Jake O'Brien, jro77
Parth Saraswat, ps978
Shivangi Gambhir, sg2439

ECE 5745 Lab 2 Report: Sorting Accelerator

1. Introduction

In the second lab, we worked to implement a medium-grain hardware accelerator for sorting a variable length integer array. The baseline design consists of a pure-software sorting microbenchmark, in which a quicksort algorithm is implemented. This program is run on a pipelined processor with its own instruction and data cache. The alternative design adds to the baseline design with a hardware accelerator and the necessary software to configure and assist the accelerator. The purpose of this lab was to give students familiarity with implementing an accelerator, as well as getting experience with hardware/software codesign. In addition, the open-ended nature of the alternative design pushed students to think critically about what designs may yield the best area, energy and performance results. Finally, lab two gave students some insight into how physically incorporating an accelerator into the design effects area, energy, and timing. In our alternative design, the accelerator loaded 8 integers from a provided source address, sorted the 8 integers, and wrote them back into the source address. It then repeated the process until the entire source array had been transformed into an array of multiple sorted blocks of 8 integers. The software then assisted the accelerator by merging the sorted blocks into the destination array. We were able to successfully complete the alternative design, and push everything through the ASIC toolflow. Based on the results gathered from pushing the designs through the flow, we found that when sorting a 128 integer array, the alternate composition provided 1.04x the performance of the baseline, while consuming 1.11x the amount of energy, and 1.04x the amount of area. We conclude that whether or not an accelerator is a good fit for a design is specific to the situation, however our results push us towards believing that the tradeoff between performance and energy/area from implementing the accelerator to not be worth it. We do however believe that this could change with further optimizations. Overall, the additional area overhead is expected because we are building an accelerator on top of the existing pmx composition. Therefore, to make the accelerator worthwhile, the focus of further work should be on reducing the energy overhead and increasing performance.

2. Alternative Design

The alternative design for this lab was a medium-grain sorting accelerator and a corresponding software microbenchmark that takes advantage of this new accelerator. Our approach was to create an FSM based software accelerator that would load 8 integers from the provided source address, sort these 8 integers, and write them back into the source address, and would keep repeating this process until it had transformed the source array into an array which included multiple sorted blocks. The software microbenchmark would then take over and merge these sorted blocks and write them to the destination array, to achieve a completely sorted destination array. Refer to Figure 4 for a short example on the functioning of this system. Let's take a closer look at some of the key characteristics of the design.

This lab involves hardware/software codesign, so let us break down the deep dive into two parts: (1) the hardware accelerator, and (2) the software microbenchmark. Let's first look at the hardware accelerator.

The hardware accelerator is mostly entirely control logic, which is why we decided not to go for a datapath/control unit split. A key component of the accelerator's datapath is our flat sorter unit. The flat sorter has a very straightforward interface: it has 8 inputs for 8 unsorted integers, and 8 outputs, where the 8 integers are available in a sorted order. The sorter unit is made up of smaller sort units which sort blocks of 4, and these smaller units are built of simple min-max units. The min-max unit simply compares two integers, and places the smaller integer on its 'min' output, and the larger on its 'max' output. The datapath diagrams for the min-max unit, the 4-element sorter and the 8-element sorter can be seen in Figure 1, 2, and 3 respectively. A working example of the 8-element sorter is shown in Figure 5. Note that there is a set of pipeline registers between stages 1 and 2 of the sorter, this was in order to cut our critical path in order to meet timing. We did not make any changes to the accelerator protocol. That is, the processor writes the base address of the source array to xr1, size is written into xr2, and any write to xr0 is used to tell the accelerator to start. The accelerator writes to xr0 in order to indicate that it is done once it has sorted the source array into blocks. The accelerator uses the FSM shown in Figure 6. The FSM consists of the following states: XCFG, M_READ, M_WRITE, SORT, and WAIT. XCFG is the accelerator configuration stage - it handles the protocol by reading the XcelReq message, and writing the appropriate value to the appropriate register. Once the XCFG stage realises that the XcelReq message is writing to xr0, it transitions to the M_READ stage. In the M_READ stage, the accelerator begins to read the source array from memory, and places the memory

responses onto the inputs of the sorter unit. The outputs of the sorter unit are these 8 integers in a sorted order. Once the accelerator has sent 8 requests to memory for the 8 integers, it moves into the SORT stage. In the SORT stage, the accelerator simply waits for one cycle such that the outputs of the sorter are available. Then, it moves to the M_WRITE state. In the M_WRITE state, the accelerator writes the sorted integers back into the source array. Once these 8 writes are completed, the accelerator moves into the WAIT state, where it simply has to decide whether it is done sorting the entire array into blocks or not. If the entire source array is sorted into blocks, the accelerator goes back to the XCFG state where it waits for the next XcelReq message. In case the entire source array hasn't been sorted into blocks, the accelerator goes back to the M_READ stage and reads in the next 8 integers, and the process is repeated. Next, let's take a look at the software microbenchmark.

The software microbenchmark is a C program that takes advantage of the accelerator in order to sort a source array and place the sorted array into the destination array. The microbenchmark has to handle a few corner cases based on the length of the source array in order to make sure the destination array is fully sorted: (1) the source array has < 8 elements. In this case, the accelerator is of no use and the software simply performs selection sort on the array and we are done. Selection sort was chosen for its ease of implementation. While it is an O(n^2) algorithm, the longest length array of integers it will ever have to sort is 7, so there is no point using another algorithm with higher implementation complexity. (2) The length of the array is a multiple of 8 and a) has an even number of blocks of 8, or b) has an odd number of blocks of 8. In the case of 2a), once the accelerator has sorted the source array into blocks, the software simply merges the sorted blocks into the destination array, and we are done. In the case of 2b), the sorted blocks with exception of the last block are merged. Then, the newly sorted block is merged with the last block. (3) The length of the array is not a multiple of 8 and a) has an even number of blocks of 8, or b) has an odd number of blocks of 8. In the case of 3a), the accelerator sorts the source array into blocks of 8 until fewer than 8 numbers remain in the source array. The software then performs a selection sort on this small block, and finally merges all blocks together into the destination array. In the case of 3b), the same process is followed from 3a), except that the sorted blocks with exception of the last sorted block are merged, then the last sorted block is merged into the newly sorted array, and finally the now sorted block of less than 8 integers is merged into the destination array. For a better understanding of these corner cases, reference Figures 12 - 16. While this software implementation does result in a fully sorted array, there are some inefficiencies in terms of when merging happens that could be addressed in the future. For example, in 2b) by waiting to merge the last block until all the other blocks have merged there is a potential inefficiency by merging a very large block with a small block, especially if the small block contains most of the high integers.

Our design of the accelerator and the microbenchmark follow a few key design principles. The accelerator and the software demonstrate modularity. A key component of the accelerator is the sorter unit, which is made up of smaller 4-unit sorters, which are made up of smaller min-max units. The software includes multiple functions that perform their individual functions, and there is no stray code which does not belong to a particular function. The design also exhibits extensibility: if you wished to extend the accelerator to now handle 16 element blocks, you simply create a new 16 element sorter which can very easily be composed of the 8-element sorter we have created. The accelerator also exhibits regularity by reusing the min-max unit multiple times in order to build the 4-element sorter, and then the 8-element sorter. Finally, the accelerator and the functions in the C code also abstract their internal working from the outside world. There is no need to understand the implementation of the functions or the sorter unit in order to make use of them.

Finally, the reason why we chose this design approach was that the FSM design was very easy to implement in an iterative fashion. We started off by only implementing the XCFG stage and the M_READ stage to ensure that we were able to read messages from memory. We then implemented the M_WRITE stage, and finally incorporated our sorter unit into the datapath. The entire process allowed us to catch bugs at early stages and ensure a systematic approach throughout the implementation of the accelerator.

3. Evaluation

Let us begin our evaluation with some energy and power analysis.Refer to Plot 1a in the appendix: Plot 1a is our first *design-space exploration* plot, and plots the *performance* of the baseline and alternative pmx compositions against the *energy* consumed. An ideal design would consume low energy and offer high performance. That is, ideally, we would want to be in the bottom right of the plot, and avoid being anywhere in the top left. Consider the baseline pmx (no sorting accelerator, software running quicksort on an array of length 128) which is towards the top left of the plot. This is an undesirable location to be in as the design consumes high energy while also providing low performance. Next, consider the alternative design (pmx with sorting accelerator and software microbenchmark running merge sort). This design provides relatively higher performance, but at the cost of some extra energy overhead. Our intuition told us that the reason for the alternative composition consuming more energy had to do with the extra registers required in the sorter unit. We confirmed these beliefs by looking at the power reports. Refer to the power report in Figure 18 to see the breakdown of the power taken

up by the alternative pmx composition. Within the composition, most of the power is taken up by the processor and the cache.The accelerator accounts for about 15.5% of the total power consumed by the composition. Most of the power in the accelerator is consumed by the sorter (12.4% of the 15.5%). Upon a closer look, we see that the functional units account for ~1.5% of this 12.4% - which shows that the registers in the sorter account for 10.9% of the power consumed by the PMX composition.

Quantitatively, the alternate composition provides **1.04x the performance** of the baseline design, while consuming **1.11x the amount of energy**. The question naturally arises whether the extra performance is worth the extra energy spent -- in our opinion, this does come down to the priorities of the user, but as a general suggestion we would say that as the amount of energy required is rising faster than the amount of performance provided, this is probably not a tradeoff we would make. This data also helps us in realising that if we were to continue working on this accelerator, we would focus on reducing the energy consumed or increasing performance by continuously streaming data through a DMA module. An interesting thing to notice is how the multipliers in the processor consume a significant amount of power as compared to the register files which consume less power even though we practically don't have any multiplication operations and use the register files frequently. One possible explanation which could be given is that the tool applies clock gating to the register files and not on the multipliers which can be seen in the power report of the processor in Figure 20 and Figure 21 for both baseline as well as alternative. This gives us another optimization to look forward to in the future as to how clock gating or data grating can be added to non-functional units like multipliers in this case to reduce the dynamic power consumption. This approach can help us optimize the power consumption of the processor and in effect reduce the energy utilized by the entire system i.e. the processor-memory-accelerator.

Let us move to some area analysis. Plot 1b is our second design-space exploration plot, and plots the performance of each design against the area needed. An ideal design would offer high performance while taking up a low area. That is, it would want to be in the bottom right of the plot. Anything in the top left would be undesirable. Once again, consider the baseline pmx (top left). This is an undesirable location to be in as the design occupied high area while also providing low performance. Next, consider the alternative design - this design provides higher performance **(1.04x)**, while taking slightly more area **(1.04x).** The extra overhead in terms of area can be tracked down to the sorter. Refer to Figure 19 for the area breakdown which shows how the sorter occupies ~70% area of the accelerator. On analysing further we can see that the actual sorting units occupy more area compared to the pipelining registers which we add to optimize our design. We believe this trade-off is slightly more nuanced, as it is not possible to *reduce* the area as compared to the area taken by the baseline in this case, as we are building an accelerator on top of the existing pmx composition. Hence, the extra area taken is to be expected and unavoidable. Therefore, in our opinion, the extra area overhead is well justified by the extra performance that the accelerator helps us achieve.

We also generated amoeba plots for the baseline and the alternative designs to get a visual representation of all the data we collected. Refer to Figure 7 in the appendix: it clearly shows how the memory and processor take up maximum area in the baseline while placing the null accelerator right in the center. It is also worth noticing how the layout of the caches (data + instruction) is a lot more regular compared to everything else - primarily due to memory modules being so dense. Figure 10 shows a breakdown of the area consumed by the processor, memory and accelerator in the alternate design. We once again see how the tool tries to evenly distribute the memory and places the accelerator at the center where the null accelerator was present for the baseline design. Figure 11 in the appendix shows the amoeba plot of the accelerator in isolation - an interesting observation is that the pipeline registers actually do not consume a lot of area as compared to the sorter unit.

Finally, let us focus our discussion on cycle time. Figure 8 highlights the critical path for the baseline which runs from the instruction cache to the processor, and Figure 9 shows a comparison of the critical paths between our initial design with no pipelining registers (which did not meet timing) and the final design (which included pipeline registers to meet timing). An interesting problem which we ran into while pushing our initial Sorting Accelerator through the flow was that we were able to meet a time constraint of 0.65ns or a frequency of ~1.5GHz which seemed pretty fast. Unfortunately, things stopped making sense when we pushed the pmx for this design through the flow and noticed a large negative slack with a clock constraint of 2.5ns. On digging deeper through the Synopsis DC log and timing reports, we found that there were inferred latches in our design which messed up the timing reports generated. After we removed the inferred latches by systematically modelling registers in the Accelerator RTL, we pushed the design through the flow and failed to meet timing for the accelerator in isolation and got a critical path highlighted on the left of Figure 9. In order to fix this, we simply added pipeline registers between multiple functional units in the sorter. This helped us optimize our critical path and we could achieve a much shorter critical path, which is highlighted on the right of Figure 9.

Appendix:

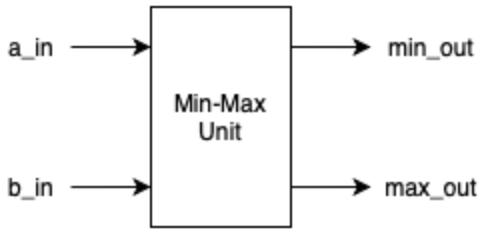


Figure 1: Min-max unit

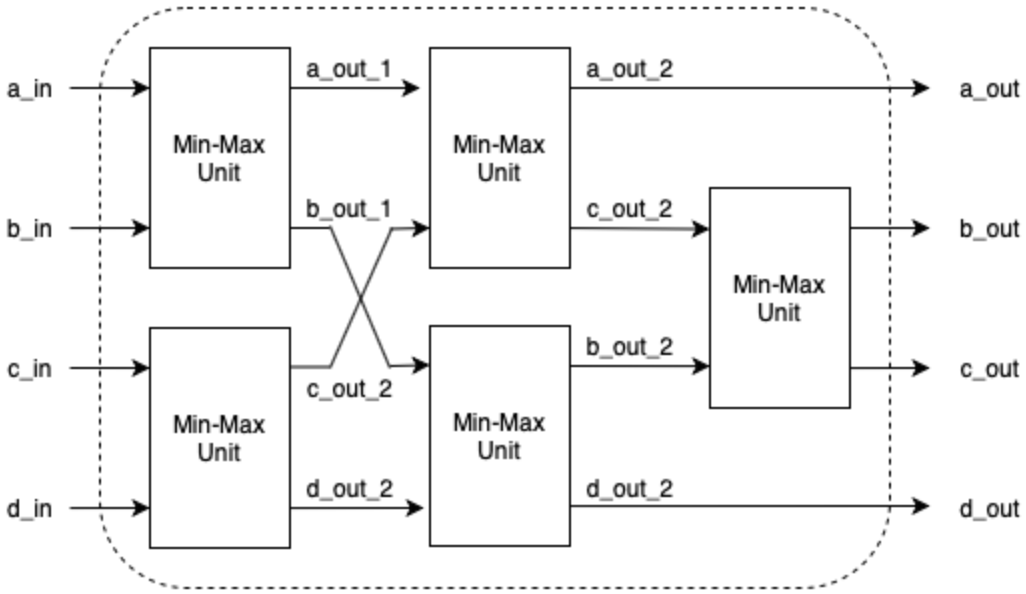


Figure 2: Functional Unit (sorts four numbers)

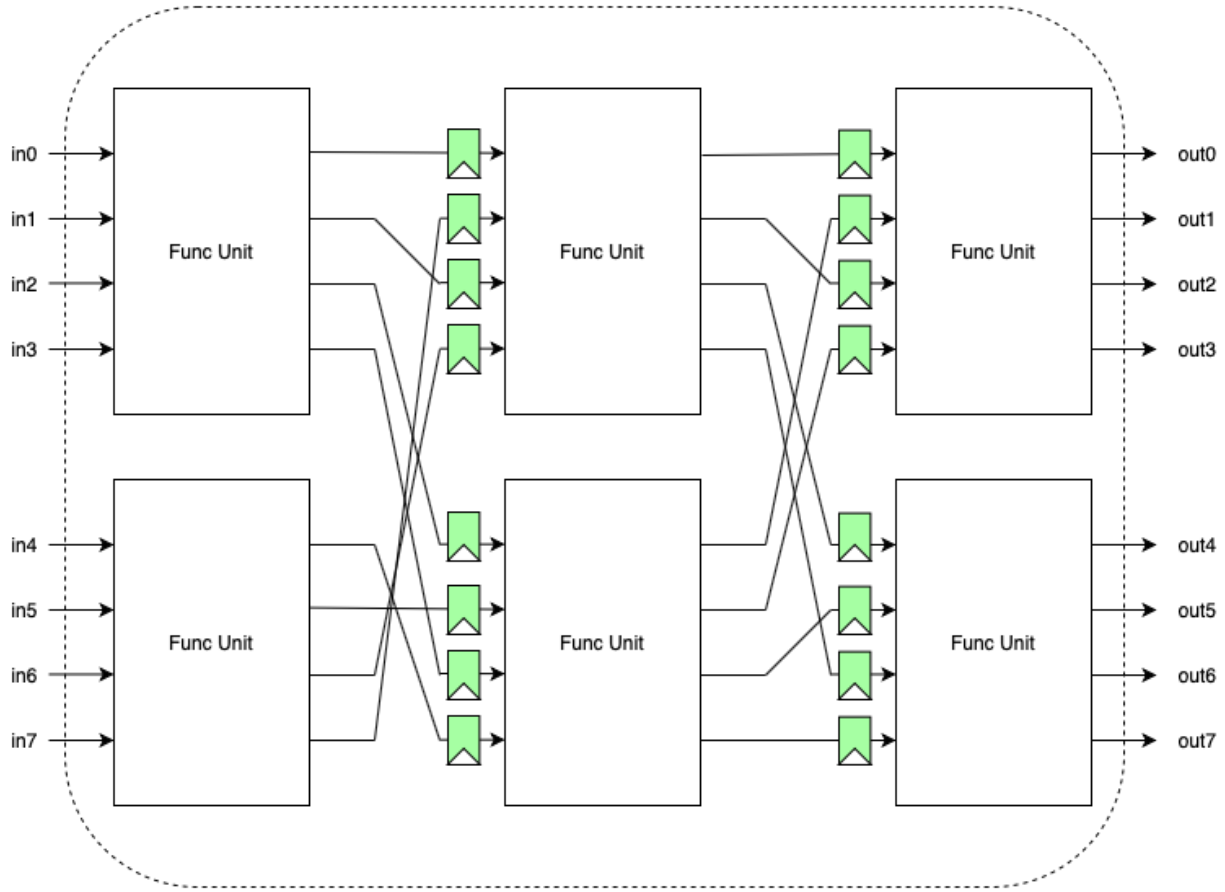


Figure 3: Functional Unit - 8 (sorts 8 numbers): *note the pipeline registers - added in order to shorten the critical path and meet timing for the accelerator*

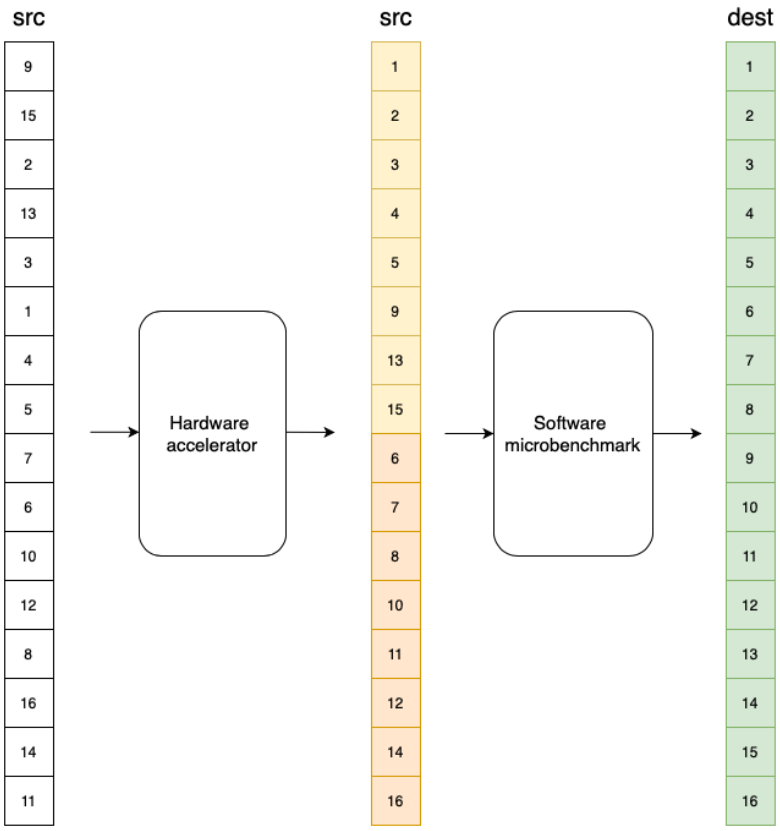


Figure 4: Example of system working

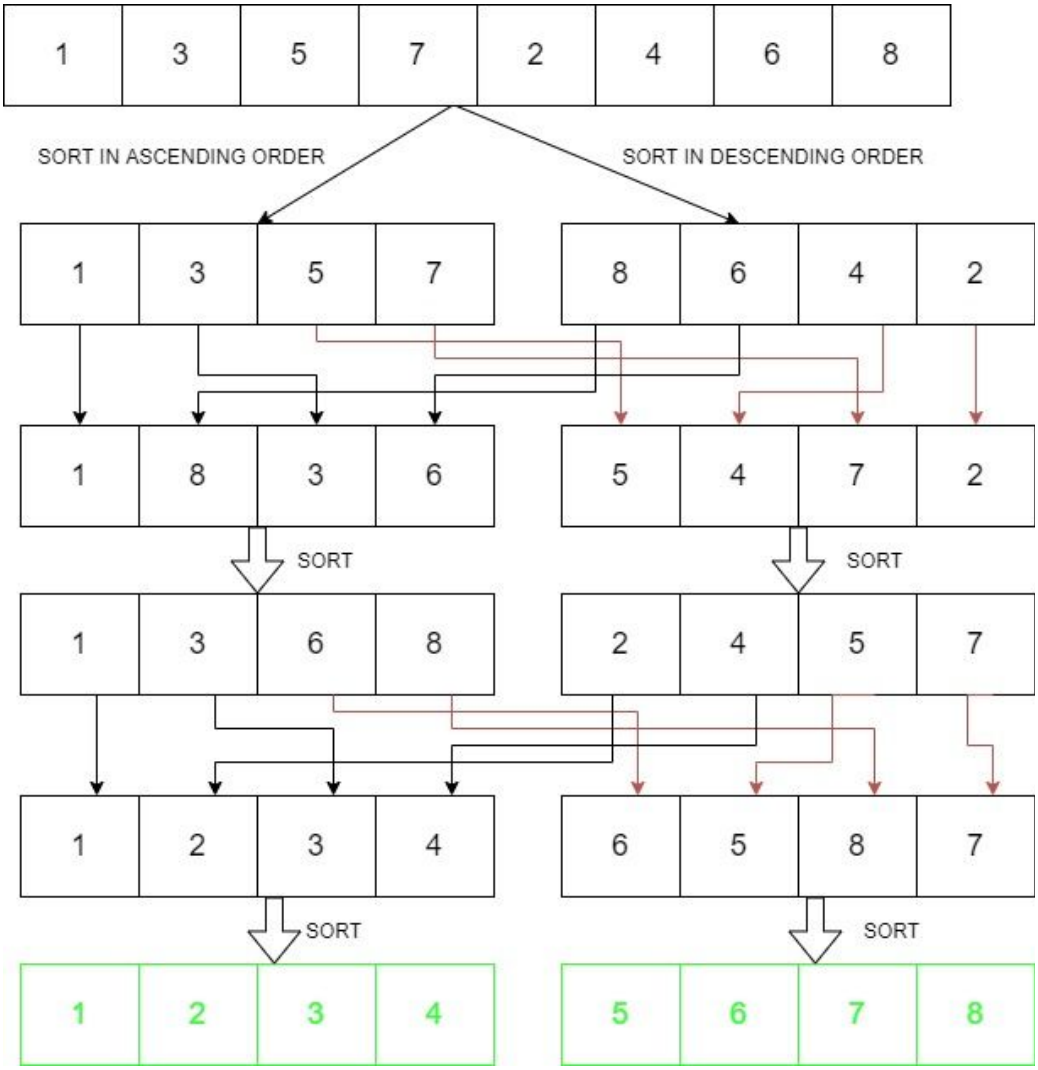


Figure 5: Working Example of 8-element sorter

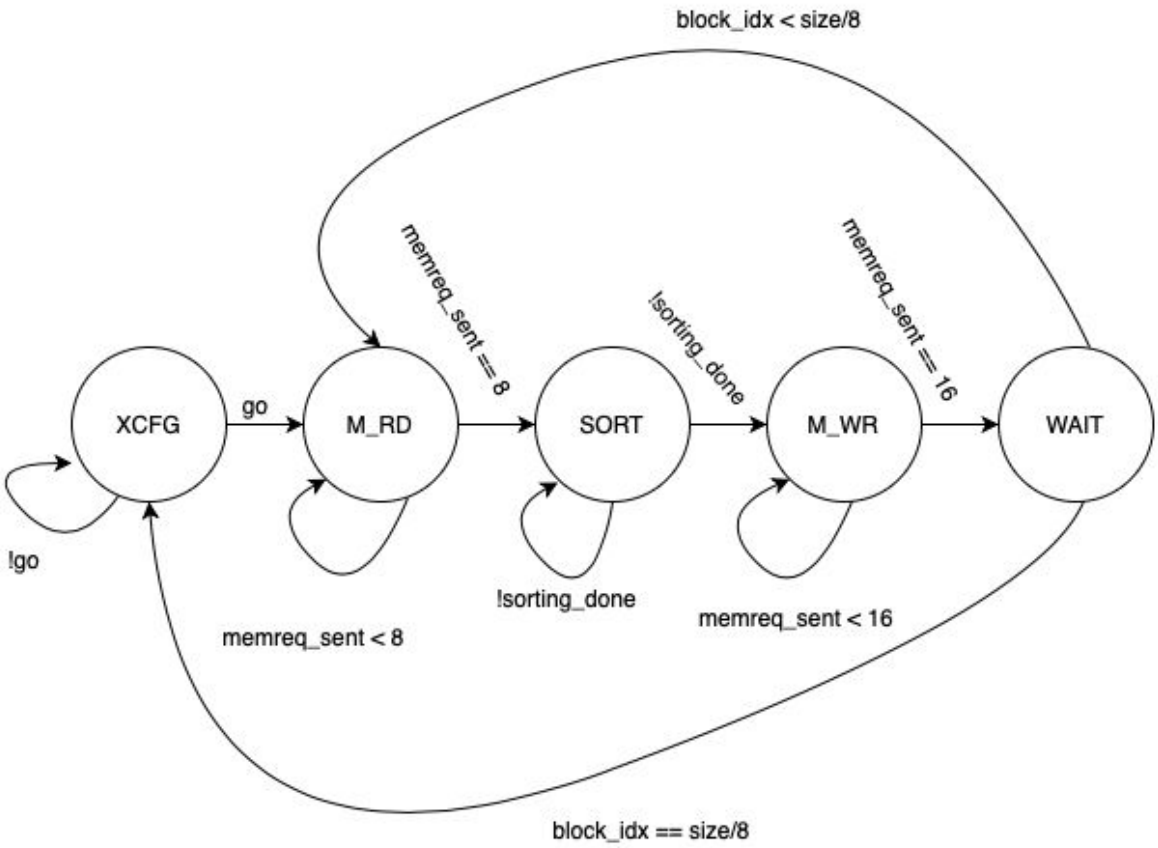


Figure 6: FSM used by the accelerator

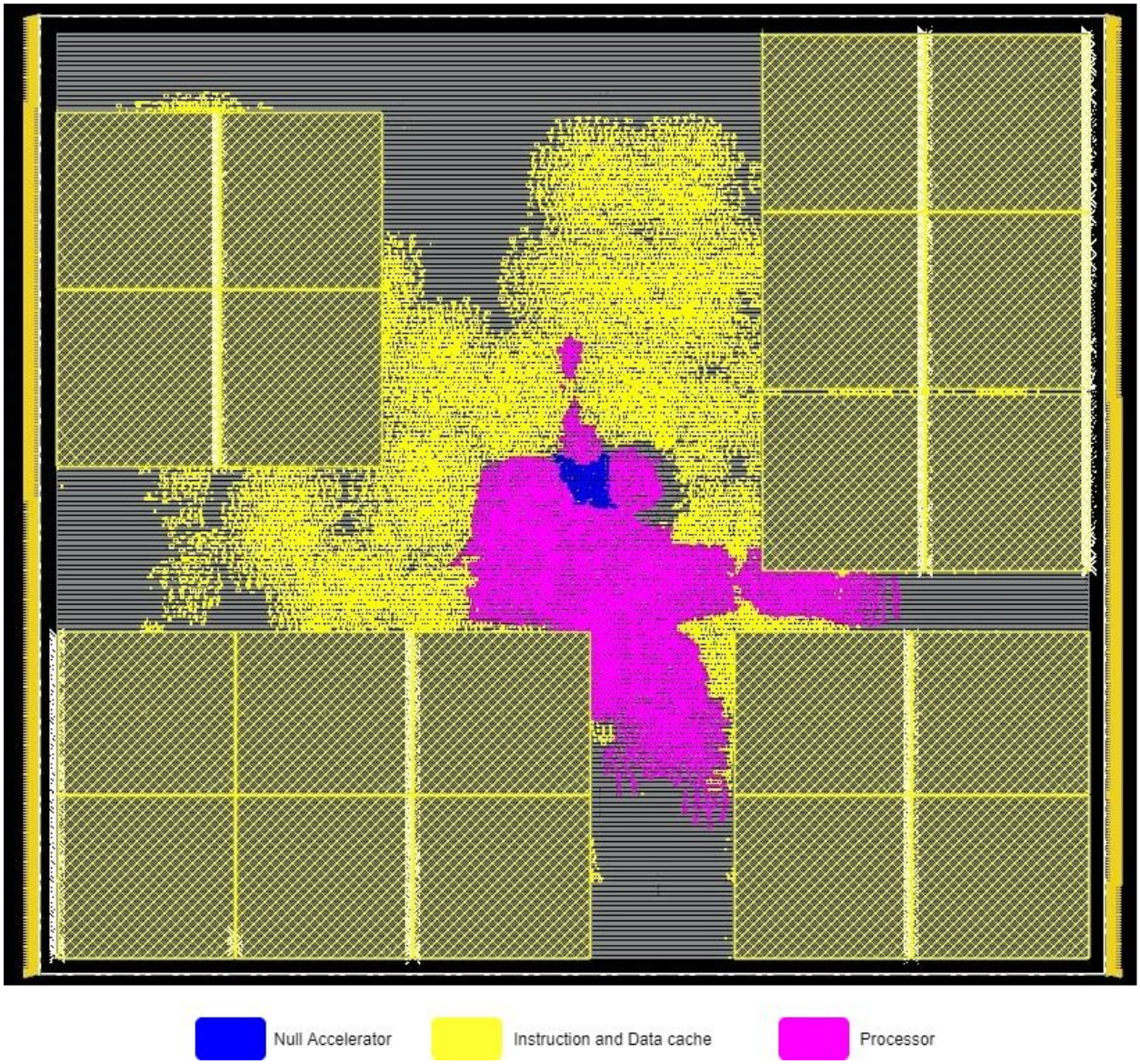


Figure 7: Processor, Memory and Null Accelerator highlighted for the baseline design

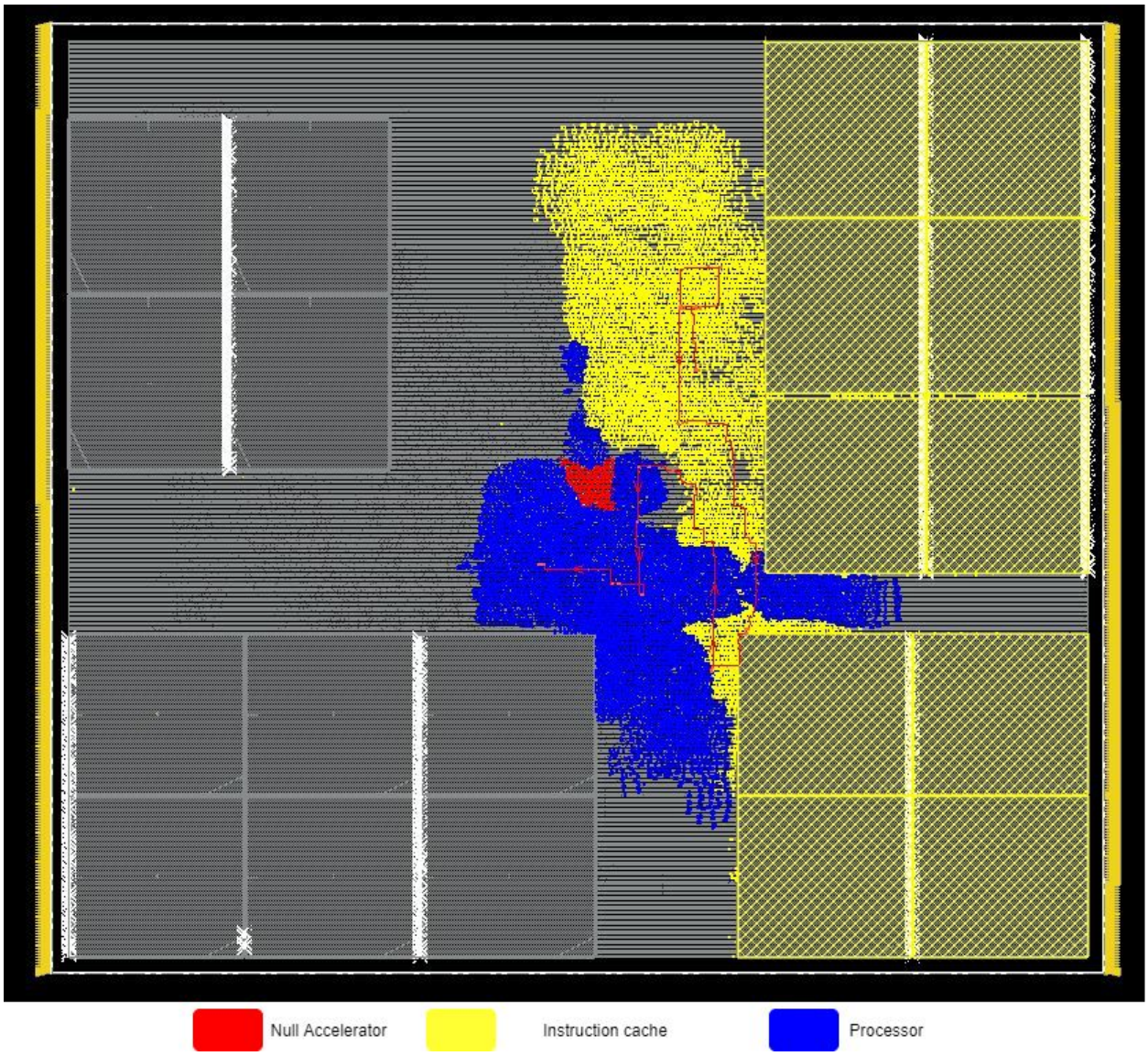


Figure 8: Critical Path for the baseline design

(Note how the critical path starts lies along the instruction cache and processor only and doesn't cross the null accelerator)

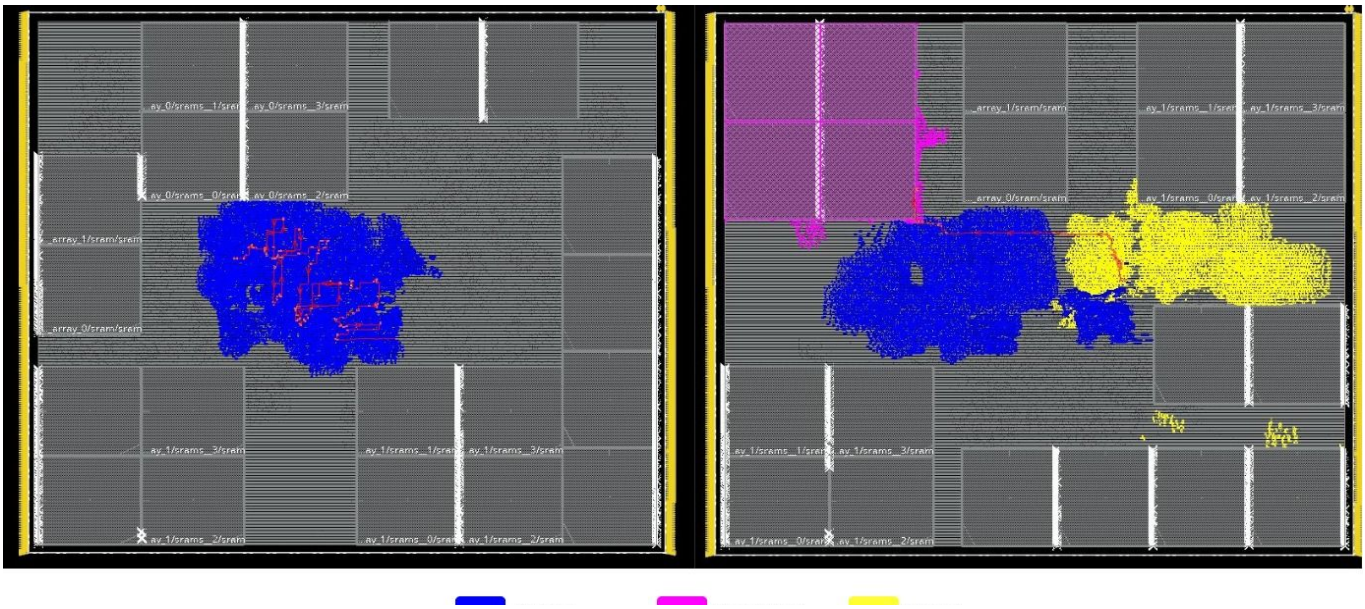


Figure 9:
Left : Critical Path for pmx with non-pipelined sorter
Right: Critical path for pmx with pipelined sorter
(Note how the critical path changes and shortens when we add pipeline registers to the functional sorting unit)

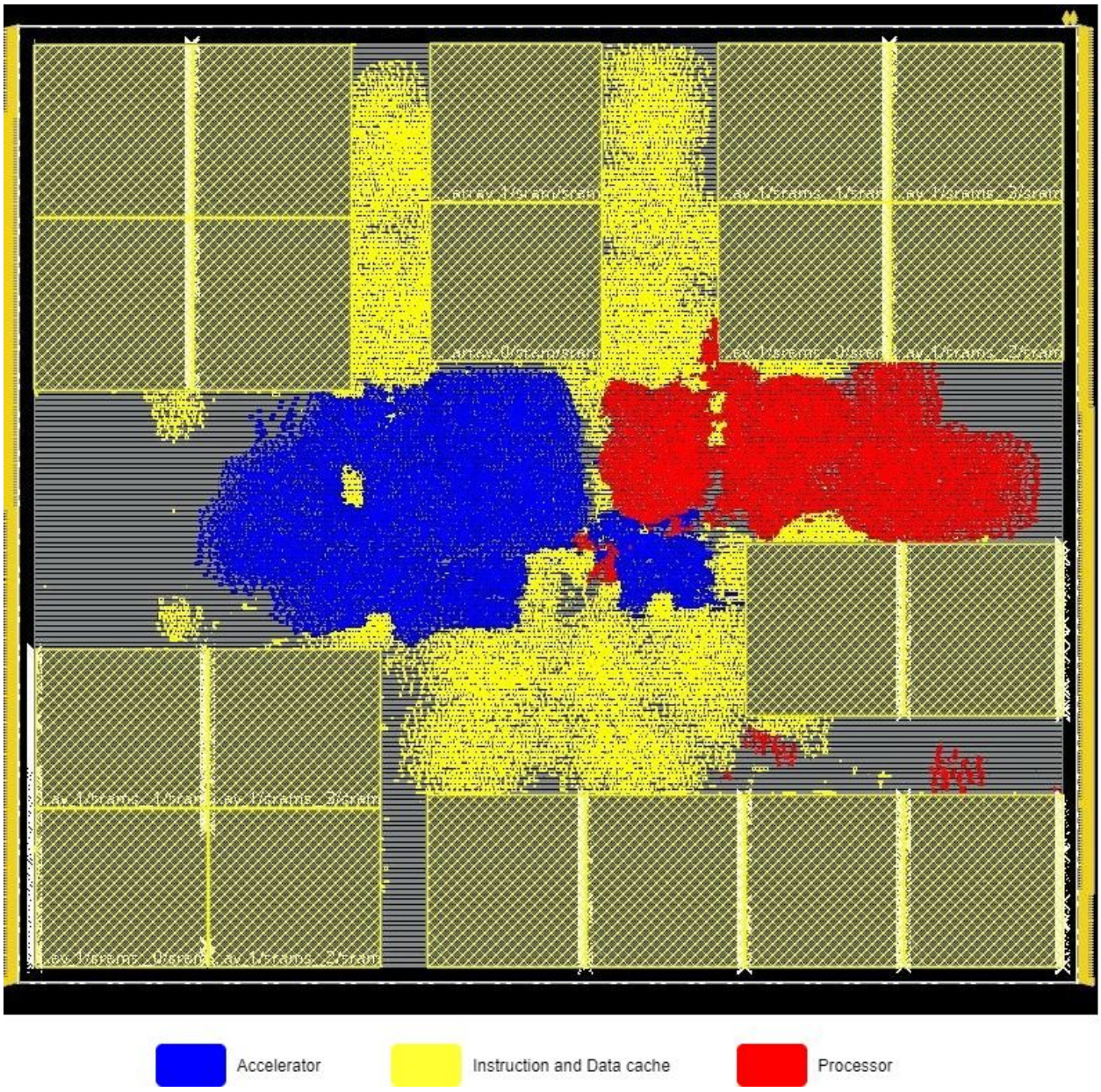


Figure 10: Processor, Memory and Pipelined Accelerator highlighted for the alternative design

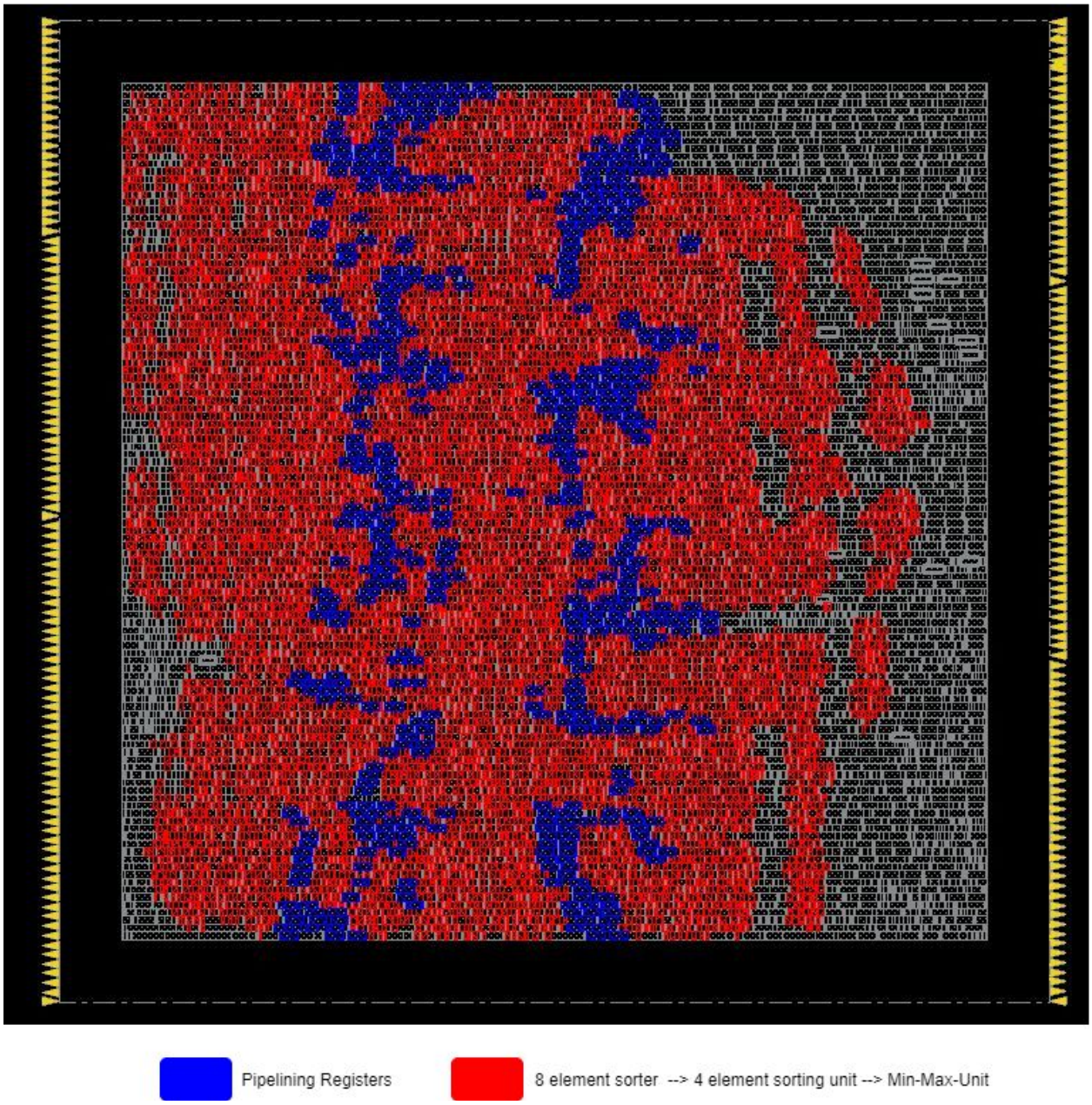


Figure 11: Pipelined Accelerator

(Note how maximum area of accelerator is occupied by the sorting unit comprised of min-max units and the pipelining registers occupy very less area giving us a good area/performance trade-off)

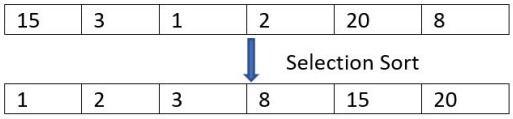


Figure 12: Selection sort used when the input array is less than 8 integers

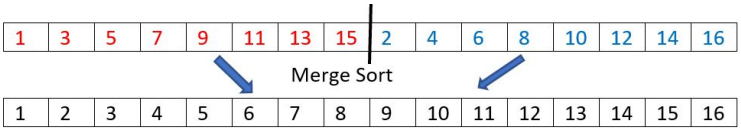


Figure 13: Number of integers is a multiple of 8, and there is an even number of blocks.

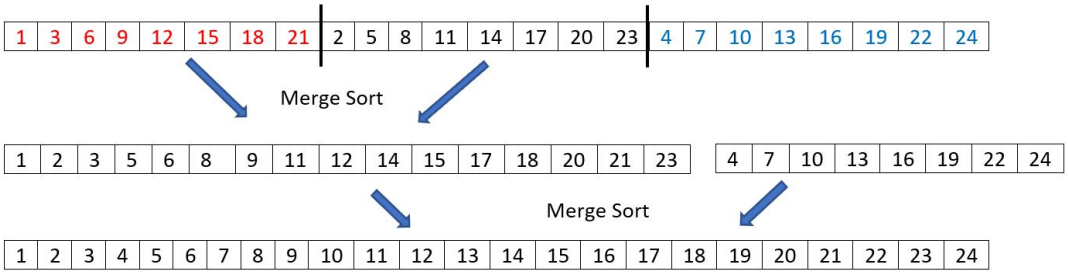


Figure 14: Number of integers is a multiple of 8, and there is an odd number of blocks.

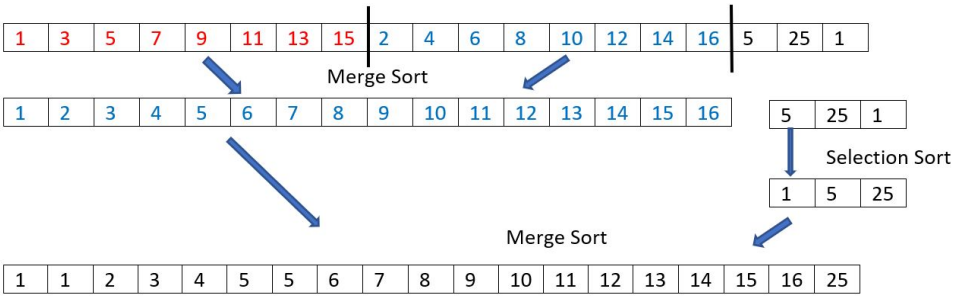


Figure 15: Number of integers is not a multiple of 8, and there is an even number of blocks

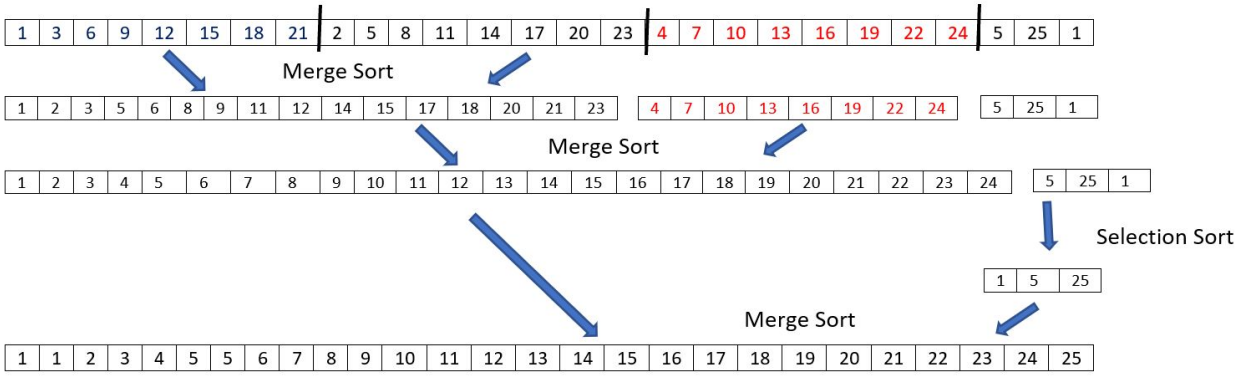
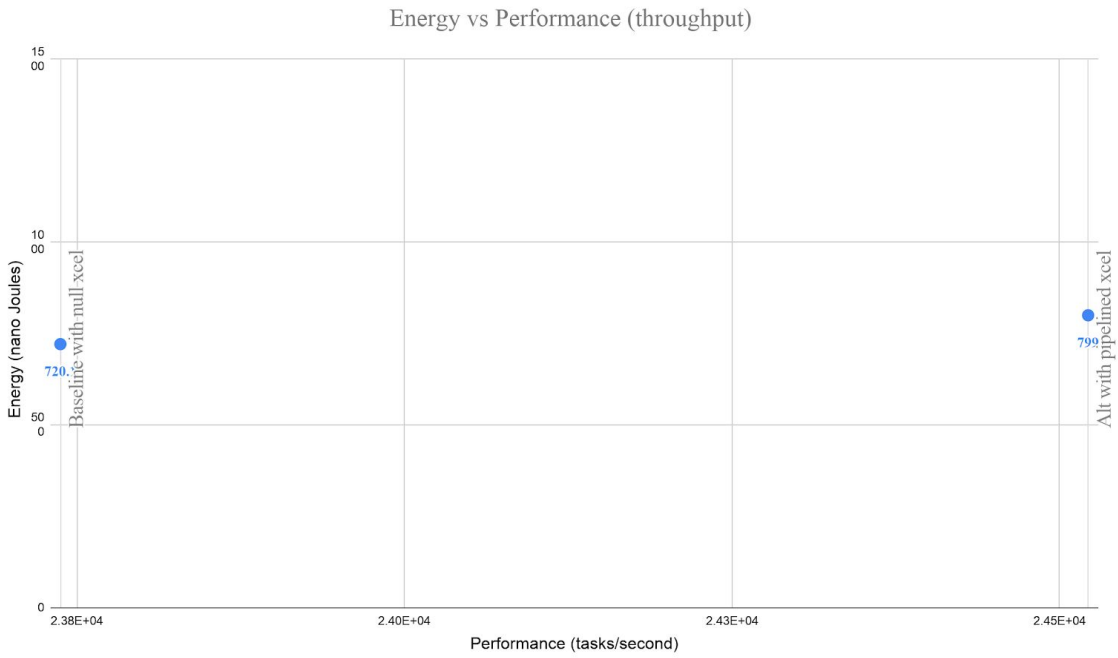
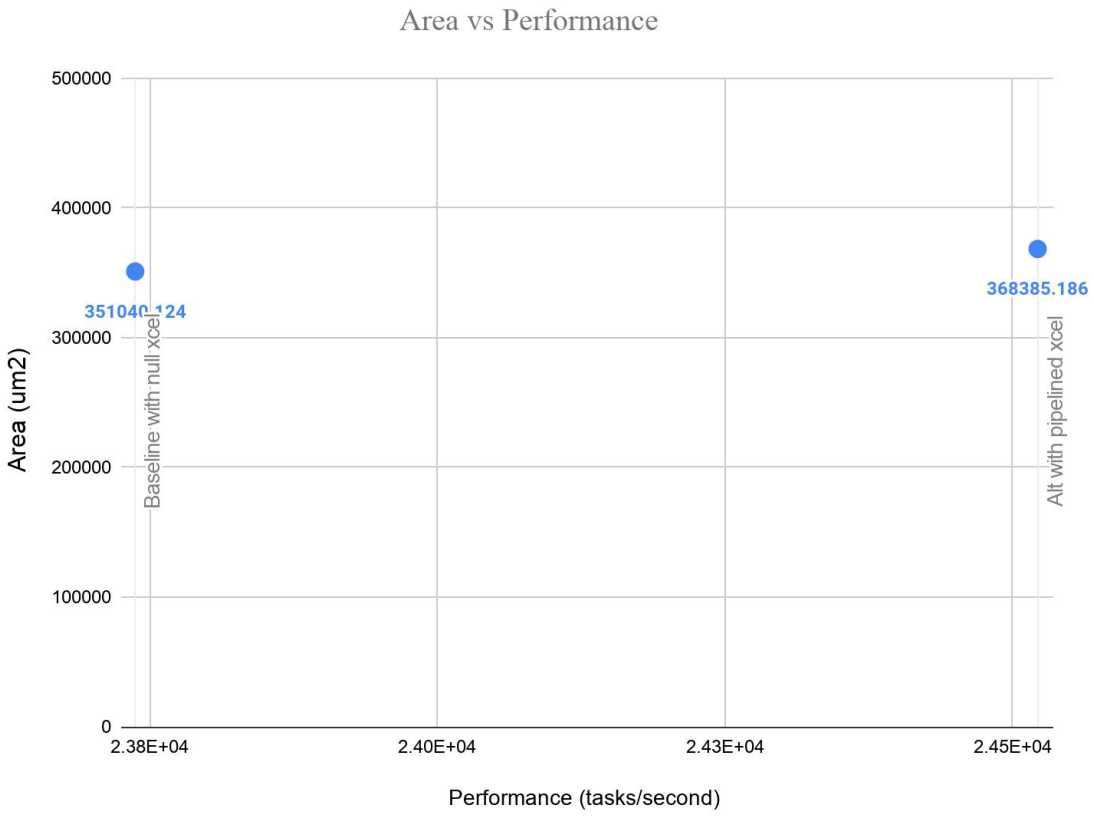


Figure 16: Number of integers is not a multiple of 8, and there is an odd number of blocks



Plot 1a: Design-space exploration plot: Energy vs Performance



Plot 1b: Design-space exploration: Area vs Performance

Design	Design area (um^2)
Baseline Design (Processor + Memory + Null Xcel)	351040.124
Accelerator	16325.218
Processor + Memory + Accelerator	368385.186

Table 1: Area report for Baseline and Alternative Design

Metric	Pure-software	Accelerated	Relative
Area (um^2)	351040.124	368385.186	1.04x
Cycle Time (ns)	2.5	2.5	1.00x
Cycles	16851	16312	0.96x
Energy (nJ)	720.1	799.8	1.11x

Table 2: Summary of the performance, area, energy, and timing comparison between the pure-software sort microbenchmark and the accelerated sort microbenchmark

Hierarchy	Int Power	Switch Power	Leak Power	Total Power	%
ProcMemXcel_null_rtl	7.33e-03	6.36e-03	1.03e-03	1.47e-02	100.0
icache (BlockingCacheRTL__num_banks_0_0)	2.09e-03	1.57e-03	3.17e-04	3.98e-03	27.0
dcache (BlockingCacheRTL__num_banks_0_1)	1.58e-03	1.43e-03	3.69e-04	3.38e-03	22.9
xcel (NullXcelRTL__nbits_32)	1.80e-05	1.68e-07	1.08e-05	2.89e-05	0.2
proc (ProcRTL__num_cores_1)	3.40e-03	2.85e-03	3.10e-04	6.55e-03	44.5

Figure 17: Power Report for the Baseline Design i.e. Processor, Memory and Null Accelerator
(Note how the Memory and Processor consume ~95% of the power)

Hierarchy	Int Power	Switch Power	Leak Power	Total Power	%
ProcMemXcel_sort_rtl	9.16e-03	6.17e-03	1.45e-03	1.68e-02	100.0
icache (BlockingCacheRTL__num_banks_0_0)	2.24e-03	1.44e-03	3.14e-04	3.99e-03	23.8
dcache (BlockingCacheRTL__num_banks_0_1)	1.25e-03	1.06e-03	3.83e-04	2.69e-03	16.0
xcel (lab2_xcel_SortXcelVRTL_wrapper)	1.81e-03	3.92e-04	4.01e-04	2.60e-03	15.5
sorter (lab2_FuncUnitSort8VRTL)	1.49e-03	3.07e-04	2.86e-04	2.08e-03	12.4
C1A (lab2_FuncUnitSortVRTL_5)	3.12e-07	6.05e-07	4.08e-05	4.17e-05	0.2
C1B (lab2_FuncUnitSortVRTL_4)	3.08e-07	7.24e-07	4.05e-05	4.15e-05	0.2
C2A (lab2_FuncUnitSortVRTL_3)	5.50e-07	1.09e-06	4.06e-05	4.22e-05	0.3
C2B (lab2_FuncUnitSortVRTL_2)	4.55e-07	9.46e-07	4.05e-05	4.19e-05	0.2
C3A (lab2_FuncUnitSortVRTL_1)	6.61e-07	1.10e-06	4.27e-05	4.45e-05	0.3
C3B (lab2_FuncUnitSortVRTL_0)	6.62e-07	1.32e-06	4.16e-05	4.36e-05	0.3
proc (ProcRTL__num_cores_1)	3.39e-03	2.61e-03	3.20e-04	6.32e-03	37.7

Figure 18: Power Report for Alternative Design i.e. Processor, Memory and Pipelined Accelerator
(Note how the memory and processor occupy ~80% of power and the accelerator occupies 15.5% of power)

Hierarchy	Area (um^2)	%
lab2_xcel_SortXcelRTL	16325.218	100.0
sorter(lab2_FuncUnitSort8VRTL)	11305.532	69.0
C3A (lab2_FuncUnitSortVRTL_1)	1530.032	13.5
C2B (lab2_FuncUnitSortVRTL_2)	1489.866	13.2
C3B (lab2_FuncUnitSortVRTL_0)	1492.792	13.2
C1B (lab2_FuncUnitSortVRTL_4)	1485.876	13.1
C2A (lab2_FuncUnitSortVRTL_3)	1484.812	13.1
C1A (lab2_FuncUnitSortVRTL_5)	1485.61	13.1

Figure 19: Area Report for Accelerator

Hierarchy	Int Power	Switch Power	Leak Power	Total Power	%
ProcMemXcel_null_rtl	7.33e-03	6.36e-03	1.03e-03	1.47e-02	100.0
proc (ProcRTL__num_cores_1)	3.40e-03	2.85e-03	3.10e-04	6.55e-03	44.5
imul (IntMulScycleRTL)	1.06e-03	9.94e-04	4.98e-05	2.10e-03	14.3
rf (RegisterFile__cf7325a12c1669d6)	4.22e-04	3.09e-04	1.17e-04	8.47e-04	5.8
clk_gate_regs_reg_20_ (SNPS_CLOCK_GATE_HIGH_RegisterFile__cf7325a12c1669d6_12)	2.53e-06	2.88e-07	1.24e-07	2.94e-06	0.0
clk_gate_regs_reg_21_ (SNPS_CLOCK_GATE_HIGH_RegisterFile__cf7325a12c1669d6_11)	2.52e-06	3.11e-07	1.24e-07	2.96e-06	0.0
clk_gate_regs_reg_22_ (SNPS_CLOCK_GATE_HIGH_RegisterFile__cf7325a12c1669d6_10)	2.45e-06	1.94e-07	1.24e-07	2.76e-06	0.0

Figure 20: Power Report for the processor for Baseline Design

Hierarchy	Int Power	Switch Power	Leak Power	Total Power	%
ProcMemXcel_sort_rtl	9.16e-03	6.17e-03	1.45e-03	1.68e-02	100.0
proc (ProcRTL__num_cores_1)	3.39e-03	2.61e-03	3.20e-04	6.32e-03	37.7
imul (IntMulScycleRTL)	9.03e-04	8.11e-04	4.89e-05	1.76e-03	10.5
rf (RegisterFile__cf7325a12c1669d6)	4.07e-04	3.01e-04	1.17e-04	8.25e-04	4.9
clk_gate_regs_reg_20_ (SNPS_CLOCK_GATE_HIGH_RegisterFile__cf7325a12c1669d6_12)	2.73e-06	5.40e-07	1.24e-07	3.40e-06	0.0
clk_gate_regs_reg_21_ (SNPS_CLOCK_GATE_HIGH_RegisterFile__cf7325a12c1669d6_11)	2.60e-06	3.70e-07	1.24e-07	3.10e-06	0.0

Figure 21: Power Report for the processor for Alternative Design