Lab2 Sorting AcceleratorReport

Yibang Xiao (yx455), Yuxiang Long (yl3377)

Introduction

As we all know, sorting an array of a large number of values can be somehow time-consuming. So if an accelerator could be implemented, the processor can send the unsorted array to the accelerator and keep on executing other instructions instead of just waiting for the sorting process. As a result, the efficiency could be greatly improved. In this lab, we are required to design an accelerator that is used for doing this work and is also suitable to sort the random length of the array. This lab could also help us better understand how the hardware could be used to speed up some specific functions. And during the lab section, we successfully completed the alternative design, all the test cases have been passed, and the RTL design is also successfully pushed through the ASIC tool flow. The main principle of our sorting function is that the accelerator takes in data, compare it with the existing data in the registers with index ranging from small to large until this data becomes the smaller one. Then this input data will be stored in the register with the corresponding index. And in order to better evaluate and optimize the performance of our accelerator, we set the sorting length of our accelerator to be 16, 32, 64, 128, 256,... which means the maximum length of the array that the accelerator could sort at one time. There are certainly some trade-offs. When the sorting length becomes larger, the number of sorted subarrays could be less, and this can save execution time, but larger sorting length also means a larger hardware area. We need to think about balancing them.

Alternative Design

Accelerator Hardware Part

Our alternative design is a sorting accelerator working as a co-processor besides the processor. It is a medium-grain accelerator which is initialized by a writing-to-xr0 instruction sent from the processor. This accelerator receives the address of the first element in an array and the size of it. After these two configurations, the accelerator will iteratively read data from the data cache, sort them, and finally write them back to replace the original out-of-order array. The accelerator has three layers. The first layer is a large state machine with five states to receive configuration from the processor, read data from the data cache, pass the data to the sort-unit, and write the in-order array back to the cache. The second layer is a module that combines the data-path and control part of sort-unit. The last layer is the detailed implementation of the data-path and control of our sort-unit. We will introduce the third layer first, then talk about the second layer, and lastly we will discuss the first layer. The reason for this reverse sequence is that the third layer which is the datapath and control part determines the design of our first layer, the FSM.

The third layer (the sort-unit datapath and control module) is the implementation of the insertion sort in the hardware. The insertion sorting was chosen because the input data can be compared with all of the sorted numbers stored in the sort-unit and inserted to the desired position of the array in one cycle by using lots of comparators and multiplexers. When the new data comes in, the hardware has already known where it should be inserted and the rest of the register units have three possibilities: hold its own value (freeze), replace with the inserted data (insert), or accept the number from its neighbor register and pass its own data to its next one (descent). There is one more possibility: when we first initialize the accelerator, zero should be inserted in every register so that the comparison can generate the correct result and also the hardware can clear the results for the next sorting iteration. Therefore, it needs a 4-input multiplexer with the proper selection signal to choose the right case. For example, we have a sort-unit with sorting length four shown in Figure 1. The sorting length four means that it can sort 4 numbers in one round, or iteration. The 4-input mux connected with a Flip Flop could write data into each register. If looking at the second mux from the right-hand side (result[2]), the first input is zero for initialization; the second input is the old data it holds (freeze); the third input is the number from its neighbor, result[3] (descent); the last input is the new data coming in (insert). For the first mux from the right-hand side (result[3]), the descent input is null because it does not have its neighbor on the right and it always stores the largest number in the current round. A Flip Flop has to be placed at the output of the mux because it can make all registers in the sort-unit update the data synchronously. This is the datapath for the sort-unit and all mux selection signals are generated from the control module. In the control module, the new number read from the data cache will be compared to from the right-most register (in our example, it is the result[3]). If it is smaller than the value in result[3], then it will be compared to the result[2]. If this new data is larger or equal than the value in result[2], the control module will set the selection signal of mux2 to 3 (insert). In the meanwhile, all of the mux_sel on the left-hand side of mux_sel[2] (mux_sel[0] ~ [1]) will be set to 2 (descent). They will pass the value to their left neighbor so that the sorted array will not be disturbed. Finally, all of the mux_sel on the right-hand side of mux_sel[2] will set to 1 (freeze) because there is no change of sequence on those numbers which are larger than the inserted data. Also, the control module has a counter to count how many numbers it has sorted and can assert a 'full' signal if they reach the maximum capacity, which is the sorting length we defined as a global parameter.

The second layer shown in Figure 2 is a module that combines the datapath and control module of the sort-unit. The datapath outputs the numbers in the result[] to the control module and the control uses behaviour style to compare the new data with all the numbers stored in the sort-unit. Once it finds where this new number should go, it will generate the correct selection signals and output them to the datapath. There is one more function of this layer. This layer can see which round the sort-unit is doing so that it is able to distribute the 128 numbers to the output of this layer in the right sequence. For example, in the first round of sorting 200 numbers, the sort-unit loads 128 numbers and sorts them from small to large. When writing them back, this layer starts output the sorted numbers from the beginning of the result[] (result[0]). In the second round, the sort-unit only needs to sort the rest of the number, which has 200-128=72 numbers. Therefore, when outputting them, we should start from the result[56] (128-72=56) instead of the result[0].

The first layer is a big Finite State Machine with five states. The state diagram is in Figure 3. The X stage is used to let the processor configure the accelerator. The commands from the processor can write to the xr1 and xr2 register of the accelerator. The xr1 is the base address register which stores the address of the first element of the array. The xr2 is the size register which holds the total size of the array that it needs to sort. If the instruction is a write to xr0, then the 'go' signal will be asserted and the FSM moves from X stage to the MEM_RD state. In the MEM_RD state, a memory read request message which contains the initial address will be sent to the data cache to request one data. If the FSM receives the data coming from the cache, it will pass it to the sort-unit (the second layer), and assert the signal 'sorting_go' to initiate the sorting process. Also, this state will increment the reading counter and the byte offset (+4) the next time the hardware goes in this state. After the sorting process is started, the new data will be inserted in the correct position in the data array in sort-unit in one cycle. Then if the sort-unit is full, the hardware cannot sort any more numbers and it must go to the MEM_WR state to write the sorted number back. If it is not full, then the FSM will go back to the MEM_RD states, read one more number from cache, and sort it. This read-sort-read-sort progress will not stop until the sort-unit is full. The FSM must go to the MEM_WR state after the sort-unit is full. In MEM_WR state, a memory write request message which contains the same address from MEM_RD state will be sent to the memory. If the memory request ready signal is high, which means the memory is ready to receive the message, the request will be sent and the FSM moves to the MEM_WAIT state. The FSM does not leave this state until a memory response ready signal is received. There are three possible states for the FSM to go. The first one is back to MEM_WR. There is a counter which counts how many numbers have been written back to the memory. If the counter does not reach the value of the sorting length of the sort-unit, it means that not all sorted numbers have been sent back. Therefore, the FSM needs to go back to MEM_WR to continue sending data back to memory. When the counter reaches the sorting length, which means all sorted numbers have been sent back, it will move to either X state or MEM_RD state depending on the 'iteration' counter. When the size is written to the xr2 register in X state, the hardware can calculate the number of rounds, or called iteration, by using size_in divided by the sorting length. For example, if the size_in is 64 and our sorting length is 32, then the total number of iterations needed is 64/32=2. In the first round, the "iteration" is 1, which is smaller than 2. Therefore, when the FSM enters the MEM_WAIT state in the first iteration, it knows that there is one more to go before it goes back to the X state. As a result, the FSM moves to the MEM_RD state to read the next round numbers.

Our three layers (including the datapath and the control) are seperated in different files. If more sort-units are added to realize some parallel processing in the future, the developer can just call sort-unit twice in layer two without touching the details implementation in the sort-unit. The whole accelerator in isolation is portable if the same xcel/mem communication interface is used. The datapath and the control module of the sort-unit is parameterized. To test and evaluate the different length accelerator, a small adjustment on the parameter 'c_sorting_length' in the first layer is needed.

Accelerator Software Part

The accelerator we designed could sort a selected length of the array at one time, it is also able to sort an array which is shorter than the selected length. And if the array is longer than the selected length, this array will be separated into several sub-arrays, then the accelerator could sort them respectively. But the sorted result is not the final result, we need to use software to merge these sorted arrays together to form the final sorted array. The merge function we use is to merge two sorted subarrays at one time and then merge another one subarray with the previous sorted array. The algorithm we use to merge is to compare the left-most value of the two arrays first, then choose the smaller one and put it into another blank array which is used for the temporal merge result. After this, the index of the subarray that holds the smaller value will be added so the next value in this array will be used to compare. To be more detailed, we can use an example to explain it: [1, 3, 5], [2, 4, 6]. The index of these two arrays is initialized to be 0. And so the first value of these two arrays will be compared first. 1 is smaller than 2, so 1 will be put into the temporal array [1]. The index of the first array will be added 1 in the next iteration. Then 3 will be used to compare with 2, and 2 is smaller than 3, so the temporal array will become [1,2, ...]. When the index of either array reaches its end, the software will copy the remaining values of the other array into the temporal array. Then this temporal array will be used to merge with another subarray. This software is also capable of merging two arrays of different lengths. When all the arrays are merged, the software will output the final result and compare it with the reference.

Evaluation

Chart 1 displays the relation between the cycle number and the sorting length (or called sorting capacity C) of our isolated accelerator. The accelerator we designed has a fixed number of sorting cycles under every specific length of the array. If the length of the out-of-order array is N, then the number of cycles it took to finish sorting and writing them back to memory is around 4N under the xcel isolation test (assume test memory has a few latency and 0.5 as stall possibility). Because we are now doing evaluation of the single accelerator without the software mergesort, our test array sets are equal or smaller than the sorting capacity. The reason for 4N can be explored from the FSM of this design. The first state is the initial state "XCFG" which waits for the configuration commands from the processor. After the accelerator initiates, it then goes into the state of "MEM_RD" where the accelerator starts reading data from the memory. After one data arrives at the accelerator (assume no memory latency), the state goes into "SORT", where the accelerator starts inserting this new data into the sorted register arrays in the sort-unit, and this can be done in one clock cycle. These two states which sort one number take 2N cycles. Then the accelerator will write the sorted array back into the memory, we can see from the state diagram that this writing process will also take 2N cycles (again, assume no memory latency). We can confirm this by evaluating execution cycles under different kinds of arrays. We tested our accelerator with some different sorting capacity of one round: 16, 32, 64, 128, and 512 by feeding them the unsorted dataset with the same length of each case. And the result showed that the execution time increases linearly from 77 cycles to 2061 as the length of the array goes up. Take the sorting length of 16 and 512 as examples. The 16 numbers come in and it takes 4*16=64 cycles to read them, sort them, and write them back. The resting 77-64=13 cycles is the overhead time to configure the accelerator in "XCFG" state. For 512 numbers, it takes 512*4=2048 cycles and plus the overhead time which is 2048+13=2061 cycles. Our analysis matched with the cycles generated by the xcel-simulator. It is worth mentioning that the Chart 1 also shows the best and worst cases of our design. The sorting cycles of the random array, forward array, and reverse array are the same because our sort-unit always needs one cycle in "SORT" state to update the new sequence no matter the data coming in is the smallest or the largest. In other words, we do not have any best cases and worst cases for this accelerator. It will always take 4N cycles to sort N numbers if N is less than or equal to the sort-unit capacity, C.

Chart 2 shows the graph between the cycle numbers and the process-memory-xcel (PMX) system with different accelerator capacities. With the assistance of the processor and memory, we can now sort N numbers which N can be larger than the sort-unit capacity by adding a software merge sort running on the RISC-V processor. We can see that the number of cycles decreases as the sorting-unit capacity increases and when the PMX has sorting capacity 64, its cycles is less than the quicksort-v3 (only half of the quicksort-v3) which is a sorting algorithm optimized on the processor. Starting from 64, the accelerator brings the effect of speeding up. The reason for this is that a sort-unit with larger sorting capacity means there are more sorted datas and less number of subarrays for the merge sort function. As a result, there is less work remaining to be done by the merge function. And as we discussed before, the accelerator only takes 4N clock cycles for sorting N numbers each time so it helps the merge sort reduce the sorting pressure. Also from Chart 2, we can see that if the accelerator is not implemented, the best performance of the quick sorting (v3) algorithm is 12076 cycles. Then the best performance when the accelerator is implemented could be less than 1608 cycles, which means the sorting speed could be improved for 7.5 times. The reason that sorting time is nearly the same for accelerators with 128 and 256 sorting length is that the length of the data array input is kept the same: 128. In conclusion, the larger the sorting capacity is, the less total cycle number it will take. There are two scenarios when the accelerator does sorting: first one: the number of incoming dataset, N, is less than or equal to the sorting capacity, and the second one: N is larger than the sorting length. We added several data randomly in the 128-data array in the .dat file and fed them to our xcel with 128 capacity. The result compared with the original 128-data array is in the Chart 3. The input datasets are 128-data array, 129-data, 130-data, 131-data, and 132-data. As shown on the graph, there is a sharp increase when the number of data inputs change from 128 to 129. The reason is that now we set our sorting capacity as 128. If we have 129 data to be sorted, the merge sort from the software side has to involve in to merge the sorted 128-data-array with a 1-data-array. The merge sort takes nearly 5000 cycles to load 128 sorted data from the memory, compare it with the single one data, and write them back. Chart 4 is the comparison which removes the cycle number when sorting 128 dataset. There is a small increase in the cycles when we add one more data. Their relation is linear so we generate a linear equation between the cycle number and the input number of data X, where X is larger than the sort-unit capacity. The equation is y=54.2x-524.13. We can predict the cycle it takes if we know the number of data we are going to feed. For example, if we feed the PMX 256 numbers, it will take around 54.2*256-524.13=13351 cycles for the accelerator hardware and the merge sort to sort them.

Although the accelerator could improve speed, it will add more area to the hardware. So we evaluated the design area of different schematics (basic components are processor and cache, and then the varied part is no accelerator, accelerator with sorting length of 16, 32, 64, 128, 256), the results are shown below in Chart 5 and Chart 6. We can see that if only the area of the accelerator is considered, the area is nearly proportional to the sorting length of the accelerator, ranging from 12404 um2 to 121640 um2 as the sorting length increases from 16 to 256. The reason is obvious because larger sorting length requires more registers to store the sorted number in sort-unit, more multiplexers to choose the right inputs, and more comparators in the control module. Everytime we double the sorting capacity, the design area of the isolated xcel will also be doubled. Then we also evaluated the total area which includes the processor, cache and the accelerator, as shown in Chart 6, we can see that when no accelerator is implemented, the area is more than 350000 um2, and then the area goes up as the sorting length of the accelerator increases. We combined the area evaluation result with the result from Chart 2, and found that if the sorting length of the accelerator is set to be 128, compared to no accelerator, the performance could be improved for 7.5 times for the number of dataset which is smaller than the sorting capacity and improved for almost 2 times for those dataset who has more number than the capacity, while the area only increases for 20% to be 420000 um2. From the area report of the accelerator part, the total area of the accelerator is 64927 um2. Also, we can see that the main area is the third layer, the datapath and control module of the sort-unit which plays the most important role in sorting the array, and the area of it is 48217 um^2 which is 74.9% of the total area. This sort-unit consists of the datapath which is 31428 um^2 (48.4% of the total area and 65.2% of the sort-unit) and the control logic which is 16036 um^2 (24.7% of the total area and 33.3% of the sort-unit). If we increase our sort-unit capacity to 256, it can sort 7.5X faster if the number of data is less than 256, while the PMX area increases to around 495000 um^2 which increases another 17.8% from PMX-128. As a result, we want to balance both the speed and the area so we think the accelerator with 128 sorting length could be the appropriate design.

Another reason we chose 128 as the appropriate design is the clock slack. When the clock period is fixed at 2.5 ns , as the sorting length of the goes up, the clock slack will go down, as shown below in Chart 7, which means it will be harder to meet the timing constraint. Therefore, there is a trade-off between execution speed and timing, we need to find a balance between them. We set the Max_Delay equal to 0.9 for Xcel-16 to Xcel-128 to constrain the synthesis harder when the tool synthesises the RTL. However, the Xcel-256 and Xcel-512 cannot meet the timing closure no matter if there is Max_Delay or not. We checked the critical path and the result shows that there is a long combination logic path which uses modulus operation to calculate the number of the rest of the unsorted data in "MEM_RD" state. We think that the modulus operation, similar to the divide operation, has a critical impact on the timing so we break this path by implementing the first part in "MEM_RD" state and the second part in the "Sort" state because the value does not change until leaving the "Sort" state. Therefore, this long path is done by two clock cycles and it does relieve the timing pressure. After this change, Xcel-256 and Xcel-512 can meet the timing closure. We dig into the timing report and it shows our critical path in Xcel-128 is in the first layer we talked about in the alternative design. The critical path starts from the clock tree, through the input/output 'iteration' of the second and third layer, to the control module of the sort-unit, and stops at the multiplexer selection signal. In this path, the device that consumes the longest time is the cascaded full adder with subtraction operation. The critical path plot from Cadence and Amoeba plot from Cadence are also shown below in Figure 11 and Figure 12.

Then comes the power and energy evaluation. As shown below in Chart 8 and Chart 9, if the length of the input array is fixed at 128, we can see that when no accelerator is implemented, the energy consumption is 523.9 nJ, and then when an accelerator with sorting length of 16, the energy consumption will rise to be 956.5 because there are extra hardware such as flip flop, muxes, and registers. And then the energy consumption will decrease as the sorting length of the accelerator goes up. When the sorting length reaches 128, the energy consumption will drop to 347.6 nJ. The reason for this drop is that although there is much more extra hardware in Xcel-128, it can sort the 128-dataset without using the merge sort. The merge sort still consumes energy on the processor. In other words, reducing the work on merge sort on the software side can decrease the energy that the system used. As a result, the total energy consumed by Xcel-128 could be the lowest in this case. Another thing we can see is that when the sorting length changes from 128 to 256, the energy rises significantly, this is because our test array length is only 128, which is less than 256, so it does not use the merge sort either. However 256 sorting length means double the number of the registers, muxes, and the flip flop on the datapath of the sort-unit, which causes more energy consumed by the hardware. Chart 9 shows the relation between the energy and the performance (the cycle numbers to finish sorting 128-dataset, less cycles means better performance), we can see that for the case of 128-dataset, PMX-128 and PMX-256 owns the best performance. However, PMX-256 has larger energy than the PMX-128 does because PMX-256 needs more energy to manage the 256 registers in sort-unit. In order to dig deep into the energy evaluation, we also generate the power result of accelerators with different sorting length when the array length is fixed at 128, as shown in Chart 10. As we can see, as the sorting capacity of the accelerator goes up, the power also goes up, which seems proportional to the sorting length, this result also proves that more hardware will bring more energy consumption.

In conclusion, everything comes with a trade-off. Although the area and power of the PMX-128 is larger than the design with smaller sorting length, the execution time and energy consumption of the PMX-128 are all lower than designs with larger sorting length. In addition, the PMX-128 design could also minimize the workload of the merge function, thus could further save execution time and energy. So the PMX-128 design is the most optimized choice for the 128-dataset input. It maintains a good balance between the area, the energy, and the cycle numbers.

Apendix

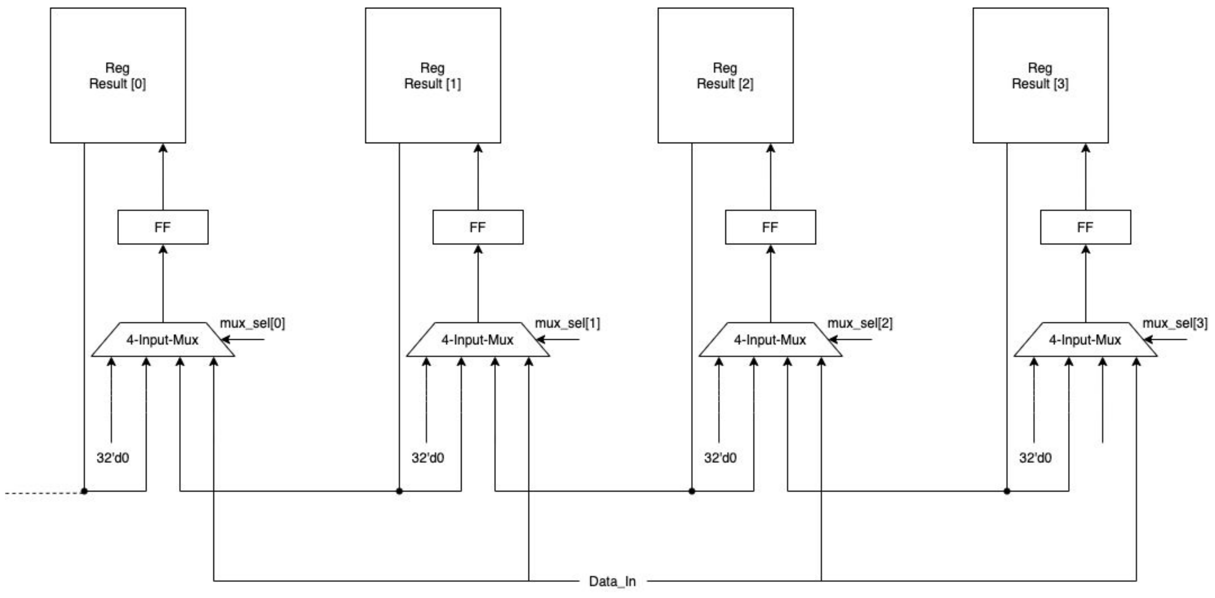

Figure 1: Third Layer: Datapath of Sort-Unit with sorting length of four

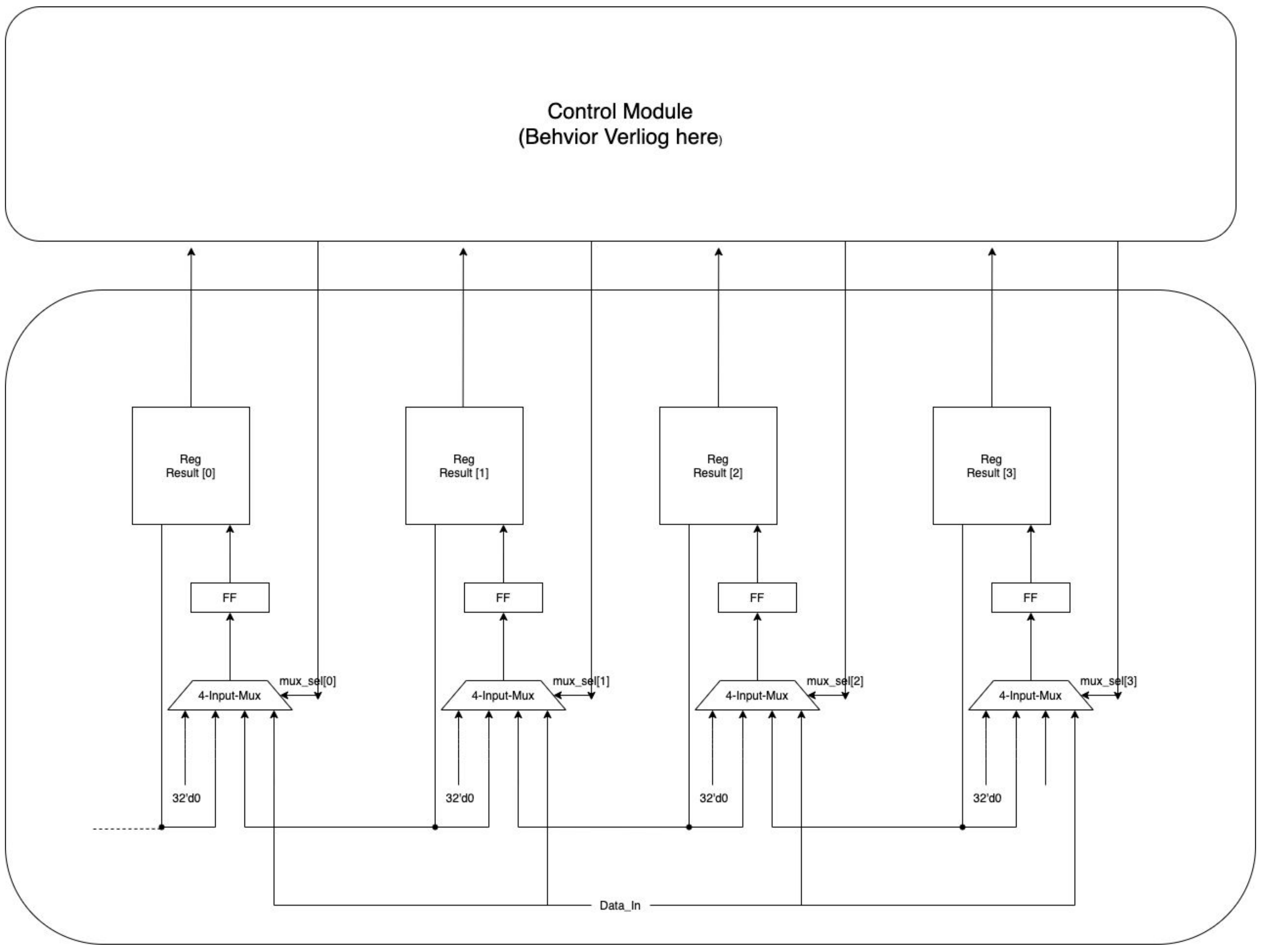

Figure 2: Second Layer: Integrated sort-unit (datapath and control module)

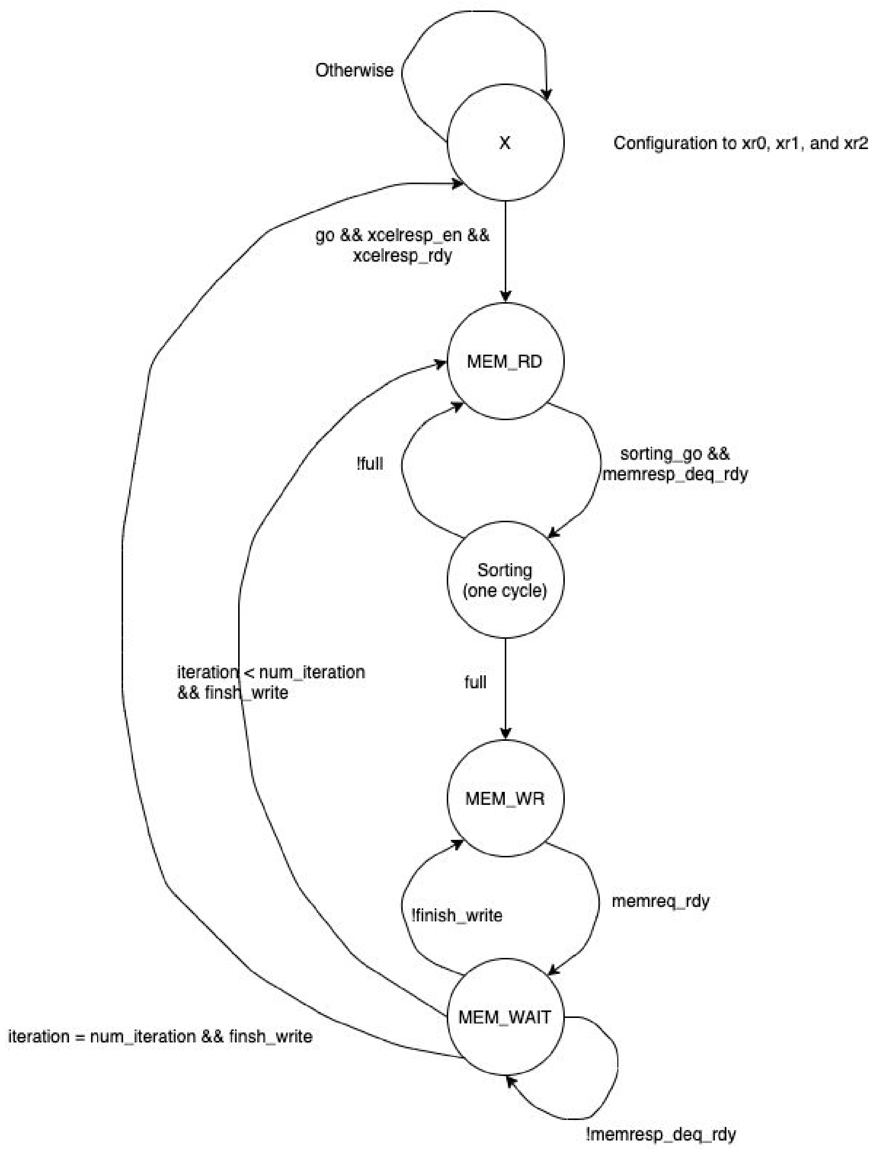
Otherwise
X
Configuration to xr0, xr1, and xr2
go && xcelresp_en && xcelresp_rdy
MEM_RD
!full
sorting_go && memresp_deq_rdy
Sorting (one cycle)
iteration < num_iteration && finsh_write
full
MEM_WR
memreq_rdy
!finish_write
MEM_WAIT
iteration = num_iteration && finsh_write
!memresp_deq_rdy

Figure 3: First Layer: Finite State Machine

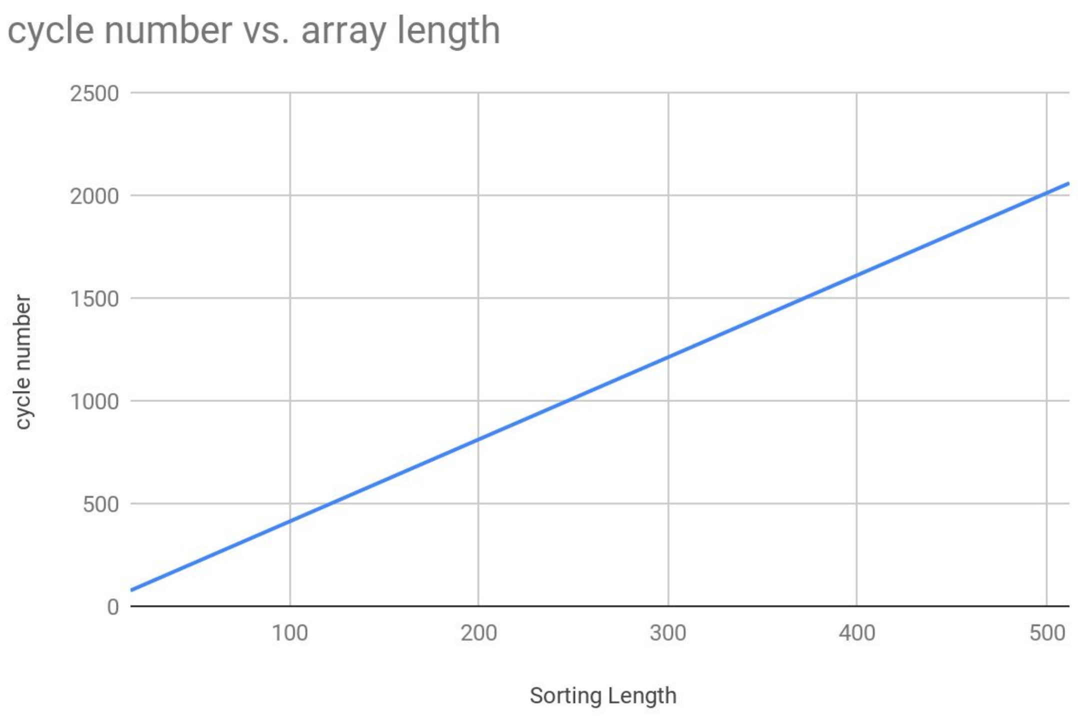

Chart 1: Execution Cycles under different sorting length of accelerator (array length = sorting length)

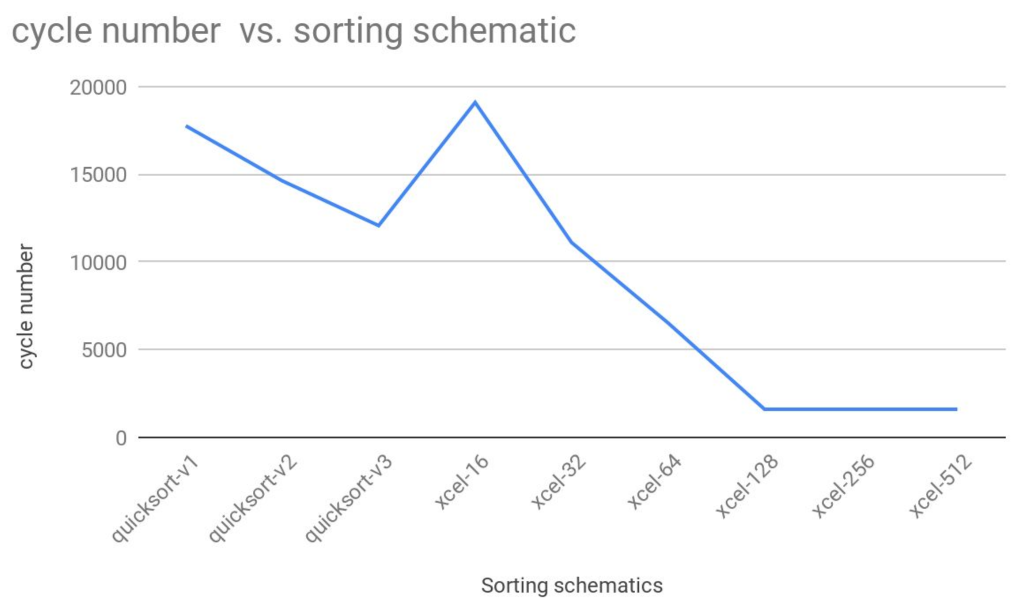

Chart 2 : Execution Cycles under different Processor-Memory-Xcel system (Capacity varied)

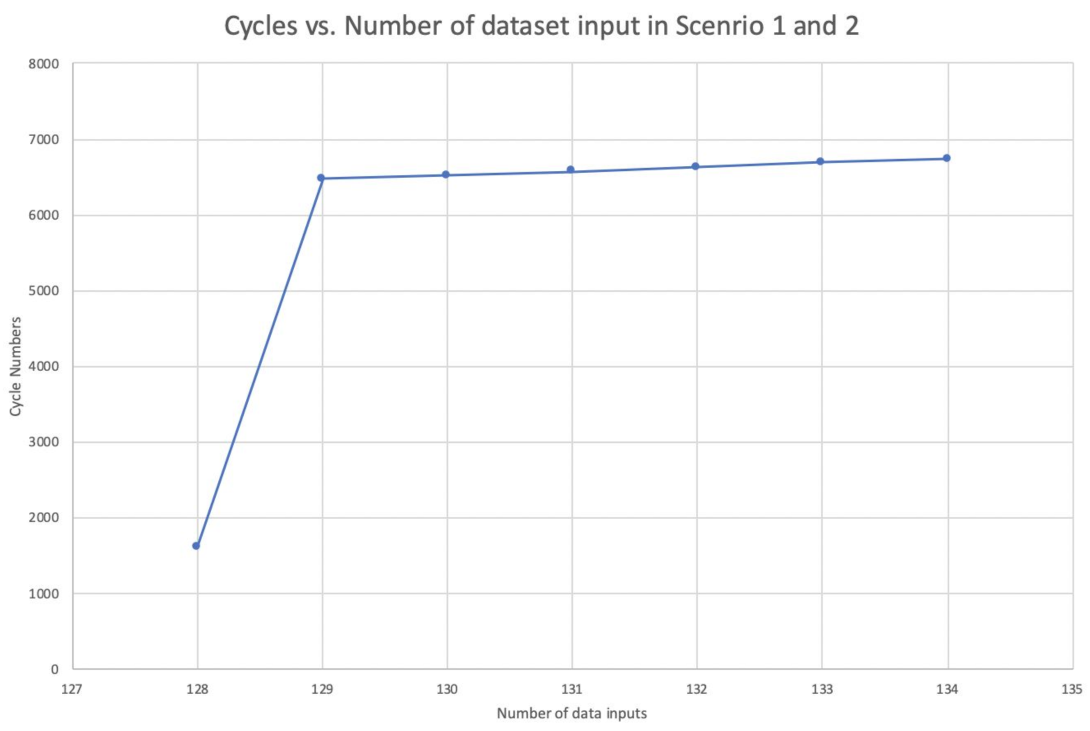

Chart 3: Cycle numbers vs. Number of dataset input in Scenario 1 and Scenario 2

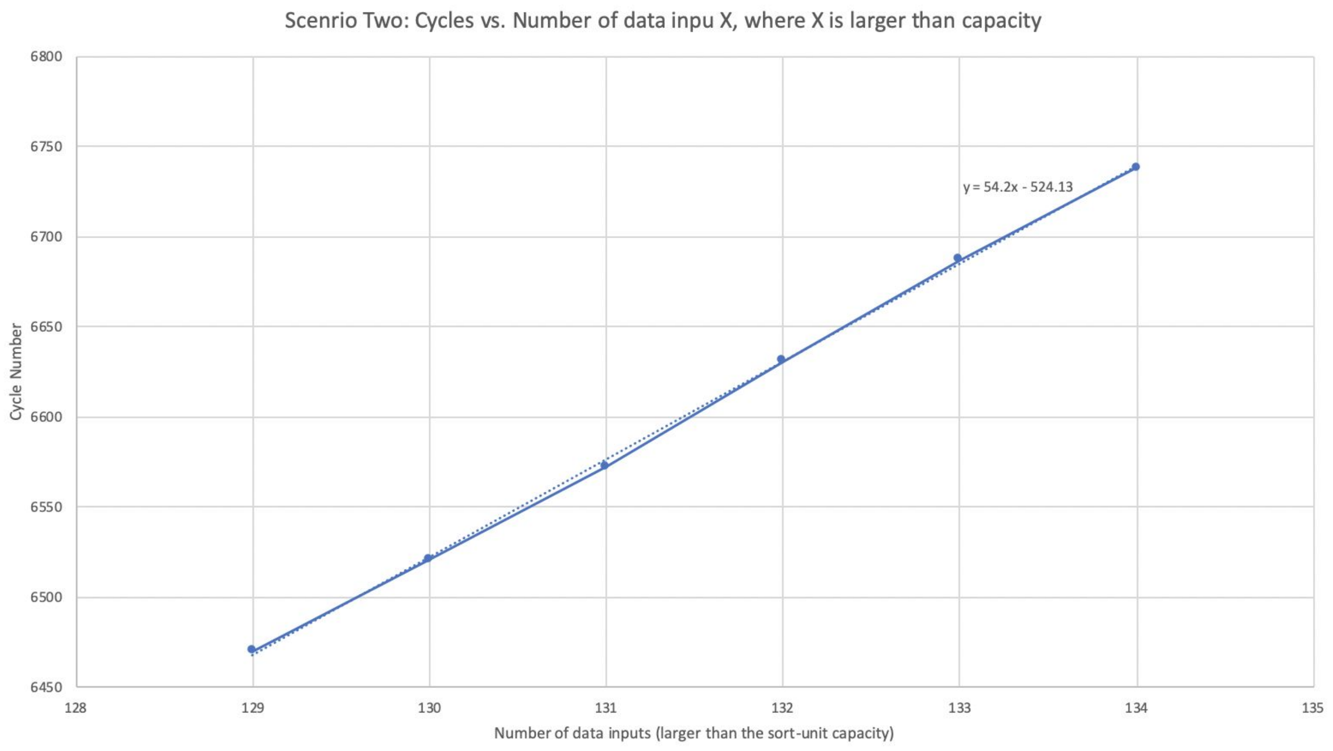

Chart 4: Cycle numbers vs. Number of dataset input in Scenario 2

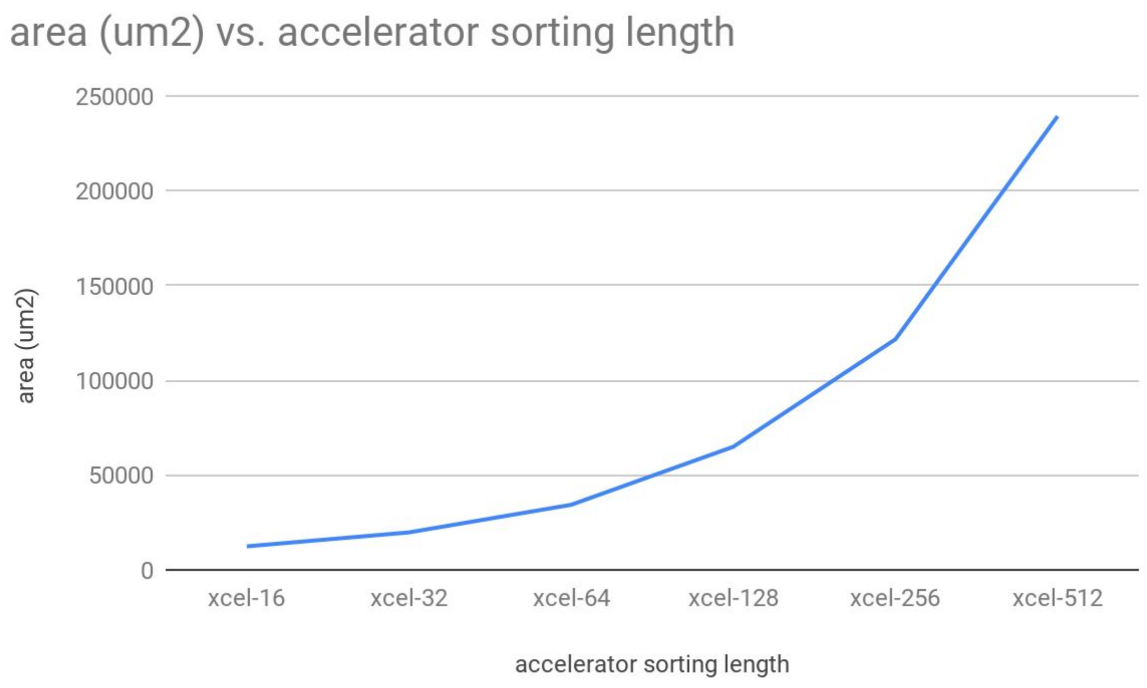

Chart 5 : Area (um2) of the accelerators under different accelerator sorting length

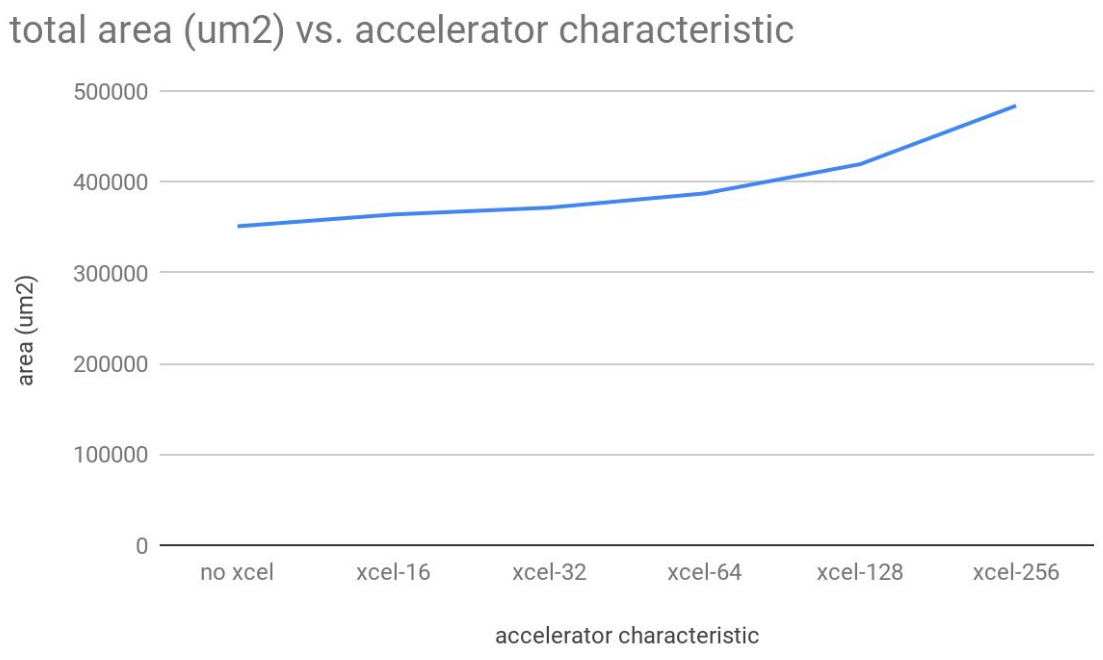

Chart 6 : Area (um2) of the whole system which includes processor, cache & accelerator with different sorting length

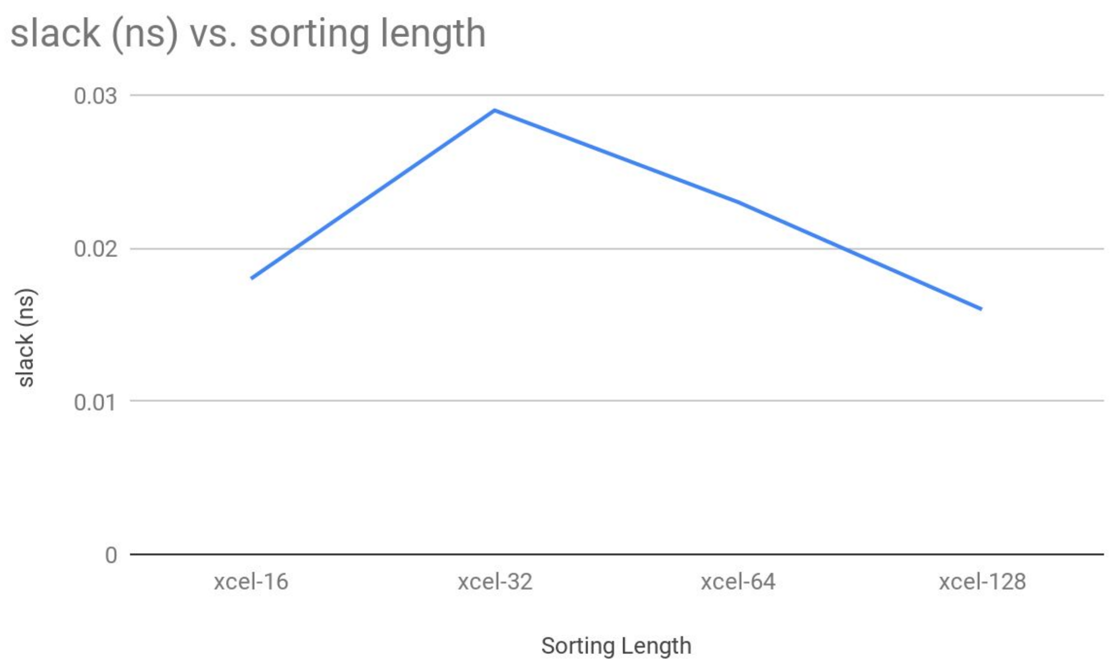

Chart 7 : The clock slack (ns) of accelerator with different sorting length (only accelerator)

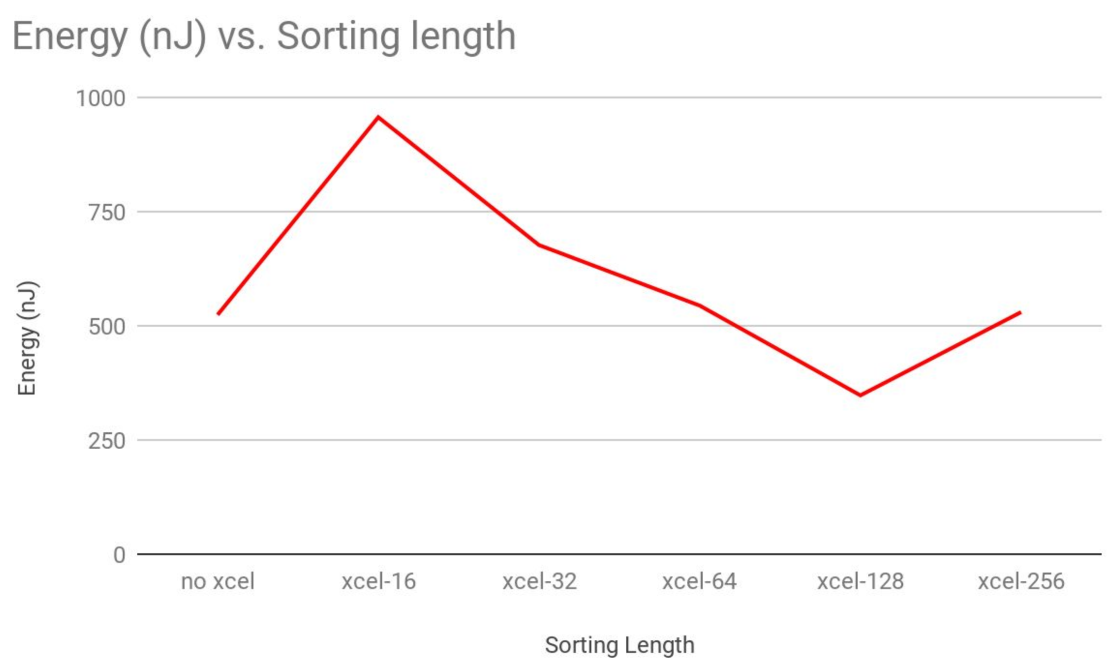

Chart 8 :Energy (nJ) consumption of accelerator with different sorting length for the whole system(processor + cache + accelerator)

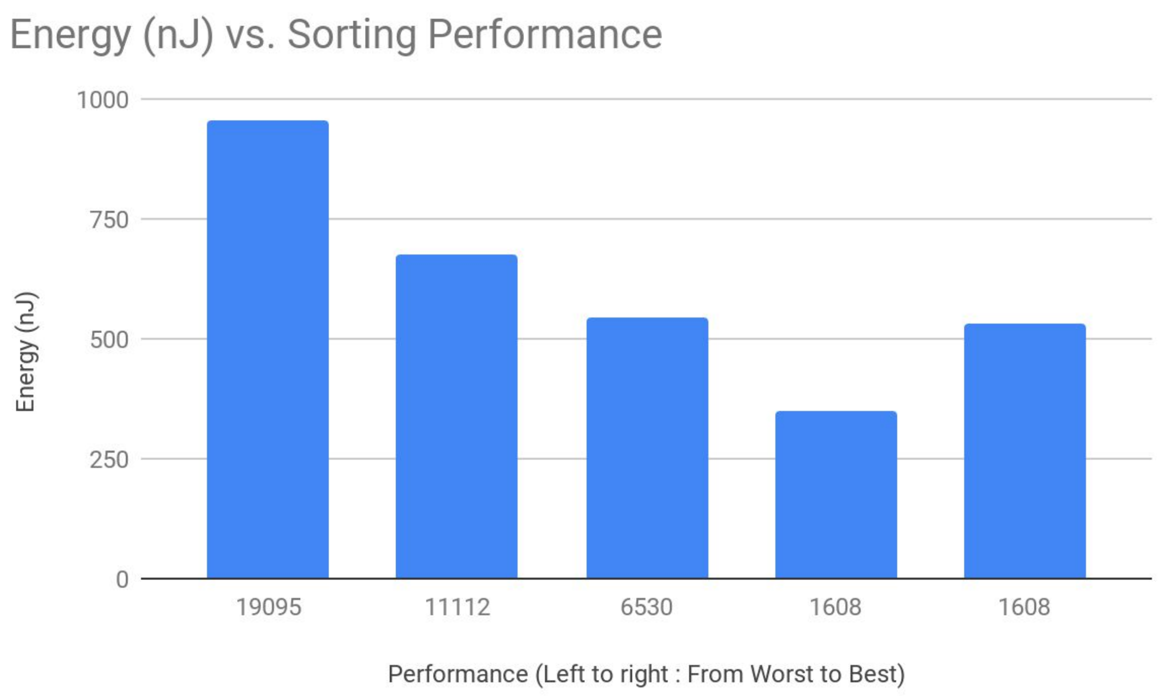

Chart 9 : Energy (nJ) of different performance of the whole system (processor + cache + accelerator)
left to right: sorting length - 16, 32, 64, 128, 256

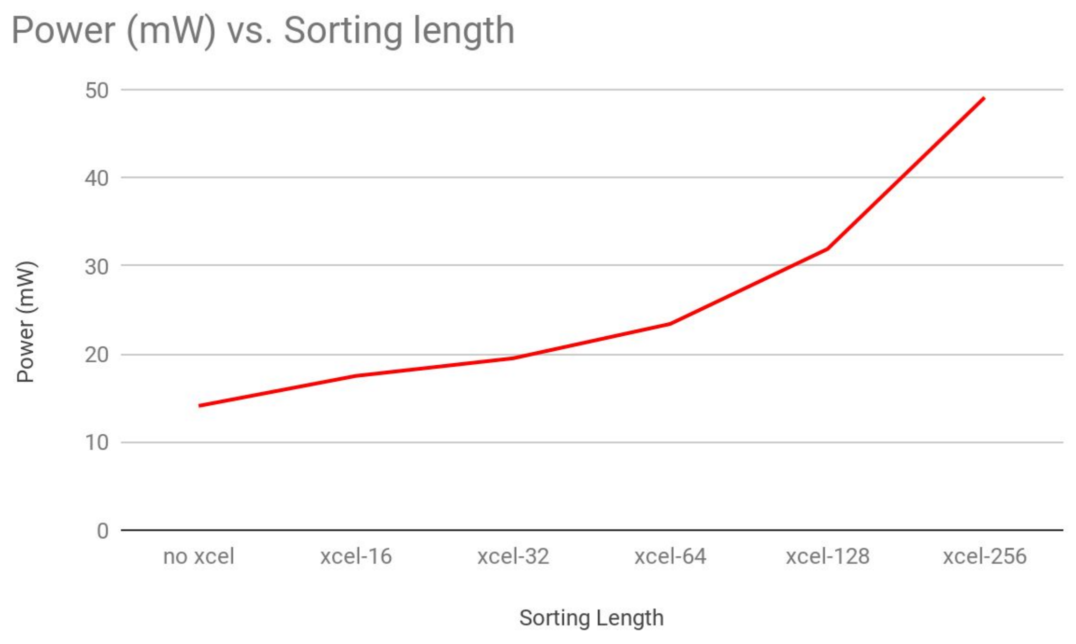

Chart 10 :Power (nJ) of accelerator with different sorting length for the whole system(processor + cache + accelerator)

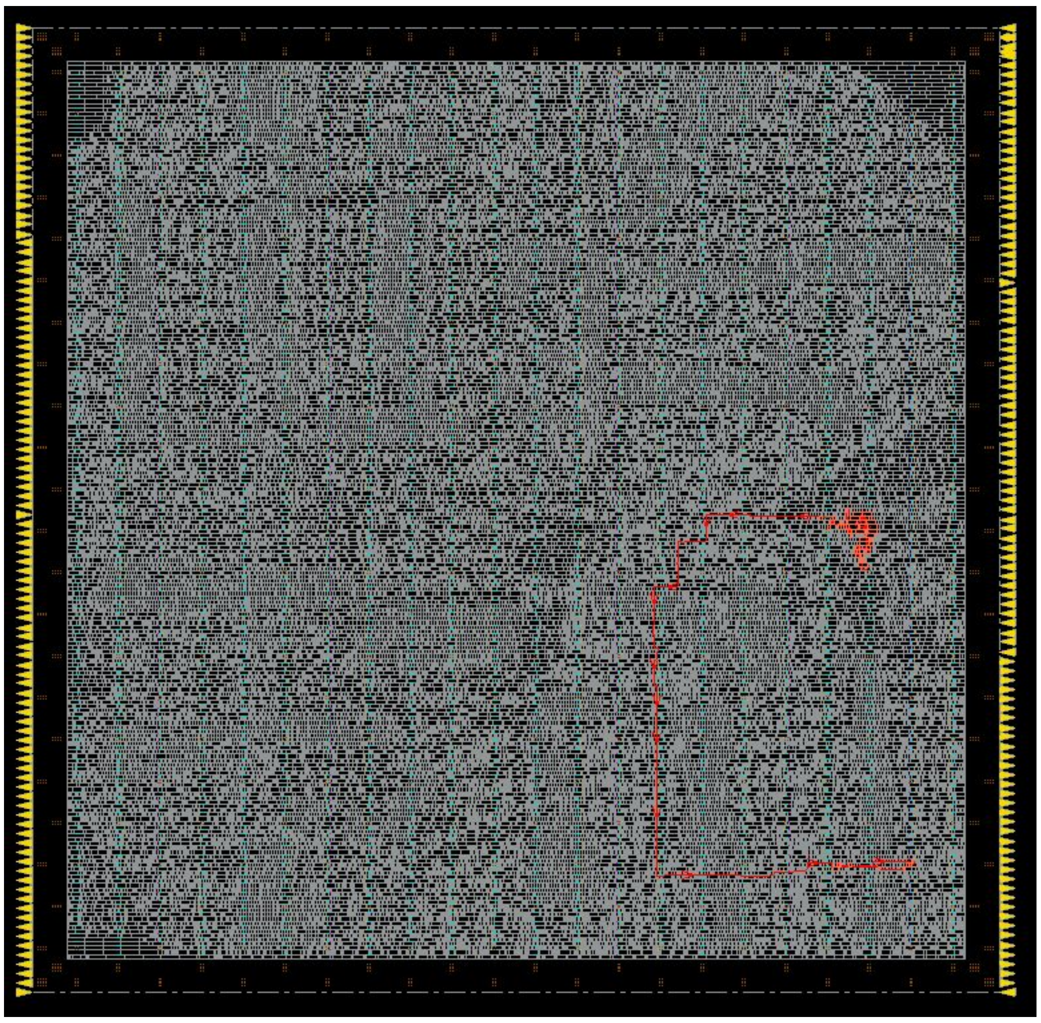

Figure 11: Critical path from the Cadence - Accelerator of 128 sorting length

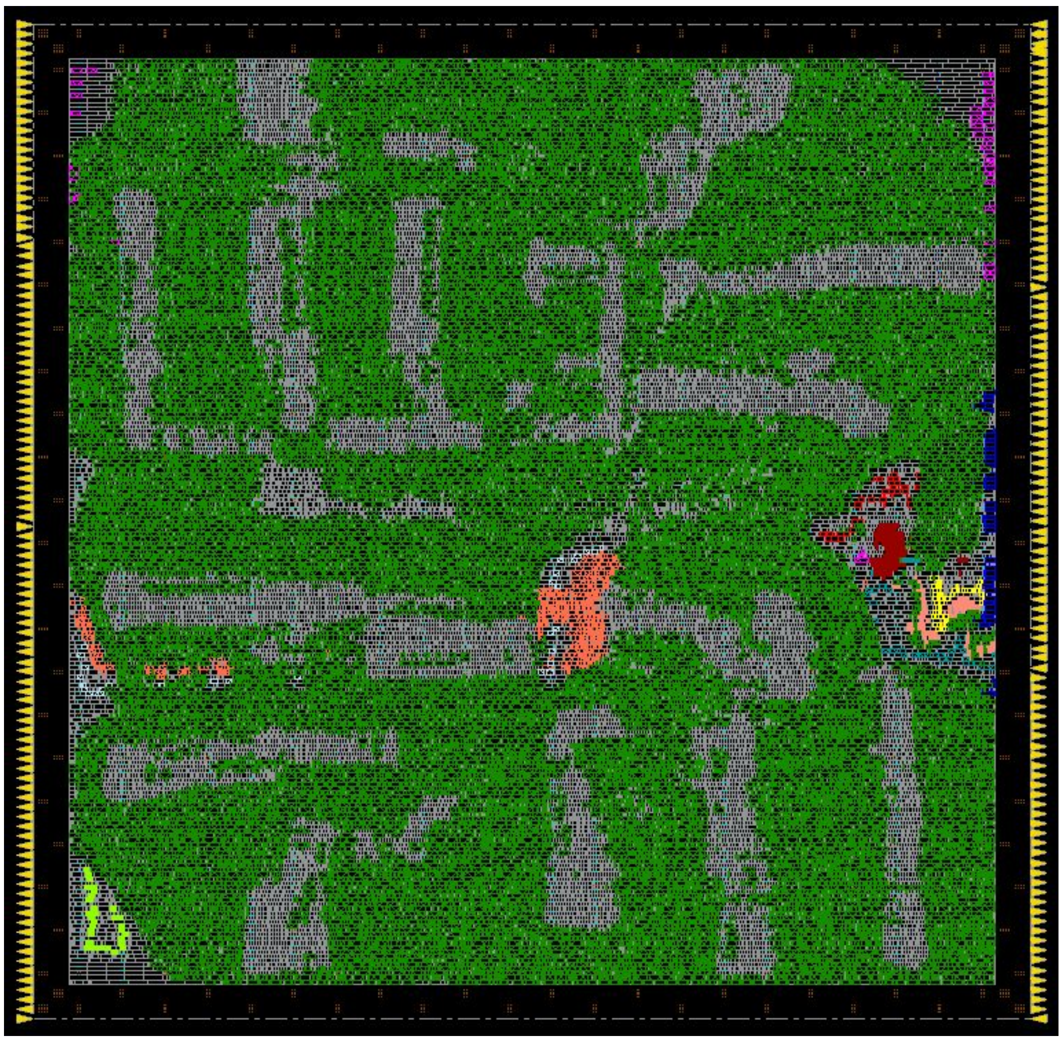

Figure 12: Amoeba plot from the Cadence - Accelerator of 128 sorting length