.

In tbl. 3.1 we can see the overview of the number of files and lines of code for each implementation. The number of lines can be an indicator of complexity, but it must be used carefully as code can, of course, be optimized for size. In our case, we have tried to follow a consistent formatting style in addition to not optimizing for size. Another important note is that we are only counting lines of code, not comments or blank lines, and also only code which is part of the implementation of the algorithm, so no test benches or code for displaying arrays etc.

Table 3.1: Amount of source code for each implementation

Implementation	Files	Lines
Selection sort (hardware)	7	270
Linear cell sort (hardware)	8	360
Odd-even sort (hardware)	4	216
Selection sort (software)	1	14
Linear cell sort (software)	1	21
Odd-even merge sort (software)	1	40

In tbl. 3.2 and fig. 3.1 we can see the usage of hardware resources for each algorithm across multiple input sizes. This was done to enable us to see the increase in hardware resources as the amount of signals to be sorted was increased. It's worth noting that each measurement was done with a data size of 8 bytes. Further in fig. 3.2 we can see the on-chip power usage of each implementation.

Table 3.2: Overview of cells, IO-ports, nets, LUTs and FFs used across multiple input sizes for hardware implementations

Implementation	Input size	Cells	IO-ports	Nets	LUT	FF
Selection sort	4	10	15	51	18	25
	8	10	27	87	37	45
	16	10	51	159	72	85
Linear cell sort	4	9	12	46	41	27
	8	13	20	112	119	81
	16	21	36	338	404	283
Odd-even sort (N layers)	4	25	37	120	86	17
	8	81	133	712	860	65
	16	289	517	4872	6744	258
Odd-even sort (3 layers)	4	24	37	104	78	17
	8	76	133	392	431	65
	16	276	517	1544	1583	257

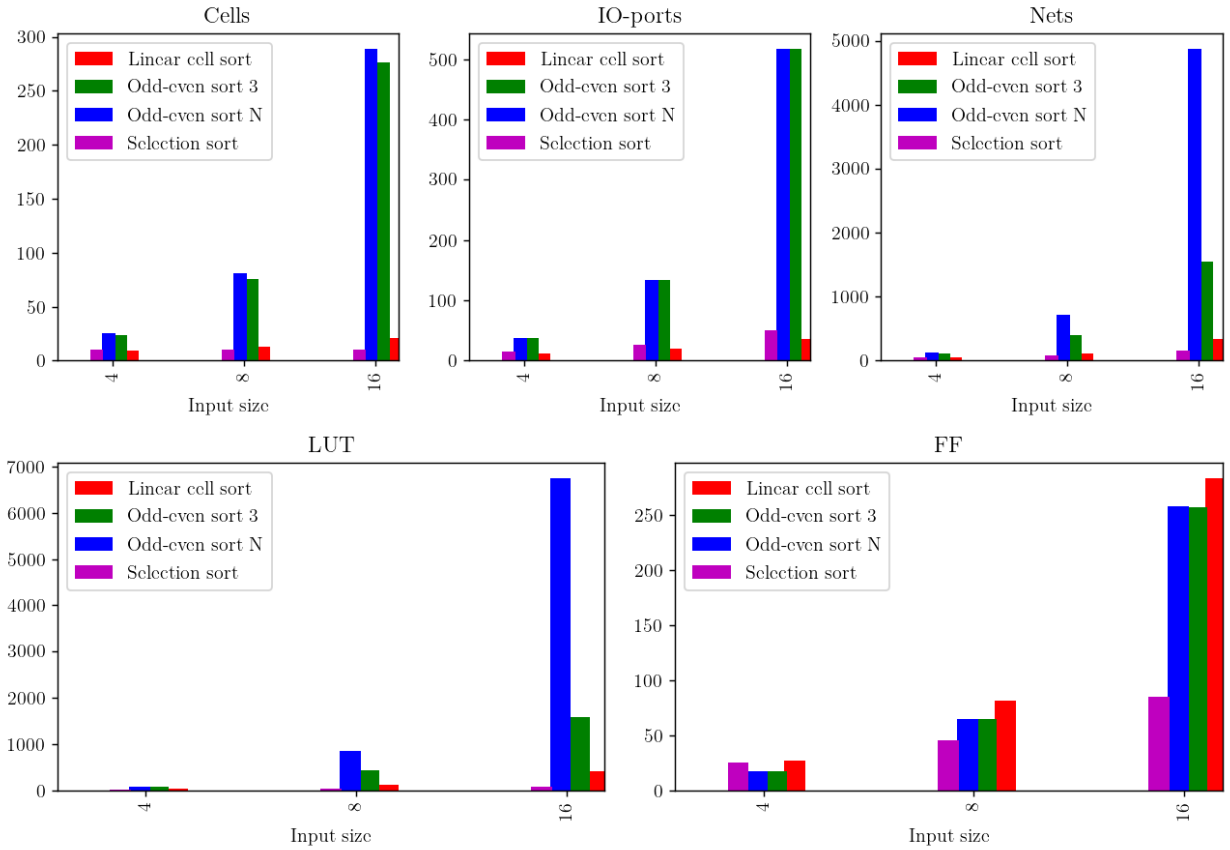


Figure 3.1: Graph of cells, IO-ports, nets, LUTs and FFs used across multiple input sizes for hardware implementations

In tbl. 3.3 we can see the best and worst time complexity of each implementation. It's worth noting that the time complexity for an implementation isn't necessarily the same as the time

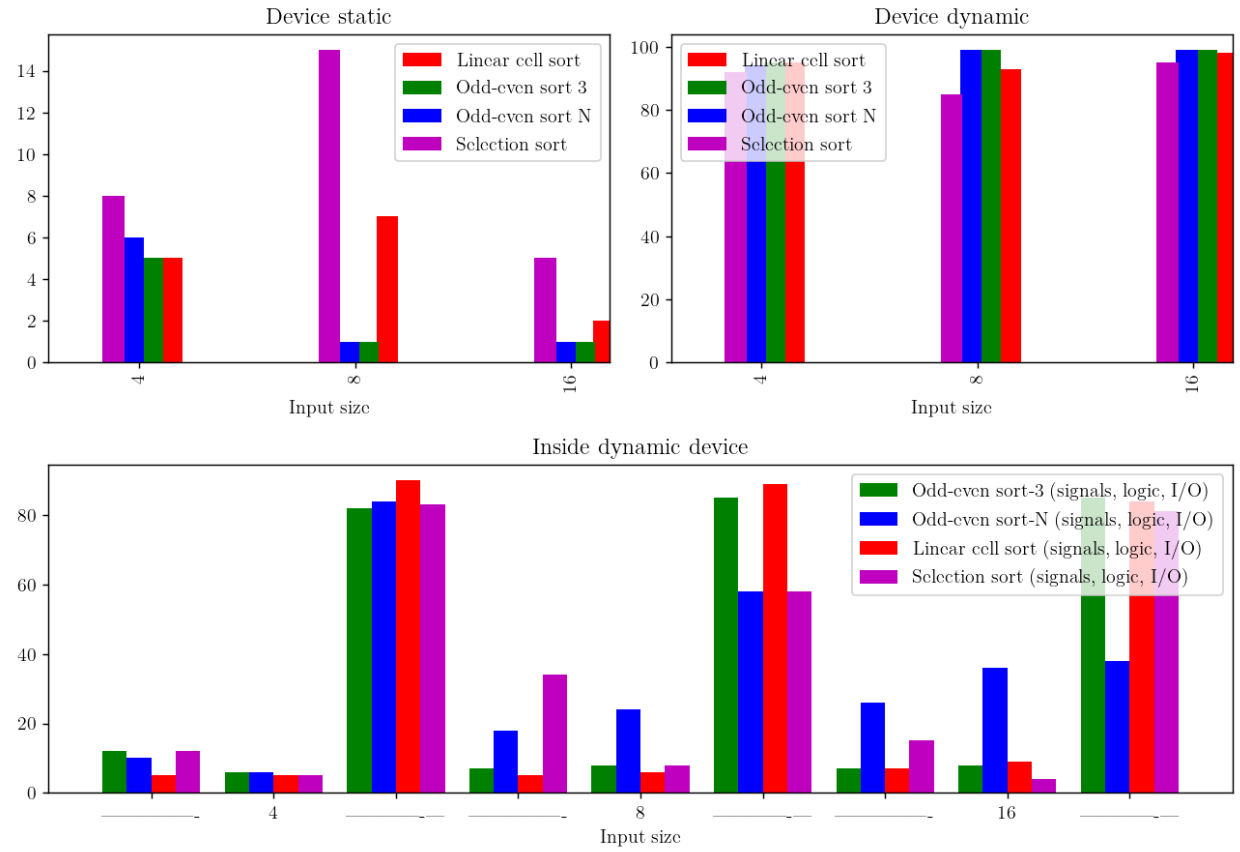


Figure 3.2: Graph of power usage across multiple input sizes for hardware implementations

complexity for the algorithm as a whole, because e.g. we used parallelization in the hardware implementations. On particular odd point is the hardware implementation of odd-even sort, which would normally have an $O(1)$, if it was a full sorting network. In our case, we have a combination of iterative and sorting network. This means that k, the amount of layers in the sorting network, is a part of determining the time complexity. Further in tbl. 3.4 we have calculated the exact amount of cycles each hardware implementation would need to fully sort the specified data.

Table 3.3: Overview of time complexity for the different implementations

Implementation	Worst	Best
Selection sort (hardware/software)	$O(n^2)$	$O(n^2)$
Linear cell sort (hardware)	$O(n)$	$O(n)$
Linear cell sort (software)	$O(n^2)$	$O(n^2)$
Odd-even sort (hardware)	$O(n-k)$	$O(1)$
Odd-even merge sort (software)	$O(n(\log n)^2)$	$O(n(\log n)^2)$

Table 3.4: Amount of clock cycles for hardware implementations, calculated based on ASMD charts

Implementation	Worst	Best
Selection sort	$1 + N^2 + 4(N-1)$	$1 + N^2 + 4(N-1)$
Linear cell sort	N	N
Odd-even sort	$N - k$	1

3.2 Selection sort

The Selection Sort is the most straightforward sorting algorithm. Our implementation will identify the minimum element in the array and swap it with the element in the primary position. Then it will identify the second position minimum element and swap it with the element in the second location, and it will continue executing this until the entire array is sorted. It has an $O(n^2)$ time complexity, which means it is inefficient on larger arrays. The input array divides into two subarrays, a sorted subarray of elements built up from top to bottom, and the remaining unsorted elements occupy the rest of the array.

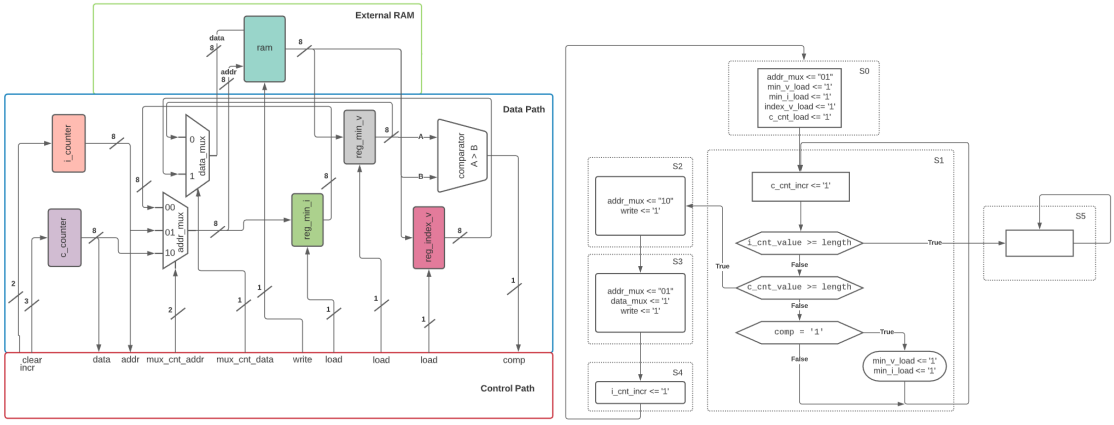
See appendix A.1 for a visual explanation of the algorithm.

3.2.1 Hardware Implementation

For the first hardware implementation, we followed the *Vivado Quick Start Tutorial* by Gericota for running VHDL code on the Zybo board using Vivado [4].

We have created a generic counter and register in the hardware implementation, which we want to reuse as much code as possible. The comparing counter is set to 1 as a default value, and the output of the RAM will be the first element in the array when we run the program. We temporarily store this index value of this element in a register and increment the index counter to compare the elements to find the smallest element in the array. Again, we temporarily store the index and the value of the smallest element in registers, then we swap those elements till the array is sorted. We have removed the RAM from the design file into the test bench file, which we wanted an external RAM instead of an internal RAM.

The design charts can be found in fig. 3.3, the schematic of the elaborated design in fig. 3.4 and finally a slice of the waveform diagram of simulating the implementation can be found in fig. 3.5.



(a) FSMD chart (b) ASMD chart

Figure 3.3: Design charts for selection sort

3.2.2 Software Implementation

For the first software implementation, we followed the *Vivado Quick Start Tutorial* by Gericota for using the microprocessor on the Zybo board [5].

The implementation of the algorithm in software was quick to write and certainly inspired by the hardware implementation. To keep it consistent, we decided to stick with similar names

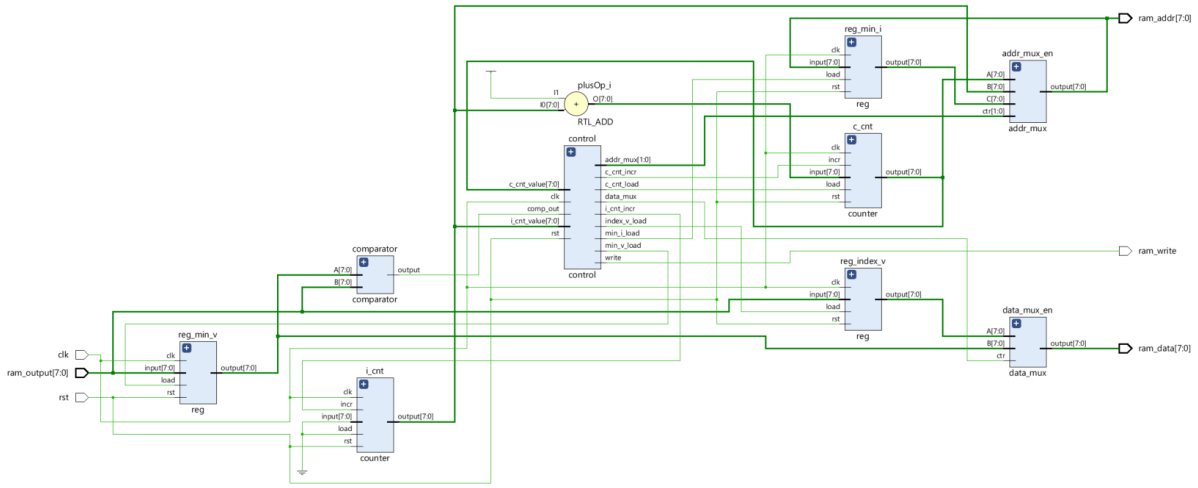


Figure 3.4: Schematic of elaborated design for selection sort

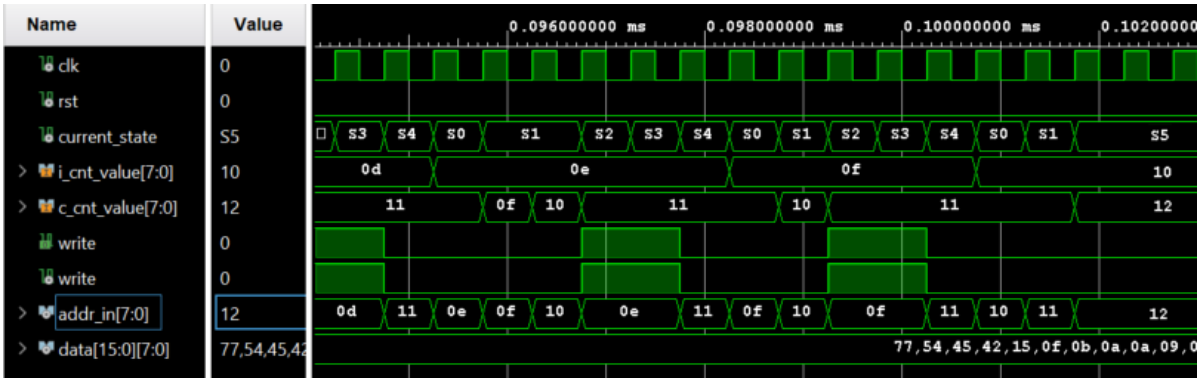


Figure 3.5: Part of waveform diagram for selection sort

for the different components (in particular `index_counter` and `comparing_index_counter`). This means that it should be easy to compare the implementations.

We have tested the software implementation on the Zybo board and it worked perfectly, as seen in fig. 3.6. The code can be found in lst. B.1.

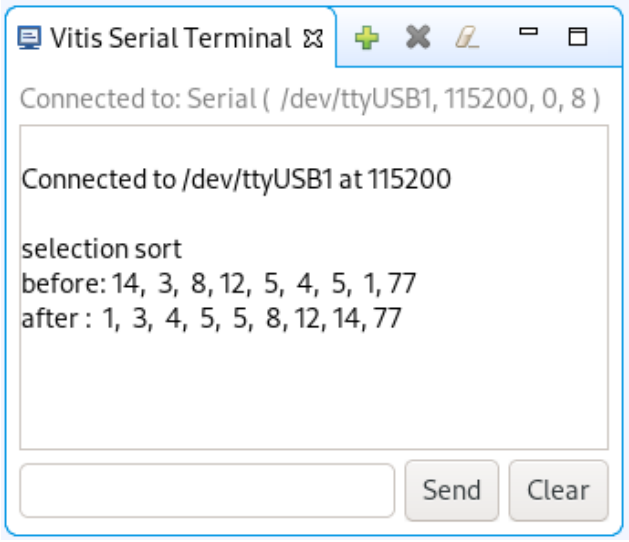


Figure 3.6: Results in serial terminal from running selection sort

3.3 Linear cell sort

Linear cell sort, as detailed by Vasquez's article [1], receives data once per clock cycle and sorts the data while it is being clocked in. This means that the algorithm only needs the N clock cycles to sort the data, giving it a time complexity of $O(N)$.

Since we decided to make the algorithm generic, it will let the user decide the array's size and length. The number of cells will be the same as the array size. New incoming data will be placed to the cell from top to bottom with increasing size. When all cells are empty, the first element will automatically take the first place. Second incoming data will be compared with the first element; if it is smaller than the first element, then the first element will be moved to the second cell, and the new data will be placed to the first cell. Third incoming data will be compared with the other cells; if the incoming data is smaller than the first cell, we have a full and pushed. The first cell's data will be pushed to the second cell, and the data in the second cell will be pushed to the third cell, and the new incoming data will be placed to the first cell. The sorting algorithm will continue like this until the whole array is sorted.

So in essence, the algorithm only has four rules:

1. If a cell is unoccupied, it will only be populated if the cell above is full.
2. If a cell is full, the cell data will be replaced if both the incoming data is less than the stored data, and the cell above is not pushing its data.
3. If the cell over the current cell is pushing out its stored data, then the current cell has to replace the current data with the cell data above.
4. If a cell is occupied and accepts new data either from the above cell or from the incoming data), it must push out the current data.

See appendix A.2 for a visual explanation of the algorithm.

3.3.1 Hardware implementation

The implementation of linear cell sort algorithms was more complicated than selection sort. We needed to draw two FSMDs and ASMDs charts, first for a general cell and then for the top level connection of cells. This design charts can be found in fig. 3.7.

The elaborated schematic can be found in fig. 3.8. The multiplexer in the schematic turned out to be quite large, this was a result of us having to use `std_logic`, and some other ugly workarounds, to make it work with generics. A waveform diagram of simulating the implementation can be found in fig. 3.9.

3.3.2 Software implementation

Since this algorithm is parallel by nature, there are some trade-offs to be made when implementing it in software. As we only have a single core to work with, we have chosen to simply transform it into a sequential algorithm. This means that instead of $O(N)$ time complexity, it will be $O(N^2)$ time complexity (as we have to iterate through every cell on every insertion). As such, we chose to handle the algorithm by having a ROM and a pointer to the "incoming" input, and instead of using cells, we chose to use an array to be simulated as multiple cells.

We have tested the software implementation on the Zybo board and it worked perfectly, as seen in fig. 3.10. The code can be found in lst. B.2.

3.4 Odd-even sort

The implementation and idea for the algorithm come from Skilarova's presentation of sorting networks [2].

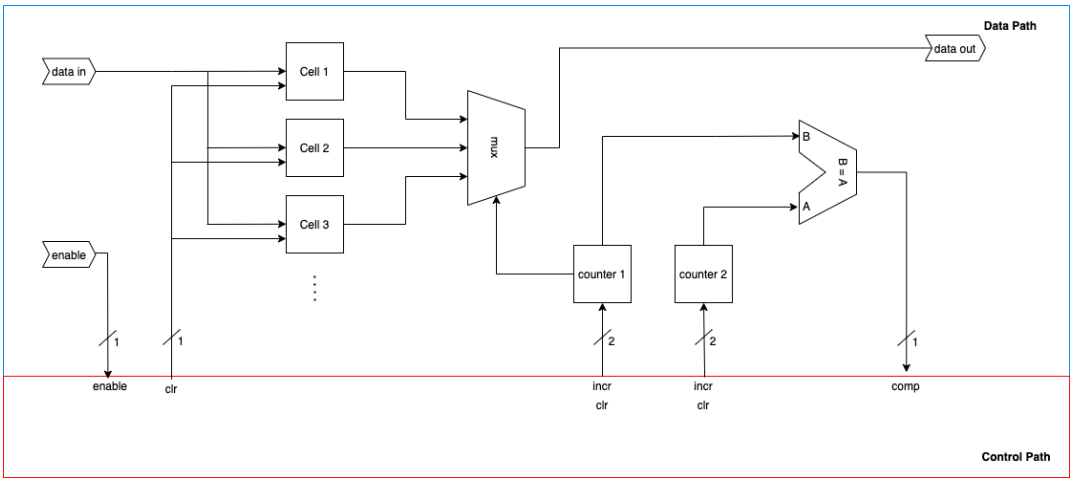
The algorithm is inspired by bubble sort and is a relatively straight forward. Bubble sort functioning by comparing adjacent elements; if the array elements are sorted, no swapping is terminated. Contrarily, the elements need to be switched. The even-odd transposition sort algorithm operates by comparing all odd/even listed pairs of neighbouring elements in the array if the match is in incorrect order; in other words, the primary element is bigger than the second the elements are swapped. The second step is to compare all even/odd listed matches of adjoining elements. These two steps are repeating until the array is sorted.

Knuth goes deeply into how sorting networks can be optimized and are created in his masterpiece; *The Art of Computer Programming* [6]. Further, the parallelization of the algorithm is well explained in the book from Nvidia called *GPU Gems 2* [7]. It details optimized sorting on GPUs using sorting networks and parallel comparisons. The parallelization aspect is quite similar between GPUs and FPGAs.

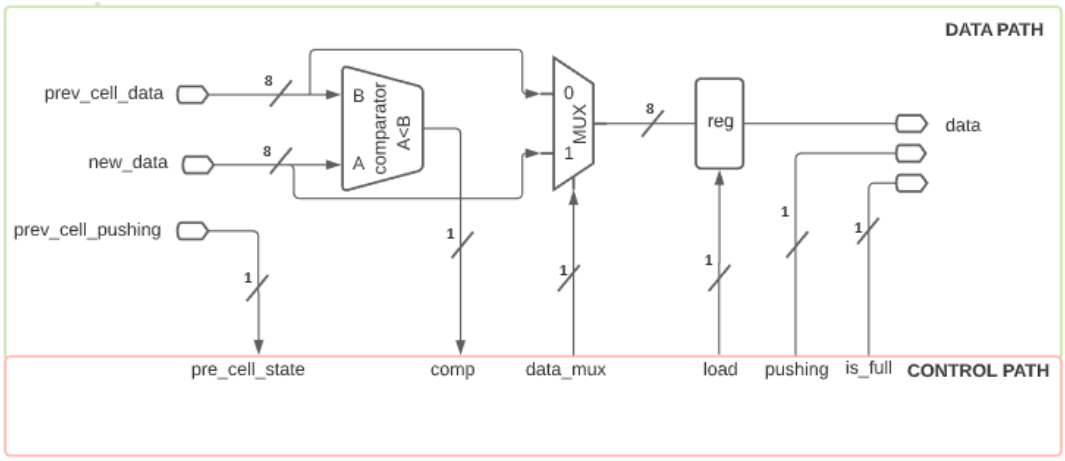
See appendix A.3 for a visual explanation of the algorithm.

3.4.1 Hardware implementation

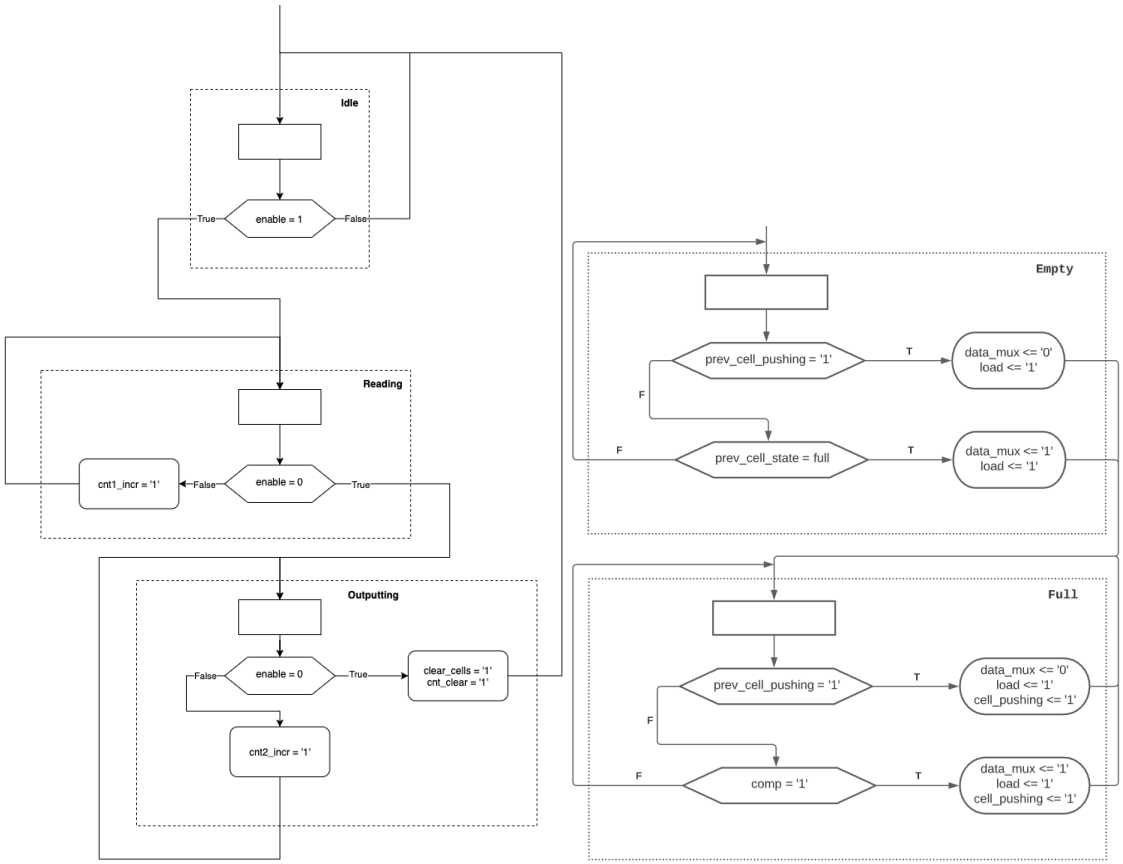
The hardware implementation of this algorithm was more comfortable than we thought, and the time complexity of this is $O(N)$. We parallelized our approach since it was easy and compared swap functions performed simultaneously on each element's match. We implemented this to use a generic model, which is more natural to resize the input N. The depth is the length of the input data, and it takes N stages that data is sorted. Furthermore, odd-



(a) FSMD chart



(b) FSMD chart for single cell



(c) ASMD chart

(d) ASMD chart for single cell

Figure 3.7: Design charts for linear cell sort

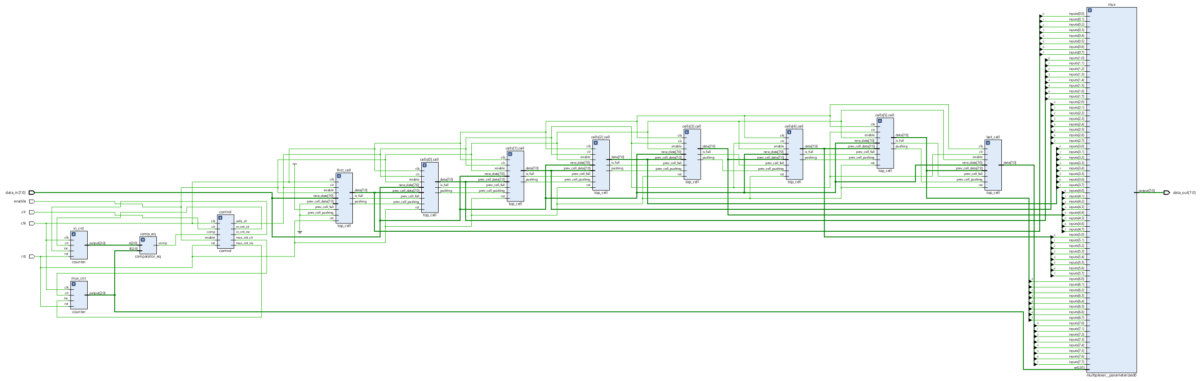


Figure 3.8: Schematic of elaborated design for linear cell sort

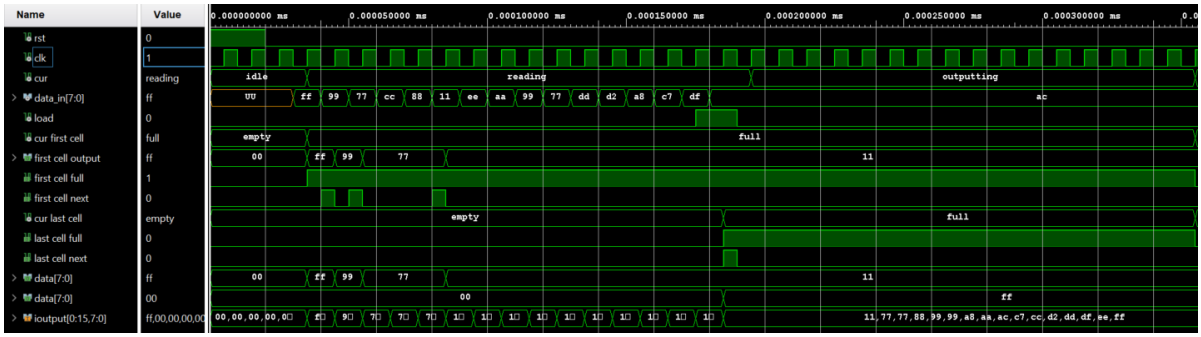


Figure 3.9: Waveform diagram for linear cell sort

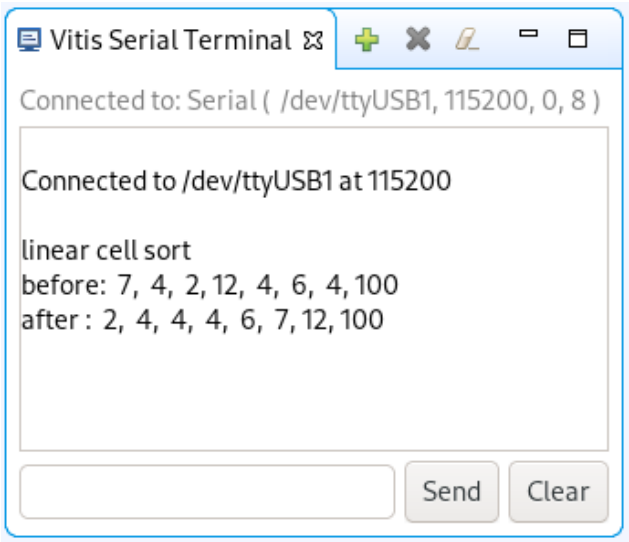


Figure 3.10: Results in serial terminal from running linear cell sort

even sort using more comparators, and we can calculate that straightforward. Let's take an example.

If we want to sort an array of N elements, we can calculate how deep and how many comparators we would need using the formulas from Skliarova's presentation [2; p. 8], seen in eq. (3.1). As an example, for 10 input signals, we would need 45 comparators spread out over 10 layers (eq. (3.2)).

$$\begin{aligned} D &= \text{depth / amount of layers} \\ C &= \text{comparators} \end{aligned}$$

$$\begin{aligned} D(N) &= N \\ C(N) &= \frac{N(N-1)}{2} \end{aligned} \tag{3.1}$$

$$\begin{aligned} D(10) &= N = 10 \\ C(10) &= \frac{N(N-1)}{2} = \frac{10(10-1)}{2} = 45 \end{aligned} \tag{3.2}$$

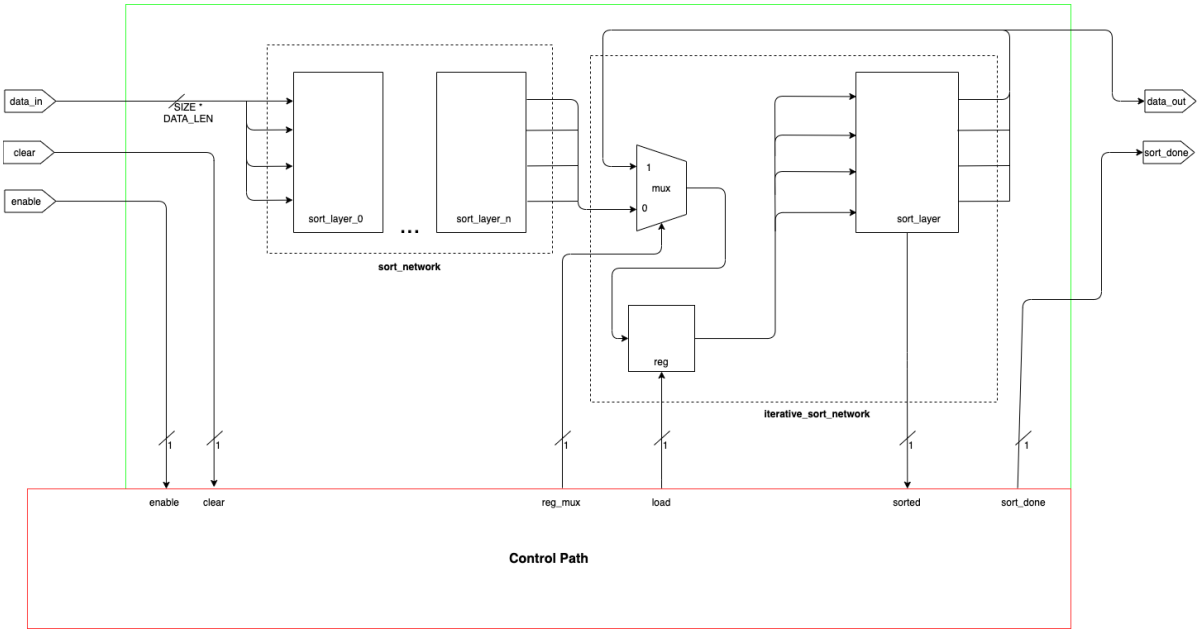
In Skilarova's presentation she also talks about iterative sorting networks, which reuses a single sort layer with some registers [2; p. 24]. Seeing this, we decided to use a combination of a sorting network and iterative layer in our implementation. The benefit of this is that the user of the VHDL module could tune the sorting network to be as quick and small as desirable. By increasing the amount of sorting layers, the network gets faster, however it also uses more hardware resources.

The design charts can be found in fig. 3.11 and a schematic of the elaborated design can be found in fig. 3.12. Lastly, a waveform diagram of simulating the implementation can be found in fig. 3.13.

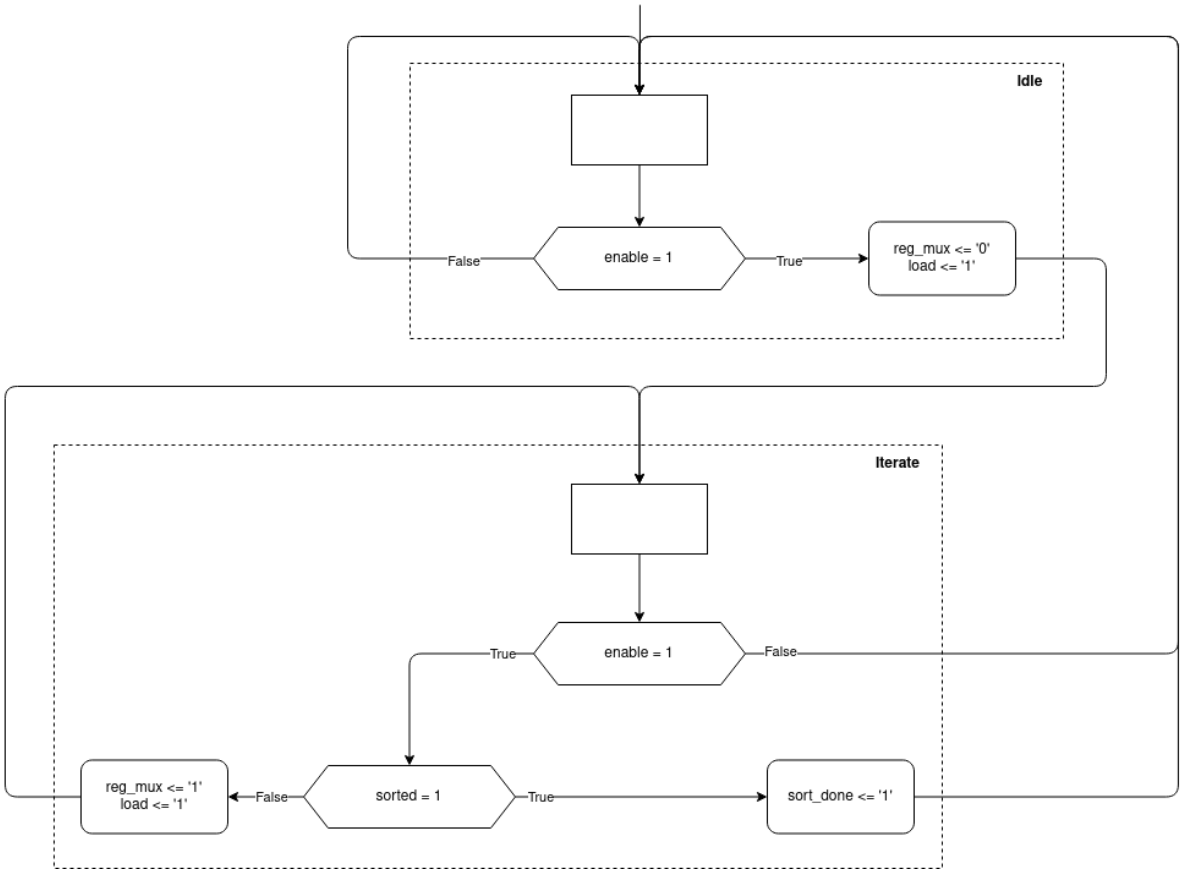
3.4.2 Software implementation

The main challenge of this algorithm is calculating the correct neighbouring indices for comparisons. As this is already a solved problem, we simply translated the code-shared by Bekbolatov [8] into C to be usable for our purpose. The function simply takes the current signal index, the current layer and the internal layer index and returns the index for the signal to compare to, or itself if there is non for this layer.

We have tested the software implementation on the Zybo board and it worked perfectly, as seen in fig. 3.14. The code can be found in lst. B.3.



(a) FSMD chart



(b) ASMD chart

Figure 3.11: Design charts for odd-even sort

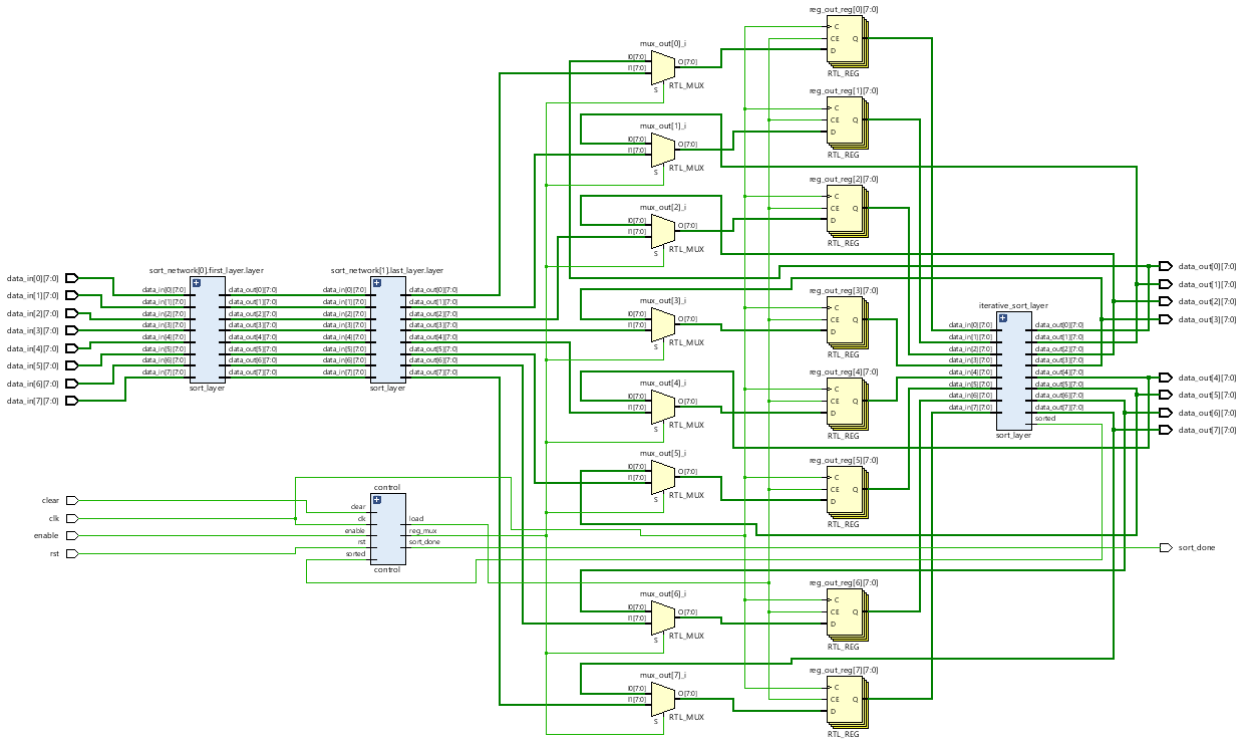


Figure 3.12: Schematic of elaborated design for odd-even sort

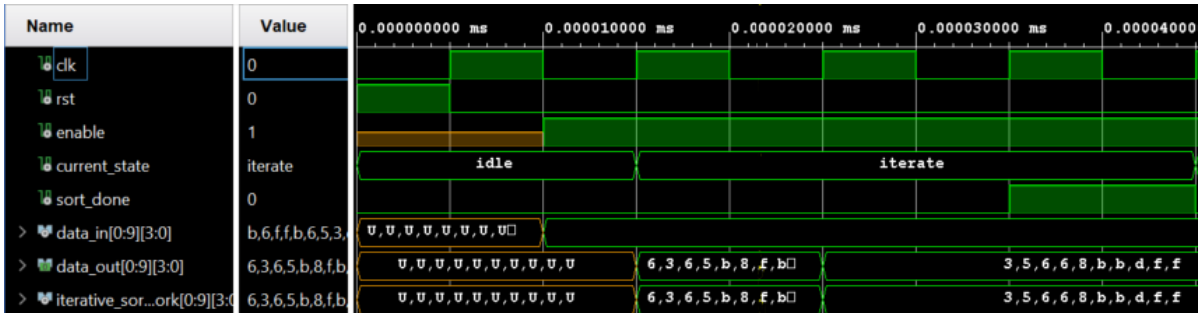


Figure 3.13: Waveform diagram for odd-even sort

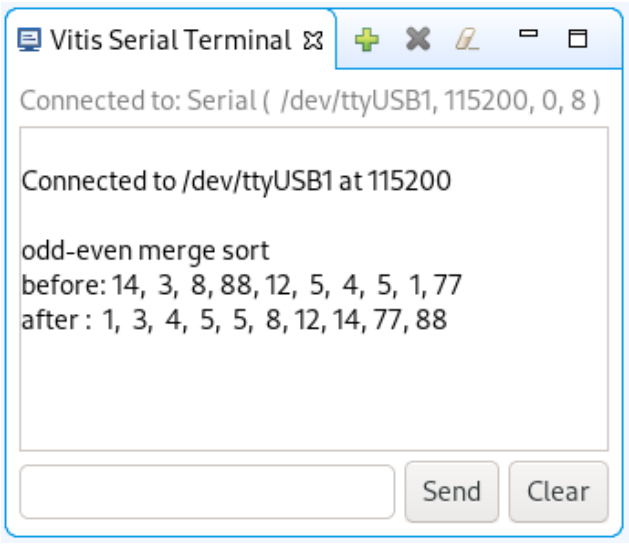


Figure 3.14: Results in serial terminal from running odd-even sort

Chapter 4

Discussion

Through our exploration, we managed to get all three algorithms working in both hardware and software. Further, we also found some clear distinctions between the algorithms in terms of development complexity, performance, efficiency, resource requirements and parallelization. We will now discuss and compare the different algorithms and implementations.

4.1 Differing development effort

The effort required by hardware and software development were quite differing. As an example, we spent nearly two full days of collaboration and pair programming to implement selection sort in VHDL. On the contrary, it took only about an hour to write the software implementation in C and running it on the Zybo board. We believe there are several reasons for this gap in development time.

The entire group did not have a long track record with VHDL and hardware development. Hence we spent time learning and developing our knowledge next to the actual implementation work. As we got more experienced with hardware development the work got more focused and hence more effective. This can be seen as the implementation of our last algorithm, odd-even sort, is both quite complex and modular, especially compared to our first implementation of selection sort. In the software domain, the group is quite well versed, hence the implementations were quickly developed by individuals.

Another aspect that affected development time for hardware was the extensive development activities conducted before writing a single line of code. We followed a lower-level approach, hence we firstly created an FSMD chart, then an ASMD chart and finally converting them into code. Further, per the development technique, we created separated files and entities for each component in the FSMD chart, hence there was quite a bit of work for each component. This is further emphasized by the difference in the amount of code for each implementation in hardware and software, as can be seen in tbl. 3.1. We did not conduct similar development activities when implementing the algorithm in software, because it's at a much higher level of abstraction. It is also possible to work at a higher level of abstraction when implementing in hardware, however, we did not explore this possibility due to time restrictions within the project. Although it would have been interessting to see the outcome this type of implementation.

One aspect that might have impacted development effort is that we consequently did the hardware implementation before the software implementation. As the course has been mostly

focused on hardware and VHDL, we wanted to prioritize completing the hardware implementations as a group. By doing it as a group we could take advantage of discussions and collaboration to learn optimally. As we started each new algorithm by working together as a group, we naturally also started with the hardware implementation. After implementing the algorithm in hardware one can argue that we had a much better understanding of the algorithm which would mean that the following implementation in software would be easier. However, since the algorithms are fairly trivial the knowledge gained from implementing it in hardware is minuscule, and therefore it is unlikely that this had a big impact on the development effort.

Lastly, despite gaining proficiency in using the tools for hardware development, we spent a lot of time figuring out cryptic error messages. One would think that this would improve with experience, however, as we started to used more complex features we also consistently hit new errors. As an example, we started using generics in our second algorithm to make it more reusable.

4.2 Multiplexing in time vs space

Hardware is by nature parallel, while software is, in general, sequential[1]. The perhaps biggest benefit of implementing algorithms in hardware is that we can utilize the built-in disposition for parallelism to execute multiple actions at once. This is done by creating several components separate in space which operate independently of each other, hence instead of multiplexing actions over time, we multiplex them over space. An example of a concrete benefit can be seen in tbl. 3.4, where the speed of odd-even merge sort in hardware can at best take only 1 clock cycle while selection sort requires at least N^2 cycles.

Although utilizing parallelism in hardware has many benefits, it also comes with the drawback of requiring specialized hardware resources. In fig. 3.1 we can see that the resources needed for odd-even sort, which is highly parallelized, are substantially larger than selection sort and linear cell sort, which don't utilize parallelism to the same degree. Further, the pure software implementation of either algorithm requires no extra hardware resources besides the generic processor.

[1] A software program can be created to run tasks concurrently, either on separate processors or through time-slicing, however for our comparison we will consider a single-core computer running a sequential program.

Chapter 5

Conclusion

We found that the development efforts between software and hardware were particularly highlighted in our project due to lack of knowledge. However, the extensive development activities are still a major factor causing hardware development to be more labour-intensive than software development. Further utilizing parallelism in hardware can substantially increase the speed of the algorithm at the cost of requiring more hardware resources. Hence there is a trade-off to make between multiplexing the algorithm in time or space depending on the context of the application.

References

[1] J. Vasquez, "Sort faster with FPGAs," Jan. 20, 2016. https://hackaday.com/2016/01/20/a-linear-time-sorting-algorithm-for-fpgas/ (accessed Sep. 17, 2020).

[2] I. Skliarova, Ed., *Parallel data processing in reconfigurable systems*. Universidade de Aveiro, 2015.

[3] D. Inc., "Zybo reference manual." https://reference.digilentinc.com/reference/programmable-logic/zybo/reference-manual (accessed Sep. 19, 2020).

[4] M. Gericota, "Vivado quick start tutorial for the Digilent Zybo Z7-10 Zynq-7000 ARM/FPGA SoC platform board – a hardware approach," University of South-Eastern Norway, 2020.

[5] M. Gericota, "Vivado/Vitis Zynq quick start tutorial for the Digilent Zybo Z7-10 Zynq 7000 ARM/FPGA SoC platform board – using the Zynq," University of South-Eastern Norway, 2020.

[6] D. E. Knuth, "Networks for sorting," in *The art of computer programming*, Second., vol. 3, Stanford University; Addison Weasly Longman, 1998.

[7] M. Pharr and F. Randima, "Improved gpu sorting," in *GPU gems 2 : Programming techniques for high-performance graphics and general-purpose computation*, Addison-Wesley, 2005.

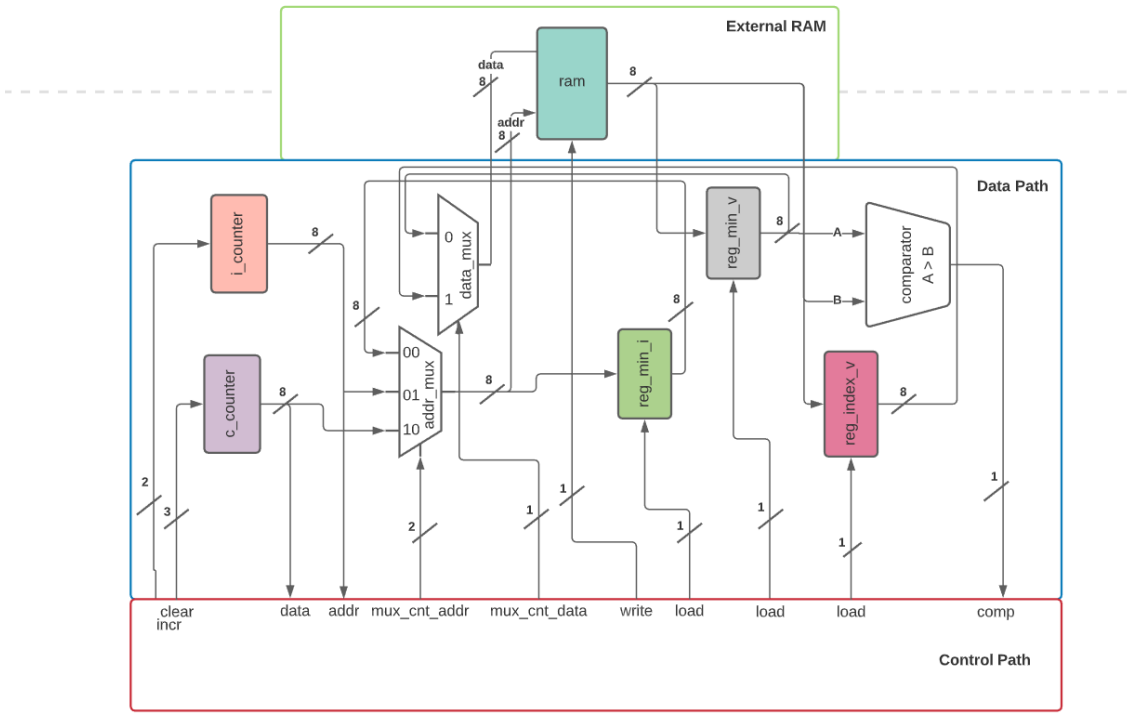
[8] R. Bekbolatov, "Sorting network from batcher's odd-even merge: Partner calculation," 2015. https://gist.github.com/Bekbolatov/c8e42f5fcaa36db38402 (accessed Sep. 17, 2020).

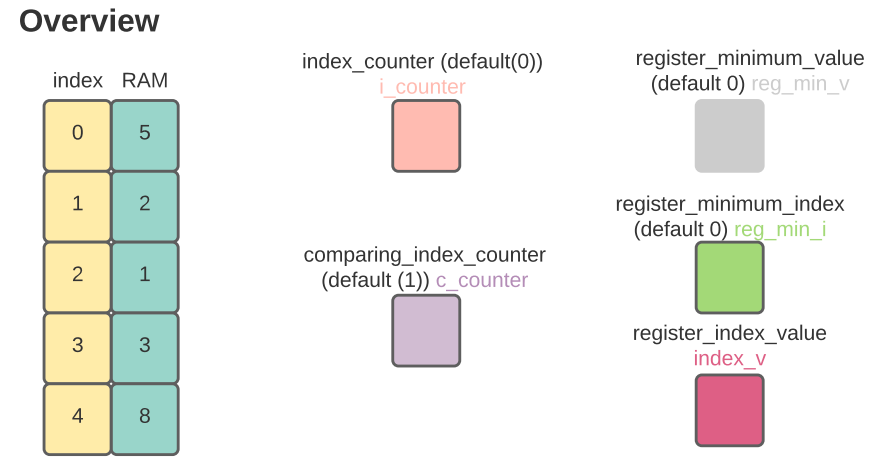
[9] M. Gericota, "Creating, packaging and controlling an IP in Vivado/Vitis - Quick start tutorial for the Digilent Zybo Z7-10 Zynq-7000 ARM/FPGA," University of South-Eastern Norway, 2020.

Appendix A

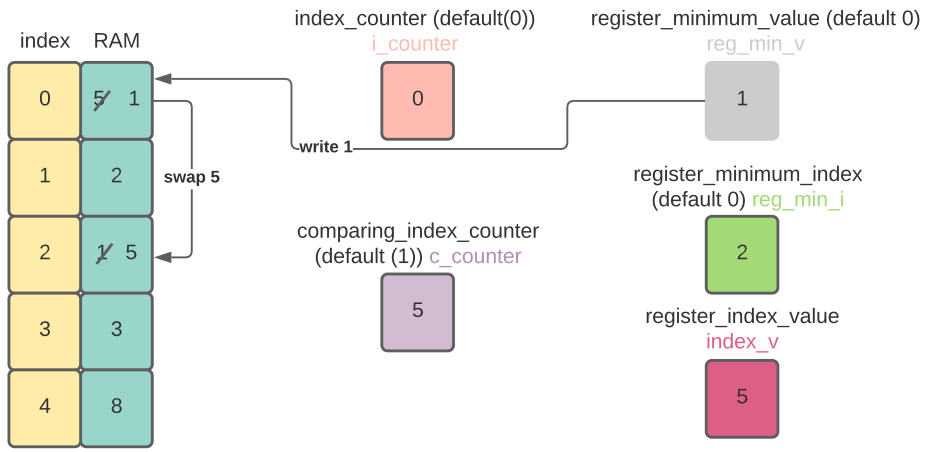
Visual explanations of the sorting algorithms

A.1 Selection sort

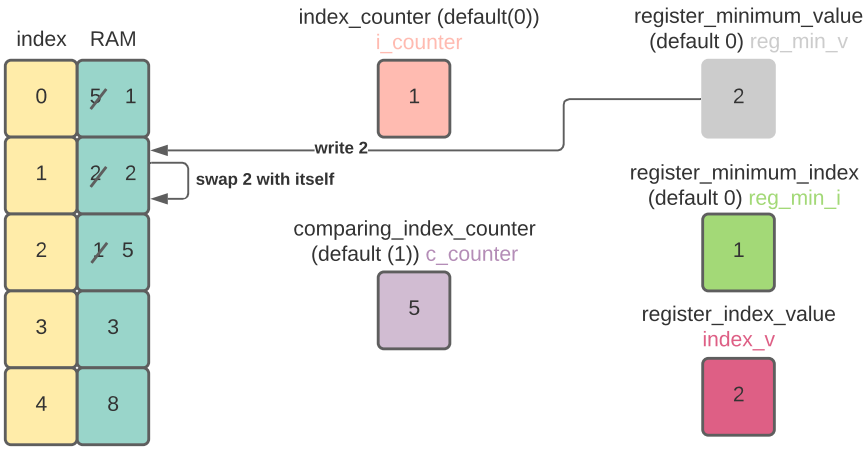


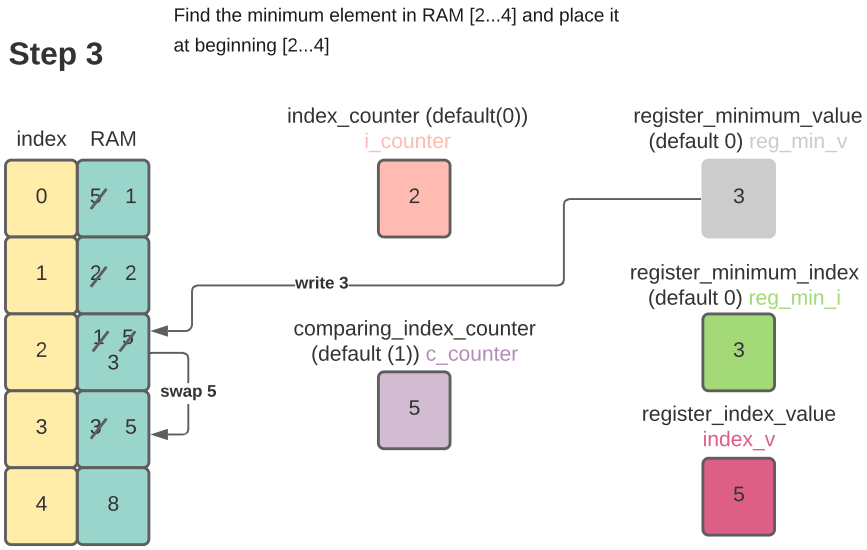


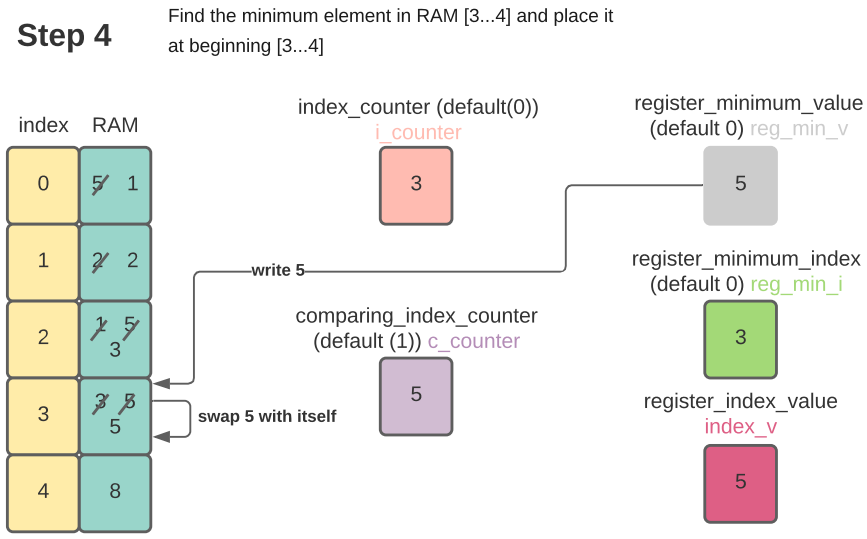
Find the minimum element in RAM [0...4] and place it at beginning



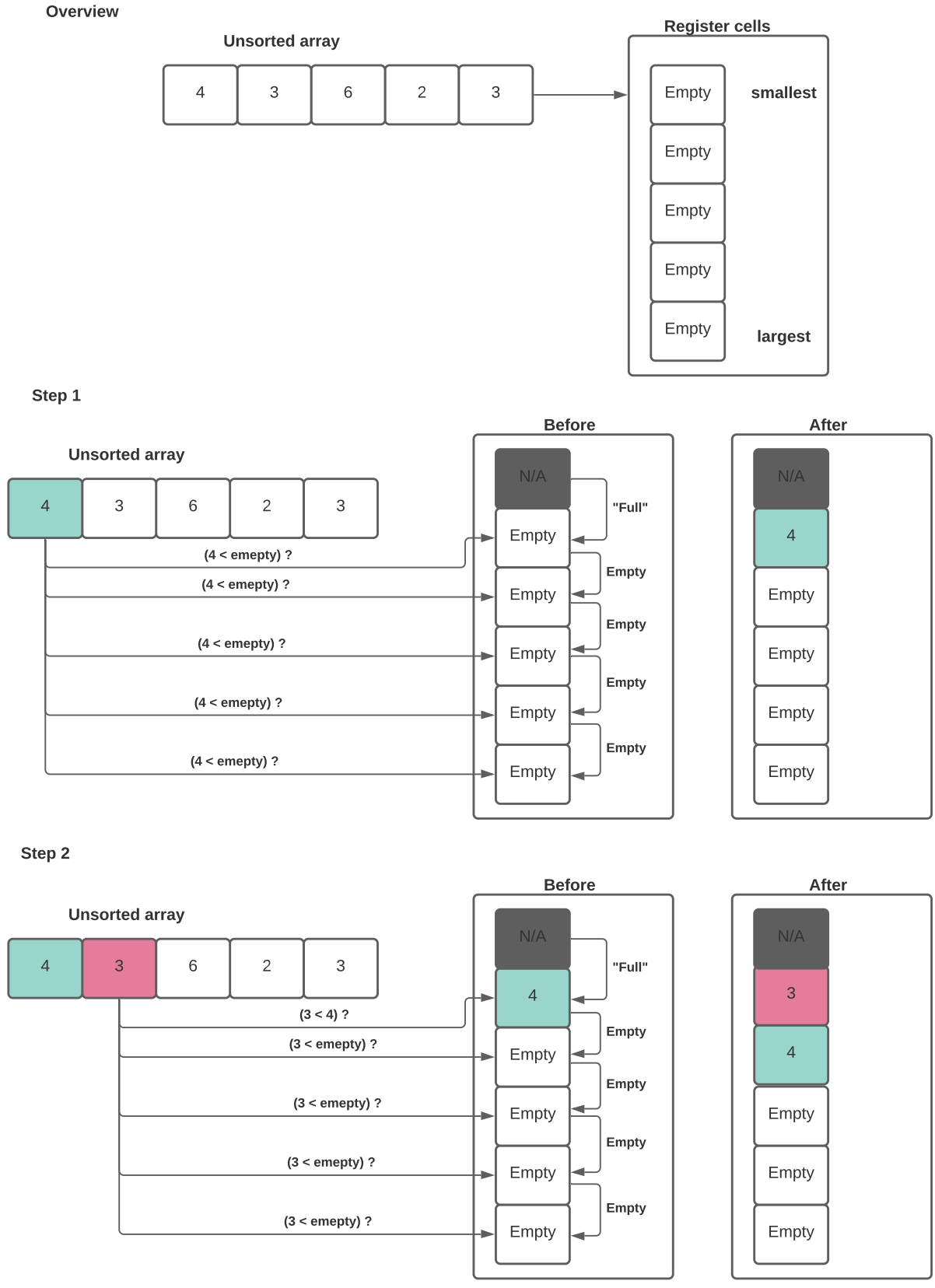
Find the minimum element in RAM [1...4] and place it at beginning [1...4]

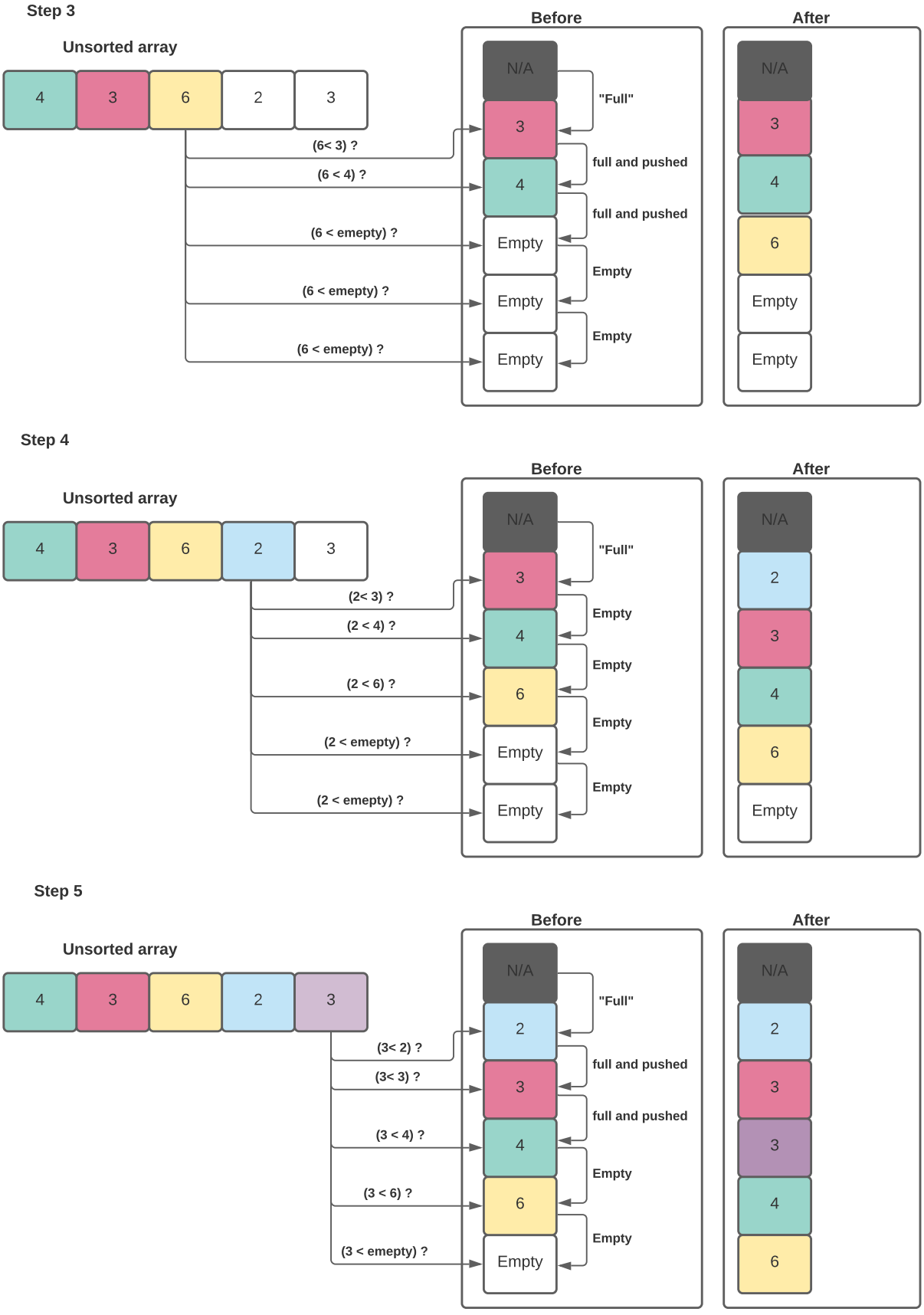



Step 3
Find the minimum element in RAM [2...4] and place it at beginning [2...4]
index
RAM
index_counter (default(0))
i_counter
register_minimum_value
(default 0) reg_min_v
write 3
register_minimum_index
(default 0) reg_min_i
comparing_index_counter
(default (1)) c_counter
swap 5
register_index_value
index_v


Step 4
Find the minimum element in RAM [3...4] and place it at beginning [3...4]
index
RAM
index_counter (default(0))
i_counter
register_minimum_value
(default 0) reg_min_v
write 5
register_minimum_index
(default 0) reg_min_i
comparing_index_counter
(default (1)) c_counter
swap 5 with itself
register_index_value
index_v

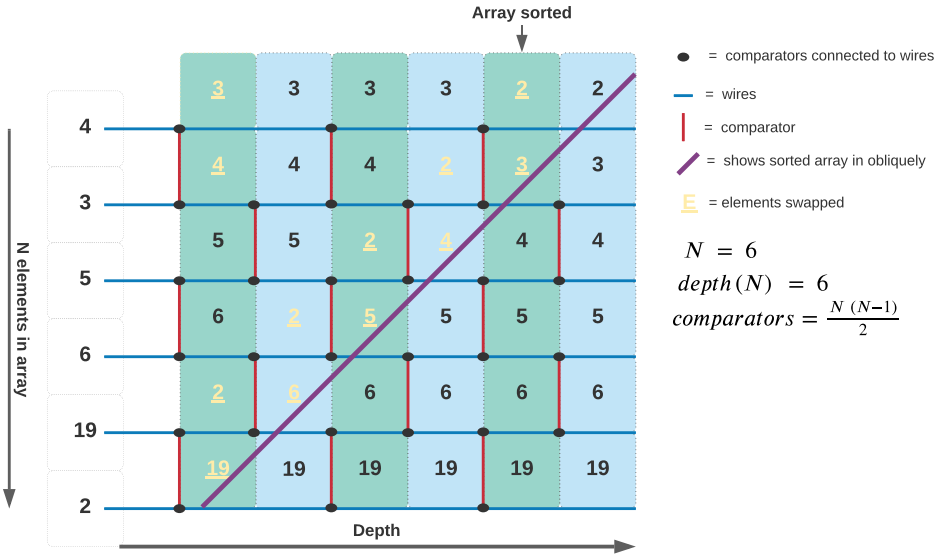
A.2 Linear cell sort




Step 3
Unsorted array
4
3
6
2
3
Before
After
N/A
3
4
Empty
Empty
Empty
"Full"
full and pushed
full and pushed
Empty
Empty
(6< 3) ?
(6 < 4) ?
(6 < emepty) ?
(6 < emepty) ?
(6 < emepty) ?
Step 4
Unsorted array
4
3
6
2
3
Before
After
N/A
3
4
6
Empty
Empty
"Full"
Empty
Empty
Empty
(2< 3) ?
(2 < 4) ?
(2 < 6) ?
(2 < emepty) ?
(2 < emepty) ?
N/A
2
3
4
6
Empty
Step 5
Unsorted array
4
3
6
2
3
Before
After
N/A
2
3
4
Empty
"Full"
full and pushed
full and pushed
Empty
Empty
(3< 2) ?
(3< 3) ?
(3 < 4) ?
(3 < 6) ?
(3 < emepty) ?
N/A
2
3
3
4
6

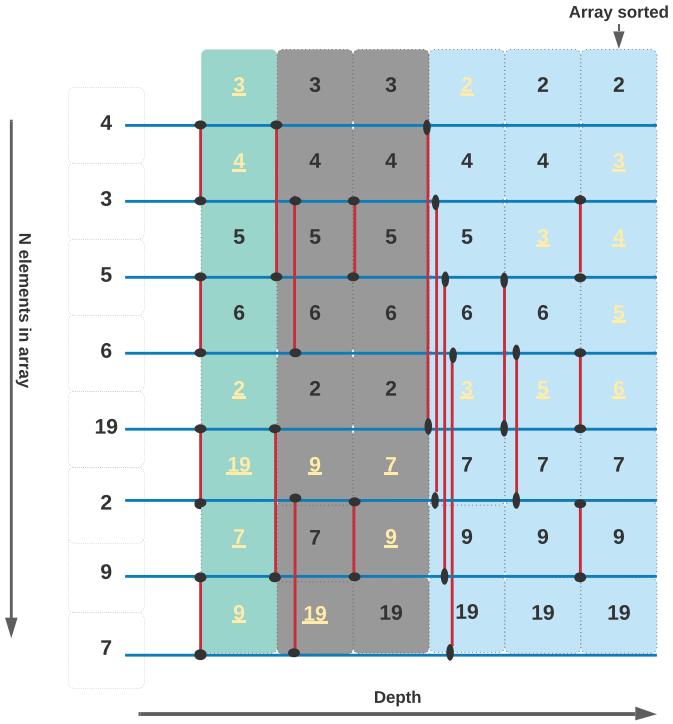
A.3 Odd-even transposition and merge sort

Even-odd transposition sort network



$$N = 6$$
$$depth(N) = 6$$
$$comparators = \frac{N\,(N-1)}{2}$$

Even-odd merge sort network



$$N = 8$$
$$depth(N = 2^p) = \frac{p\,(p+1)}{2}$$
$$depth\ (N) = 6$$

$$comparators\ (N = 2^p) = \left(p^2 - p + 4\right) 2^{p-2} - 1$$
$$comparators\ (N) = 19$$

Appendix B

The code

We have only included the code for the software implementations due to the hardware implementations having several hundred lines. As the code is not the primary product of this paper, it will only be available as supplementary ZIP-files.

B.1 Selection sort

```
1  #include "platform.h"
2  #include "xil_printf.h"
3  #include <stdio.h>
4
5  typedef unsigned int uint;
6
7  #define DATA_LEN 9
8  uint data[DATA_LEN] = {14, 3, 8, 12, 5, 4, 5, 1, 77};
9
10 void sort(uint *array, uint length);
11 void print_array(uint *array, uint length);
12
13 int main() {
14   init_platform();
15
16   fputs("\rselection sort\rbefore: ", stdout);
17   print_array(data, DATA_LEN);
18
19   sort(data, DATA_LEN);
20
21   fputs("\rafter : ", stdout);
22   print_array(data, DATA_LEN);
23
24   cleanup_platform();
25   return 0;
26 }
27
28 // Sorts array inplace using selection sort.
29 void sort(uint *array, uint length) {
```

```
30
31   for (int index_counter = 0; index_counter < length; ++index_counter) {
32     // Set index_counter element as smallest found element
33     uint smallest_index = index_counter;
34
35     // Compare smallest element with rest of array and keep the smallest
           element
36     // found (including index)
37     for (int comparing_index_counter = index_counter + 1;
38          comparing_index_counter < length; ++comparing_index_counter) {
39       if (array[comparing_index_counter] < array[smallest_index]) {
40         smallest_index = comparing_index_counter;
41       }
42     }
43
44     // Swap first and smallest elements
45     uint temp = array[index_counter];
46     array[index_counter] = array[smallest_index];
47     array[smallest_index] = temp;
48   }
49 }
50
51 void print_array(uint *array, uint length) {
52   for (int i = 0; i < length; ++i) {
53     char buf[10];
54     sprintf(buf, "%2d%s", array[i], i + 1 == length ? "" : ", ");
55     fputs(buf, stdout);
56   }
57 }
```

Listing B.1: Code for software implementation of selection sort

B.2 Linear cell sort

```
1  #include "platform.h"
2  #include "xil_printf.h"
3  #include <stdio.h>
4
5  typedef unsigned int uint;
6
7  #define DATA_LEN 8
8  uint data[DATA_LEN] = {7, 4, 2, 12, 4, 6, 4, 100};
9
10 void sort(uint *dest, const uint *src, uint length);
11 void print_array(uint *array, uint length);
12
13 void shift_and_replace(uint *data, uint length, uint from_position,
14                         uint new_value);
15
16 int main() {
17   init_platform();
18
19   fputs("\rlinear cell sort\rbefore: ", stdout);
20   print_array(data, DATA_LEN);
21
22   uint sorted_data[DATA_LEN] = {};
23   sort(sorted_data, data, DATA_LEN);
24
25   fputs("\rafter : ", stdout);
26   print_array(sorted_data, DATA_LEN);
27
28   cleanup_platform();
29   return 0;
30 }
31
32 // Moves data from src into dest so that dest remains sorted.
33 //
34 // NB! dest has to be zero-initialized.
35 void sort(uint *dest, const uint *src, uint length) {
36   // First value always goes into first cell
37   dest[0] = src[0];
38
39   // Loop through each input value in src and process it through the "cells"
40   for (int i = 1; i < length; i++) {
41
42     // The following loop is done in parallel in hardware, however in software
43     // we have implemented it sequentially because the syncronization needed
          for
44     // doing this in parallel is quite tricky.
45     for (int j = 0; j < length; j++) {
46
47       // The rules for a cell encoded into C code. It is only if we make an
```

```
48        // action that we move on to the next "cell" using break.
49        if (dest[j] > src[i]) {
50          // New element is smaller so move every element in the array to the
51          // right and insert as new value in front.
52          shift_and_replace(dest, DATA_LEN, j, src[i]);
53          break;
54        } else if (dest[j] == 0 && (j == 0 || dest[j - 1] != 0)) {
55          // New element is bigger or equal, current "cell" is empty and
                previous
56          // cell is full, so we place it in the empty "cell".
57          dest[j] = src[i];
58          break;
59        }
60      }
61    }
62  }
63
64  void shift_and_replace(uint *data, uint length, uint from_position,
65                         uint new_value) {
66    for (int i = DATA_LEN - 1; i > from_position; i--) {
67      data[i] = data[i - 1];
68    }
69    data[from_position] = new_value;
70  }
71
72  void print_array(uint *array, uint length) {
73    for (int i = 0; i < length; ++i) {
74      char buffer[10];
75      sprintf(buffer, "%2d%s", array[i], i + 1 == length ? "" : ", ");
76      fputs(buffer, stdout);
77    }
78  }
```

Listing B.2: Code for software implementation of linear cell sort

B.3 Odd-even sort

```
1  #include "platform.h"
2  #include "xil_printf.h"
3  #include <stdio.h>
4
5  typedef unsigned int uint;
6
7  #define DATA_LEN 10
8  uint data[DATA_LEN] = {14, 3, 8, 88, 12, 5, 4, 5, 1, 77};
9
10 void sort(uint *array, uint length);
11 void print_array(uint *array, uint length);
12
13 uint partner(uint index, uint layer, uint in_layer);
14 void swap(uint *a, uint *b);
15 uint hibit(uint n);
16
17 int main() {
18   init_platform();
19
20   fputs("\rodd-even merge sort\rbefore: ", stdout);
21   print_array(data, DATA_LEN);
22
23   sort(data, DATA_LEN);
24
25   fputs("\rafter : ", stdout);
26   print_array(data, DATA_LEN);
27
28   cleanup_platform();
29   return 0;
30 }
31
32 // Sorts array inplace using odd-even-merge sort.
33 void sort(uint *array, uint length) {
34
35   uint layers = hibit(length);
36
37   // If length is not exactly 2^layers, then add another layer. This is to
38   // ensure that the entire array is sorted.
39   if (length - (1 << layers) > 0)
40     layers++;
41
42   // Iterate over sorting layers
43   for (uint layer = 1; layer <= layers; ++layer) {
44
45     // Iterate over in-layer steps
46     for (uint in_layer = 1; in_layer <= layer; ++in_layer) {
47
48       // Iterate over comparators for current layer and step
```

```
49      for (uint i = 0; i < length; ++i) {
50        uint p = partner(i, layer, in_layer);
51
52        if (i != p && p < length && array[i] < array[p]) {
53          swap(&array[i], &array[p]);
54        }
55      }
56    }
57  }
58 }
59
60 void swap(uint *a, uint *b) {
61   uint temp = *a;
62   *a = *b;
63   *b = temp;
64 }
65
66 // Gets position of highest order bit in number
67 //
68 // E.g. 8 -> 3, 9 -> 3, 16 -> 4
69 uint hibit(uint n) {
70   n |= (n >> 1);
71   n |= (n >> 2);
72   n |= (n >> 4);
73   n |= (n >> 8);
74   n |= (n >> 16);
75   return n - (n >> 1);
76 }
77
78 // Gets partner index based on index, layer and in_layer indicies.
79 uint partner(uint index, uint layer, uint in_layer) {
80   if (in_layer == 1) {
81     return index ^ (1 << (layer - 1));
82   } else {
83     uint scale = 1 << (layer - in_layer);
84     uint box = 1 << in_layer;
85     uint sn = index / scale - (index / scale / box) * box;
86
87     return sn == 0 || sn == box - 1
88                ? index
89                : sn % 2 == 0 ? index - scale : index + scale;
90   }
91 }
92
93 void print_array(uint *array, uint length) {
94   for (int i = 0; i < length; ++i) {
95     char buf[10];
96     sprintf(buf, "%2d%s", array[i], i + 1 == length ? "" : ", ");
97     fputs(buf, stdout);
```

```
98   }
99 }
```

Listing B.3: Code for software implementation of Batcher's odd-even merge sort

Appendix C

Division of work

In this project, we collaborated closely together as a group to maximize learning and discussion. Most of the tasks were done together at the University of South-Eastern Norway. This included discussions about which algorithms we would choose and why, and also the hardware implementation activities. Further, we also completed two of the three software implementations together. The work on the report was done separately, but with close collaboration through Slack.

ASMD and FSMD diagrams were firstly drawn on a blackboard and then digitized by Rahmat and Anders. Everyone participated in the creation of the first and the second draft of the report, while Anders, Rahmat and Ole finalized the report. Anders implemented the software code for the linear cell sort algorithm. Ole worked on the IP implementation and structured the final report. Rahmat also worked on IP implementation, created a visual explanation for the algorithms and extracted the utilization data from the different implementations.

Appendix D

Results from attempting IP creation

In the IP implementation, we followed the *Vivado Quick Start Tutorial* by Gericota for creating, packaging and controlling an IP [9].

We declared some output ports and port mapped those in the `selection_sort_IP_v1_0.vhd`. Next, we made a component declaration and created some signals for inputs and outputs in `selection_sort_IP_v1_0_S00_AXI.vhd`. The VHDL description files created for the hardware implementation of the selection sort algorithm we copied those files into the IP directory and created a new AXI4 Peripheral for IP.

After this, we created a new block design to integrate our IP, added, and customized the `ZYNQ7 Processing System`. Our next step was that we added `selection_sort_IP_v1_0.vhd` into our design and created HDL Wrapper. Further, we added the sorting controller and the block memory IP blocks into our design. The block memory generator is the previously explained external RAM and the sorting controller enables us to inspect the memory after it has been sorted. This is done by simply enabling our software running on the ZYNQ processor to read the memory through AXI.

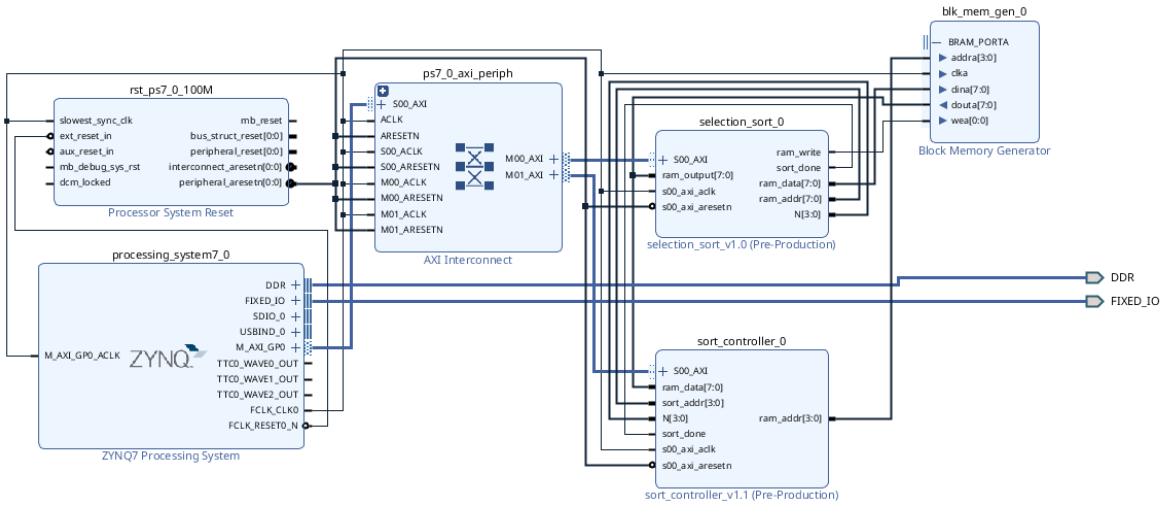


Figure D.1: IP block design for selection sort

Finally, after putting together the different IP blocks and having the block diagram in fig. D.1, we generated a bitstream. Then we exported the hardware design to Vitis IDE. In Vitis IDE we first created a project platform using the XSA-file, which was exported from the

Vivado. After building the platform, we created a new application project to test our IP implementation using software.

To be able to display the sorted values in the serial terminal, we need to communicate with the sorting controller from the ZYNQ processing unit through the AXI interface. The code that has to run on the processing unit can be found in lst. D.1. The function `Xil_In32`, provided by the platform, reads a value from the AXI interface. By reading slave register 2 of the sorting controller, we can tell if the sorting is done, as the first bit represents the `sort_done` signal. Further, by then repeatedly reading slave register 1 we will get the contents of the memory block as the sorting controller continuously updates the RAM address and reads the data into the slave register.

However, despite the promise of this solution in theory, we could not get it working in pratice. Our best guess as to why it wasn't working was that since the sort controller updated the slave register with the RAM data value too quickly, as it's updated every clock cycle in hardware. Another possible reason for the error was that we could be reading from the wrong register at the AXI interface. Nonetheless, getting to the current non-working solution took us a considerable amount of effort, and we concluded that it was too time consuming to continue. If we were to continue trying, I think we would have redesign the sort controller to rather read an address from the AXI interface, which was then used to fetch a value from RAM. This would allow the software on the Zynq processor to iterate over the RAM positions and allow the data values to stay longer in the slave register.

```
1  include "xparameters.h"
2  include "xuartps_hw.h"
3
4  int main(){
5      xil_printf("Start selection sort\n\n\r");
6
7      xil_printf("Sorting");
8      u32 slave_reg_2;
9      do {
10         slave_reg_2 = Xil_In32(XPAR_SORT_CONTROLLER_0_S00_AXI_BASEADDR + 8);
11         xil_printf(".");
12     } while ((slave_reg_2 & 0x1) == 0);
13     xil_printf("\n\r");
14
15     xil_printf("Printing\n\r");
16     u32 slave_reg_1;
17     do {
18         slave_reg_2 = Xil_In32(XPAR_SORT_CONTROLLER_0_S00_AXI_BASEADDR + 8);
19         slave_reg_1 = Xil_In32(XPAR_SORT_CONTROLLER_0_S00_AXI_BASEADDR + 4);
20         xil_printf("%lx ", slave_reg_1);
21     } while ((slave_reg_2 & 0x1) == 1);
22
23     return 0;
24 }
```

Listing D.1: Code for communicating with the sort controller