FPGA Implementation and Comparison of Popular Sorting Algorithms

Candidate 6000
CS4110 Hardware/Software Co-Design
University of South-Eastern Norway
Department of Science and Industry Systems
Kongsberg, Norway

Candidate 6002
CS4110 Hardware/Software Co-Design
University of South-Eastern Norway
Department of Science and Industry Systems
Kongsberg, Norway

Candidate 6014
CS4110 Hardware/Software Co-Design
University of South-Eastern Norway
Department of Science and Industry Systems
Kongsberg, Norway

***Abstract*—Sorting algorithms are algorithms that organize elements in a list into an order, often ascending or descending. It is widely used in computer science for various applications. Efficient sorting is an important aspect when optimizing the algorithm. This paper examines various implementations of the sorting algorithms to run on a System-on-Chip containing a FPGA and a CPU. This paper is focused on the implementation of three popular sorting algorithms: merge sort, radix sort and bitonic sort. Each sorting algorithm are implemented using Register Transfer Level (RTL) implementation, High Level Synthesis (HLS) implementation and software implementation.**

Keywords—FPGA, VHDL, SoC, Sorting Algorithm, Hardware, Software

I. Introduction

This paper is going in-depth at three completely different sorting algorithms and their implementations in software and hardware. The algorithms this paper is going to focus on is merge-, radix- and bitonic-sort. In this context merge sort is an algorithm that heavily uses comparisons and is a divide and conquer based type of algorithm that breaks down the problem into sub-problems recursively. Radix on the other hand is a non-comparative algorithm and uses the radixes of each number to sort. It sorts recursively from the least significant digit to the most significant digit. Bitonic sort is the last algorithm and is extremely concurrent and designed to be ran in parallel. The algorithm builds itself a sorting network that compromises of comparators. The two other algorithms are mostly sequential and is based on two completely different sorting ideas.

The purpose of this is to compare and review each one and compare resource usage, speed and effectiveness. The solutions will be implemented on the Basys-3 and Zybo boards. The background for this paper is a project assigned to this group in the CS4110 course.

II. Merge Sort

A. Introduction to Merge Sort Algorithm

Merge sort is an algorithm based on divide-and-conquer method [1]. An unsorted array is continuously split in half until it cannot be further divided, either because the array is empty or because there is only one element left. After splitting, the merge operation starts. Two and two smaller sorted arrays are combined to create a larger sorted array. An array with only one element is considered sorted. The merge operations continue until there is only one sorted array left. This is the final sorted array. The sorting is illustrated in Figure 1.

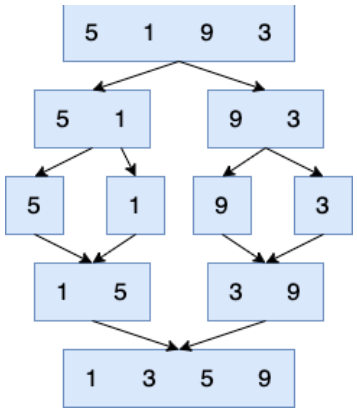


Figure 1: Merge sort

Pseudo code [2] is shown below.

```
split each element into partitions of size 1
   recursively merge adjacent partitions
     for i = leftPartIdx to rightPartIdx
       if leftPartHeadValue <= rightPartHeadValue
         copy leftPartHeadValue
       else:
          copy rightPartHeadValue;
          Increase InvIdx
     copy elements back to original
```

B. HLS Design and Implementation

Vitis HLS, Vivado and Vitis IDE 2022.02 were used for the software implementation of merge sort. C was chosen as programming language for HLS implementation of merge sort, for both HLS and the top-level software. The board chosen for the HLS implementation is the Zybo board.

1) HLS

The C code for HLS implementation is a modified version of the C code from [4]. The HLS was designed as a block handling the full sorting algorithm. Merge sort is often implemented with recursive function call, but to simplify the design for the synthesis, the algorithm is implemented with nested for-loops for the sorting. For further simplification of the design, the HLS code is limited to handle arrays with 8 elements, where each element is 8 bits. The elements can be either characters interpreted as ascii values or numbers.

The top-function is merge sort, and is the main entry point for the HLS code. This function takes an unsorted array and returns a sorted array. The function has an input array and an output array. The input array is divided into smaller arrays by nested for loops. The arrays are merged until the whole array is sorted. The HLS synthesis timing estimate is shown in Table 1.

Table 1: HLS Synthesis Timing Estimate

Target	Estimate	Uncertainty
10.00 ns	7,064 ns	2,70 ns

2) *Co-simulation*

The results of the co-simulation is shown in Figure 2.

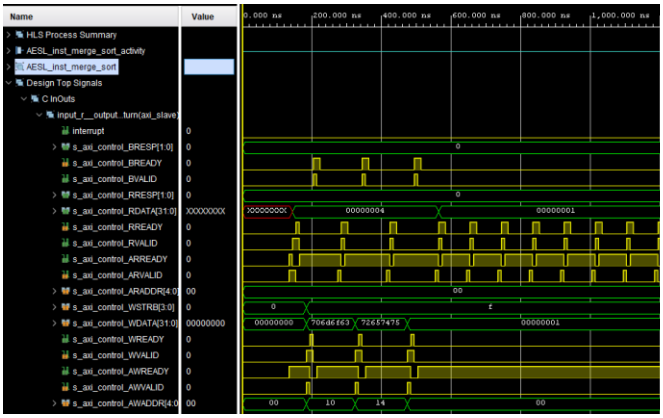


Figure 2: Co-Simulation waveform

3) *Hardware Design*

The block design is based on the Zybo board, and the IP generated from the HLS. The block design is shown in Figure 3.

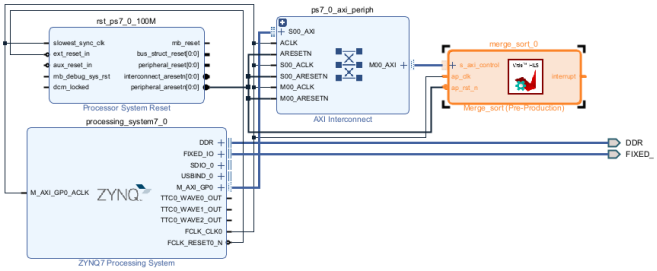


Figure 3: HLS Block Design

The utilization of the HLS implementation is shown in Figure 4.

Resource	Utilization	Available	Utilization %
LUT	848	17600	4.82
LUTRAM	92	6000	1.53
FF	867	35200	2.46
BRAM	1	60	1.67

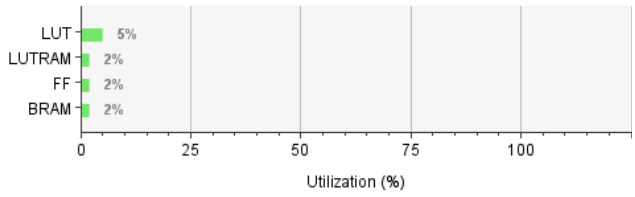


Figure 4: HLS Utilization

4) *Software*

The software implementation of merge sort is written in C. The implementation is based on the same architecture as the HLS implementation. The main function reads from input UART and passes the array to the merge sort function. Once the output array is sorted, the main function writes the array to the output UART. For simplification, the software implementation handles arrays with 8 elements, where each element is 8 bits.

C. *RTL Design and Implementation*

The FSMD and ASMD for the RTL design is shown in Appendix A and B. For simplification the design expects 2^A, where A is a positive integer. The implementation of the hardware design is not complete in Vivado. Most blocks are implemented, but the top-level block and FSM block is yet to be implemented.

1) *Counter Block*

The counter block consists of the input signals inc, clr, set, cnt_in and the output signal cnt_out. If the inc signal is triggered, the counter increases by one. If the clr signal is triggered the counter is reset. If the set signal is triggered the counter is set to the value from cnt_in. The output value is cnt_out.

2) *MUX block*

The MUX block consists of the input signals ctrl, data_in and ascii_r, and the output signal data_out. The signal ctrl controls whether data from receiving UART or data from temporary RAM block is written to RAM.

3) *Comparator Block*

The comparator block consists of the input signals left_value, right_value, ctrl, and output signal sel and value_out. The ctrl signal controls whether the comparator should set value_out straight to left_value or right_value, or if the input values should be compared. If the two input values are compared, the smallest value of the two becomes value_out. The sel signal tells which value was forwarded to value_out.

4) *RAM block*

The RAM block consists of the input signals data_in, addr_left, addr_right and wr, and the output signals data_out_left and data_out_right. The signal data_in is alwas written to addr_left whenever the wr signal is triggered. The signals data_out_left and data_out_right holds the value stored in the addresses where addr_left and addr_right points to.

III. Radix Sort

A. *Introduction to Radix Sort Algorithm*

The Radix sort algorithm is an algorithm that sorts integers. This is done by placing each digit in the corresponding bucket, starting from the least significant digit to the most significant bits. This process is repeated until the array is sorted. This done without comparing, so it is a non-comparative sorting algorithm.

Historically radix sort has been used inside card sorters. Where it was used to sort punched cards. The cards could only be punched in 12 different spots, so it was sorted on the 13 different states a card could have. Then the operator could collect cards which had the first row punched, second row and so on.

Pseudo code [2] is shown below.

```
create 8 buckets (queues) for each digit (0 to 7)
   for each digit place in
     for each element in list
       move element into respective bucket
     for each bucket, starting from smallest digit
       while bucket is non-empty
         restore element to list
```

A small example of how the algorithm sorts a small set of numbers can be seen in figure 5.

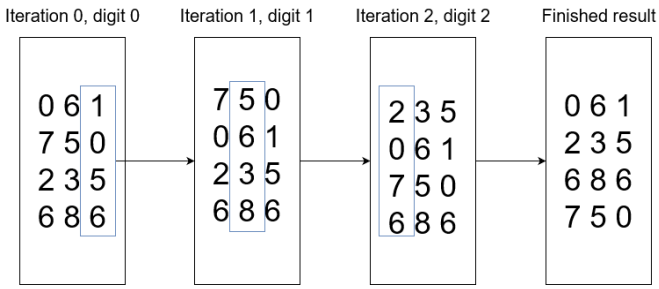


Figure 5 Example of radix sort

B. HLS Design and Implementation

A working C code example of the sorting algorithm was used. [3] The file was modified to accept serial communication and to be implemented in the Zybo board. This meant that the input had to be changed, as well as the output. To further complete the implementation a testbench needed to be created. However, there was a problem with the HLS software that did not make it possible to complete the simulation, this is further elaborated in chapter six. If the testbench was made it would been possible to test the code

After simulation, the next step is a Co-Simulation. The point of this operation is to verify the RTL result of the synthesis. This means that it is a way to verify that the hardware does the same and acts the same way as the C code.

C. RTL Design and Implementation

The design process of creating the hardware for the sorting algorithm was to begin to draw a sketch of the datapath. Everything from the receiver to the transmitter. The first iteration needed a divisor block to divide the numbers to get the modulus, and with this find the radix. The divisor was necessary because the circuit based the calculations on the decimal system (Base 10), and was meant to replace the need of modulus. With further discussion and reflection within the group, the octal system (Base 8) replaced the decimal system.

With this design it was no longer needed to use a divisor, and in its place a shift register could be used. The bits could be subjected to a right shift of $n * 3$ bits. With this the correct radix would be the three least significant bits. Therefore, space and resources were saved with using a shift register instead of a divisor, and two less RAM blocks with counters.

After finishing the sketch of the datapath, and ASMD chart was created. This chart would be an overview over the circuit over what happens when and where. This would form the basis of the different states and the state machine in its entirety. The ASMD chart was created with following the datapath and what path the data would take to be correct. The first iteration of the ASMD was partially incorporated into the fsm.vhd file, with some pseudo code at the end. This is because the limitations of the first design became apparent. Therefore, a new design was made, this design was fully implemented in a pseudo fsm. This version relies more on the concurrent nature of VHDL.

After both the datapath and ASMD chart, the work of creating the circuit in VHDL started. Most of the components used templates from the book. But the shift register had to be created from scratch, and the control path needed to be heavily modified for the initial draft.

With the VHDL for the component in its place, the creation of the top level design file was the next step. This required a lot of trial and error as this was the step that unified the components to one entity. Some of the ports had different bit width and addresses were still not perfect for the RAM/ROM components.

Since neither of the solutions was possible to simulate/test there is no conclusive evidence to compare.

IV. Bitonic Sort

A. Introduction to Bitonic Sort Algorithm

Bitonic sort is an algorithm which is highly parallelizable. Given an input of a known size, the comparisons happen in a fixed order, independent of the input values. This lends itself

well to parallel operations, which could be highly optimized on a device like an FPGA.

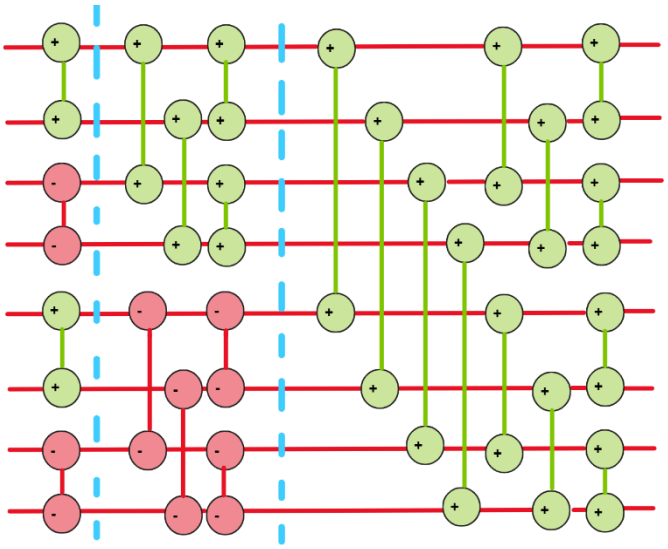


Figure 7 Bitonic sort algorithm for 8 values

The algorithm works by subdividing a list into smaller lists, and ordering these into bitonic lists. A list being bitonic means that the list is strictly increasing then decreasing. In other words, given a list a of length N, there is a value k so that all values before a_k are increasing, and all values after are decreasing

$$a_0 \leq \cdots \leq a_k \geq \cdots \geq a_{k+i}, \quad 0 \leq k \leq N$$

When we perform a bitonic sort, we split it into stages and passes. The number of passes in each stage increases linearly with the number of stages performed. Before starting, we split the list into sub-lists of size 2. From the definition of bitonic lists, all these sub-lists are guaranteed to be bitonic.

For the first stage of a bitonic sort, we want to combine pairs of the bitonic sub-lists of size 2, into a list of size 4. This is done through keeping track of a direction. The first of the pairs should be in increasing order, the second of the pairs should be in decreasing orders.

In the second stage we continue merging pairs of bitonic lists, this time pairs of lists of size 4 into a list of size 8. Each list is sorted into a strictly increasing or decreasing list, which together form a bitonic list. Figure [6] shows the comparisons performed for a bitonic sort with 8 inputs. The stages are separated by dashed lines. Green indicates increasing order swap, red is in decreasing order.

B. HLS Design and Implementation

The HLS implementation is based on pseudocode for a bitonic sort without recursion written in C [5].

The HLS implementation defines a code block which has an array with a fixed size as an input, and an array as a fixed size as an output. The sort_list function performs the bitonic sort algorithm, and the order function swaps the values depending on the values of i and k.

```
void sort_list(ap_int<DATA_WIDTH> arr[DATA_SIZE]) {
  int i,j,k;
  for (k = 2; k <= DATA_SIZE; k *= 2) {
    for (j = k / 2; j > 0; j /= 2) {
      for (i=0; i < DATA_SIZE; i++) {
        int l = i ^ j;
        if (l > i) {
          order(arr[i], arr[l], (i & k)  != 0);
        }
      }
    }
  }
}
```

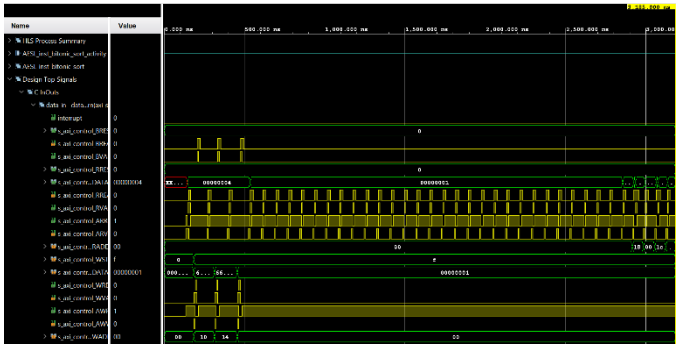


Figure 6 Waveform from HLS cosimulation of bitonic sort

```
char c = inbyte();

XBitonic_sort_Write_data_in_Bytes(&bitonic_sort,
offset, &c, 1);

if(offset < 8) {

  continue;

}

XBitonic_sort_Start(&bitonic_sort);

while(!XBitonic_sort_IsDone(&bitonic_sort));

char output[8];

XBitonic_sort_Read_data_out_Bytes(&bitonic_sort,  0,
output, 8);

for(int i = 0; i < 8; i++) {

  outbyte(output[i]);

}
```

The implementation assigns a fixed memory space for the input and output arrays, which are accessible later via Vitis IDE.

In Vitis IDE we read the values from the UART, and write them to the memory space dedicated to the input list. Once the list is filled up, we start the function, and wait until the process is finished. When the process is finished, we can return the result sequentially through TX

Resource	Utilization	Available	Utilization %
LUT	962	17600	5.47
LUTRAM	124	6000	2.07
FF	905	35200	2.57
BRAM	0.50	60	0.83

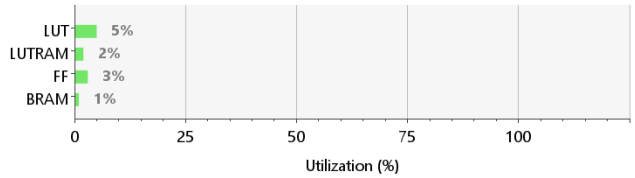


Figure 8 Resource usage of HLS implementation of bitonic sort

C. RTL Design and Implementation

Since the algorithm is done by doing a predetermined sequence of comparisons and swaps, our goal was to implement a completely parallel comparison network. This would allow us to instantly sort a list the moment it is registered. The only time required to sort would be the time required to send the input and output values through the serial interface.

While this gives a simple top level and FSM design, it greatly complicates the generation of the sorting network. To make it a bit easier to design, we have decided that the size of the list N has to be a power of 2

1) Generating the comparator net

Since the pre-determined sorts depends on the size of the input, and the positions to compare and how to sort them changes in each stage of the process, the sorting network had to be generated based on input size. Each of the comparators have 2 inputs, and 2 outputs. The higher value output will always be the same for any single comparator, depending on how it was instantiated.

Through analyzing the structure of the algorithm, we identified 4 main loops that could be used to generate the required comparators:

1. The stages, containing one or more passes
2. The passes per stage
3. The groups of comparators
4. Each comparator within a group, offset by only one

The number of iterations for each of these loops is as follows:

$$number\ of\ stages = \log_2 N$$

$$number\ of\ passes = stage\ number$$

$$number\ of\ comparator\ groups = \frac{N}{2^{current\ stage - current\ pass}}$$

$$number\ of\ comparators\ in\ group = 2^{current\ stage - current\ pass}$$

With these numbers, we can now generate the correct number of comparators and can know where in the network they each is placed, but we have not connected them yet. We need to connect the inputs of the first comparators and connect each stage of comparators with the next. To do this, we set up an array able to fit all the input values, intermediate values, and output values. This requires us to set up an array that can fit all of this.

$$number\ of\ comparators = \frac{N \log_2 N\ (\log_2 N + 1)}{4}$$

$$array\ size = number\ of\ comparators * 2 + N$$

This array size can fit all of the comparator outputs, in addition to the initial input list. We are now ready to route all the signals now that we have an array, and a loop to generate all the comparators. By deciding to map each of the passes sequentially in the memory, this makes it easy to find the input address if the output is known.

$$input\ list\ address \in [0, N]$$

$$\begin{aligned} comparator\ output\ address\ 1 &= 2 * 2^{current\ stage - current\ pass} * comparator\ group \\ &+ comparator\ offset \\ &+ \left(\frac{((current\ stage)^2 + current\ stage + 2)}{2} - 1\right) N \\ &+ current\ pass * N + N \end{aligned}$$

$$\begin{aligned} comparator\ output\ address\ 2 &= comparator\ output\ address\ 2 \\ &+ 2^{current\ stage - current\ pass} \end{aligned}$$

$$comparator\ input\ address\ 1 = comparator\ output\ address\ 1 - N$$

$$comparator\ input\ address\ 2 = comparator\ output\ address\ 2 - N$$

These work in the following way

$$\begin{aligned} &Number\ of\ outputs\ in\ earlier\ stages \\ &= \left(\frac{((current\ stage)^2 + current\ stage + 2)}{2} - 1\right) N + N \end{aligned}$$

$$Number\ of\ outputs\ earlier\ in\ this\ stage = current\ pass * N$$

$$\begin{aligned} &Offset\ for\ current\ group\ of\ comparators \\ &= 2 * 2^{current\ stage - current\ pass} * comparator\ group \end{aligned}$$

For the second output, the position is offset with the distance between the second and first outputs. This gives us a way of computing where in the array to read values, and where to place the results. The only remaining step now is determining if a comparator should sort low to high, or high to low. This changes based on the current size of bitonic lists, and which comparator block is currently being generated.

$$direction = \frac{comparator\ group}{2^{current\ pass}} \bmod 2$$

This is integer division, so the result of the division is truncated. The sorting direction can be changed by deciding what direction means up and down.

The rest of the connections are much simpler after this. The output from the sort is the last N values from the sorting network, which can then be sent to the parallel to serial register for serial output.

2) Serial to parallel, parallel to serial

While the inputs and outputs from the UART interface is serial, the bitonic sort design has a completely parallel input and output. The data has to be stored and output in a way that makes it possible to read serial data, and output the data as serial. The simple solution to achieve this is a serial to parallel shift register, and an equivalent solution in reverse for the output. To avoid

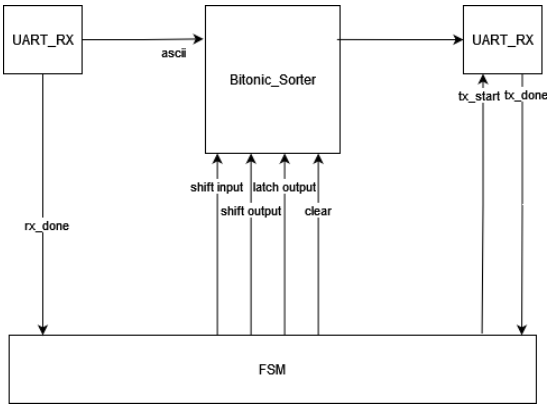


Figure 9 Top level design of bitonic sort

having a coupled input and output, the bitonic sort module needs a control signal to say if the block should read a value to the input or shift the output. It should also be possible to set the output register to contain the sorted input.

This leads to having at least the following controls:

- An input data port
- An output data port
- A way to shift the input
- A way to shift the output
- A way to store the sorted list in the output

In addition to this, we could benefit from having a way of clearing the output. This leads to a design of the bitonic sort block with 6 ports. From this we get our design of Figure [10], where the comparator network is generated as previously described.

Serial to parallel is a simple process, and simply requires storing every value in the next block in the chain. This can be achieved by chaining 1 word data storages together, connecting the output of the data storage to the input of the next. The number of data latches is determined by the capacity of the sorting block. The data that passes through the last data register is discarded, meaning that the first data sent is discarded when the data exceeds the capacity.

Parallel to serial is similar, but there has to be a way to control if data is stored in the register, or if it's shifted to the next value. There is a variable amount of data registers depending on the capacity of the sorting block.

To decide between reading the sorted list, and shifting the output to the next bit, there is a data multiplexer which swaps between inputs, and the output of the registers. The data from the registers is shifted so the data is sequentially output in the correct order when it is shifted through.

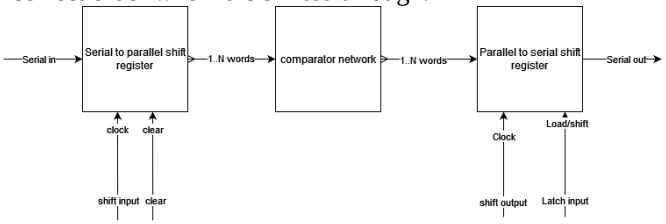


Figure 12 Internal design of the bitonic sort block

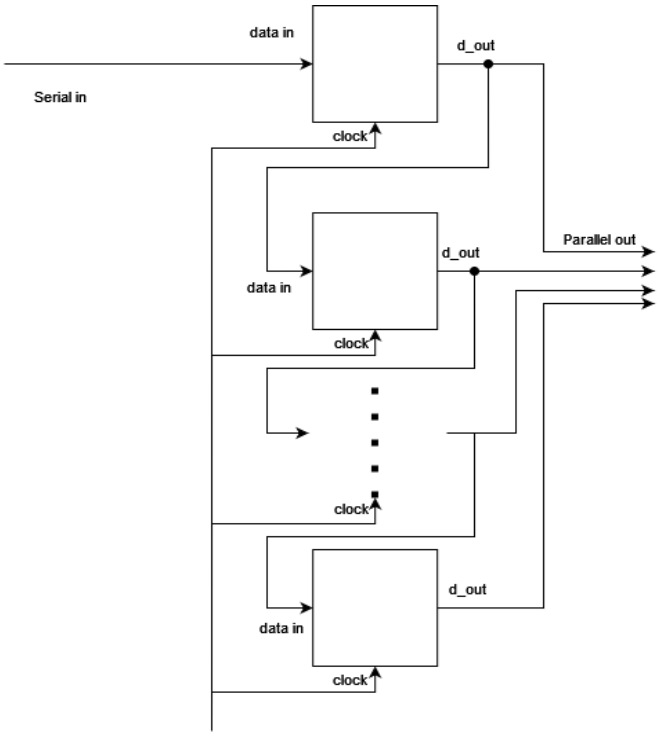


Figure 10 Serial to parallel output

3) Testbench results

Given the lack of states involved in the actual sorting algorithm, no FSM was required to perform the sort itself. While it would be required to implement a full solution with UART communication, the bitonic sort block is stateless.

To test it, we can create a bitonic sort component, and pass in an unsorted list through the serial interface.as the numbers are passed in, we should be able to observe a change every time in

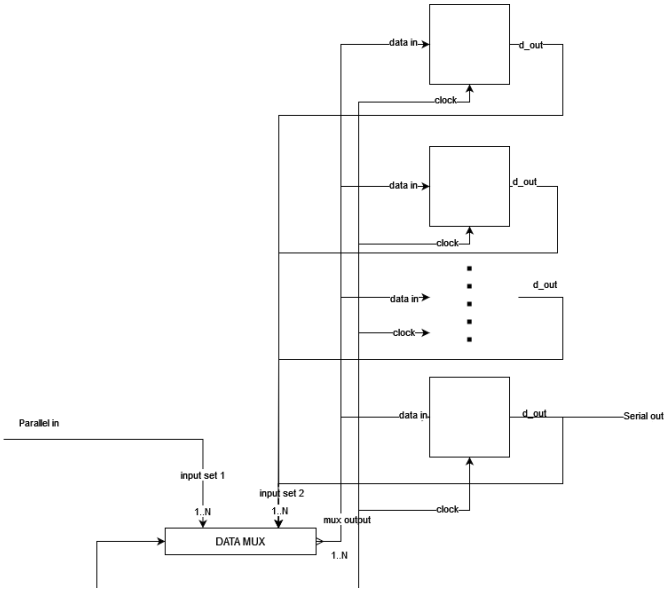


Figure 11 Parallel to serial output

Resource	Utilization	Available	Utilization %
LUT	368	20800	1.77
FF	136	41600	0.33
IO	20	106	18.87

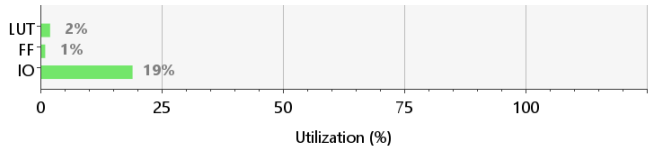


Figure 13 Resource usage of RTL design with 8 values

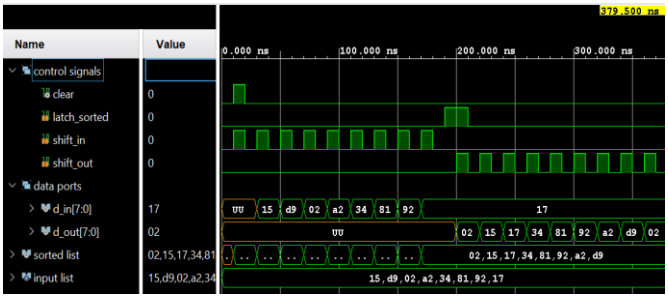


Figure 14 Waveform output of simulation of bitonic sort in Vivado

the internal output of the comparator network. Once all values have been input, we can write the sorted list to the output register, and sequentially go through and read all values in the sorted list.

The waveform output of the testbench is seen in Figure [14] and shows the list updating whenever a new value is inserted. Since the output from the sorting network is a result of a fixed sequence of comparisons, there is no delay in the output, and the list is sorted instantly. When the last number is inserted after 170ns, the list is already sorted.

D. Results

Number of elements	Number of comparators
4	6
8	24
16	80
32	240
64	672
128	1792

V. While the time required to sort using this RTL implementation is only limited by internal signal propagation, it is not a very effective use of registers. The number of comparators required to sort a list, scales with a complexity of $O(nlog^2n)$, and can quickly consume all resources available on the device. In the case of this implementation, the LUTs utilized exceeds the available resources after increasing the list size to only 64 elements.Conclusion

A. Merge Sort

The result of the merge sort algorithm with RTL vs. HLS implementation is not conclusive. This is because the work with the RTL implementation is not complete. The results of the HLS implementation are covered in chapter IV. B. HLS Design and Implementation.

B. Radix Sort

The result of the radix sort algorithm is not conclusive, because of the fact that neither one ran on a device or in a simulation. This is because several reasons discussed in detail in “chapter 6. Future Work”.

C. Bitonic Sort

Overall, bitonic sort is an algorithm well suited for parallel processing of the type one normally does on an FPGA or GPU. Because of the predetermined order of operations, it’s easy to split into sub tasks and run in multiple simultaneous processes.

Even if it is well suited for parallel processing, with the scaling of the number of comparators this RTL implementation is not a very practical implementation of the algorithm unless speed is crucial to the system. The algorithm does not seem to be the best algorithm to use either, as there are other algorithms with a similar working principle that use less comparators total.

The difference between the RTL and HLS implementation shows the affect pipelining can have on the timing but will also lead to less resource intensive implementation. The HLS implementation compares and swaps the numbers in place, which overall leads to a slower implementation.

The main benefit of the RTL implementation is that it scales better with resource usage and larger lists to sort, so it can handle more data with less FPGA hardware resources. This is at the cost of speed however, and there is a larger latency because of the time it takes to perform the sorting operation

VI. Future Work

A. Merge Sort

Both the HLS implementation and RTL implementation has limitations with regards to number of inputs. The HLS implementation requires exact 8 element input, while the RTL requires 2^A number of inputs, where A must be a positive integer. This has been done to simplify the design, but for future improvement, the design should be able to handle a various number of inputs.

For future improvements, the RTL implementation work should also be completed.

B. Radix Sort

To improve on the current solution of radix sort, a redesign of the current hardware design may be a good move. The current one has multiple pieces that could increase complexity. For example, all the different ram blocks that is in use. Each one requires a counter and several signals to the control path. This may be improved by using a solution with a centralized control block for the counters, or a centralized ram block. However, if these solutions are improvements or even viable are unknown.

Table 2 Number of comparators for each number of elements

In addition, another point of improvement may be using a modulus function in the VHDL instead of using a shift. How this will affect performance and resources is also unknown.

The control path is also incomplete due to the complexity and poor design of signals. The next iteration requires more status signals, signaling if processes is completed or not. This simplifies the transitions between states.

Vitis HLS was not completed as well because of software troubles. Vitis could complete the simulation because of errors encountered during the csim.exe generation. Xilinx support forum recommended doing the simulation on a Linux machine. Therefore because of operating system incompatibility the simulation was not done. For future work there may be an advantage to use a Linux machine.

C. Bitonic Sort

As mentioned, there are better options for sorting algorithms to use. A couple of better alternatives includes batcher's odd-even mergesort and pairwise sorting network. Both have the same performance and complexity, but they need less comparators than the bitonic mergesort network.

In addition to this, one can pipeline the sorting instead of doing everything in parallel through a sorting network, which could reduce system usage to manageable amounts even for larger lists of data, at the cost of speed. This is more like what the HLS implementation does, which is also why it takes longer to sort the list.

REFERENCES

[1] GeeksforGeeks. (2022). GeeksforGeeks, [online]. Available: https://www.geeksforgeeks.org/merge-sort/

[2] VISUALGO.NET. (2022). VISUALGO.NET, [online]. Available: https://visualgo.net/en/sorting

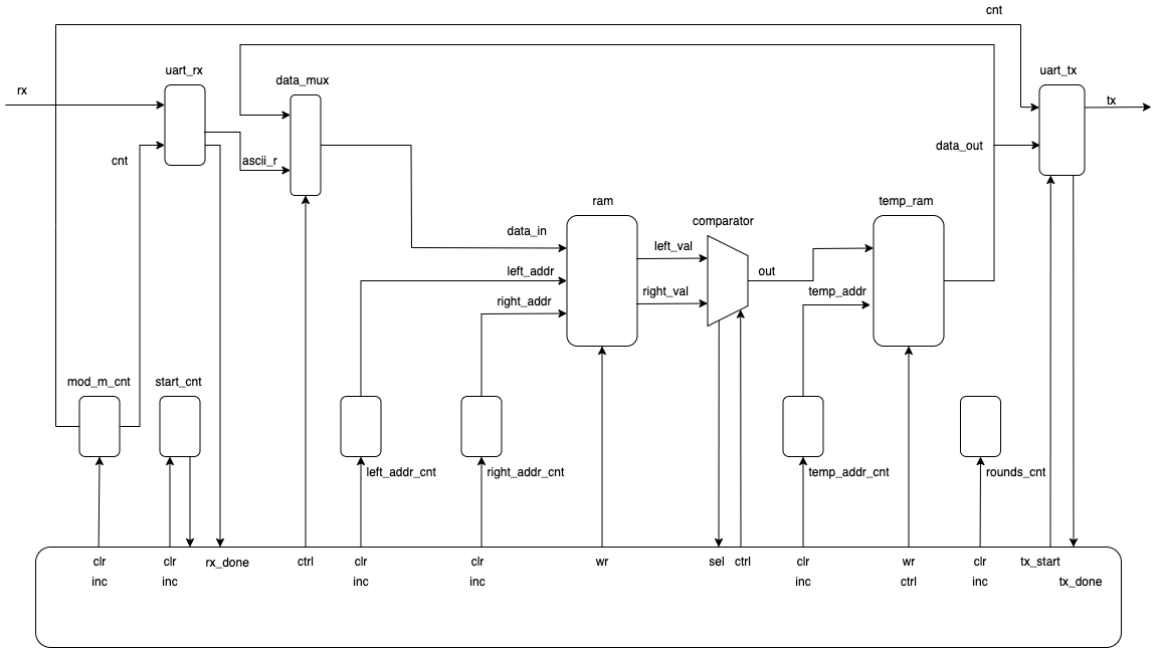
[3] Programiz. (2022). Programiz.net, [online]. Available: https://www.programiz.com/dsa/radix-sort

[4] R. Kastner, J. Matai and S. Neuendorffer, "Parallel Programming for FPGAs", 2018, p. 191.

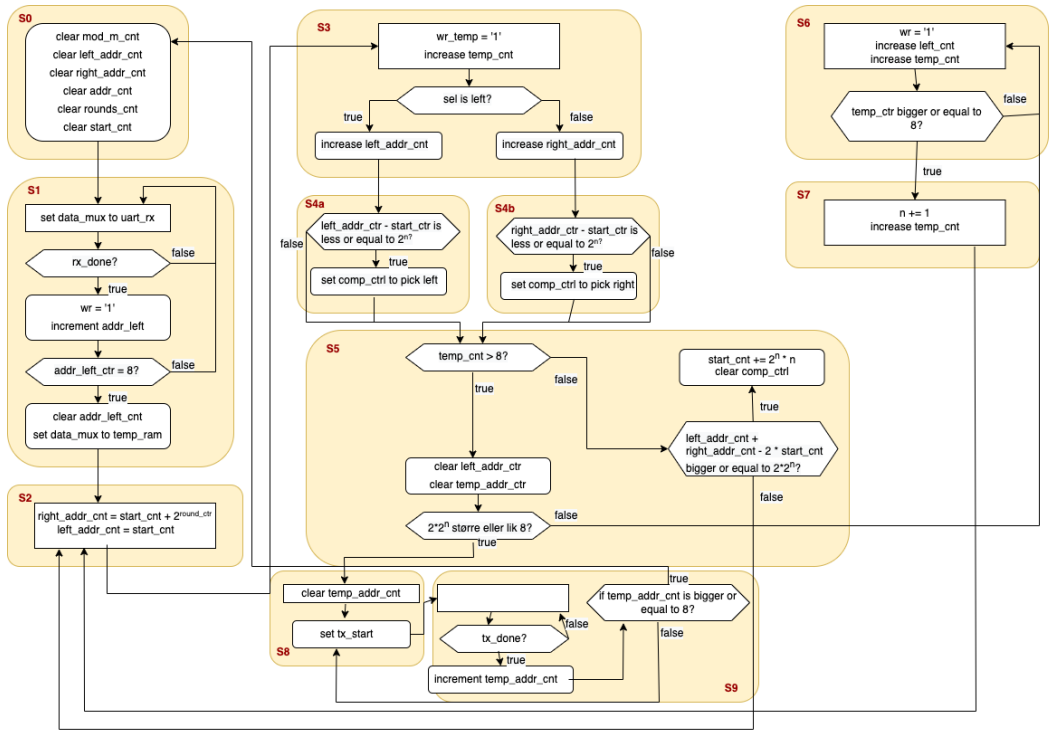
[5] N.Pitsianis (2022) Duke [online]. Available: https://www2.cs.duke.edu/courses/fall08/cps196.1/Pthreads/bitonic.c

APPENDIX

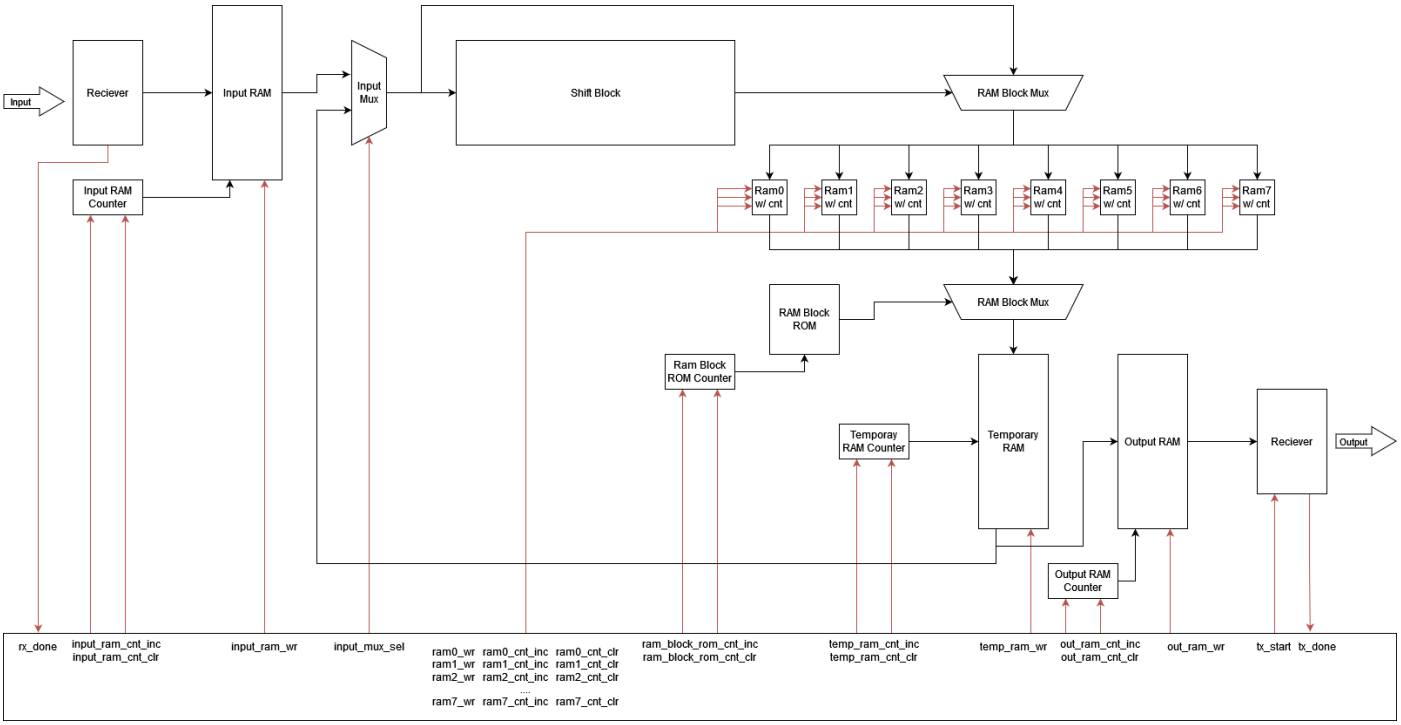
MERGE SORT FSMD



MERGE SORT ASMD



RADIX SORT FSMD



RADIX SORT ASMD