1.0 Introduction

The following document serves as the lab report for assignment 3 of ECE 1756 (Fall 2021). This report is written by Sheng Zhao (1003273913).

Section 2.0 describes the C++ implementation of the desired logical RAM mapping application as well as the the means by which it is further optimized. Section 3.0 then presents and evaluates the mappings produced by this implementation. Finally, a brief Section 4.0 covers the potential future work that can be done to improve upon the current implementation.

2.0 RAM Mapper

The objective of the assignment is to create a RAM mapping tool that produces a legal mapping of desired logical RAMs to available physical RAMs. A total of roughly 15000 different logical RAMs across 69 circuits serves as the benchmark of the implementation, with the area metric calculated by the checker executable provided by the course serving as the evaluation criteria. Details of the problem statement can be found within the lab manual.

The RAM mapping functionality is implemented using C++ in an Object Oriented Programming fashion. This section details the process that led to the final implementation. Skip to subsection 2.4 for the final implementation. Big O notation will only be given for the final implementation's core calculations (initialization portions are skipped). Note that due to time constraints, the implementation only ever map logical RAMs to a single type of physical RAM.

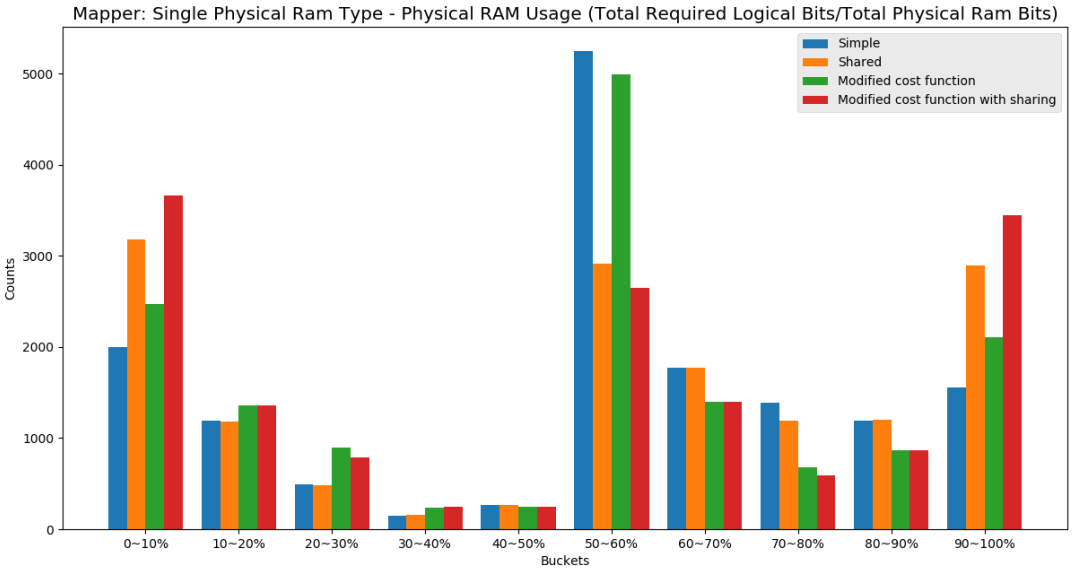

Figure 1: Physical RAM usage for single ram type mapping

To help illustrate how well the instantiated physical RAMs are being utilized, Fig. 1 (python code used to generate this can be found in Appendix A) was generated by first computing the percentage of total logical bits over the total physical bits within the RAM that it is mapped to. The 15000 total data points are then placed into ten buckets based on this percentage value. In essence, the intuition is that the larger the utilization the better (and less area is wasted) and this is represented with a higher count in the lower buckets (on the left).

In general, it would seem like the final implementation (in red) has shown an improvement when compared to the initial implementation (in blue). There is a decrease in counts for buckets for 50% 90% and increase in counts for buckets of lower wastage percentages. However, the final bucket for 90% 100% showed an increase which suggest that some other optimization method other than sharing and improving the cost function (explained in the following subsections) might be required.

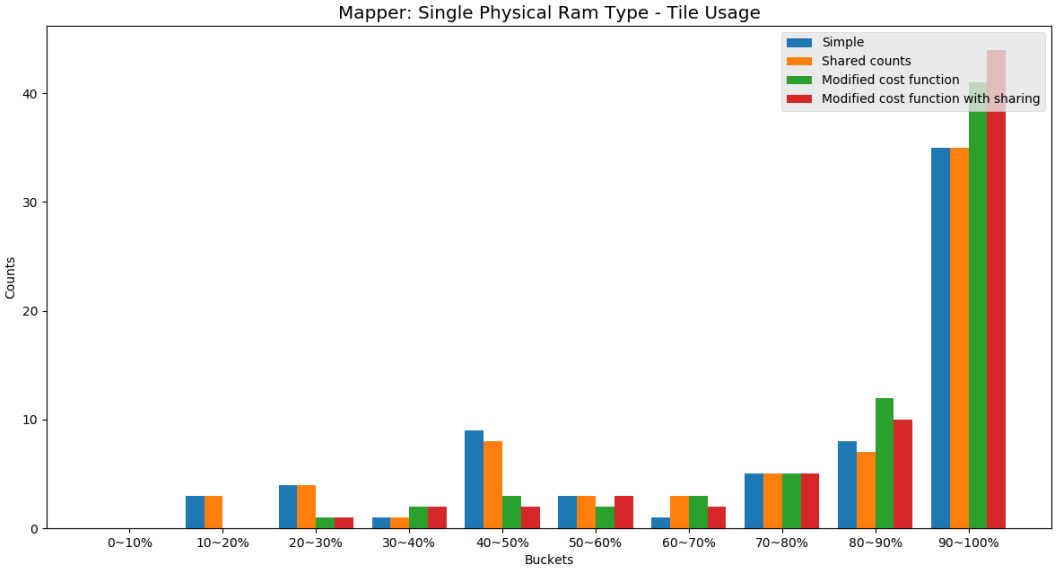

Figure 2: Tile usage for single ram type mapping

A similar computation is done for Fig. 2 (python code in Appendix B) for how well the logic block count relates to the total final tile count. Having a large discrepancy between the two values for each circuit (graphically this means more data points falling in the lower buckets, i.e. lower usage) means that one physical RAM type might be overused, leading to more tiles being needed. As shown in the figure, the final implementation (in red) showed improvement over the initial (in blue) with more circuits utilizing more of the available tiles.

2.1 Single Ram Type - Simple

The RAM mapping tool began with just a simple idea, to map each logical RAM, regardless of which circuit they belong to. The main computation essentially produces a solution for each logical RAM requested for each type of physical RAM, and the lowest cost solution is then chosen to be the final solution. In the event that the physical RAM can operate in different widths, all combinations that do not violate any legality checks are considered. To keep computations simple, the cost function used is simply the area of the physical RAM used (i.e. logic block areas for LUTRAMs, BRAM areas for BRAMs, additional required LUTs are included).

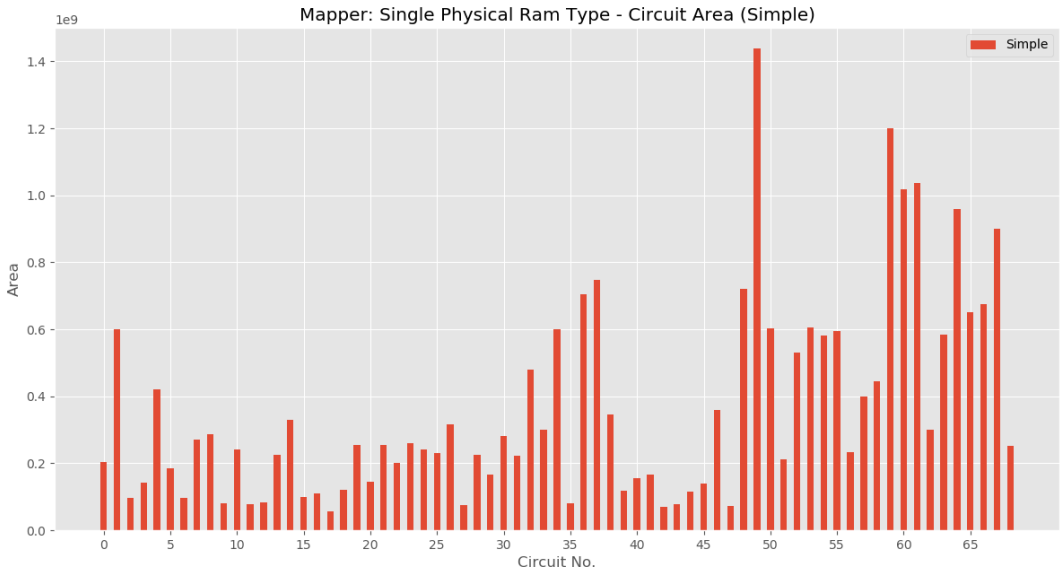

Figure 3: Area usage for simple implementation

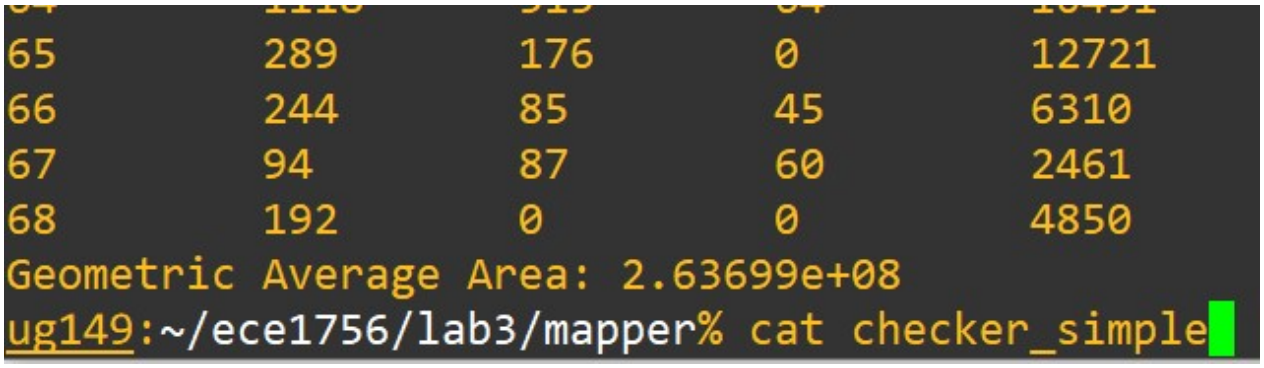

Figure 4: Geometric average for simple implementation

With the simple implementation, a mapping was computed and evaluated using the provided checker executable. A geometric average area of $2.636\,99 \times 10^8$ MWTAs (Fig. 4) is obtained and Fig. 3 (python script in Appendix C) shows the area usage for each of the 69 circuits.

In addition, with respect to Fig. 1, The simple implementation (shown in blue) seems to be under utilizing the physical RAMs that are instantiated as most of these physical RAMs are seeing more than 50% of their available space wasted.

2.2 Single Ram Type - Shared

With under utilization in mind, an effort was made to share the physical RAMs are much as possible. To facilitate the sharing of physical RAM, a map of logical RAM width to a vector of logical RAM IDs is created during initialization when the test cases are being read in. The vectors of logical RAMs are also sorted based on the logical RAM depth in descending order.

Building on top of the simple implementation, every time a solution is explored for the current logical RAM, a lookup is also performed into this newly created map. This is done first with the width that is currently being explored and, should no valid candidate be found, smaller widths in powers of two. Each candidate is checked for validity based on their required modes, the physical RAM's current operating mode, whether if their width is possible with the physical RAM in TrueDualPort mode, whether if they are already mapped, etc. The cost function for the current logical RAM is adjusted according to the ratio of the two logical RAM's total bit requirements. This translates to a strict decrease in cost in all cases. Should a solution with RAM sharing be chosen, this entry is recorded twice, once for each logical RAM, and both logical RAMs are marked as done.

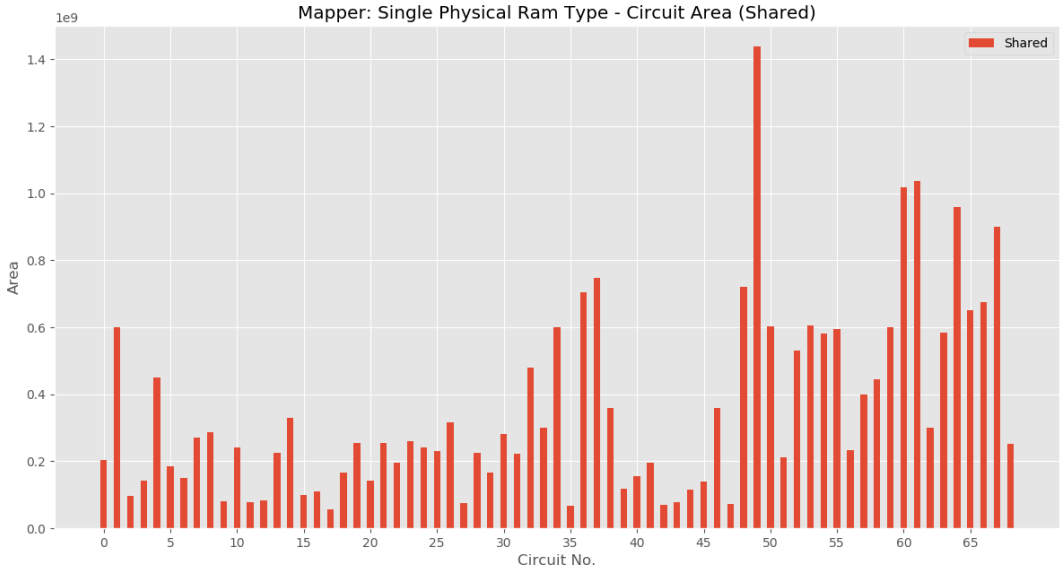

Figure 5: Area usage for shared implementation

```
65          289         176         0           12721       130
66          244         85          45          6310        135
67          94          87          60          2461        180
68          192         0           0           4850        504
Geometric Average Area: 2.64334e+08
ug149:~/ece1756/lab3/mapper% cat checker_shared
```

Figure 6: Geometric average for shared implementation

With the shared implementation, a geometric average area of $2.643\,34 \times 10^8$ MWTAs (Fig. 4) is obtained which is higher than before. The breakdown of area usage in Fig. 3 shows that while circuit 59 has seen a halving in area requirement, many other circuits are in fact using more area than before. This is likely due to the inaccurate cost function which only accounts for the current logical RAM that is being evaluated.

It is also interesting to note that Fig. 1 shows that sharing physical RAM does indeed increase utilization. However, the increase in geomtric average area suggests that the initial intuition of percentage of physical RAM utilized is closely related to area is untrue.

2.3 Single Ram Type - Modified Cost Function

With the cost function in mind, an implementation was done on top of the simple implementation as a proof of concept with lower complexity in the code. The improved cost function does not only account for the area required by the physical RAMs themselves, but also any additional area incurred when the use of the desired physical RAM requires more logic blocks. For example, if based on the current number of logic blocks there are only X number of BRAM8192 blocks and the current solution (that is being evaluated) requires Y more than the limit, the cost function will also include the instantiation of an addition Y times the ratio (10 in this case for Stratix-IV) number of logic blocks.

The modified cost function begins by computing the existing amount of each type of physical RAM based on the logic blocks used for the circuit logic. With each iteration, the number and type of physical RAM chosen for each solution is recorded. When a potential solution seeks to use a RAM type that is not available anymore, the addition cost is accounted for. This has the effect of delaying the use of unavailable physical RAM type. When a solution is chosen, and should it exceed the currently available amount, a new total of each RAM type is computed based on the new logic block count (e.g. using an extra BRAM128K causes 300 more logic blocks to be added, which is 30 more BRAM8192 blocks).

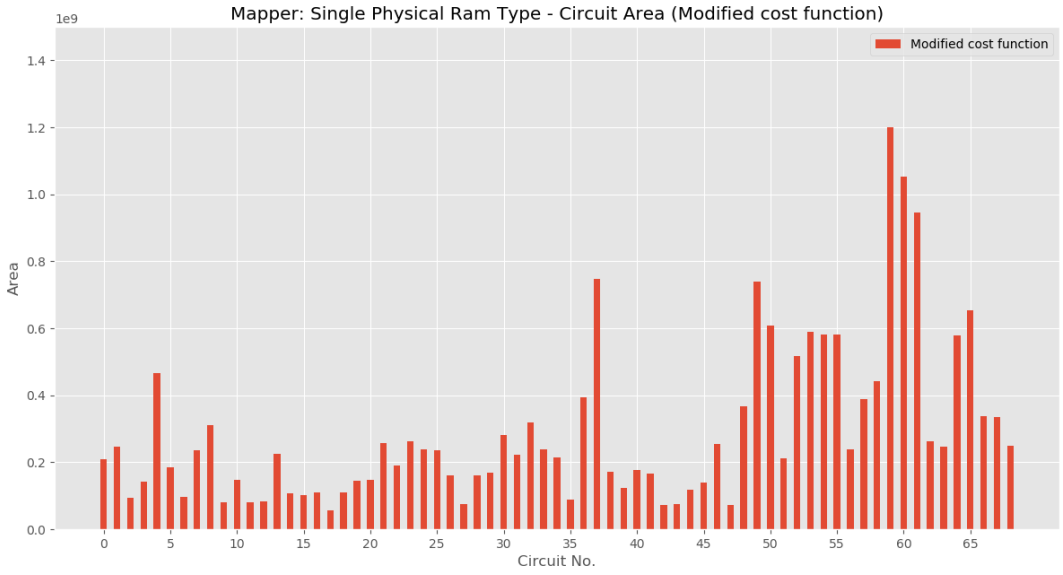

Figure 7: Area usage for modified cost function implementation

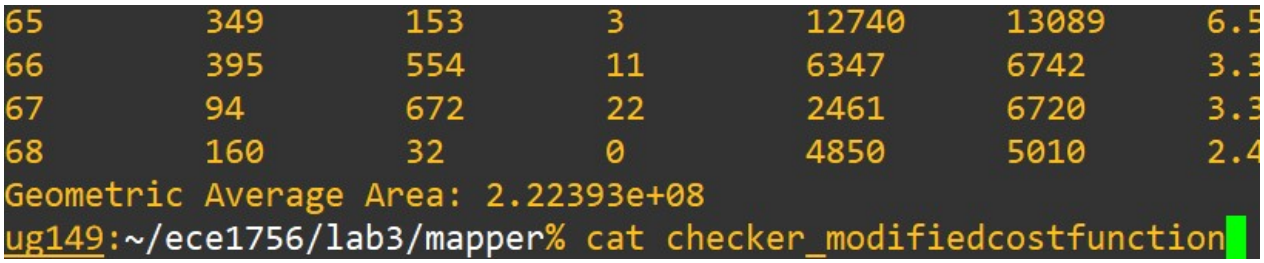

Figure 8: Geometric average for modified cost function implementation

The modified cost function is a success and brought the geometric average area down to $2.223\,93 \times 10^8$ MWTAs (Fig. 8) and Fig. 7 also shows a significant decrease in area used for many circuits.

2.4 Single Ram Type - Modified Cost Function with Sharing

Getting to the final implementation, the modified cost function and physical RAM sharing components are merged into a single function with both implementations still operating the same way they did before.

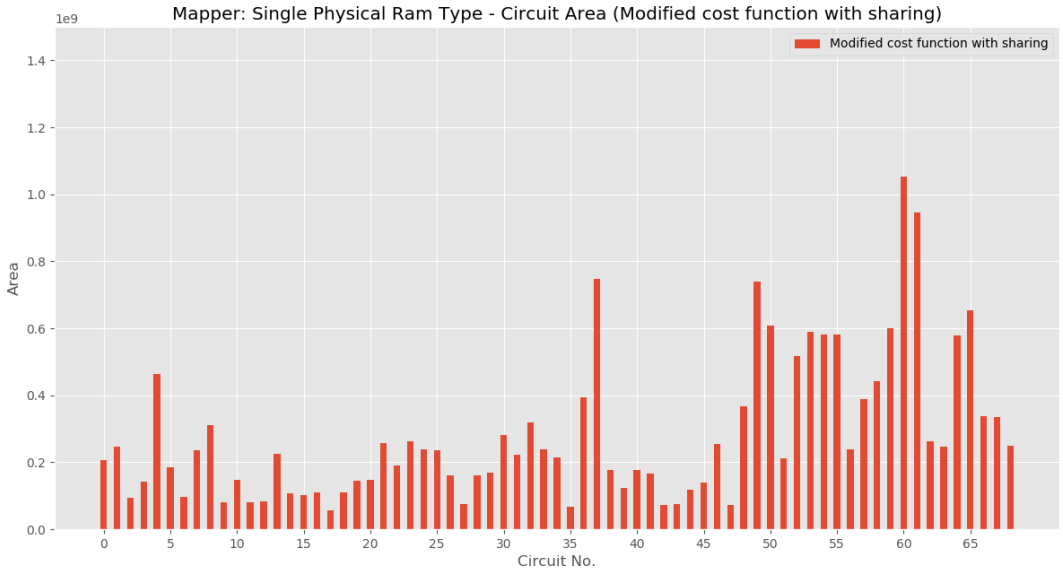

Figure 9: Area usage for modified cost function with physical RAM sharing implementation

```
62          353         461         15          4921        5274        2.63118e+08
63          0           491         8           4857        4910        2.45141e+08
64          1120        1010        33          10453       11573       5.7802e+08
65          349         153         3           12740       13089       6.53702e+08
66          395         554         11          6347        6742        3.36614e+08
67          94          672         22          2461        6720        3.35595e+08
68          160         32          0           4850        5010        2.49857e+08
Geometric Average Area: 2.1939e+08
ug149:~/ece1756/lab3/mapper% cat checker_modifiedcostfunction_withsharing
```

Figure 10: Geometric average for modified cost function with physical RAM sharing implementation

The addition of the physical RAM sharing portion further improved the geometric average area to 2.1939×10^8 MWTAs (Fig. 10). Similar to when we switched from simple to shared implementations Fig. 9 shows the same decrease for circuit 59 and a very slight increase in area usage for circuits 38.

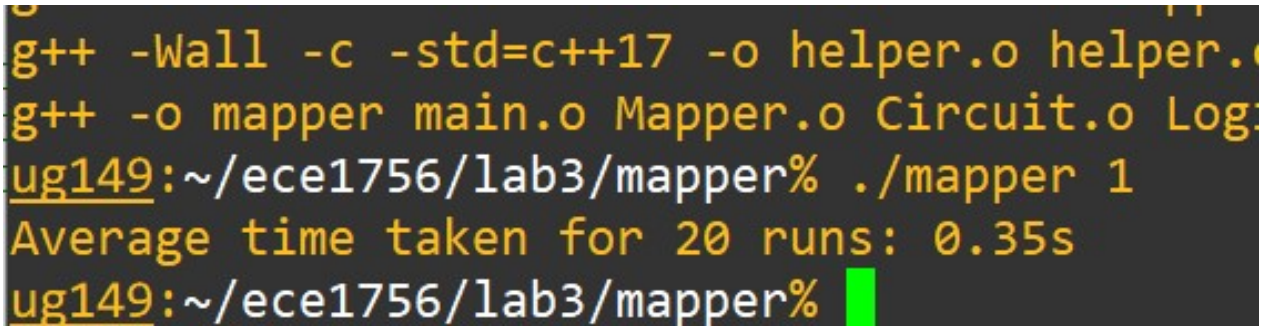

Figure 11: Average time taken

The final implementation took an average of 0.35 seconds to complete the entire computation (including initialization) over 20 runs (Fig. 11). The command needed to run the mapping tool with Stratix-IV architecture is ./mapper 1.

2.4.1 Big O

```
for circuit in all_circuits
    for logical_ram in all_logical_rams
        ## First two for loops combine to be ~15000 O(N)

        for physical_ram in all_physical_rams
            ## Up to 3 types O(T)

            for width_setting in all_possible_widths
                ## Up to 8 for 128bit width
                ## If combined with BRAM type loop O(W)

                // Compute P depending on logical width
                // Compute S depending on logical depth
                // Compute additional LUTs depending on S
                // Compute area based on P, S, additional LUTs
                // Compute cost based on area, physical_ram availability by type
                ## Constant time computations above O(C)


                // Search for compatible sharing target
                for width in widths less than or equal to width_setting
                    ## Up to 7 for 64bit width, since TrueDualPort cannot run at 128bit width
                    O(W)

                    for logical_ram in logical_ram_ROM_SinglePORT
                        ## Cannot be more than 15000 (N)

                        // Check for legality
                        // If found, insert potential match, goto EndOfLoop;
                        ## Constant time computations above O(C)
                        === O(N*T*W*W*N)

                // Insert potential solution
            // Get lowest cost solution across all width_setting
            ## Sort used, O(W log W), W is size of solution array, same as potential widths
            ## W up to 8 for 128 bit width O(W)
            === O(N*T*W*logW)

        // Get lowest cost solution across all_physical_rams
        ## Sort used, O(T log T), T is size of solution array, same as BRAM types
        ## T up to 3 O(T)
        === O(N*T*logT)

        // Choose lowest cost solution
        // If sharing, update other logical_ram to be solved
        ## Constant time computations above O(C)

        for physical_ram in all_physical_rams
            ## Up to 3 types O(T)
            === O(N*T)

            // Update available physical_ram counts by type
            ## Constant time computations above O(C)
```

Figure 12: Pseudocode for main computation

With the following variable definitions:

- N - Total number of test cases

- T - Total number of physical RAM types
- W - Total valid width settings for a given physical RAM type
- C - Denoting constant time computation

The Big-O notation for the main computation is given by $O(N^2 * T * W^2 + N * T * W \log W + N * T \log T + N * T)$.

3.0 Results

3.1 Stratix-IV Architecture

Circuit	Type 1	Type 2	Type 3	Blocks	Tiles	Area	
0	1150	231	3	3016	4166	2.07448e+08	Pass
1	808	492	12	2950	4920	2.45613e+08	Pass
2	72	14	0	1836	1908	9.49983e+07	Pass
3	57	50	1	2811	2868	1.42819e+08	Pass
4	1098	696	23	8177	9275	4.62833e+08	Pass
5	31	264	8	3692	3723	1.85737e+08	Pass
6	76	126	3	1853	1929	9.59789e+07	Pass
7	712	464	15	4017	4729	2.35669e+08	Pass
8	893	303	10	5342	6235	3.10976e+08	Pass
9	1	32	0	1636	1637	8.13783e+07	Pass
10	424	294	1	1428	2940	1.46291e+08	Pass
11	275	58	1	1352	1627	8.09068e+07	Pass
12	15	3	2	1632	1647	8.18498e+07	Pass
13	22	18	0	4498	4520	2.259e+08	Pass
14	101	216	7	1823	2160	1.07809e+08	Pass
15	67	103	2	1956	2023	1.00469e+08	Pass
16	22	77	0	2183	2205	1.09883e+08	Pass
17	2	59	0	1165	1167	5.75142e+07	Pass
18	173	122	2	2044	2217	1.10429e+08	Pass
19	458	291	5	2251	2910	1.44877e+08	Pass
20	253	217	7	2687	2940	1.46291e+08	Pass
21	27	43	1	5105	5132	2.56441e+08	Pass
22	1048	210	3	2761	3809	1.89734e+08	Pass
23	4	204	6	5233	5237	2.61344e+08	Pass
24	351	408	13	4416	4767	2.3748e+08	Pass
25	175	43	0	4543	4718	2.3516e+08	Pass
26	183	325	5	1402	3250	1.6176e+08	Pass
27	32	0	0	1496	1528	7.62288e+07	Pass
28	181	320	7	2027	3200	1.59402e+08	Pass
29	331	224	7	3031	3362	1.67873e+08	Pass
30	202	40	1	5419	5621	2.8036e+08	Pass
31	126	1	0	4347	4473	2.22804e+08	Pass
32	182	639	18	3480	6390	3.19184e+08	Pass
33	526	344	7	4261	4787	2.38423e+08	Pass
34	51	432	13	1705	4320	2.15618e+08	Pass

Figure 13: Checker output for Stratix-IV architecture (top)

```
35        0         80        0         1360      1360      6.75333e+07          Pass
36        339       788       3         1665      7880      3.93698e+08          Pass
37        0         48        0         14969     14969     7.47457e+08          Pass
38        27        354       4         3208      3540      1.76286e+08          Pass
39        302       245       7         1863      2450      1.22335e+08          Pass
40        356       89        2         3217      3573      1.77813e+08          Pass
41        272       335       10        1993      3350      1.67326e+08          Pass
42        104       27        0         1342      1446      7.15307e+07          Pass
43        279       55        0         1212      1491      7.3701e+07           Pass
44        236       127       2         2124      2360      1.1724e+08           Pass
45        2         28        0         2782      2784      1.38897e+08          Pass
46        764       509       7         3509      5090      2.53629e+08          Pass
47        47        18        0         1439      1486      7.34169e+07          Pass
48        107       734       7         6907      7340      3.66533e+08          Pass
49        2903      1128      28        11883     14786     7.38857e+08          Pass
50        246       383       12        11915     12161     6.07468e+08          Pass
51        14        395       13        4206      4220      2.10903e+08          Pass
52        768       149       0         9603      10371     5.17956e+08          Pass
53        1001      200       0         10817     11818     5.90375e+08          Pass
54        717       156       4         10903     11620     5.80265e+08          Pass
55        1280      256       0         10341     11621     5.80303e+08          Pass
56        183       245       7         4594      4777      2.37951e+08          Pass
57        632       123       0         7145      7777      3.87923e+08          Pass
58        1174      83        2         7700      8874      4.43083e+08          Pass
59        0         1200      0         11888     12000     5.99885e+08          Pass
60        502       403       13        20544     21046     1.05191e+09          Pass
61        0         1890      62        15079     18900     9.44819e+08          Pass
62        353       461       15        4921      5274      2.63118e+08          Pass
63        0         491       8         4857      4910      2.45141e+08          Pass
64        1120      1010      33        10453     11573     5.7802e+08           Pass
65        349       153       3         12740     13089     6.53702e+08          Pass
66        395       554       11        6347      6742      3.36614e+08          Pass
67        94        672       22        2461      6720      3.35595e+08          Pass
68        160       32        0         4850      5010      2.49857e+08          Pass
Geometric Average Area: 2.1939e+08
```

Figure 14: Checker output for Stratix-IV architecture (bot)

3.2 Single Physical RAM Type w/ and w/o LUTRAMs

```
Best test case: LUTRAM: False, Size:1K, Width: 32, Ratio: 10, With geomean area: 593610000.0
Best test case: LUTRAM: False, Size:2K, Width: 32, Ratio: 10, With geomean area: 393535000.0
Best test case: LUTRAM: False, Size:4K, Width: 32, Ratio: 10, With geomean area: 293704000.0
Best test case: LUTRAM: False, Size:8K, Width: 32, Ratio: 10, With geomean area: 245787000.0
Best test case: LUTRAM: False, Size:16K, Width: 64, Ratio: 10, With geomean area: 233840000.0
Best test case: LUTRAM: False, Size:32K, Width: 64, Ratio: 10, With geomean area: 253859000.0
Best test case: LUTRAM: False, Size:64K, Width: 128, Ratio: 25, With geomean area: 300032000.0
Best test case: LUTRAM: False, Size:128K, Width: 128, Ratio: 25, With geomean area: 357256000.0
Best test case: LUTRAM: True, Size:1K, Width: 8, Ratio: 10, With geomean area: 460345000.0
Best test case: LUTRAM: True, Size:2K, Width: 16, Ratio: 10, With geomean area: 337596000.0
Best test case: LUTRAM: True, Size:4K, Width: 16, Ratio: 10, With geomean area: 255638000.0
Best test case: LUTRAM: True, Size:8K, Width: 32, Ratio: 10, With geomean area: 216381000.0
Best test case: LUTRAM: True, Size:16K, Width: 32, Ratio: 10, With geomean area: 214437000.0
Best test case: LUTRAM: True, Size:32K, Width: 64, Ratio: 25, With geomean area: 219984000.0
Best test case: LUTRAM: True, Size:64K, Width: 128, Ratio: 50, With geomean area: 236324000.0
Best test case: LUTRAM: True, Size:128K, Width: 128, Ratio: 50, With geomean area: 257645000.0
ug149:~/ece1756/lab3/mapper%
```

Figure 15: Explore summary

To facilitate the design exploration with a single type of physical RAM with and without LUTRAMs, a short python script (Appendix D) was written that works in conjunction with the mapping tool to execute the roughly 600 different combinations of max BRAM size, with different widths and ratios, as well as with and without LUTRAMs. For each max size from 1K to 128K, widths of 2 to 128 bits are tested. Each set of these different max widths are also tested with ratios of 10, 25, 50, 100 (for sizes up to 16K), 150 (for 32K), 200 (for 64K), 300 (for 128K). Finally, the whole test set is rerun with LUTRAMs enabled.

Fig. 15 shows the summary of the exploration and the full dump file can be found in Appendix E. Results shown in Fig. 15 is used to prepare Table 1 and Table 2.

BRAM Size	**Max Width**	**LBs / BRAM**	**Geometric Average FPGA Area (MWTAs) W/O BRAM128K Configured**
1 kbit	32	10	5.9361×10^8
2 kbit	32	10	$3.935\,35 \times 10^8$
4 kbit	32	10	$2.937\,04 \times 10^8$
8 kbit	32	10	$2.457\,87 \times 10^8$
16 kbit	64	10	2.3384×10^8
32 kbit	64	10	$2.538\,59 \times 10^8$
64 kbit	128	25	$3.000\,32 \times 10^8$
128 kbit	128	25	$3.572\,56 \times 10^8$

Table 1: Results without LUTRAM

BRAM Size	**Max Width**	**LBs / BRAM**	**Geometric Average FPGA Area (MWTAs) W/O BRAM128K Configured**
1 kbit	8	10	$4.603\,45 \times 10^8$
2 kbit	16	10	$3.375\,96 \times 10^8$
4 kbit	16	10	$2.556\,38 \times 10^8$
8 kbit	32	10	$2.163\,81 \times 10^8$
16 kbit	32	10	$2.144\,37 \times 10^8$
32 kbit	64	25	$2.199\,84 \times 10^8$
64 kbit	64	25	$2.363\,24 \times 10^8$
128 kbit	128	50	$2.576\,45 \times 10^8$

Table 2: Results with LUTRAM

3.2.1 W.R.T Stratix-IV

With respect to the original Stratix-IV architecture, some test results show that they can be more area efficient (BRAM sizes 8K, 16K and 32K with LUTRAM). This was an interesting observation since it suggested that simply having one type of BRAM was more efficient than what the industry actually implemented in a commercial product. The reason for a smaller area is likely due to the lack of BRAM128K, which means area that was previously dedicated to these BRAMs are now freed.

3.2.2 Trends

First, the results show that area usage for the architecture with LUTRAMs are much lower than that without. This is likely due to the increased granularity in terms of available physical RAM sizes. LUTRAMs are much more efficient for logical RAMs that have a small size.

Next, for both results with and without LUTRAM, max width is seen increasing as max size increases. This is because the depth increases with the size of a BRAM assuming width stays the same. The depth demanded by logical RAMs are only so much and at some point in time, not having enough width means that many physical RAMs are required in parallel. The additional capacity that is provided by these physical RAMs (by the means of depth) is thus wasted. With RAM sharing being only possible for up to two logical RAMs at a time, this wastage translates to more physical RAMs being required and an increase in area usage.

As for logic blocks per BRAM, the larger sized BRAMs seem to prefer a higher ratio. Once again, just like the observation with respect to Stratix-IV, having these large BRAMs too common will bump up the area since too many of them are present and not all of them might be used (as supported by examining the output table of checker executable, where many circuits do not utilize all available BRAMs).

Finally for the geometric average area computed, the trend is that the lowest values are found for medium sized BRAMs. This is likely due to them having the best balance between total capacity and area. BRAMs that are too small require many more to be instantiated in series or parallel, with additional LUTs and/or wastage of capacity adding up, leading to inefficient use of area. BRAMs that are too large also experience the same issue.

3.3 Custom Architecture

```
ug149:~/ece1756/lab3/final_mapper% cat explore_summary_custom
Best test case: LUTRAM: 50.0%, LBs/LUTRAM: 2, Size_1: 8K, Width_1: 32, Ratio_1: 10,
 Size_2: 128K, Width_2: 64, Ratio_2: 300, With geomean area: 218734000.0
Best test case: LUTRAM: 33.3333333333%, LBs/LUTRAM: 3, Size_1: 8K, Width_1: 32, Rat
io_1: 10, Size_2: 128K, Width_2: 32, Ratio_2: 150, With geomean area: 212881000.0
Best test case: LUTRAM: 25.0%, LBs/LUTRAM: 4, Size_1: 8K, Width_1: 32, Ratio_1: 10,
 Size_2: 128K, Width_2: 32, Ratio_2: 150, With geomean area: 210536000.0
Best test case: LUTRAM: 10.0%, LBs/LUTRAM: 10, Size_1: 8K, Width_1: 32, Ratio_1: 10
, Size_2: 128K, Width_2: 64, Ratio_2: 150, With geomean area: 209068000.0
ug149:~/ece1756/lab3/final_mapper%
```

Figure 16: Explore summary for custom architecture

Once again, to facilitate the exploration of custom architectures, a python script (Appendix F, full dump file in Appendix G) is written to feed custom test cases into the mapping tool as well as automatically evaluate it using the checker executable and record the results. In terms of the explored architecture, one with LUTRAM enabled and two additional types of BRAM is chosen. LUTRAM size and widths are the same as Stratix-IV and only the ratio is varied. As for BRAMs, 8K, 32K, 64K and 128K sizes with varying widths and ratios are tested due to the better performance as seen from before.

Of all the architectures tested, the best result came from one with 10% LUTRAMs, BRAM8192 (max width 32, ratio 10), BRAM128K (max width 64, ratio 150) with a geometric average area of $2.090\,68 \times 10^8$.

Area usage across the 69 circuits for both the Stratix-IV and custom architectures are plotted in Fig. 17 (python script in Appendix H). The figure shows that most circuits gained an area reduction while a select few ended up with larger areas than before. A large portion of area reduction is likely related to the lower ratio of BRAM128K. Taking circuits 32, 49 (Fig. 18) and 67 for instance, these circuits show the largest area reduction and closer inspection into the results generated by the checker executable shows that the number of BRAM128K used is much more than for Stratix-IV. On the other hand, other circuits like circuit 0 (Fig. 19) suffered an area increase due to the lower LUTRAM count.

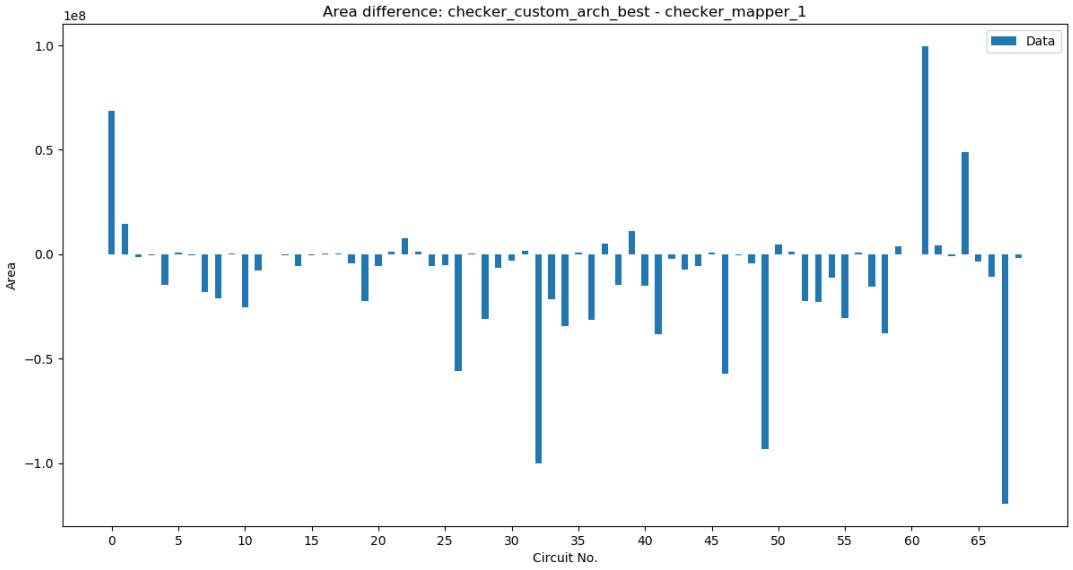

Figure 17: Comparison between Stratix-IV and custom architecture

```
ug149:~/ece1756/lab3/final_mapper% grep "49        2903" checker_mapper_1
49        2903      1128      28        11883     14786     7.38857e+08          Pass
ug149:~/ece1756/lab3/final_mapper% grep "49        960" checker_custom_arch_best
49        960       963       85        11883     12843     6.45672e+08          Pass
ug149:~/ece1756/lab3/final_mapper%
```

Figure 18: Comparison between circuit 49s

```
ug149:~/ece1756/lab3/final_mapper% grep "0         548" checker_custom_arch_best
0         548       550       33        3025      5500      2.7621e+08           Pass
ug149:~/ece1756/lab3/final_mapper% grep "0         1150" checker_mapper_1
0         1150      231       3         3016      4166      2.07448e+08          Pass
ug149:~/ece1756/lab3/final_mapper%
```

Figure 19: Comparison between circuit 0s

4.0 Future Work

Cost function can be improved to account for sharing better. The current implementation only considers the computation for the current logical RAM being evaluated. The other chosen logical RAM gets mapped to the same physical RAM if the cost of the sharing solution is the lowest for the current logical RAM regardless if there might be a better solution for the other one. A better solution would perhaps hold off on deciding a solution until both are evaluated. However, this leads to a much more complex algorithm which the current implementation is unsuited for. Such an algorithm would not choose any solution until all potential ones are computed instead of doing so one by one like the current implementation.

Cost function also does not care about utilization of physical RAM directly during the mapping of logical RAMs, which are evaluated in the order in which they are entered in the test case file. As a result, physical RAMs might not be utilized in the best manner especially when it comes to having to instantiate ones that are not available given the current logic block size. A smaller sized logical RAM might get mapped to a BRAM128K, since that might be the only one that is available without instantiated more logical blocks, which is inefficient if the next logical RAM can utilize the BRAM128K better. The current smaller sized one can then map to a newly available LUTRAM or BRAM8192 since 300 more logical blocks will be added.

To improve the results further, we can explore algorithms that make use of "physical" RAMs that are made of the given, actually available physical RAMs. For example, in an architecture with LUTRAMs and BRAMs of different sizes, we can take some combination of each of them to construct a new "physical" RAM which can then be added to the list of physical RAMs to be evaluated. Since there are infinitely many combinations, some amount of heuristics will have to be used to determine what some of the efficient ways logical RAMs can be grouped together. This insight and specialized mapping technique will then only be used for particular combinations logical RAMs with the right depths and widths.

5.0 Appendix

Appendix A: .py file for computing physical RAM usage

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np

# LW = []
# LD = []
# ID = []
# S = []
# P = []
# Type = []
# W = []
# D = []

filenames = ["mapping_simple", "mapping_shared",
↪  "mapping_modifiedcostfunction",
↪  "mapping_modifiedcostfunction_withsharing"]

all_data = {}

for filename in filenames:
    current_file_data = []
    data_fd = open(filename, 'r')
    for line in data_fd.readlines():
        segments = line.strip().split(" ")

        LW_n = float(segments[4])
        LD_n = float(segments[6])
        ID_n = str(segments[0])+"_"+str(segments[8])
        S_n = float(segments[10])
        P_n = float(segments[12])
        Type_n = float(segments[14])
        W_n = float(segments[18])
        D_n = float(segments[20])

        # LW.append(LW_n)
        # LD.append(LD_n)
        # ID.append(ID_n)
```

```
        # S.append(S_n)
        # P.append(P_n)
        # Type.append(Type_n)
        # W.append(W_n)
        # D.append(D_n)

        Total_logic_bits_n = LW_n*LD_n
        Total_used_bits_n = S_n*P_n*W_n*D_n

        data = {}
        data["ID"] = ID_n
        data["total_logic"] = Total_logic_bits_n
        data["total_used"] = Total_used_bits_n

        current_file_data.append(data)
    all_data[filename] = current_file_data

plot_data = {}
for filename in filenames:
    temp = []
    for x in range(10):
        temp.append(0)
    plot_data[filename] = temp

for filename in filenames:
    print(filename)
    for count, each_data_point in enumerate(all_data[filename]):
        if (count % 1000 == 0):
            print(count)
        same_id = [x for x in all_data[filename] if x["ID"] ==
        ↪  each_data_point["ID"]]

        # print(same_id)

        all_data_temp = [x for x in all_data[filename] if x not in same_id]
        all_data[filename] = all_data_temp

        total_total_logic_bits = 0
        total_used_sanity = each_data_point["total_used"]

        for each_same_id_point in same_id:
            total_total_logic_bits += each_same_id_point["total_logic"]
```

```
            if each_same_id_point["total_used"] !=
            ↪  each_data_point["total_used"]:
                print("Warning: Same ID but different physical ram!!")


        Percentage_waste_n =
        ↪  (total_used_sanity-total_total_logic_bits)/total_used_sanity *
        ↪  100

        if Percentage_waste_n >= 0 and Percentage_waste_n < 10:
            plot_data[filename][0] += 1
        elif Percentage_waste_n >= 10 and Percentage_waste_n < 20:
            plot_data[filename][1] += 1
        elif Percentage_waste_n >= 20 and Percentage_waste_n < 30:
            plot_data[filename][2] += 1
        elif Percentage_waste_n >= 30 and Percentage_waste_n < 40:
            plot_data[filename][3] += 1
        elif Percentage_waste_n >= 40 and Percentage_waste_n < 50:
            plot_data[filename][4] += 1
        elif Percentage_waste_n >= 50 and Percentage_waste_n < 60:
            plot_data[filename][5] += 1
        elif Percentage_waste_n >= 60 and Percentage_waste_n < 70:
            plot_data[filename][6] += 1
        elif Percentage_waste_n >= 70 and Percentage_waste_n < 80:
            plot_data[filename][7] += 1
        elif Percentage_waste_n >= 80 and Percentage_waste_n < 90:
            plot_data[filename][8] += 1
        elif Percentage_waste_n >= 90 and Percentage_waste_n <= 100:
            plot_data[filename][9] += 1
        else:
            print("Warning: Percentage_waste_n does not fit within buckets:
            ↪  {}", Percentage_waste_n)

        #print("Total_logic_bits:{}, Total_used_bits:{},
        ↪  Percentage_waste_n:{}".format(Total_logic_bits,
        ↪  Total_used_bits, Percentage_waste_n))


X = np.arange(10)
fig = plt.figure(dpi=100, figsize=(14, 7))
ax = fig.add_axes([0.1,0.1,0.8,0.8])
plt.style.use('ggplot')
```

```
ax.bar(X - 0.30, plot_data["mapping_simple"], width = 0.2, label="Simple")
ax.bar(X - 0.10, plot_data["mapping_shared"], width = 0.2, label="Shared")
ax.bar(X + 0.10, plot_data["mapping_modifiedcostfunction"], width = 0.2,
↪  label="Modified cost function")
ax.bar(X + 0.3, plot_data["mapping_modifiedcostfunction_withsharing"], width
↪  = 0.2, label="Modified cost function with sharing")

plt.legend(loc="upper right")
plt.title('Mapper: Single Physical Ram Type - Physical RAM Usage (Total
↪  Required Logical Bits/Total Physical Ram Bits)')
plt.xlabel('Buckets')
plt.ylabel('Counts')
plt.xticks(np.arange(10), ['0~10%', '10~20%', '20~30%', '30~40%', '40~50%',
↪  '50~60%', '60~70%', '70~80%', '80~90%', '90~100%'])
plt.tick_params(axis='both', which='major', labelsize=10)
plt.show()
plt.savefig("single_ram_pram_usage.png")
```

Appendix B: .py file for computing tile usage

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np

filenames = ["checker_simple", "checker_shared",
↪  "checker_modifiedcostfunction",
↪  "checker_modifiedcostfunction_withsharing"]

plot_data = {}
for filename in filenames:
    temp = []
    for x in range(10):
        temp.append(0)
    plot_data[filename] = temp

for filename in filenames:
    current_file_data = []

    data_fd = open(filename, 'r')
    data_section = False

    for line in data_fd.readlines():
        segments = line.strip().split()

        if (len(segments) > 0):
            if (segments[0] == "Circuit"):
                data_section = True
                continue
            if (segments[0] == "Geometric"):
                data_section = False
                continue
        if (not data_section):
            continue

        blocks = float(segments[4])
        tiles = float(segments[5])
        usage_percentage = blocks/tiles * 100

        if usage_percentage >= 0 and usage_percentage < 10:
```

```
            plot_data[filename][0] += 1
        elif usage_percentage >= 10 and usage_percentage < 20:
            plot_data[filename][1] += 1
        elif usage_percentage >= 20 and usage_percentage < 30:
            plot_data[filename][2] += 1
        elif usage_percentage >= 30 and usage_percentage < 40:
            plot_data[filename][3] += 1
        elif usage_percentage >= 40 and usage_percentage < 50:
            plot_data[filename][4] += 1
        elif usage_percentage >= 50 and usage_percentage < 60:
            plot_data[filename][5] += 1
        elif usage_percentage >= 60 and usage_percentage < 70:
            plot_data[filename][6] += 1
        elif usage_percentage >= 70 and usage_percentage < 80:
            plot_data[filename][7] += 1
        elif usage_percentage >= 80 and usage_percentage < 90:
            plot_data[filename][8] += 1
        elif usage_percentage >= 90 and usage_percentage <= 100:
            plot_data[filename][9] += 1
        else:
            print("Warning: usage_percentage does not fit within buckets:
            ↪  {}", usage_percentage)


X = np.arange(10)
fig = plt.figure(dpi=100, figsize=(14, 7))
ax = fig.add_axes([0.1,0.1,0.8,0.8])
plt.style.use('ggplot')

ax.bar(X - 0.30, plot_data["checker_simple"], width = 0.2, label="Simple")
ax.bar(X - 0.10, plot_data["checker_shared"], width = 0.2, label="Shared
↪  counts")
ax.bar(X + 0.10, plot_data["checker_modifiedcostfunction"], width = 0.2,
↪  label="Modified cost function")
ax.bar(X + 0.3, plot_data["checker_modifiedcostfunction_withsharing"], width
↪  = 0.2, label="Modified cost function with sharing")

plt.legend(loc="upper right")
plt.title('Mapper: Single Physical Ram Type - Tile Usage')
plt.xlabel('Buckets')
plt.ylabel('Counts')
```

```
plt.xticks(np.arange(10), ['0~10%', '10~20%', '20~30%', '30~40%', '40~50%',
↪  '50~60%', '60~70%', '70~80%', '80~90%', '90~100%'])
plt.tick_params(axis='both', which='major', labelsize=10)
plt.show()
plt.savefig("single_ram_tile_usage.png")
```

Appendix C: .py file for computing circuit area usage

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np

filenames = ["checker_simple", "checker_shared",
↪  "checker_modifiedcostfunction",
↪  "checker_modifiedcostfunction_withsharing"]

X = np.arange(69)
plt.style.use('ggplot')

for filename in filenames:
    data_fd = open(filename, 'r')
    data_section = False

    plot_data = []
    for line in data_fd.readlines():
        segments = line.strip().split()

        if (len(segments) > 0):
            if (segments[0] == "Circuit"):
                data_section = True
                continue
            if (segments[0] == "Geometric"):
                data_section = False
                continue
        if (not data_section):
            continue

        plot_data.append(float(segments[6]))



    fig = plt.figure(dpi=100, figsize=(14, 7))
    ax = fig.add_axes([0.1,0.1,0.8,0.8])

    plot_label = "Simple"
    if (filename == "checker_shared"):
        plot_label = "Shared"
```

```
elif (filename == "checker_modifiedcostfunction"):
    plot_label = "Modified cost function"
elif (filename == "checker_modifiedcostfunction_withsharing"):
    plot_label = "Modified cost function with sharing"

ax.bar(X, plot_data, width = 0.5, label=plot_label)
ax.set_ylim([0, 1.5e9])

plt.legend(loc="upper right")
plt.title('Mapper: Single Physical Ram Type - Circuit Area (' +
↪  plot_label+')')
plt.xlabel('Circuit No.')
plt.ylabel('Area')
plt.xticks(np.arange(0, 70, 5))
plt.tick_params(axis='both', which='major', labelsize=10)
plt.show()
plt.savefig("single_ram_circuit_area_"+filename+".png")
```

Appendix D: .py file for exploring single type of physical ram

```
import os

ljust_lutram_option = 15
ljust_size = 15
ljust_width = 15
ljust_ratio = 15
ljust_geomean = 45

all_tests = []

tests_1k = []
tests_1k.append({'max_bits_K': 1, 'max_width': 2, 'ratio': 10})
tests_1k.append({'max_bits_K': 1, 'max_width': 4, 'ratio': 10})
tests_1k.append({'max_bits_K': 1, 'max_width': 8, 'ratio': 10})
tests_1k.append({'max_bits_K': 1, 'max_width': 16, 'ratio': 10})
tests_1k.append({'max_bits_K': 1, 'max_width': 32, 'ratio': 10})
tests_1k.append({'max_bits_K': 1, 'max_width': 64, 'ratio': 10})
tests_1k.append({'max_bits_K': 1, 'max_width': 128, 'ratio': 10})
tests_1k.append({'max_bits_K': 1, 'max_width': 2, 'ratio': 25})
tests_1k.append({'max_bits_K': 1, 'max_width': 4, 'ratio': 25})
tests_1k.append({'max_bits_K': 1, 'max_width': 8, 'ratio': 25})
tests_1k.append({'max_bits_K': 1, 'max_width': 16, 'ratio': 25})
tests_1k.append({'max_bits_K': 1, 'max_width': 32, 'ratio': 25})
tests_1k.append({'max_bits_K': 1, 'max_width': 64, 'ratio': 25})
tests_1k.append({'max_bits_K': 1, 'max_width': 128, 'ratio': 25})
tests_1k.append({'max_bits_K': 1, 'max_width': 2, 'ratio': 50})
tests_1k.append({'max_bits_K': 1, 'max_width': 4, 'ratio': 50})
tests_1k.append({'max_bits_K': 1, 'max_width': 8, 'ratio': 50})
tests_1k.append({'max_bits_K': 1, 'max_width': 16, 'ratio': 50})
tests_1k.append({'max_bits_K': 1, 'max_width': 32, 'ratio': 50})
tests_1k.append({'max_bits_K': 1, 'max_width': 64, 'ratio': 50})
tests_1k.append({'max_bits_K': 1, 'max_width': 128, 'ratio': 50})
tests_1k.append({'max_bits_K': 1, 'max_width': 2, 'ratio': 100})
tests_1k.append({'max_bits_K': 1, 'max_width': 4, 'ratio': 100})
tests_1k.append({'max_bits_K': 1, 'max_width': 8, 'ratio': 100})
tests_1k.append({'max_bits_K': 1, 'max_width': 16, 'ratio': 100})
tests_1k.append({'max_bits_K': 1, 'max_width': 32, 'ratio': 100})
tests_1k.append({'max_bits_K': 1, 'max_width': 64, 'ratio': 100})
tests_1k.append({'max_bits_K': 1, 'max_width': 128, 'ratio': 100})
```

```
tests_2k = []
tests_2k.append({'max_bits_K': 2, 'max_width': 2, 'ratio': 10})
tests_2k.append({'max_bits_K': 2, 'max_width': 4, 'ratio': 10})
tests_2k.append({'max_bits_K': 2, 'max_width': 8, 'ratio': 10})
tests_2k.append({'max_bits_K': 2, 'max_width': 16, 'ratio': 10})
tests_2k.append({'max_bits_K': 2, 'max_width': 32, 'ratio': 10})
tests_2k.append({'max_bits_K': 2, 'max_width': 64, 'ratio': 10})
tests_2k.append({'max_bits_K': 2, 'max_width': 128, 'ratio': 10})
tests_2k.append({'max_bits_K': 2, 'max_width': 2, 'ratio': 25})
tests_2k.append({'max_bits_K': 2, 'max_width': 4, 'ratio': 25})
tests_2k.append({'max_bits_K': 2, 'max_width': 8, 'ratio': 25})
tests_2k.append({'max_bits_K': 2, 'max_width': 16, 'ratio': 25})
tests_2k.append({'max_bits_K': 2, 'max_width': 32, 'ratio': 25})
tests_2k.append({'max_bits_K': 2, 'max_width': 64, 'ratio': 25})
tests_2k.append({'max_bits_K': 2, 'max_width': 128, 'ratio': 25})
tests_2k.append({'max_bits_K': 2, 'max_width': 2, 'ratio': 50})
tests_2k.append({'max_bits_K': 2, 'max_width': 4, 'ratio': 50})
tests_2k.append({'max_bits_K': 2, 'max_width': 8, 'ratio': 50})
tests_2k.append({'max_bits_K': 2, 'max_width': 16, 'ratio': 50})
tests_2k.append({'max_bits_K': 2, 'max_width': 32, 'ratio': 50})
tests_2k.append({'max_bits_K': 2, 'max_width': 64, 'ratio': 50})
tests_2k.append({'max_bits_K': 2, 'max_width': 128, 'ratio': 50})
tests_2k.append({'max_bits_K': 2, 'max_width': 2, 'ratio': 100})
tests_2k.append({'max_bits_K': 2, 'max_width': 4, 'ratio': 100})
tests_2k.append({'max_bits_K': 2, 'max_width': 8, 'ratio': 100})
tests_2k.append({'max_bits_K': 2, 'max_width': 16, 'ratio': 100})
tests_2k.append({'max_bits_K': 2, 'max_width': 32, 'ratio': 100})
tests_2k.append({'max_bits_K': 2, 'max_width': 64, 'ratio': 100})
tests_2k.append({'max_bits_K': 2, 'max_width': 128, 'ratio': 100})

tests_4k = []
tests_4k.append({'max_bits_K': 4, 'max_width': 2, 'ratio': 10})
tests_4k.append({'max_bits_K': 4, 'max_width': 4, 'ratio': 10})
tests_4k.append({'max_bits_K': 4, 'max_width': 8, 'ratio': 10})
tests_4k.append({'max_bits_K': 4, 'max_width': 16, 'ratio': 10})
tests_4k.append({'max_bits_K': 4, 'max_width': 32, 'ratio': 10})
tests_4k.append({'max_bits_K': 4, 'max_width': 64, 'ratio': 10})
tests_4k.append({'max_bits_K': 4, 'max_width': 128, 'ratio': 10})
tests_4k.append({'max_bits_K': 4, 'max_width': 2, 'ratio': 25})
tests_4k.append({'max_bits_K': 4, 'max_width': 4, 'ratio': 25})
tests_4k.append({'max_bits_K': 4, 'max_width': 8, 'ratio': 25})
```

```
tests_4k.append({'max_bits_K': 4, 'max_width': 16, 'ratio': 25})
tests_4k.append({'max_bits_K': 4, 'max_width': 32, 'ratio': 25})
tests_4k.append({'max_bits_K': 4, 'max_width': 64, 'ratio': 25})
tests_4k.append({'max_bits_K': 4, 'max_width': 128, 'ratio': 25})
tests_4k.append({'max_bits_K': 4, 'max_width': 2, 'ratio': 50})
tests_4k.append({'max_bits_K': 4, 'max_width': 4, 'ratio': 50})
tests_4k.append({'max_bits_K': 4, 'max_width': 8, 'ratio': 50})
tests_4k.append({'max_bits_K': 4, 'max_width': 16, 'ratio': 50})
tests_4k.append({'max_bits_K': 4, 'max_width': 32, 'ratio': 50})
tests_4k.append({'max_bits_K': 4, 'max_width': 64, 'ratio': 50})
tests_4k.append({'max_bits_K': 4, 'max_width': 128, 'ratio': 50})
tests_4k.append({'max_bits_K': 4, 'max_width': 2, 'ratio': 100})
tests_4k.append({'max_bits_K': 4, 'max_width': 4, 'ratio': 100})
tests_4k.append({'max_bits_K': 4, 'max_width': 8, 'ratio': 100})
tests_4k.append({'max_bits_K': 4, 'max_width': 16, 'ratio': 100})
tests_4k.append({'max_bits_K': 4, 'max_width': 32, 'ratio': 100})
tests_4k.append({'max_bits_K': 4, 'max_width': 64, 'ratio': 100})
tests_4k.append({'max_bits_K': 4, 'max_width': 128, 'ratio': 100})

tests_8k = []
tests_8k.append({'max_bits_K': 8, 'max_width': 2, 'ratio': 10})
tests_8k.append({'max_bits_K': 8, 'max_width': 4, 'ratio': 10})
tests_8k.append({'max_bits_K': 8, 'max_width': 8, 'ratio': 10})
tests_8k.append({'max_bits_K': 8, 'max_width': 16, 'ratio': 10})
tests_8k.append({'max_bits_K': 8, 'max_width': 32, 'ratio': 10})
tests_8k.append({'max_bits_K': 8, 'max_width': 64, 'ratio': 10})
tests_8k.append({'max_bits_K': 8, 'max_width': 128, 'ratio': 10})
tests_8k.append({'max_bits_K': 8, 'max_width': 2, 'ratio': 25})
tests_8k.append({'max_bits_K': 8, 'max_width': 4, 'ratio': 25})
tests_8k.append({'max_bits_K': 8, 'max_width': 8, 'ratio': 25})
tests_8k.append({'max_bits_K': 8, 'max_width': 16, 'ratio': 25})
tests_8k.append({'max_bits_K': 8, 'max_width': 32, 'ratio': 25})
tests_8k.append({'max_bits_K': 8, 'max_width': 64, 'ratio': 25})
tests_8k.append({'max_bits_K': 8, 'max_width': 128, 'ratio': 25})
tests_8k.append({'max_bits_K': 8, 'max_width': 2, 'ratio': 50})
tests_8k.append({'max_bits_K': 8, 'max_width': 4, 'ratio': 50})
tests_8k.append({'max_bits_K': 8, 'max_width': 8, 'ratio': 50})
tests_8k.append({'max_bits_K': 8, 'max_width': 16, 'ratio': 50})
tests_8k.append({'max_bits_K': 8, 'max_width': 32, 'ratio': 50})
tests_8k.append({'max_bits_K': 8, 'max_width': 64, 'ratio': 50})
tests_8k.append({'max_bits_K': 8, 'max_width': 128, 'ratio': 50})
tests_8k.append({'max_bits_K': 8, 'max_width': 2, 'ratio': 100})
```

```
tests_8k.append({'max_bits_K': 8, 'max_width': 4, 'ratio': 100})
tests_8k.append({'max_bits_K': 8, 'max_width': 8, 'ratio': 100})
tests_8k.append({'max_bits_K': 8, 'max_width': 16, 'ratio': 100})
tests_8k.append({'max_bits_K': 8, 'max_width': 32, 'ratio': 100})
tests_8k.append({'max_bits_K': 8, 'max_width': 64, 'ratio': 100})
tests_8k.append({'max_bits_K': 8, 'max_width': 128, 'ratio': 100})

tests_16k = []
tests_16k.append({'max_bits_K': 16, 'max_width': 2, 'ratio': 10})
tests_16k.append({'max_bits_K': 16, 'max_width': 4, 'ratio': 10})
tests_16k.append({'max_bits_K': 16, 'max_width': 8, 'ratio': 10})
tests_16k.append({'max_bits_K': 16, 'max_width': 16, 'ratio': 10})
tests_16k.append({'max_bits_K': 16, 'max_width': 32, 'ratio': 10})
tests_16k.append({'max_bits_K': 16, 'max_width': 64, 'ratio': 10})
tests_16k.append({'max_bits_K': 16, 'max_width': 128, 'ratio': 10})
tests_16k.append({'max_bits_K': 16, 'max_width': 2, 'ratio': 25})
tests_16k.append({'max_bits_K': 16, 'max_width': 4, 'ratio': 25})
tests_16k.append({'max_bits_K': 16, 'max_width': 8, 'ratio': 25})
tests_16k.append({'max_bits_K': 16, 'max_width': 16, 'ratio': 25})
tests_16k.append({'max_bits_K': 16, 'max_width': 32, 'ratio': 25})
tests_16k.append({'max_bits_K': 16, 'max_width': 64, 'ratio': 25})
tests_16k.append({'max_bits_K': 16, 'max_width': 128, 'ratio': 25})
tests_16k.append({'max_bits_K': 16, 'max_width': 2, 'ratio': 50})
tests_16k.append({'max_bits_K': 16, 'max_width': 4, 'ratio': 50})
tests_16k.append({'max_bits_K': 16, 'max_width': 8, 'ratio': 50})
tests_16k.append({'max_bits_K': 16, 'max_width': 16, 'ratio': 50})
tests_16k.append({'max_bits_K': 16, 'max_width': 32, 'ratio': 50})
tests_16k.append({'max_bits_K': 16, 'max_width': 64, 'ratio': 50})
tests_16k.append({'max_bits_K': 16, 'max_width': 128, 'ratio': 50})
tests_16k.append({'max_bits_K': 16, 'max_width': 2, 'ratio': 100})
tests_16k.append({'max_bits_K': 16, 'max_width': 4, 'ratio': 100})
tests_16k.append({'max_bits_K': 16, 'max_width': 8, 'ratio': 100})
tests_16k.append({'max_bits_K': 16, 'max_width': 16, 'ratio': 100})
tests_16k.append({'max_bits_K': 16, 'max_width': 32, 'ratio': 100})
tests_16k.append({'max_bits_K': 16, 'max_width': 64, 'ratio': 100})
tests_16k.append({'max_bits_K': 16, 'max_width': 128, 'ratio': 100})

tests_32k = []
tests_32k.append({'max_bits_K': 32, 'max_width': 2, 'ratio': 10})
tests_32k.append({'max_bits_K': 32, 'max_width': 4, 'ratio': 10})
tests_32k.append({'max_bits_K': 32, 'max_width': 8, 'ratio': 10})
tests_32k.append({'max_bits_K': 32, 'max_width': 16, 'ratio': 10})
```

```
tests_32k.append({'max_bits_K': 32, 'max_width': 32, 'ratio': 10})
tests_32k.append({'max_bits_K': 32, 'max_width': 64, 'ratio': 10})
tests_32k.append({'max_bits_K': 32, 'max_width': 128, 'ratio': 10})
tests_32k.append({'max_bits_K': 32, 'max_width': 2, 'ratio': 25})
tests_32k.append({'max_bits_K': 32, 'max_width': 4, 'ratio': 25})
tests_32k.append({'max_bits_K': 32, 'max_width': 8, 'ratio': 25})
tests_32k.append({'max_bits_K': 32, 'max_width': 16, 'ratio': 25})
tests_32k.append({'max_bits_K': 32, 'max_width': 32, 'ratio': 25})
tests_32k.append({'max_bits_K': 32, 'max_width': 64, 'ratio': 25})
tests_32k.append({'max_bits_K': 32, 'max_width': 128, 'ratio': 25})
tests_32k.append({'max_bits_K': 32, 'max_width': 2, 'ratio': 50})
tests_32k.append({'max_bits_K': 32, 'max_width': 4, 'ratio': 50})
tests_32k.append({'max_bits_K': 32, 'max_width': 8, 'ratio': 50})
tests_32k.append({'max_bits_K': 32, 'max_width': 16, 'ratio': 50})
tests_32k.append({'max_bits_K': 32, 'max_width': 32, 'ratio': 50})
tests_32k.append({'max_bits_K': 32, 'max_width': 64, 'ratio': 50})
tests_32k.append({'max_bits_K': 32, 'max_width': 128, 'ratio': 50})
tests_32k.append({'max_bits_K': 32, 'max_width': 2, 'ratio': 100})
tests_32k.append({'max_bits_K': 32, 'max_width': 4, 'ratio': 100})
tests_32k.append({'max_bits_K': 32, 'max_width': 8, 'ratio': 100})
tests_32k.append({'max_bits_K': 32, 'max_width': 16, 'ratio': 100})
tests_32k.append({'max_bits_K': 32, 'max_width': 32, 'ratio': 100})
tests_32k.append({'max_bits_K': 32, 'max_width': 64, 'ratio': 100})
tests_32k.append({'max_bits_K': 32, 'max_width': 128, 'ratio': 100})
tests_32k.append({'max_bits_K': 32, 'max_width': 2, 'ratio': 150})
tests_32k.append({'max_bits_K': 32, 'max_width': 4, 'ratio': 150})
tests_32k.append({'max_bits_K': 32, 'max_width': 8, 'ratio': 150})
tests_32k.append({'max_bits_K': 32, 'max_width': 16, 'ratio': 150})
tests_32k.append({'max_bits_K': 32, 'max_width': 32, 'ratio': 150})
tests_32k.append({'max_bits_K': 32, 'max_width': 64, 'ratio': 150})
tests_32k.append({'max_bits_K': 32, 'max_width': 128, 'ratio': 150})

tests_64k = []
tests_64k.append({'max_bits_K': 64, 'max_width': 2, 'ratio': 10})
tests_64k.append({'max_bits_K': 64, 'max_width': 4, 'ratio': 10})
tests_64k.append({'max_bits_K': 64, 'max_width': 8, 'ratio': 10})
tests_64k.append({'max_bits_K': 64, 'max_width': 16, 'ratio': 10})
tests_64k.append({'max_bits_K': 64, 'max_width': 32, 'ratio': 10})
tests_64k.append({'max_bits_K': 64, 'max_width': 64, 'ratio': 10})
tests_64k.append({'max_bits_K': 64, 'max_width': 128, 'ratio': 10})
tests_64k.append({'max_bits_K': 64, 'max_width': 2, 'ratio': 25})
tests_64k.append({'max_bits_K': 64, 'max_width': 4, 'ratio': 25})
```

```
tests_64k.append({'max_bits_K': 64, 'max_width': 8, 'ratio': 25})
tests_64k.append({'max_bits_K': 64, 'max_width': 16, 'ratio': 25})
tests_64k.append({'max_bits_K': 64, 'max_width': 32, 'ratio': 25})
tests_64k.append({'max_bits_K': 64, 'max_width': 64, 'ratio': 25})
tests_64k.append({'max_bits_K': 64, 'max_width': 128, 'ratio': 25})
tests_64k.append({'max_bits_K': 64, 'max_width': 2, 'ratio': 50})
tests_64k.append({'max_bits_K': 64, 'max_width': 4, 'ratio': 50})
tests_64k.append({'max_bits_K': 64, 'max_width': 8, 'ratio': 50})
tests_64k.append({'max_bits_K': 64, 'max_width': 16, 'ratio': 50})
tests_64k.append({'max_bits_K': 64, 'max_width': 32, 'ratio': 50})
tests_64k.append({'max_bits_K': 64, 'max_width': 64, 'ratio': 50})
tests_64k.append({'max_bits_K': 64, 'max_width': 128, 'ratio': 50})
tests_64k.append({'max_bits_K': 64, 'max_width': 2, 'ratio': 100})
tests_64k.append({'max_bits_K': 64, 'max_width': 4, 'ratio': 100})
tests_64k.append({'max_bits_K': 64, 'max_width': 8, 'ratio': 100})
tests_64k.append({'max_bits_K': 64, 'max_width': 16, 'ratio': 100})
tests_64k.append({'max_bits_K': 64, 'max_width': 32, 'ratio': 100})
tests_64k.append({'max_bits_K': 64, 'max_width': 64, 'ratio': 100})
tests_64k.append({'max_bits_K': 64, 'max_width': 128, 'ratio': 100})
tests_64k.append({'max_bits_K': 64, 'max_width': 2, 'ratio': 150})
tests_64k.append({'max_bits_K': 64, 'max_width': 4, 'ratio': 150})
tests_64k.append({'max_bits_K': 64, 'max_width': 8, 'ratio': 150})
tests_64k.append({'max_bits_K': 64, 'max_width': 16, 'ratio': 150})
tests_64k.append({'max_bits_K': 64, 'max_width': 32, 'ratio': 150})
tests_64k.append({'max_bits_K': 64, 'max_width': 64, 'ratio': 150})
tests_64k.append({'max_bits_K': 64, 'max_width': 128, 'ratio': 150})
tests_64k.append({'max_bits_K': 64, 'max_width': 2, 'ratio': 200})
tests_64k.append({'max_bits_K': 64, 'max_width': 4, 'ratio': 200})
tests_64k.append({'max_bits_K': 64, 'max_width': 8, 'ratio': 200})
tests_64k.append({'max_bits_K': 64, 'max_width': 16, 'ratio': 200})
tests_64k.append({'max_bits_K': 64, 'max_width': 32, 'ratio': 200})
tests_64k.append({'max_bits_K': 64, 'max_width': 64, 'ratio': 200})
tests_64k.append({'max_bits_K': 64, 'max_width': 128, 'ratio': 200})

tests_128k = []
tests_128k.append({'max_bits_K': 128, 'max_width': 2, 'ratio': 10})
tests_128k.append({'max_bits_K': 128, 'max_width': 4, 'ratio': 10})
tests_128k.append({'max_bits_K': 128, 'max_width': 8, 'ratio': 10})
tests_128k.append({'max_bits_K': 128, 'max_width': 16, 'ratio': 10})
tests_128k.append({'max_bits_K': 128, 'max_width': 32, 'ratio': 10})
tests_128k.append({'max_bits_K': 128, 'max_width': 64, 'ratio': 10})
tests_128k.append({'max_bits_K': 128, 'max_width': 128, 'ratio': 10})
```

```
tests_128k.append({'max_bits_K': 128, 'max_width': 2, 'ratio': 25})
tests_128k.append({'max_bits_K': 128, 'max_width': 4, 'ratio': 25})
tests_128k.append({'max_bits_K': 128, 'max_width': 8, 'ratio': 25})
tests_128k.append({'max_bits_K': 128, 'max_width': 16, 'ratio': 25})
tests_128k.append({'max_bits_K': 128, 'max_width': 32, 'ratio': 25})
tests_128k.append({'max_bits_K': 128, 'max_width': 64, 'ratio': 25})
tests_128k.append({'max_bits_K': 128, 'max_width': 128, 'ratio': 25})
tests_128k.append({'max_bits_K': 128, 'max_width': 2, 'ratio': 50})
tests_128k.append({'max_bits_K': 128, 'max_width': 4, 'ratio': 50})
tests_128k.append({'max_bits_K': 128, 'max_width': 8, 'ratio': 50})
tests_128k.append({'max_bits_K': 128, 'max_width': 16, 'ratio': 50})
tests_128k.append({'max_bits_K': 128, 'max_width': 32, 'ratio': 50})
tests_128k.append({'max_bits_K': 128, 'max_width': 64, 'ratio': 50})
tests_128k.append({'max_bits_K': 128, 'max_width': 128, 'ratio': 50})
tests_128k.append({'max_bits_K': 128, 'max_width': 2, 'ratio': 100})
tests_128k.append({'max_bits_K': 128, 'max_width': 4, 'ratio': 100})
tests_128k.append({'max_bits_K': 128, 'max_width': 8, 'ratio': 100})
tests_128k.append({'max_bits_K': 128, 'max_width': 16, 'ratio': 100})
tests_128k.append({'max_bits_K': 128, 'max_width': 32, 'ratio': 100})
tests_128k.append({'max_bits_K': 128, 'max_width': 64, 'ratio': 100})
tests_128k.append({'max_bits_K': 128, 'max_width': 128, 'ratio': 100})
tests_128k.append({'max_bits_K': 128, 'max_width': 2, 'ratio': 150})
tests_128k.append({'max_bits_K': 128, 'max_width': 4, 'ratio': 150})
tests_128k.append({'max_bits_K': 128, 'max_width': 8, 'ratio': 150})
tests_128k.append({'max_bits_K': 128, 'max_width': 16, 'ratio': 150})
tests_128k.append({'max_bits_K': 128, 'max_width': 32, 'ratio': 150})
tests_128k.append({'max_bits_K': 128, 'max_width': 64, 'ratio': 150})
tests_128k.append({'max_bits_K': 128, 'max_width': 128, 'ratio': 150})
tests_128k.append({'max_bits_K': 128, 'max_width': 2, 'ratio': 200})
tests_128k.append({'max_bits_K': 128, 'max_width': 4, 'ratio': 200})
tests_128k.append({'max_bits_K': 128, 'max_width': 8, 'ratio': 200})
tests_128k.append({'max_bits_K': 128, 'max_width': 16, 'ratio': 200})
tests_128k.append({'max_bits_K': 128, 'max_width': 32, 'ratio': 200})
tests_128k.append({'max_bits_K': 128, 'max_width': 64, 'ratio': 200})
tests_128k.append({'max_bits_K': 128, 'max_width': 128, 'ratio': 200})
tests_128k.append({'max_bits_K': 128, 'max_width': 2, 'ratio': 300})
tests_128k.append({'max_bits_K': 128, 'max_width': 4, 'ratio': 300})
tests_128k.append({'max_bits_K': 128, 'max_width': 8, 'ratio': 300})
tests_128k.append({'max_bits_K': 128, 'max_width': 16, 'ratio': 300})
tests_128k.append({'max_bits_K': 128, 'max_width': 32, 'ratio': 300})
tests_128k.append({'max_bits_K': 128, 'max_width': 64, 'ratio': 300})
tests_128k.append({'max_bits_K': 128, 'max_width': 128, 'ratio': 300})
```

```
all_tests.append(tests_1k)
all_tests.append(tests_2k)
all_tests.append(tests_4k)
all_tests.append(tests_8k)
all_tests.append(tests_16k)
all_tests.append(tests_32k)
all_tests.append(tests_64k)
all_tests.append(tests_128k)

all_final_prints = []

with open('explore_details', 'w+') as explore_details_fd:
    for use_lutram in [False, True]:
        for each_test_set in all_tests:

            lowest_area = 10e10
            lowest_area_test_case = None

            table_header_str = '{}{}{}{}{}{}'.format(
                'Use LUTRAM'.ljust(ljust_lutram_option),
                'BRAM Size'.ljust(ljust_size),
                'Max Width'.ljust(ljust_width),
                'LBs/BRAM'.ljust(ljust_ratio),
                'Geomean Area (MWTA)'.ljust(ljust_geomean),
                )
            print(table_header_str)
            explore_details_fd.write(table_header_str)
            explore_details_fd.write("\n")

            for each_test in each_test_set:
                lutram_option = '2'
                if use_lutram:
                    lutram_option = '3'

                run_mapper_cmd = './mapper ' + lutram_option + ' ' +
                ↪  str(each_test['max_bits_K']) + ' ' +
                ↪  str(each_test['max_width']) + ' ' +
                ↪  str(each_test['ratio']) + ' > temp'
```

```
        run_checker_cmd = './checker -l 1 1 -b ' +
        ↪  str(each_test['max_bits_K']*1024) + ' ' +
        ↪  str(each_test['max_width']) + ' ' +
        ↪  str(each_test['ratio']) + ' 1 logical_rams.txt
        ↪  logic_block_count.txt temp > temp2'

        os.system(run_mapper_cmd)
        os.system(run_checker_cmd)

        geomean_area = ''
        with open('temp2', 'r') as f:
            lines = f.read().splitlines()
            geomean_area = lines[-1].split()[-1]

        lutram_str = 'False'
        if use_lutram:
            lutram_str = 'True'
        size_str = str(each_test['max_bits_K']) + "K"
        table_entry_str = '{}{}{}{}{}'.format(
            lutram_str.ljust(ljust_lutram_option),
            size_str.ljust(ljust_size),
            str(each_test['max_width']).ljust(ljust_width),
            str(each_test['ratio']).ljust(ljust_ratio),
            geomean_area.ljust(ljust_geomean),
            )
        print(table_entry_str)
        explore_details_fd.write(table_entry_str)
        explore_details_fd.write("\n")

        if float(geomean_area) < lowest_area:
            lowest_area = float(geomean_area)
            lowest_area_test_case = each_test

    final_print = 'Best test case: LUTRAM: ' + lutram_str + ',
    ↪  Size:' + str(lowest_area_test_case['max_bits_K']) + "K" + ',
    ↪  Width: ' + str(lowest_area_test_case['max_width']) + ',
    ↪  Ratio: ' + str(lowest_area_test_case['ratio']) + ', With
    ↪  geomean area: ' + str(lowest_area)
    all_final_prints.append(final_print)

    print("")
    explore_details_fd.write("\n")
```

```
with open('explore_summary', 'w+') as explore_summary_fd:
    for each_final_print in all_final_prints:
        print(each_final_print)
        explore_summary_fd.write(each_final_print)
        explore_summary_fd.write("\n")
```

Appendix E: Dump file for exploring single type of physical ram

Use LUTRAM (MWTA)	BRAM Size	Max Width	LBs/BRAM	Geomean Area
False	1K	2	10	1.31441e+09
False	1K	4	10	8.90003e+08
False	1K	8	10	6.85732e+08
False	1K	16	10	6.08716e+08
False	1K	32	10	5.9361e+08
False	1K	64	10	6.28889e+08
False	1K	128	10	7.26289e+08
False	1K	2	25	3.06737e+09
False	1K	4	25	2.0616e+09
False	1K	8	25	1.54622e+09
False	1K	16	25	1.29432e+09
False	1K	32	25	1.18637e+09
False	1K	64	25	1.18025e+09
False	1K	128	25	1.25669e+09
False	1K	2	50	6.06834e+09
False	1K	4	50	4.03941e+09
False	1K	8	50	3.01883e+09
False	1K	16	50	2.48627e+09
False	1K	32	50	2.21989e+09
False	1K	64	50	2.15246e+09
False	1K	128	50	2.19854e+09
False	1K	2	100	1.20745e+10
False	1K	4	100	8.0324e+09
False	1K	8	100	5.99544e+09
False	1K	16	100	4.92196e+09
False	1K	32	100	4.33089e+09
False	1K	64	100	4.09875e+09
False	1K	128	100	4.10436e+09

Use LUTRAM (MWTA)	BRAM Size	Max Width	LBs/BRAM	Geomean Area
False	2K	2	10	1.13825e+09
False	2K	4	10	7.12522e+08
False	2K	8	10	5.03943e+08
False	2K	16	10	4.16096e+08
False	2K	32	10	3.93535e+08
False	2K	64	10	4.128e+08
False	2K	128	10	4.76126e+08

False	2K	2	25	2.60739e+09
False	2K	4	25	1.60287e+09
False	2K	8	25	1.09538e+09
False	2K	16	25	8.40843e+08
False	2K	32	25	7.41587e+08
False	2K	64	25	7.17498e+08
False	2K	128	25	7.52444e+08
False	2K	2	50	5.10847e+09
False	2K	4	50	3.10686e+09
False	2K	8	50	2.11038e+09
False	2K	16	50	1.60031e+09
False	2K	32	50	1.36386e+09
False	2K	64	50	1.26862e+09
False	2K	128	50	1.27626e+09
False	2K	2	100	1.01477e+10
False	2K	4	100	6.16692e+09
False	2K	8	100	4.14894e+09
False	2K	16	100	3.13534e+09
False	2K	32	100	2.62968e+09
False	2K	64	100	2.38458e+09
False	2K	128	100	2.34753e+09

Use LUTRAM	BRAM Size	Max Width	LBs/BRAM	Geomean Area (MWTA)
False	4K	2	10	1.05919e+09
False	4K	4	10	6.29249e+08
False	4K	8	10	4.21139e+08
False	4K	16	10	3.25702e+08
False	4K	32	10	2.93704e+08
False	4K	64	10	3.02766e+08
False	4K	128	10	3.46027e+08
False	4K	2	25	2.37217e+09
False	4K	4	25	1.38229e+09
False	4K	8	25	8.70927e+08
False	4K	16	25	6.15025e+08
False	4K	32	25	5.0609e+08
False	4K	64	25	4.77048e+08
False	4K	128	25	4.95668e+08
False	4K	2	50	4.60405e+09
False	4K	4	50	2.64435e+09
False	4K	8	50	1.6465e+09
False	4K	16	50	1.14535e+09

False	4K	32	50	9.0526e+08
False	4K	64	50	8.15538e+08
False	4K	128	50	8.01714e+08
False	4K	2	100	9.1175e+09
False	4K	4	100	5.20164e+09
False	4K	8	100	3.20443e+09
False	4K	16	100	2.21346e+09
False	4K	32	100	1.7288e+09
False	4K	64	100	1.51535e+09
False	4K	128	100	1.44979e+09

Use LUTRAM	BRAM Size	Max Width	LBs/BRAM	Geomean Area (MWTA)
False	8K	2	10	1.05909e+09
False	8K	4	10	6.04366e+08
False	8K	8	10	3.86555e+08
False	8K	16	10	2.83095e+08
False	8K	32	10	2.45787e+08
False	8K	64	10	2.46048e+08
False	8K	128	10	2.77243e+08
False	8K	2	25	2.2963e+09
False	8K	4	25	1.27783e+09
False	8K	8	25	7.65874e+08
False	8K	16	25	5.07546e+08
False	8K	32	25	3.9698e+08
False	8K	64	25	3.60464e+08
False	8K	128	25	3.69339e+08
False	8K	2	50	4.39388e+09
False	8K	4	50	2.40797e+09
False	8K	8	50	1.42432e+09
False	8K	16	50	9.18147e+08
False	8K	32	50	6.84626e+08
False	8K	64	50	5.90704e+08
False	8K	128	50	5.68452e+08
False	8K	2	100	8.6503e+09
False	8K	4	100	4.69301e+09
False	8K	8	100	2.73363e+09
False	8K	16	100	1.74381e+09
False	8K	32	100	1.28232e+09
False	8K	64	100	1.07543e+09
False	8K	128	100	1.0056e+09

Use LUTRAM (MWTA)	BRAM Size	Max Width	LBs/BRAM	Geomean Area
False	16K	2	10	1.14769e+09
False	16K	4	10	6.36531e+08
False	16K	8	10	3.9356e+08
False	16K	16	10	2.782e+08
False	16K	32	10	2.34955e+08
False	16K	64	10	2.3384e+08
False	16K	128	10	2.61593e+08
False	16K	2	25	2.35813e+09
False	16K	4	25	1.26492e+09
False	16K	8	25	7.26533e+08
False	16K	16	25	4.6271e+08
False	16K	32	25	3.49267e+08
False	16K	64	25	3.06826e+08
False	16K	128	25	3.07799e+08
False	16K	2	50	4.41934e+09
False	16K	4	50	2.33103e+09
False	16K	8	50	1.31908e+09
False	16K	16	50	8.13425e+08
False	16K	32	50	5.82136e+08
False	16K	64	50	4.83729e+08
False	16K	128	50	4.53902e+08
False	16K	2	100	8.60466e+09
False	16K	4	100	4.48013e+09
False	16K	8	100	2.49484e+09
False	16K	16	100	1.52117e+09
False	16K	32	100	1.05997e+09
False	16K	64	100	8.5521e+08
False	16K	128	100	7.78485e+08

Use LUTRAM (MWTA)	BRAM Size	Max Width	LBs/BRAM	Geomean Area
False	32K	2	10	1.34831e+09
False	32K	4	10	7.45632e+08
False	32K	8	10	4.5119e+08
False	32K	16	10	3.09418e+08
False	32K	32	10	2.57198e+08
False	32K	64	10	2.53859e+08
False	32K	128	10	2.80897e+08
False	32K	2	25	2.54137e+09
False	32K	4	25	1.34796e+09

False	32K	8	25	7.49939e+08
False	32K	16	25	4.60168e+08
False	32K	32	25	3.37398e+08
False	32K	64	25	2.88302e+08
False	32K	128	25	2.8405e+08
False	32K	2	50	4.58943e+09
False	32K	4	50	2.39313e+09
False	32K	8	50	1.30635e+09
False	32K	16	50	7.74894e+08
False	32K	32	50	5.36569e+08
False	32K	64	50	4.33343e+08
False	32K	128	50	3.99161e+08
False	32K	2	100	8.75332e+09
False	32K	4	100	4.50575e+09
False	32K	8	100	2.41674e+09
False	32K	16	100	1.41486e+09
False	32K	32	100	9.51707e+08
False	32K	64	100	7.44588e+08
False	32K	128	100	6.64585e+08
False	32K	2	150	1.29172e+10
False	32K	4	150	6.6478e+09
False	32K	8	150	3.5261e+09
False	32K	16	150	2.04862e+09
False	32K	32	150	1.37337e+09
False	32K	64	150	1.05681e+09
False	32K	128	150	9.31764e+08

Use LUTRAM	BRAM Size	Max Width	LBs/BRAM	Geomean Area (MWTA)
False	64K	2	10	1.75402e+09
False	64K	4	10	9.67906e+08
False	64K	8	10	5.82273e+08
False	64K	16	10	3.91716e+08
False	64K	32	10	3.2047e+08
False	64K	64	10	3.16372e+08
False	64K	128	10	3.44088e+08
False	64K	2	25	2.92509e+09
False	64K	4	25	1.54329e+09
False	64K	8	25	8.51573e+08
False	64K	16	25	5.13236e+08
False	64K	32	25	3.67193e+08
False	64K	64	25	3.07516e+08

False	64K	128	25	3.00032e+08
False	64K	2	50	4.96807e+09
False	64K	4	50	2.57731e+09
False	64K	8	50	1.39178e+09
False	64K	16	50	8.01555e+08
False	64K	32	50	5.38277e+08
False	64K	64	50	4.25492e+08
False	64K	128	50	3.87127e+08
False	64K	2	100	9.13196e+09
False	64K	4	100	4.677e+09
False	64K	8	100	2.48106e+09
False	64K	16	100	1.40344e+09
False	64K	32	100	9.16629e+08
False	64K	64	100	7.00097e+08
False	64K	128	100	6.17617e+08
False	64K	2	150	1.32959e+10
False	64K	4	150	6.80832e+09
False	64K	8	150	3.57177e+09
False	64K	16	150	2.00512e+09
False	64K	32	150	1.30156e+09
False	64K	64	150	9.75813e+08
False	64K	128	150	8.49817e+08
False	64K	2	200	1.74597e+10
False	64K	4	200	8.93964e+09
False	64K	8	200	4.6871e+09
False	64K	16	200	2.608e+09
False	64K	32	200	1.68551e+09
False	64K	64	200	1.25837e+09
False	64K	128	200	1.08215e+09

Use LUTRAM	BRAM Size	Max Width	LBs/BRAM	Geomean Area (MWTA)
False	128K	2	10	2.55665e+09
False	128K	4	10	1.40936e+09
False	128K	8	10	8.45136e+08
False	128K	16	10	5.64291e+08
False	128K	32	10	4.56165e+08
False	128K	64	10	4.45829e+08
False	128K	128	10	4.72604e+08
False	128K	2	25	3.68424e+09
False	128K	4	25	1.94307e+09
False	128K	8	25	1.06853e+09

```
False           128K            16              25              6.41679e+08
False           128K            32              25              4.50035e+08
False           128K            64              25              3.70331e+08
False           128K            128             25              3.57256e+08
False           128K            2               50              5.71715e+09
False           128K            4               50              2.9654e+09
False           128K            8               50              1.59306e+09
False           128K            16              50              9.10787e+08
False           128K            32              50              6.00531e+08
False           128K            64              50              4.64923e+08
False           128K            128             50              4.18027e+08
False           128K            2               100             9.88104e+09
False           128K            4               100             5.06042e+09
False           128K            8               100             2.67093e+09
False           128K            16              100             1.49669e+09
False           128K            32              100             9.5333e+08
False           128K            64              100             7.11578e+08
False           128K            128             100             6.21763e+08
False           128K            2               150             1.40449e+10
False           128K            4               150             7.19174e+09
False           128K            8               150             3.75418e+09
False           128K            16              150             2.08793e+09
False           128K            32              150             1.31654e+09
False           128K            64              150             9.63954e+08
False           128K            128             150             8.32368e+08
False           128K            2               200             1.82088e+10
False           128K            4               200             9.32306e+09
False           128K            8               200             4.86404e+09
False           128K            16              200             2.68133e+09
False           128K            32              200             1.68343e+09
False           128K            64              200             1.22766e+09
False           128K            128             200             1.04597e+09
False           128K            2               300             2.65366e+10
False           128K            4               300             1.35857e+10
False           128K            8               300             7.08665e+09
False           128K            16              300             3.86694e+09
False           128K            32              300             2.40724e+09
False           128K            64              300             1.75296e+09
False           128K            128             300             1.4713e+09

Use LUTRAM      BRAM Size       Max Width       LBs/BRAM        Geomean Area
↪  (MWTA)
```

```
True            1K              2               10              5.38579e+08
True            1K              4               10              4.83018e+08
True            1K              8               10              4.60345e+08
True            1K              16              10              4.62694e+08
True            1K              32              10              4.82354e+08
True            1K              64              10              5.24125e+08
True            1K              128             10              6.07707e+08
True            1K              2               25              9.6386e+08
True            1K              4               25              8.54016e+08
True            1K              8               25              7.88752e+08
True            1K              16              25              7.71039e+08
True            1K              32              25              7.77498e+08
True            1K              64              25              8.04137e+08
True            1K              128             25              8.62213e+08
True            1K              2               50              1.56375e+09
True            1K              4               50              1.38604e+09
True            1K              8               50              1.27643e+09
True            1K              16              50              1.21826e+09
True            1K              32              50              1.20422e+09
True            1K              64              50              1.22252e+09
True            1K              128             50              1.26711e+09
True            1K              2               100             2.59807e+09
True            1K              4               100             2.29829e+09
True            1K              8               100             2.11605e+09
True            1K              16              100             2.01477e+09
True            1K              32              100             1.94876e+09
True            1K              64              100             1.95331e+09
True            1K              128             100             1.98505e+09

Use LUTRAM      BRAM Size       Max Width       LBs/BRAM        Geomean Area
↪  (MWTA)
True            2K              2               10              4.16092e+08
True            2K              4               10              3.68636e+08
True            2K              8               10              3.44145e+08
True            2K              16              10              3.37596e+08
True            2K              32              10              3.48928e+08
True            2K              64              10              3.73339e+08
True            2K              128             10              4.28432e+08
True            2K              2               25              7.06044e+08
True            2K              4               25              6.10196e+08
True            2K              8               25              5.45513e+08
True            2K              16              25              5.16088e+08
```

True	2K	32	25	5.16357e+08
True	2K	64	25	5.31308e+08
True	2K	128	25	5.68826e+08
True	2K	2	50	1.1166e+09
True	2K	4	50	9.60533e+08
True	2K	8	50	8.54185e+08
True	2K	16	50	7.8945e+08
True	2K	32	50	7.74441e+08
True	2K	64	50	7.82611e+08
True	2K	128	50	8.1086e+08
True	2K	2	100	1.8202e+09
True	2K	4	100	1.56315e+09
True	2K	8	100	1.39e+09
True	2K	16	100	1.28258e+09
True	2K	32	100	1.23009e+09
True	2K	64	100	1.21508e+09
True	2K	128	100	1.23612e+09

Use LUTRAM	BRAM Size	Max Width	LBs/BRAM	Geomean Area (MWTA)
True	4K	2	10	3.38698e+08
True	4K	4	10	2.93209e+08
True	4K	8	10	2.67904e+08
True	4K	16	10	2.55638e+08
True	4K	32	10	2.61108e+08
True	4K	64	10	2.81775e+08
True	4K	128	10	3.24858e+08
True	4K	2	25	5.44594e+08
True	4K	4	25	4.63772e+08
True	4K	8	25	4.06424e+08
True	4K	16	25	3.74342e+08
True	4K	32	25	3.67134e+08
True	4K	64	25	3.76758e+08
True	4K	128	25	3.99348e+08
True	4K	2	50	8.35514e+08
True	4K	4	50	7.03225e+08
True	4K	8	50	6.09109e+08
True	4K	16	50	5.46459e+08
True	4K	32	50	5.21498e+08
True	4K	64	50	5.23116e+08
True	4K	128	50	5.40507e+08
True	4K	2	100	1.32788e+09

True	4K	4	100	1.11627e+09
True	4K	8	100	9.64254e+08
True	4K	16	100	8.63207e+08
True	4K	32	100	8.0339e+08
True	4K	64	100	7.89319e+08
True	4K	128	100	8.01116e+08

Use LUTRAM	BRAM Size	Max Width	LBs/BRAM	Geomean Area (MWTA)
True	8K	2	10	3.18014e+08
True	8K	4	10	2.59061e+08
True	8K	8	10	2.32004e+08
True	8K	16	10	2.16832e+08
True	8K	32	10	2.16381e+08
True	8K	64	10	2.31707e+08
True	8K	128	10	2.66267e+08
True	8K	2	25	4.76626e+08
True	8K	4	25	3.72393e+08
True	8K	8	25	3.20942e+08
True	8K	16	25	2.91598e+08
True	8K	32	25	2.81444e+08
True	8K	64	25	2.86658e+08
True	8K	128	25	3.03134e+08
True	8K	2	50	7.09935e+08
True	8K	4	50	5.42204e+08
True	8K	8	50	4.63544e+08
True	8K	16	50	4.09794e+08
True	8K	32	50	3.82894e+08
True	8K	64	50	3.7979e+08
True	8K	128	50	3.91929e+08
True	8K	2	100	1.11805e+09
True	8K	4	100	8.37248e+08
True	8K	8	100	7.06792e+08
True	8K	16	100	6.16383e+08
True	8K	32	100	5.61096e+08
True	8K	64	100	5.44335e+08
True	8K	128	100	5.49186e+08

Use LUTRAM	BRAM Size	Max Width	LBs/BRAM	Geomean Area (MWTA)
True	16K	2	10	3.39268e+08
True	16K	4	10	2.70202e+08

```
True            16K             8               10              2.36045e+08
True            16K             16              10              2.16975e+08
True            16K             32              10              2.14437e+08
True            16K             64              10              2.28265e+08
True            16K             128             10              2.57692e+08
True            16K             2               25              4.72567e+08
True            16K             4               25              3.41345e+08
True            16K             8               25              2.75711e+08
True            16K             16              25              2.45343e+08
True            16K             32              25              2.33315e+08
True            16K             64              25              2.32637e+08
True            16K             128             25              2.42826e+08
True            16K             2               50              6.84657e+08
True            16K             4               50              4.74483e+08
True            16K             8               50              3.72159e+08
True            16K             16              50              3.24195e+08
True            16K             32              50              3.00763e+08
True            16K             64              50              2.95669e+08
True            16K             128             50              3.01567e+08
True            16K             2               100             1.05472e+09
True            16K             4               100             7.11824e+08
True            16K             8               100             5.45358e+08
True            16K             16              100             4.70357e+08
True            16K             32              100             4.22073e+08
True            16K             64              100             4.03115e+08
True            16K             128             100             4.01553e+08

Use LUTRAM      BRAM Size       Max Width       LBs/BRAM        Geomean Area
↪  (MWTA)
True            32K             2               10              3.99863e+08
True            32K             4               10              3.17272e+08
True            32K             8               10              2.70417e+08
True            32K             16              10              2.47535e+08
True            32K             32              10              2.45435e+08
True            32K             64              10              2.57239e+08
True            32K             128             10              2.85629e+08
True            32K             2               25              5.01232e+08
True            32K             4               25              3.54478e+08
True            32K             8               25              2.76756e+08
True            32K             16              25              2.37542e+08
True            32K             32              25              2.23118e+08
True            32K             64              25              2.19984e+08
```

True	32K	128	25	2.29902e+08
True	32K	2	50	6.99765e+08
True	32K	4	50	4.69853e+08
True	32K	8	50	3.41424e+08
True	32K	16	50	2.78901e+08
True	32K	32	50	2.54106e+08
True	32K	64	50	2.45376e+08
True	32K	128	50	2.46737e+08
True	32K	2	100	1.04976e+09
True	32K	4	100	6.86507e+08
True	32K	8	100	4.78106e+08
True	32K	16	100	3.79325e+08
True	32K	32	100	3.37066e+08
True	32K	64	100	3.20223e+08
True	32K	128	100	3.17614e+08
True	32K	2	150	1.36385e+09
True	32K	4	150	8.81721e+08
True	32K	8	150	6.03267e+08
True	32K	16	150	4.70572e+08
True	32K	32	150	4.17001e+08
True	32K	64	150	3.86277e+08
True	32K	128	150	3.79844e+08

Use LUTRAM	BRAM Size	Max Width	LBs/BRAM	Geomean Area (MWTA)
True	64K	2	10	5.2313e+08
True	64K	4	10	4.12408e+08
True	64K	8	10	3.52249e+08
True	64K	16	10	3.21799e+08
True	64K	32	10	3.15643e+08
True	64K	64	10	3.25767e+08
True	64K	128	10	3.55664e+08
True	64K	2	25	5.77569e+08
True	64K	4	25	4.05898e+08
True	64K	8	25	3.14902e+08
True	64K	16	25	2.64396e+08
True	64K	32	25	2.4457e+08
True	64K	64	25	2.39716e+08
True	64K	128	25	2.49062e+08
True	64K	2	50	7.57994e+08
True	64K	4	50	4.99779e+08
True	64K	8	50	3.54894e+08

True	64K	16	50	2.78561e+08
True	64K	32	50	2.46801e+08
True	64K	64	50	2.36499e+08
True	64K	128	50	2.36324e+08
True	64K	2	100	1.09493e+09
True	64K	4	100	7.01424e+08
True	64K	8	100	4.73443e+08
True	64K	16	100	3.48399e+08
True	64K	32	100	2.91793e+08
True	64K	64	100	2.74192e+08
True	64K	128	100	2.70358e+08
True	64K	2	150	1.40408e+09
True	64K	4	150	8.83661e+08
True	64K	8	150	5.85757e+08
True	64K	16	150	4.17649e+08
True	64K	32	150	3.45328e+08
True	64K	64	150	3.15789e+08
True	64K	128	150	3.09548e+08
True	64K	2	200	1.69828e+09
True	64K	4	200	1.05444e+09
True	64K	8	200	6.93977e+08
True	64K	16	200	4.8985e+08
True	64K	32	200	3.99175e+08
True	64K	64	200	3.65936e+08
True	64K	128	200	3.54342e+08

Use LUTRAM	BRAM Size	Max Width	LBs/BRAM	Geomean Area (MWTA)
True	128K	2	10	7.67282e+08
True	128K	4	10	6.11207e+08
True	128K	8	10	5.19855e+08
True	128K	16	10	4.80791e+08
True	128K	32	10	4.64893e+08
True	128K	64	10	4.72402e+08
True	128K	128	10	5.0514e+08
True	128K	2	25	7.26822e+08
True	128K	4	25	5.13325e+08
True	128K	8	25	3.95748e+08
True	128K	16	25	3.30385e+08
True	128K	32	25	3.02924e+08
True	128K	64	25	2.97258e+08
True	128K	128	25	3.07551e+08

True	128K	2	50	8.71856e+08
True	128K	4	50	5.75778e+08
True	128K	8	50	4.06243e+08
True	128K	16	50	3.1671e+08
True	128K	32	50	2.73754e+08
True	128K	64	50	2.58349e+08
True	128K	128	50	2.57645e+08
True	128K	2	100	1.18406e+09
True	128K	4	100	7.59029e+08
True	128K	8	100	5.0343e+08
True	128K	16	100	3.61746e+08
True	128K	32	100	2.93151e+08
True	128K	64	100	2.66089e+08
True	128K	128	100	2.61761e+08
True	128K	2	150	1.4828e+09
True	128K	4	150	9.33459e+08
True	128K	8	150	6.09061e+08
True	128K	16	150	4.21824e+08
True	128K	32	150	3.28969e+08
True	128K	64	150	2.88808e+08
True	128K	128	150	2.82062e+08
True	128K	2	200	1.77061e+09
True	128K	4	200	1.09906e+09
True	128K	8	200	7.09466e+08
True	128K	16	200	4.85457e+08
True	128K	32	200	3.68087e+08
True	128K	64	200	3.22233e+08
True	128K	128	200	3.09237e+08
True	128K	2	300	2.32178e+09
True	128K	4	300	1.41535e+09
True	128K	8	300	8.95024e+08
True	128K	16	300	6.03192e+08
True	128K	32	300	4.48375e+08
True	128K	64	300	3.85422e+08
True	128K	128	300	3.67363e+08

Appendix F: .py file for exploring custom architecture

```
import os

ljust_lutram_percentage = 15
ljust_lb_per_lutram = 15
ljust_size = 12
ljust_width = 12
ljust_ratio = 12
ljust_geomean = 20

all_tests = []

tests_lutram_50 = []
# Section 1
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 16, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})  # Best
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 16, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})
# Stratix-IV
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 16, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300})

tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})
```

```
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300})

tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 32,
↪  'ratio_2': 150})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 32,
↪  'ratio_2': 150})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 64,
↪  'ratio_2': 150})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 64,
↪  'ratio_2': 150})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2':
↪  128, 'ratio_2': 150})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2':
↪  128, 'ratio_2': 150})

# Section 2 // Best from section 1 but vary BRAM1 & BRAM2 ratio
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 5, 'max_bits_K_2': 128, 'max_width_2': 64,
↪  'ratio_2': 100})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 5, 'max_bits_K_2': 128, 'max_width_2': 64,
↪  'ratio_2': 150})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 5, 'max_bits_K_2': 128, 'max_width_2': 64,
↪  'ratio_2': 200})
```

```
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 5, 'max_bits_K_2': 128, 'max_width_2': 64,
↪  'ratio_2': 250})

tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 100})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 150})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 200})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 250})

tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 100})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 150})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 200})
tests_lutram_50.append({'lb_per_lutram': 2, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 250})

tests_lutram_33 = []
# Section 1 // Repeat tests_lutram_50's section 1 with 33%
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300}) # Best
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 8,
↪  'max_width_1': 16, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})
```

```
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 8,
↪  'max_width_1': 16, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 8,
↪  'max_width_1': 16, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300})

tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300})

tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 32,
↪  'ratio_2': 150})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 32,
↪  'ratio_2': 150})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 64,
↪  'ratio_2': 150})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 64,
↪  'ratio_2': 150})
```

```
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2':
↪  128, 'ratio_2': 150})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2':
↪  128, 'ratio_2': 150})

# Section 2 // Best from section 1 but vary various
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 8,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})

tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 100})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 150}) # Best
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 200})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 250})

tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 100})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 150})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 200})
tests_lutram_33.append({'lb_per_lutram': 3, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 250})

tests_lutram_25 = []
# Section 1 // Repeat tests_lutram_50's section 1 with 25%
```

```
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300}) # Best
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 8,
↪  'max_width_1': 16, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 8,
↪  'max_width_1': 16, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 8,
↪  'max_width_1': 16, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300})

tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300})

tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 32,
↪  'ratio_2': 150})
```

```
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 32,
↪  'ratio_2': 150})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 64,
↪  'ratio_2': 150})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 64,
↪  'ratio_2': 150})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2':
↪  128, 'ratio_2': 150})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2':
↪  128, 'ratio_2': 150})

# Section 2 // Best from section 1 but vary various
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 8,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})

tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 100})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 150}) # Best
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 200})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 250})

tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 100})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 150})
```

```
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 200})
tests_lutram_25.append({'lb_per_lutram': 4, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 250})

tests_lutram_10 = []
# Section 1 // Repeat tests_lutram_50's section 1 with 10%
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 16, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 16, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300}) # Best
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 16, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300})

tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  32, 'ratio_2': 300})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 300})
```

```
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 300})

tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 32,
↪  'ratio_2': 150})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 32,
↪  'ratio_2': 150})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 64,
↪  'ratio_2': 150})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2': 64,
↪  'ratio_2': 150})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 32,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2':
↪  128, 'ratio_2': 150})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 32,
↪  'max_width_1': 64, 'ratio_1': 10, 'max_bits_K_2': 64, 'max_width_2':
↪  128, 'ratio_2': 150})

# Section 2 // Best from section 1 but vary various
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 100})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 150}) # Best
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 200})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 250})
```

```
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 100})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 150})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 200})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 10, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 250})

tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 100})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 150})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 200})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  128, 'ratio_2': 250})

tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 100})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 150})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 200})
tests_lutram_10.append({'lb_per_lutram': 10, 'max_bits_K_1': 8,
↪  'max_width_1': 32, 'ratio_1': 25, 'max_bits_K_2': 128, 'max_width_2':
↪  64, 'ratio_2': 250})

all_tests.append(tests_lutram_50)
all_tests.append(tests_lutram_33)
all_tests.append(tests_lutram_25)
```

```
all_tests.append(tests_lutram_10)

all_final_prints = []

with open('explore_details_custom', 'w+') as explore_details_fd:
    for each_test_set in all_tests:

        lowest_area = 10e10
        lowest_area_test_case = None

        table_header_str = '{}{}{}{}{}{}{}{}{}{}'.format(
            'LUTRAM %'.ljust(ljust_lutram_percentage),
            'LBs / LUTRAM'.ljust(ljust_lb_per_lutram),
            'BRAM_Size_1'.ljust(ljust_size),
            'Max Width_1'.ljust(ljust_width),
            'LBs/BRAM_1'.ljust(ljust_ratio),
            'BRAM_Size_2'.ljust(ljust_size),
            'Max_Width_2'.ljust(ljust_width),
            'LBs/BRAM_2'.ljust(ljust_ratio),
            'Geomean Area (MWTA)'.ljust(ljust_geomean),
            )
        print(table_header_str)
        explore_details_fd.write(table_header_str)
        explore_details_fd.write("\n")

        for each_test in each_test_set:

            run_mapper_cmd = './mapper 4 ' + str(each_test['lb_per_lutram'])
            ↪  + ' ' + str(each_test['max_bits_K_1']) + ' ' +
            ↪  str(each_test['max_width_1']) + ' ' +
            ↪  str(each_test['ratio_1']) + ' ' +
            ↪  str(each_test['max_bits_K_2']) + ' ' +
            ↪  str(each_test['max_width_2']) + ' ' +
            ↪  str(each_test['ratio_2']) + ' > temp'
```

```
run_checker_cmd = './checker -t -l ' +
↪  str(each_test['lb_per_lutram']-1) + ' 1 -b ' +
↪  str(each_test['max_bits_K_1']*1024) + ' ' +
↪  str(each_test['max_width_1']) + ' ' +
↪  str(each_test['ratio_1']) + ' 1 -b ' +
↪  str(each_test['max_bits_K_2']*1024) + ' ' +
↪  str(each_test['max_width_2']) + ' ' +
↪  str(each_test['ratio_2']) + ' 1 logical_rams.txt
↪  logic_block_count.txt temp > temp2'

os.system(run_mapper_cmd)
os.system(run_checker_cmd)

geomean_area = ''
with open('temp2', 'r') as f:
    lines = f.read().splitlines()
    geomean_area = lines[-1].split()[-1]

lutram_percentage = 1.0/float(each_test['lb_per_lutram'])*100
size_str_1 = str(each_test['max_bits_K_1']) + "K"
size_str_2 = str(each_test['max_bits_K_2']) + "K"
table_entry_str = '{}{}{}{}{}{}{}{}{}{}'.format(
    str(lutram_percentage).ljust(ljust_lutram_percentage),
    str(each_test['lb_per_lutram']).ljust(ljust_lb_per_lutram),
    size_str_1.ljust(ljust_size),
    str(each_test['max_width_1']).ljust(ljust_width),
    str(each_test['ratio_1']).ljust(ljust_ratio),
    size_str_2.ljust(ljust_size),
    str(each_test['max_width_2']).ljust(ljust_width),
    str(each_test['ratio_2']).ljust(ljust_ratio),
    geomean_area.ljust(ljust_geomean),
    )
print(table_entry_str)
explore_details_fd.write(table_entry_str)
explore_details_fd.write("\n")

if float(geomean_area) < lowest_area:
    lowest_area = float(geomean_area)
    lowest_area_test_case = each_test
```

```
        final_print = 'Best test case: LUTRAM: ' +
        ↪  str(1.0/float(lowest_area_test_case['lb_per_lutram'])*100) + '%,
        ↪  LBs/LUTRAM: ' + str(lowest_area_test_case['lb_per_lutram']) + ',
        ↪  Size_1: ' + str(lowest_area_test_case['max_bits_K_1']) + "K" +
        ↪  ', Width_1: ' + str(lowest_area_test_case['max_width_1']) + ',
        ↪  Ratio_1: ' + str(lowest_area_test_case['ratio_1']) + ', Size_2:
        ↪  ' + str(lowest_area_test_case['max_bits_K_2']) + "K" + ',
        ↪  Width_2: ' + str(lowest_area_test_case['max_width_2']) + ',
        ↪  Ratio_2: ' + str(lowest_area_test_case['ratio_2']) + ', With
        ↪  geomean area: ' + str(lowest_area)
        all_final_prints.append(final_print)

        print("")
        explore_details_fd.write("\n")


with open('explore_summary_custom', 'w+') as explore_summary_fd:
    for each_final_print in all_final_prints:
        print(each_final_print)
        explore_summary_fd.write(each_final_print)
        explore_summary_fd.write("\n")
```

Appendix G: Dump file for exploring custom architecture

```
LUTRAM %       LBs / LUTRAM   BRAM_Size_1 Max Width_1 LBs/BRAM_1
↪  BRAM_Size_2 Max_Width_2 LBs/BRAM_2  Geomean Area (MWTA)
50.0           2              8K          32          10          128K
↪  32          300         2.18741e+08
50.0           2              8K          16          10          128K
↪  32          300         2.19912e+08
50.0           2              8K          32          10          128K
↪  64          300         2.18734e+08
50.0           2              8K          16          10          128K
↪  64          300         2.19588e+08
50.0           2              8K          32          10          128K
↪  128         300         2.1939e+08
50.0           2              8K          16          10          128K
↪  128         300         2.19732e+08
50.0           2              32K         32          10          128K
↪  32          300         2.54814e+08
50.0           2              32K         64          10          128K
↪  32          300         2.66505e+08
50.0           2              32K         32          10          128K
↪  64          300         2.54962e+08
50.0           2              32K         64          10          128K
↪  64          300         2.66893e+08
50.0           2              32K         32          10          128K
↪  128         300         2.55678e+08
50.0           2              32K         64          10          128K
↪  128         300         2.67737e+08
50.0           2              32K         32          10          64K
↪  32          150         2.54756e+08
50.0           2              32K         64          10          64K
↪  32          150         2.66612e+08
50.0           2              32K         32          10          64K
↪  64          150         2.55634e+08
50.0           2              32K         64          10          64K
↪  64          150         2.67787e+08
50.0           2              32K         32          10          64K
↪  128         150         2.57291e+08
50.0           2              32K         64          10          64K
↪  128         150         2.70011e+08
50.0           2              8K          32          5           128K
↪  64          100         2.54495e+08
```

```
50.0            2               8K          32          5           128K
↪  64           150         2.45325e+08
50.0            2               8K          32          5           128K
↪  64           200         2.40896e+08
50.0            2               8K          32          5           128K
↪  64           250         2.37849e+08
50.0            2               8K          32          10          128K
↪  64           100         2.23858e+08
50.0            2               8K          32          10          128K
↪  64           150         2.18863e+08
50.0            2               8K          32          10          128K
↪  64           200         2.22932e+08
50.0            2               8K          32          10          128K
↪  64           250         2.20142e+08
50.0            2               8K          32          25          128K
↪  64           100         2.21697e+08
50.0            2               8K          32          25          128K
↪  64           150         2.20102e+08
50.0            2               8K          32          25          128K
↪  64           200         2.26247e+08
50.0            2               8K          32          25          128K
↪  64           250         2.34834e+08

LUTRAM %        LBs / LUTRAM    BRAM_Size_1 Max Width_1 LBs/BRAM_1
↪  BRAM_Size_2 Max_Width_2 LBs/BRAM_2  Geomean Area (MWTA)
33.3333333333   3               8K          32          10          128K
↪  32           300         2.13968e+08
33.3333333333   3               8K          16          10          128K
↪  32           300         2.14458e+08
33.3333333333   3               8K          32          10          128K
↪  64           300         2.14278e+08
33.3333333333   3               8K          16          10          128K
↪  64           300         2.14095e+08
33.3333333333   3               8K          32          10          128K
↪  128          300         2.14565e+08
33.3333333333   3               8K          16          10          128K
↪  128          300         2.14097e+08
33.3333333333   3               32K         32          10          128K
↪  32           300         2.48503e+08
33.3333333333   3               32K         64          10          128K
↪  32           300         2.60496e+08
```

```
33.3333333333  3              32K          32           10           128K
↪  64           300          2.4855e+08
33.3333333333  3              32K          64           10           128K
↪  64           300          2.60941e+08
33.3333333333  3              32K          32           10           128K
↪  128          300          2.49365e+08
33.3333333333  3              32K          64           10           128K
↪  128          300          2.61662e+08
33.3333333333  3              32K          32           10           64K
↪  32           150          2.48587e+08
33.3333333333  3              32K          64           10           64K
↪  32           150          2.60751e+08
33.3333333333  3              32K          32           10           64K
↪  64           150          2.49463e+08
33.3333333333  3              32K          64           10           64K
↪  64           150          2.61824e+08
33.3333333333  3              32K          32           10           64K
↪  128          150          2.51128e+08
33.3333333333  3              32K          64           10           64K
↪  128          150          2.63938e+08
33.3333333333  3              8K           64           10           128K
↪  32           300          2.27765e+08
33.3333333333  3              8K           32           10           128K
↪  32           100          2.17633e+08
33.3333333333  3              8K           32           10           128K
↪  32           150          2.12881e+08
33.3333333333  3              8K           32           10           128K
↪  32           200          2.16022e+08
33.3333333333  3              8K           32           10           128K
↪  32           250          2.14434e+08
33.3333333333  3              8K           32           25           128K
↪  32           100          2.17014e+08
33.3333333333  3              8K           32           25           128K
↪  32           150          2.19796e+08
33.3333333333  3              8K           32           25           128K
↪  32           200          2.28021e+08
33.3333333333  3              8K           32           25           128K
↪  32           250          2.33861e+08

LUTRAM %       LBs / LUTRAM   BRAM_Size_1 Max Width_1 LBs/BRAM_1
↪  BRAM_Size_2 Max_Width_2 LBs/BRAM_2  Geomean Area (MWTA)
```

```
25.0            4               8K          32          10          128K
↪  32           300         2.11162e+08
25.0            4               8K          16          10          128K
↪  32           300         2.12102e+08
25.0            4               8K          32          10          128K
↪  64           300         2.12309e+08
25.0            4               8K          16          10          128K
↪  64           300         2.11688e+08
25.0            4               8K          32          10          128K
↪  128          300         2.12587e+08
25.0            4               8K          16          10          128K
↪  128          300         2.11863e+08
25.0            4               32K         32          10          128K
↪  32           300         2.4465e+08
25.0            4               32K         64          10          128K
↪  32           300         2.56769e+08
25.0            4               32K         32          10          128K
↪  64           300         2.44744e+08
25.0            4               32K         64          10          128K
↪  64           300         2.57162e+08
25.0            4               32K         32          10          128K
↪  128          300         2.45481e+08
25.0            4               32K         64          10          128K
↪  128          300         2.57949e+08
25.0            4               32K         32          10          64K
↪  32           150         2.4494e+08
25.0            4               32K         64          10          64K
↪  32           150         2.57201e+08
25.0            4               32K         32          10          64K
↪  64           150         2.45626e+08
25.0            4               32K         64          10          64K
↪  64           150         2.58224e+08
25.0            4               32K         32          10          64K
↪  128          150         2.47323e+08
25.0            4               32K         64          10          64K
↪  128          150         2.60249e+08
25.0            4               8K          64          10          128K
↪  32           300         2.25944e+08
25.0            4               8K          32          10          128K
↪  32           100         2.14874e+08
25.0            4               8K          32          10          128K
↪  32           150         2.10536e+08
```

```
25.0            4               8K          32          10          128K
↪  32           200         2.13897e+08
25.0            4               8K          32          10          128K
↪  32           250         2.11694e+08
25.0            4               8K          32          25          128K
↪  32           100         2.14792e+08
25.0            4               8K          32          25          128K
↪  32           150         2.18354e+08
25.0            4               8K          32          25          128K
↪  32           200         2.25973e+08
25.0            4               8K          32          25          128K
↪  32           250         2.32483e+08

LUTRAM %        LBs / LUTRAM    BRAM_Size_1 Max Width_1 LBs/BRAM_1
↪  BRAM_Size_2 Max_Width_2 LBs/BRAM_2  Geomean Area (MWTA)
10.0            10              8K          32          10          128K
↪  32           300         2.13141e+08
10.0            10              8K          16          10          128K
↪  32           300         2.16445e+08
10.0            10              8K          32          10          128K
↪  64           300         2.1268e+08
10.0            10              8K          16          10          128K
↪  64           300         2.16609e+08
10.0            10              8K          32          10          128K
↪  128          300         2.12446e+08
10.0            10              8K          16          10          128K
↪  128          300         2.16415e+08
10.0            10              32K         32          10          128K
↪  32           300         2.4117e+08
10.0            10              32K         64          10          128K
↪  32           300         2.50597e+08
10.0            10              32K         32          10          128K
↪  64           300         2.40698e+08
10.0            10              32K         64          10          128K
↪  64           300         2.50981e+08
10.0            10              32K         32          10          128K
↪  128          300         2.41227e+08
10.0            10              32K         64          10          128K
↪  128          300         2.51681e+08
10.0            10              32K         32          10          64K
↪  32           150         2.40557e+08
```

```
10.0            10              32K             64              10              64K
↪  32           150             2.50692e+08
10.0            10              32K             32              10              64K
↪  64           150             2.41319e+08
10.0            10              32K             64              10              64K
↪  64           150             2.51764e+08
10.0            10              32K             32              10              64K
↪  128          150             2.4306e+08
10.0            10              32K             64              10              64K
↪  128          150             2.53685e+08
10.0            10              8K              32              10              128K
↪  128          100             2.13841e+08
10.0            10              8K              32              10              128K
↪  128          150             2.09968e+08
10.0            10              8K              32              10              128K
↪  128          200             2.15983e+08
10.0            10              8K              32              10              128K
↪  128          250             2.1353e+08
10.0            10              8K              32              10              128K
↪  64           100             2.11962e+08
10.0            10              8K              32              10              128K
↪  64           150             2.09068e+08
10.0            10              8K              32              10              128K
↪  64           200             2.15926e+08
10.0            10              8K              32              10              128K
↪  64           250             2.12745e+08
10.0            10              8K              32              25              128K
↪  128          100             2.18785e+08
10.0            10              8K              32              25              128K
↪  128          150             2.18654e+08
10.0            10              8K              32              25              128K
↪  128          200             2.24554e+08
10.0            10              8K              32              25              128K
↪  128          250             2.32806e+08
10.0            10              8K              32              25              128K
↪  64           100             2.18222e+08
10.0            10              8K              32              25              128K
↪  64           150             2.17539e+08
10.0            10              8K              32              25              128K
↪  64           200             2.2352e+08
10.0            10              8K              32              25              128K
↪  64           250             2.32188e+08
```

Appendix H: .py file for comparing architectures

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np

file_1_str = "checker_mapper_1"
file_2_str = "checker_custom_arch_best"

file_1_data = []
file_2_data = []

plot_data = []

with open(file_1_str, 'r') as current_fd:
    data_section = False
    for line in current_fd.readlines():
        segments = line.strip().split()

        if (len(segments) > 0):
            if (segments[0] == "Circuit"):
                data_section = True
                continue
            if (segments[0] == "Geometric"):
                data_section = False
                continue
        if (not data_section):
            continue

        file_1_data.append(float(segments[6]))

with open(file_2_str, 'r') as current_fd:
    data_section = False
    for line in current_fd.readlines():
        segments = line.strip().split()

        if (len(segments) > 0):
            if (segments[0] == "Circuit"):
                data_section = True
                continue
            if (segments[0] == "Geometric"):
```

```
                data_section = False
                continue
        if (not data_section):
            continue

        file_2_data.append(float(segments[6]))

plot_data = np.subtract(file_2_data, file_1_data)

X = np.arange(69)
fig = plt.figure(dpi=100, figsize=(14, 7))
ax = fig.add_axes([0.1,0.1,0.8,0.8])
ax.bar(X, plot_data, width = 0.5, label="Data")
plt.legend(loc="upper right")
plt.title('Area difference: ' + file_2_str + " - " + file_1_str)
plt.xlabel('Circuit No.')
plt.ylabel('Area')
plt.xticks(np.arange(0, 70, 5))
plt.tick_params(axis='both', which='major', labelsize=10)
plt.show()
plt.savefig("area_difference.png")
```

Appendix I: .cpp files for mapping tool

```
#ifndef ARCH_H_
#define ARCH_H_

#include <vector>

#include "PhysicalRam.h"

using namespace std;

class Architecture {
private:
        unsigned int k;
        unsigned int N;
        double logic_block_area;
        vector<PhysicalRam> physical_rams;

public:
        Architecture(unsigned int k_n, unsigned int N_n, double
        ↪  logic_block_area_n) : k(k_n), N(N_n),
        ↪  logic_block_area(logic_block_area_n) {}

        // Get functions
        unsigned int getK() { return k; }
        unsigned int getN() { return N; }
        double getLogicBlockArea() { return logic_block_area; }
        vector<PhysicalRam>& getPhysicalRams() { return physical_rams; }

        // Set functions
        void addPhysicalRam(PhysicalRam physical_ram_n);

        // Print functions
        void printInfo();
};


#endif /* ARCH_H_ */
```

```cpp
#include "Arch.h"
#include "helper.h"

using namespace std;

// Set functions
void Architecture::addPhysicalRam(PhysicalRam physical_ram_n) {
        physical_rams.push_back(physical_ram_n);
}

// Print functions
void Architecture::printInfo() {
        for (auto i: physical_rams) {
                cout << "Max bits: " << i.getMaxBits();
                cout << ", Max depth: " << i.getMaxDepth();
                cout << ", Max width: " << i.getMaxWidth();
                cout << ", Type: " << getPRAMType_S(i.getType());

                map<unsigned int, unsigned int>& z = i.getWidthToDepths();
                cout << ", Possible widths::depths: ";
                for (auto j: i.getPossibleWidths()) {
                        cout << j << "::" << z[j] << ", ";
                }
                cout << endl;
        }
}
```

```
#ifndef CIRCUIT_H_
#define CIRCUIT_H_

#include <vector>

#include "LogicalRam.h"
#include "MappedRam.h"

using namespace std;

class Circuit {
private:
        unsigned int id;
        unsigned int logic_block_num;

        vector<LogicalRam> logical_rams;
        vector<MappedRam> mapped_rams;

        double area;
        map<P_RamType, unsigned int> physical_ram_counts;

        // Optimization vars
        // Map of width to vector of logical ram ids with that width in
        ↪  descending order (largest depth first)
        map<unsigned int, vector<pair<unsigned int, unsigned int>>>
        ↪  width_logical_ram_map;

public:
        Circuit(unsigned int id_n, unsigned int logic_block_num_n) :
        ↪  id(id_n), logic_block_num(logic_block_num_n) {}

        // Get functions
        unsigned int getID() { return id; }
        unsigned int getLogicBlockNum() { return logic_block_num; }
        vector<LogicalRam>& getLogicalRams() { return logical_rams; }
        vector<LogicalRam> getLogicalRams_Copy() { return logical_rams; }
        vector<MappedRam>& getMappedRams() { return mapped_rams; }
        double getArea() { return area; }
        map<P_RamType, unsigned int>& getPhysicalRamCounts() { return
        ↪  physical_ram_counts; }
        map<unsigned int, vector<pair<unsigned int, unsigned int>>>&
        ↪  getWidthLogicalRamMap() { return width_logical_ram_map; }
```

```
        // Set functions
        void addLogicalRam(unsigned int logical_ram_id_n, unsigned int
        ↪  depth_n, unsigned int width_n, L_RamMode mode_n);
        void addMappedRam(MappedRam mapped_ram_n);
        void setArea(double area_n) { area = area_n; }

        // Computation functions
        void computeArea(vector<PhysicalRam>& physical_rams, unsigned int
        ↪  arch_N, double arch_LB_area);

        // Print functions
        void printInfo_LogicalRam();
        void printInfo_MappedRam();
        void printInfo_WidthLogicalRamMap();
        void printInfo_Area();
};


#endif /* CIRCUIT_H_ */
```

```cpp
#include <algorithm>
#include <cmath>

#include "Circuit.h"
#include "LogicalRam.h"
#include "MappedRam.h"
#include "helper.h"

// Set functions
void Circuit::addLogicalRam(unsigned int logical_ram_id_n, unsigned int
↪  depth_n, unsigned int width_n, L_RamMode mode_n) {
        LogicalRam new_logical_ram(logical_ram_id_n, depth_n, width_n,
        ↪  mode_n);
        logical_rams.push_back(new_logical_ram);
}

void Circuit::addMappedRam(MappedRam mapped_ram_n) {
        mapped_rams.push_back(mapped_ram_n);
}

// Computation functions

void Circuit::computeArea(vector<PhysicalRam>& physical_rams, unsigned int
↪  arch_N, double arch_LB_area) {
        map<P_RamType, unsigned int> physical_ram_to_lb_count_map;
        map<P_RamType, unsigned int> physical_ram_to_ratio_map;
        for (auto i: physical_rams) {
                physical_ram_to_lb_count_map[i.getType()] = 0;
                physical_ram_to_ratio_map[i.getType()] = i.getRatio();

                physical_ram_counts[i.getType()] = 0;
        }

        unsigned int total_add_luts = 0;
        for (auto i: mapped_rams) {
                physical_ram_to_lb_count_map[i.getType()] +=
                ↪  physical_ram_to_ratio_map[i.getType()];
                total_add_luts += i.getAddLuts();

                physical_ram_counts[i.getType()] += 1;
        }
```

```
// Now we find out which of the three is the limiting value
//P_RamType limiting_lb_PRAM_type;
unsigned int largest_lb_count = 0;
for (auto i: physical_rams) {
        if (physical_ram_to_lb_count_map[i.getType()] >
        ↪  largest_lb_count) {
                //limiting_lb_PRAM_type = i.getType();
                largest_lb_count =
                ↪  physical_ram_to_lb_count_map[i.getType()];
        }
}

// One more check against the total logic blocks needed for circuit
↪  and additional LUTs
unsigned int total_circuit_lb = logic_block_num + (unsigned
↪  int)ceil((double)total_add_luts / (double)arch_N);

// Blanket check for all non-LUTRAM memories, ensuring that we have
↪  at least enough LB for all the logic needed
if (total_circuit_lb > largest_lb_count) {
        largest_lb_count = total_circuit_lb;
}
for (auto i: physical_rams) {
        if (i.getType() == LUTRAM) {
                // If LUTRAM, we need to check how much of the LB
                ↪  count we computed is still free for logic
                // If the amount for logic is not enough, we need
                ↪  to find the new total LB count
                double used_by_lutram =
                ↪  ceil((double)physical_ram_to_lb_count_map[i.getType()]
                ↪  * (1/i.getRatio()));
                double free_for_logic = (double)largest_lb_count -
                ↪  used_by_lutram;

                if (total_circuit_lb > free_for_logic) {
                        // Remaining LB count not enough for logic
                        // Increment largest_lb_count (total
                        ↪  needed)
                        largest_lb_count = (unsigned
                        ↪  int)used_by_lutram + (unsigned
                        ↪  int)total_circuit_lb;
                }
```

```
                        break;
                }
        }

        // With the largest type and count known, we find total area
        // LB area calculation, includes LUTRAM
        area = (double)largest_lb_count * arch_LB_area;

        for (auto i: physical_rams) {
                if (i.getType() == LUTRAM) {
                        // Add on the additional 5000 area per logic block
                        ↪  that can do LUTRAM
                        area += ceil((double)largest_lb_count *
                        ↪  (1/i.getRatio())) * (i.getArea() -
                        ↪  arch_LB_area);
                        continue;
                }
                area += ceil((double)largest_lb_count/i.getRatio()) *
                ↪  i.getArea();
        }
}

// Print functions
void Circuit::printInfo_LogicalRam() {
        for (auto i: logical_rams) {
                cout << "Circuit id: " << id;
                cout << ", Logic block count: " << logic_block_num;
                cout << " :: ";
                i.printInfo();
        }
}

void Circuit::printInfo_MappedRam() {
        for (auto i: mapped_rams) {
                cout << "Circuit id: " << id;
                cout << ", Logic block count: " << logic_block_num;
                cout << " :: ";
                i.printInfo();
        }
}

void Circuit::printInfo_WidthLogicalRamMap() {
```

```
        for (auto it = width_logical_ram_map.cbegin(); it !=
        ↪  width_logical_ram_map.cend(); ++it) {
                for (auto i: it->second) {
                        cout << "Circuit id: " << id;
                        cout << ", Width: " << it->first;
                        cout << ", Logical Ram ID: " << i.first;
                        cout << ", Depth: " << i.second;
                        cout << endl;
                }
                cout << "===End of Width: " << it->first << "===" << endl;
        }
}

void Circuit::printInfo_Area() {
        cout << "Circuit id: " << id;
        cout << ", Area: " << area;
        for (auto it = physical_ram_counts.cbegin(); it !=
        ↪  physical_ram_counts.cend(); ++it) {
                cout << ", Type: " << getPRAMType_S(it->first);
                cout << ", Count: " << it->second;
        }
        cout << endl;
}
```

```
#ifndef HELPER_H_
#define HELPER_H_

#include <string>
#include <vector>

#include "LogicalRam.h"
#include "PhysicalRam.h"

string getLRAMMode_S(L_RamMode mode);
string getPRAMType_S(P_RamType type);

double geomean(vector<double>& data);

#endif /* HELPER_H_ */
```

```cpp
#include <cmath>

#include "helper.h"

string getLRAMMode_S(L_RamMode mode) {
        switch (mode) {
                case ROM:
                        return "ROM";
                case SinglePort:
                        return "SinglePort";
                case SimpleDualPort:
                        return "SimpleDualPort";
                case TrueDualPort:
                        return "TrueDualPort";
        }
        return "";
}

string getPRAMType_S(P_RamType type) {
        switch (type) {
                case LUTRAM:
                        return "LUTRAM";
                case BRAM8192:
                        return "BRAM8192";
                case BRAM128K:
                        return "BRAM128K";
                case BRAM_CUSTOM_1:
                        return "BRAM_CUSTOM_1";
                case BRAM_CUSTOM_2:
                        return "BRAM_CUSTOM_2";
        }
        return "";
}

double geomean(vector<double>& data) {
        double sum = 0.0;
        double size = data.size();
        for (auto i: data) {
                sum += log(i);
        }
        sum /= size;
        return exp(sum);
```

```
}
```

```
#ifndef LOGICALRAM_H_
#define LOGICALRAM_H_

#include <iostream>
#include <string>

using namespace std;

enum L_RamMode {
        ROM = 1,
        SinglePort,
        SimpleDualPort,
        TrueDualPort
};

class LogicalRam {
private:
        unsigned int id;
        unsigned int depth;
        unsigned int width;
        L_RamMode mode;

        // Optimization vars
        bool solved = false;
        unsigned int total_size;

public:
        LogicalRam(unsigned int id_n, unsigned int depth_n, unsigned int
        ↪  width_n, L_RamMode mode_n) : id(id_n), depth(depth_n),
        ↪  width(width_n), mode(mode_n), total_size(depth_n*width_n) {}

        // Get functions
        unsigned int getID() { return id; }
        unsigned int getDepth() { return depth; }
        unsigned int getWidth() { return width; }
        L_RamMode getMode() { return mode; }
        bool getSolved() { return solved; }
        unsigned int getTotalSize() { return total_size; }

        // Set functions
        void setSolved() { solved = true; }
```

```
        // Print functions
        void printInfo();

        // Sort functions
        bool operator < (const LogicalRam& logical_ram) {
                return total_size < logical_ram.total_size;
        }

};

#endif /* LOGICALRAM_H_ */
```

```cpp
#include <iostream>
#include <string>

#include "LogicalRam.h"
#include "helper.h"

using namespace std;

// Get functions

// Print functions
void LogicalRam::printInfo() {
        cout << "ID: " << id;
        cout << ", Depth: " << depth;
        cout << ", Width: " << width;
        cout << ", Mode: " << getLRAMMode_S(mode);
        cout << endl;
}
```

```
#ifndef MAPPEDRAM_H_
#define MAPPEDRAM_H_

#include "PhysicalRam.h"

using namespace std;

class MappedRam {
private:
        unsigned int logical_ram_id;
        unsigned int logical_ram_width;
        unsigned int logical_ram_depth;
        unsigned int id;
        unsigned int S;
        unsigned int P;
        P_RamType type;
        unsigned int W;
        unsigned int D;
        L_RamMode mode;

        unsigned int add_luts;
        double area;

        double cost;

public:
        MappedRam(unsigned int logical_ram_id_n, unsigned int
        ↪  logical_ram_width_n, unsigned int logical_ram_depth_n, unsigned
        ↪  int id_n, unsigned int S_n, unsigned int P_n, P_RamType type_n,
        ↪  unsigned int W_n, unsigned int D_n, L_RamMode mode_n, unsigned
        ↪  int add_luts_n, double area_n, double cost_n) :
        ↪  logical_ram_id(logical_ram_id_n),
        ↪  logical_ram_width(logical_ram_width_n),
        ↪  logical_ram_depth(logical_ram_depth_n), id(id_n), S(S_n),
        ↪  P(P_n), type(type_n), W(W_n), D(D_n), mode(mode_n),
        ↪  add_luts(add_luts_n), area(area_n), cost(cost_n) {}

        // Get functions
        unsigned int getLogicalRamID() { return logical_ram_id; }
        unsigned int getLogicalRamWidth() { return logical_ram_width; }
        unsigned int getLogicalRamDepth() { return logical_ram_depth; }
        unsigned int getID() { return id; }
```

```
        unsigned int getS() { return S; }
        unsigned int getP() { return P; }
        P_RamType getType() { return type; }
        P_RamType getType_output(); // Hacking this for output print
        unsigned int getW() { return W; }
        unsigned int getD() { return D; }
        L_RamMode getMode() { return mode; }
        unsigned int getAddLuts() { return add_luts; }
        double getArea() { return area; }
        double getCost() { return cost; }

        // Set functions

        // Print functions
        void printInfo();

        // Sort functions
        bool operator < (const MappedRam& mapped_ram) {
                return cost < mapped_ram.cost;
        }
};

#endif /* MAPPEDRAM_H_ */
```

```cpp
#include <iostream>

#include "LogicalRam.h"
#include "PhysicalRam.h"
#include "MappedRam.h"
#include "helper.h"

using namespace std;

// Get functions
P_RamType MappedRam::getType_output() {
        switch (type) {
                case BRAM8192:
                        return BRAM8192;
                case BRAM128K:
                        return BRAM128K;
                case BRAM_CUSTOM_1:
                        return BRAM8192;        // For checker since
                        ↪ there's no type 4
                case BRAM_CUSTOM_2:
                        return BRAM128K;        // For checker since
                        ↪ there's no type 5
                default:
                        return LUTRAM;
        }
}

// Print functions
void MappedRam::printInfo() {
        cout << "Logical Ram ID: " << logical_ram_id;
        cout << ", ID: " << id;
        cout << ", S: " << S;
        cout << ", P: " << P;
        cout << ", P_RamType: " << getPRAMType_S(type);
        cout << ", W: " << W;
        cout << ", D: " << D;
        cout << ", L_RamMode: " << getLRAMMode_S(mode);
        cout << ", Additional LUTs: " << add_luts;
        cout << ", Area: " << area;
        cout << ", Cost: " << cost;
        cout << endl;
}
```

```
#ifndef MAPPER_H_
#define MAPPER_H_

#include <vector>

#include "Circuit.h"
#include "Arch.h"

using namespace std;

enum Solution_Type {
        SingleType_Simple = 1,
        SingleType_Shared,
        SingleType_ModifiedCostFunction,
        SingleType_ModifiedCostFunctionWithSharing
};

/*
        Encompasses the entire problem space.
        Contains all the inputs in circuits and architecture.
*/
class Mapper {
private:
        vector<Circuit> circuits;
        Architecture architecture;

        Solution_Type solution_type;

public:
        Mapper(Architecture architecture_n) : architecture(architecture_n)
        ↪  {}

        // Get functions
        vector<Circuit>& getCircuits() { return circuits; }
        Architecture& getArchitecture() { return architecture; }

        // Set functions
        void addCircuit(unsigned int circuit_id_n, unsigned int
        ↪  circuit_logic_block_count_n);
        void setSolutionType(Solution_Type solution_type_n) { solution_type
        ↪  = solution_type_n; }
```

```
        // Computation functions
        void initCircuits(string logical_rams_f, string
        ↪  logic_block_count_f);
        void mapAll_Wrapper();
        void mapAll_SingleType_Simple();
        void mapAll_SingleType_Shared();
        void mapAll_SingleType_ModifiedCostFunction();
        void mapAll_SingleType_ModifiedCostFunctionWithSharing();
        void
        ↪  mapAll_SingleType_ModifiedCostFunctionWithSharing_SortedLogicalRam();

        void computeCircuitAreas();

        // Print functions
        void printCircuits();
        void printCircuitSolutions();
        void printCircuitWidthLogicalRamMap();
        void printCircuitAreas();

        void printCircuits_raw();
        void printLogicalRam_raw();
        void printCircuitSolutions_raw();
};


#endif /* MAPPER_H_ */
```

```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <cmath>
#include <algorithm>

#include "Mapper.h"
#include "helper.h"

using namespace std;

// Set functions
void Mapper::addCircuit(unsigned int circuit_id_n, unsigned int
↪  circuit_logic_block_count_n) {
        Circuit new_circuit(circuit_id_n, circuit_logic_block_count_n);
        circuits.push_back(new_circuit);
}

// Computation functions
void Mapper::initCircuits(string logical_rams_f, string logic_block_count_f)
↪  {
        ifstream logical_rams_fd(logical_rams_f);
        ifstream logic_block_count_fd(logic_block_count_f);

        string line;
        getline(logic_block_count_fd, line); // Skip first line since is
        ↪  field headings
        string circuit_id_s, circuit_logic_block_count_s;
        while (logic_block_count_fd >> circuit_id_s >>
        ↪  circuit_logic_block_count_s)
        {
                unsigned int circuit_id_n = stoi(circuit_id_s);
                unsigned int circuit_logic_block_count_n =
                ↪  stoi(circuit_logic_block_count_s);
                addCircuit(circuit_id_n, circuit_logic_block_count_n);
        }

        string logical_ram_id_s, depth_s, width_s, mode_s;
        L_RamMode mode_n;
        getline(logical_rams_fd, line);        // Skip first line since is
        ↪  total ciruit count
```

```
getline(logical_rams_fd, line);        // Skip first line since is
↪  field headings
while (logical_rams_fd >> circuit_id_s >> logical_ram_id_s >> mode_s
↪  >> depth_s >> width_s)
{
        unsigned int circuit_id_n = stoi(circuit_id_s);
        unsigned int logical_ram_id_n = stoi(logical_ram_id_s);
        unsigned int depth_n = stoi(depth_s);
        unsigned int width_n = stoi(width_s);
        if (mode_s == "ROM") {
                mode_n = ROM;
        }
        else if (mode_s == "SinglePort") {
                mode_n = SinglePort;
        }
        else if (mode_s == "SimpleDualPort") {
                mode_n = SimpleDualPort;
        }
        else if (mode_s == "TrueDualPort") {
                mode_n = TrueDualPort;
        }
        else {
                cout << "Mapper.cpp: Unknown logical ram type." <<
                ↪  endl;
        }

        // This line assumes circuits are entered in the sorted
        ↪  order
        // Assumes circuit id == position in vector array
        circuits[circuit_id_n].addLogicalRam(logical_ram_id_n,
        ↪  depth_n, width_n, mode_n);

        // Optimization var prep
        // Only do for non TrueDualPort
        if (mode_s != "TrueDualPort" && mode_s != "SimpleDualPort")
        ↪  {

                // Retrieve the corresponding map for current
                ↪  circuit
                map<unsigned int, vector<pair<unsigned int, unsigned
                ↪  int>>>& width_logical_ram_map =
                ↪  circuits[circuit_id_n].getWidthLogicalRamMap();
```

```
                        if (width_logical_ram_map.find(width_n) ==
                        ↪  width_logical_ram_map.end()) {
                                // Width not yet in map
                                vector<pair<unsigned int, unsigned int>>
                                ↪  logical_ram_same_width;
                                logical_ram_same_width.push_back(make_pair(logical_ram_i
                                ↪  depth_n));
                                width_logical_ram_map[width_n] =
                                ↪  logical_ram_same_width;
                        } else {
                                // Doing this sort manually since there's
                                ↪  some const issue with sort
                                for (size_t i = 0; i <
                                ↪  width_logical_ram_map[width_n].size();
                                ↪  i++) {
                                        if
                                        ↪  (width_logical_ram_map[width_n][i].second
                                        ↪  > depth_n) {
                                                continue;
                                        }
                                        // At this point we have reach the
                                        ↪  point where depth_n is larger
                                        ↪  than or equal to the rest
                                        width_logical_ram_map[width_n].insert(width_logi
                                        ↪  make_pair(logical_ram_id_n,
                                        ↪  depth_n));
                                        break;
                                }
                        }

                }
        }
        // sortCircuitWidthLogicalRamMap();
}


void Mapper::mapAll_Wrapper() {
        switch (solution_type) {
                case SingleType_Simple:
                        mapAll_SingleType_Simple();
                        break;
```

```
                case SingleType_Shared:
                        mapAll_SingleType_Shared();
                        break;
                case SingleType_ModifiedCostFunction:
                        mapAll_SingleType_ModifiedCostFunction();
                        break;
                case SingleType_ModifiedCostFunctionWithSharing:
                        mapAll_SingleType_ModifiedCostFunctionWithSharing();
                        break;
                default:
                        cout << "Mapper.cpp: Solution type not defined
                        ↪  within bounds: " << solution_type << endl;
        }
}

void Mapper::mapAll_SingleType_Simple() {
        for (auto& circuit: circuits) {
                unsigned int counter = 0;

                for (auto logical_ram: circuit.getLogicalRams()) {
                        if (logical_ram.getSolved()) {
                                // Skip if solved
                                continue;
                        }

                        // Find solution (as long as legal) for every
                        ↪  physical ram ...
                        vector<MappedRam> possible_sol_across_phy_ram;

                        for (auto physical_ram:
                        ↪  architecture.getPhysicalRams()) {

                                // Find solution (as long as legal) for
                                ↪  every possible width for a given
                                ↪  physical ram ...
                                vector<MappedRam>
                                ↪  possible_sol_given_phy_ram;

                                /// Legality checks
                                // If TrueDualPort mode, and current
                                ↪  physical ram is LUTRAM, we skip
```

```
if (logical_ram.getMode() == TrueDualPort &&
↪  physical_ram.getType() == LUTRAM){
        continue;
}

for (auto current_width:
↪  physical_ram.getPossibleWidths()) { //
↪  Assuming there is at least one solution
↪  where S <= 16

        /// Legality checks
        // If TrueDualPort mode, and
        ↪  current width is max, we skip
        if (logical_ram.getMode() ==
        ↪  TrueDualPort && current_width ==
        ↪  physical_ram.getMaxWidth()){
                continue;
        }

        /// Computations
        unsigned int add_decoder_luts = 0;
        unsigned int add_mux_luts = 0;

        // Initialize accordingly
        unsigned int logical_ram_id =
        ↪  logical_ram.getID();
        unsigned int logical_ram_width =
        ↪  logical_ram.getWidth();
        unsigned int logical_ram_depth =
        ↪  logical_ram.getDepth();
        unsigned int id = counter++;
        unsigned int S = 1;
        unsigned int P = 1;
        P_RamType type =
        ↪  physical_ram.getType();
        unsigned int W = current_width;
        unsigned int D =
        ↪  physical_ram.getWidthToDepths()[W];
        L_RamMode mode =
        ↪  logical_ram.getMode();

        unsigned int add_luts = 0;
```

```
double area = 0.0;

double cost = 0.0;

// First we check for width
if (logical_ram.getWidth() > W)
 ↪  {
        // If physical ram width is
         ↪  not enough, need to go
         ↪  parallel first

        // First find out how many
         ↪  in parallel
        P = (unsigned
         ↪  int)ceil((double)logical_ram.getWidt
}

// Check if depth is enough
if (logical_ram.getDepth() > D) {
        // If depth is not enough,
         ↪  we go series
        S = (unsigned
         ↪  int)ceil((double)logical_ram.getDept
        if (S > 16) {
                // Skip if S > 16
                continue;
        }
        // If some in series,
         ↪  additional luts needed
        if (S > 1) {
                // Additional
                 ↪  decoder LUTs
                 ↪  calculation
                if (S == 2) {
                        add_decoder_luts
                         ↪  = 1;
                } else {
                        // Since
                         ↪  log2(R):R
                         ↪  decoder
                         ↪  = R
                         ↪  log2(R)-LUTs
```

```
            // With R <=
            ↪ 16,
            ↪ log2(R)
            ↪ <= 4
            // FPGA LUTs
            ↪ are
            ↪ 6-LUTs
            add_decoder_luts
            ↪ = S;
    }

    // Additional mux
    ↪ LUTs
    ↪ calculation
    // One 6-LUTs makes
    ↪ a 4:1 mux at
    ↪ maximum
    if (S > 1 && S <= 4)
    ↪ {
            // One 4:1
            ↪ mux
            ↪ takes 4
            ↪ inputs
            add_mux_luts
            ↪ = 1;
    } else if (S > 4 &&
    ↪ S <= 7) {
            // Two 4:1
            ↪ mux,
            ↪ first
            ↪ one
            ↪ takes 4
            ↪ inputs,
            ↪ second
            ↪ one
            ↪ takes 3
            ↪ more
            add_mux_luts
            ↪ = 2;
    } else if (S > 7 &&
    ↪ S <= 10) {
```

```
            // Three 4:1 mux, first two takes 4 inputs each, second one takes 2 more
            add_mux_luts = 3;
        } else if (S > 10 && S <= 13) {
            // Four 4:1 mux, first three takes 4 inputs each, second one takes 1 more
            add_mux_luts = 4;
        } else if (S > 13 && S <= 16) {
            // Five 4:1 mux, first four takes 4 inputs each, second one takes none
```

```
                                add_mux_luts
                                ↪  = 5;
                        } else {
                                // Exceeding
                                ↪  limit of
                                ↪  S < 16
                                cout <<
                                ↪  "Mapper.cpp:
                                ↪  S
                                ↪  exceeds
                                ↪  limit of
                                ↪  16." <<
                                ↪  endl;
                        }
                        // Make sure to
                        ↪  account for
                        ↪  width
                        add_mux_luts *=
                        ↪  (W*P);
                }
        }

        add_luts = add_decoder_luts +
        ↪  add_mux_luts;
        // If TrueDualPort mode we need to
        ↪  double additional LUTs
        if (logical_ram.getMode() ==
        ↪  TrueDualPort) {
                add_luts *= 2;
        }

        // Area calculation
        area = ((double)S * (double)P *
        ↪  physical_ram.getArea()) +
        ↪  (ceil((double)add_luts /
        ↪  (double)architecture.getN()) *
        ↪  architecture.getLogicBlockArea());
        cost = area;
```

```
                                        MappedRam
                                        ↪ new_mapped_ram(logical_ram_id,
                                        ↪ logical_ram_width,
                                        ↪ logical_ram_depth, id, S, P,
                                        ↪ type, W, D, mode, add_luts,
                                        ↪ area, cost);
                                        possible_sol_given_phy_ram.push_back(new_mapped_
                                } /* End for (auto current_width:
                                ↪ physical_ram.getPossibleWidths()) */

                                // At this point we have all possible
                                ↪ solutions for current physical ram
                                // Sort and take least area solution
                                if (!possible_sol_given_phy_ram.empty()) {
                                        sort(possible_sol_given_phy_ram.begin(),
                                        ↪ possible_sol_given_phy_ram.end());
                                        possible_sol_across_phy_ram.push_back(possible_s
                                }

                        } /* End for (auto physical_ram:
                        ↪ architecture.getPhysicalRams()) */

                        // At this point we have all best solutions for
                        ↪ each type of physical ram
                        // Sort and take smallest solution
                        if (!possible_sol_across_phy_ram.empty()) {
                                sort(possible_sol_across_phy_ram.begin(),
                                ↪ possible_sol_across_phy_ram.end());
                                circuit.addMappedRam(possible_sol_across_phy_ram[0]);
                                logical_ram.setSolved();
                        }

                } /* End for (auto logical_ram: circuit.getLogicalRams())
                ↪ */

                // At this point we have all solutions for current circuit
                // Carrying on with next circuit ...

        } /* End for (auto circuit: circuits) */
}

void Mapper::mapAll_SingleType_Shared() {
```

```
for (auto& circuit: circuits) {
        unsigned int counter = 0;
        map<unsigned int, vector<pair<unsigned int, unsigned int>>>&
        ↪  width_logical_ram_map = circuit.getWidthLogicalRamMap();

        for (auto& logical_ram: circuit.getLogicalRams()) {
            if (logical_ram.getSolved()) {
                // Skip if solved
                continue;
            }

            // Find solution (as long as legal) for every
            ↪  physical ram ...
            vector<MappedRam> possible_sol_across_phy_ram;
            // Map for <MappedRam ID, LogicalRam ID that's to
            ↪  share physical ram>
            map<unsigned int, unsigned int> possible_ram_share;

            for (auto physical_ram:
            ↪  architecture.getPhysicalRams()) {

                // Find solution (as long as legal) for
                ↪  every possible width for a given
                ↪  physical ram ...
                vector<MappedRam>
                ↪  possible_sol_given_phy_ram;

                /// Legality checks
                // If TrueDualPort mode, and current
                ↪  physical ram is LUTRAM, we skip
                if (logical_ram.getMode() == TrueDualPort &&
                ↪  physical_ram.getType() == LUTRAM){
                    continue;
                }

                for (auto current_width:
                ↪  physical_ram.getPossibleWidths()) { //
                ↪  Assuming there is at least one solution
                ↪  where S <= 16

                    /// Legality checks
```

```
// If TrueDualPort mode, and
↪  current width is max, we skip
if (logical_ram.getMode() ==
↪  TrueDualPort && current_width ==
↪  physical_ram.getMaxWidth()){
        continue;
}

/// Computations
unsigned int add_decoder_luts = 0;
unsigned int add_mux_luts = 0;

// Initialize accordingly
unsigned int logical_ram_id =
↪  logical_ram.getID();
unsigned int logical_ram_width =
↪  logical_ram.getWidth();
unsigned int logical_ram_depth =
↪  logical_ram.getDepth();
unsigned int id = counter++;
unsigned int S = 1;
unsigned int P = 1;
P_RamType type =
↪  physical_ram.getType();
unsigned int W = current_width;
unsigned int D =
↪  physical_ram.getWidthToDepths()[W];
L_RamMode mode =
↪  logical_ram.getMode();

unsigned int add_luts = 0;
double area = 0.0;

double cost = 0.0;

// First we check for width
if (logical_ram.getWidth() > W)
↪  {
        // If physical ram width is
        ↪  not enough, need to go
        ↪  parallel first
```

```
                // First find out how many
                ↪  in parallel
                P = (unsigned
                ↪  int)ceil((double)logical_ram.getWidt
        }

        // Check if depth is enough
        if (logical_ram.getDepth() > D) {
                // If depth is not enough,
                ↪  we go series
                S = (unsigned
                ↪  int)ceil((double)logical_ram.getDept
                if (S > 16) {
                        // Skip if S > 16
                        continue;
                }
                // If some in series,
                ↪  additional luts needed
                if (S > 1) {
                        // Additional
                        ↪  decoder LUTs
                        ↪  calculation
                        if (S == 2) {
                                add_decoder_luts
                                ↪  = 1;
                        } else {
                                // Since
                                ↪  log2(R):R
                                ↪  decoder
                                ↪  = R
                                ↪  log2(R)-LUTs
                                // With R <=
                                ↪  16,
                                ↪  log2(R)
                                ↪  <= 4
                                // FPGA LUTs
                                ↪  are
                                ↪  6-LUTs
                                add_decoder_luts
                                ↪  = S;
                        }
```

```
// Additional mux LUTs calculation
// One 6-LUTs makes a 4:1 mux at maximum
if (S > 1 && S <= 4) {
        // One 4:1 mux takes 4 inputs
        add_mux_luts = 1;
} else if (S > 4 && S <= 7) {
        // Two 4:1 mux, first one takes 4 inputs, second one takes 3 more
        add_mux_luts = 2;
} else if (S > 7 && S <= 10) {
        // Three 4:1 mux, first two takes 4 inputs each, second one takes 2 more
```

```
            add_mux_luts
             ↪  = 3;
        } else if (S > 10 &&
         ↪  S <= 13) {
            // Four 4:1
             ↪  mux,
             ↪  first
             ↪  three
             ↪  takes 4
             ↪  inputs
             ↪  each,
             ↪  second
             ↪  one
             ↪  takes 1
             ↪  more
            add_mux_luts
             ↪  = 4;
        } else if (S > 13 &&
         ↪  S <= 16) {
            // Five 4:1
             ↪  mux,
             ↪  first
             ↪  four
             ↪  takes 4
             ↪  inputs
             ↪  each,
             ↪  second
             ↪  one
             ↪  takes
             ↪  none
            add_mux_luts
             ↪  = 5;
        } else {
            // Exceeding
             ↪  limit of
             ↪  S < 16
```

```
                                cout <<
                                ↪  "Mapper.cpp:
                                ↪  S
                                ↪  exceeds
                                ↪  limit of
                                ↪  16." <<
                                ↪  endl;
                        }
                        // Make sure to
                        ↪  account for
                        ↪  width
                        add_mux_luts *=
                        ↪  (W*P);
                }
        }

        add_luts = add_decoder_luts +
        ↪  add_mux_luts;
        // If TrueDualPort mode we need to
        ↪  double additional LUTs
        if (logical_ram.getMode() ==
        ↪  TrueDualPort) {
                add_luts *= 2;
        }

        // Area calculation
        area = ((double)S * (double)P *
        ↪  physical_ram.getArea()) +
        ↪  (((double)add_luts /
        ↪  (double)architecture.getN()) *
        ↪  architecture.getLogicBlockArea());
        cost = area;

        // At this point we have a
        ↪  potential solution
        // To optimize mapping, we want to
        ↪  see if we can fit in another
        ↪  logical ram in this
        // if this is not LUTRAM and not
        ↪  already in TrueDualPort mode
        ↪  and not operating at the max
        ↪  width
```

```
if (physical_ram.getType() != LUTRAM
↪  && logical_ram.getMode() !=
↪  SimpleDualPort &&
↪  logical_ram.getMode() !=
↪  TrueDualPort && current_width !=
↪  physical_ram.getMaxWidth()){

    for (unsigned int width_i =
    ↪  current_width;
    ↪  current_width >= 1;
    ↪  current_width /= 2) {

        // Look for other
        ↪  logical rams
        ↪  that has the
        ↪  same (current)
        ↪  width
        vector<pair<unsigned
        ↪  int, unsigned
        ↪  int>>
        ↪  potential_matches
        ↪  =
        ↪  width_logical_ram_map[width_

        unsigned int
        ↪  remaining_depth
        ↪  = D*S -
        ↪  logical_ram_depth;
        unsigned int
        ↪  remaining_space
        ↪  = S*P*W*D -
        ↪  logical_ram_depth*logical_ra
        for (auto
        ↪  potential_match:
        ↪  potential_matches)
        ↪  {
```

```
// For each potential match, we check if the depth can fit
// Note that potential_matches is in sorted order: descending depth size

if (potential_match.fir == logical_ram_id) {
        // Skip if found self
        continue;
}

// Not enough space/depth left, skip
if (potential_match.sec > remaining_depth) {
        continue;
}
```

```
if
 ↪  (width_i*potential_m
 ↪  >
 ↪  remaining_space)
 ↪  {
        continue;
}

// Assuming
 ↪  there's
 ↪  a
 ↪  possible
 ↪  match
// Retrieve
 ↪  from
 ↪  circuit's
 ↪  logical_ram
 ↪  vector
 ↪  based on
 ↪  id
if
 ↪  (circuit.getLogicalR
        //
         ↪  If
         ↪  solved,
         ↪  move
         ↪  on
        continue;
}

// At this
 ↪  point we
 ↪  have a
 ↪  logical
 ↪  ram that
 ↪  can
 ↪  share
 ↪  physical
 ↪  ram
```

```
// We record this possible ram share in our map
// Map is used for look up later after all widths are explored
possible_ram_share[id] = potential_match.firs

// Update cost value
cost = area * ((double)logical_ram

// If choose to share, mode changes to either TrueDualPort or ROM
mode = TrueDualPort;
```

```
                                        // Only can
                                         ↪ share
                                         ↪ ram with
                                         ↪ one
                                         ↪ other
                                         ↪ logical
                                         ↪ ram
                                        // Since
                                         ↪ potential_matches
                                         ↪ is
                                         ↪ already
                                         ↪ sorted,
                                         ↪
                                        // the first
                                         ↪ one we
                                         ↪ find is
                                         ↪ the
                                         ↪ largest
                                         ↪ possible
                                         ↪ depth
                                         ↪ that
                                         ↪ fits
                                        break;
                                    }
                                    // Note that if no
                                     ↪ potential match
                                     ↪ is found, we
                                     ↪ leave the loop
                                     ↪ and there is no
                                     ↪ difference
                                     ↪ compared
                                    // to the simple
                                     ↪ single ram
                                     ↪ mapping function
                                     ↪ above
                                }
                            }
```

```
                        MappedRam
                        ↪ new_mapped_ram(logical_ram_id,
                        ↪ logical_ram_width,
                        ↪ logical_ram_depth, id, S, P,
                        ↪ type, W, D, mode, add_luts,
                        ↪ area, cost);
                        possible_sol_given_phy_ram.push_back(new_mapped_


                    } /* End for (auto current_width:
                    ↪ physical_ram.getPossibleWidths()) */

                    // At this point we have all possible
                    ↪ solutions for current physical ram
                    // Sort and take least area solution
                    if (!possible_sol_given_phy_ram.empty()) {
                            sort(possible_sol_given_phy_ram.begin(),
                            ↪ possible_sol_given_phy_ram.end());
                            possible_sol_across_phy_ram.push_back(possible_s
                    }

            } /* End for (auto physical_ram:
            ↪ architecture.getPhysicalRams()) */

            // At this point we have all best solutions for
            ↪ each type of physical ram
            // Sort and take smallest solution
            if (!possible_sol_across_phy_ram.empty()) {
                    sort(possible_sol_across_phy_ram.begin(),
                    ↪ possible_sol_across_phy_ram.end());

                    // Check if chosen solution does any ram
                    ↪ sharing
                    if
                    ↪ (possible_ram_share.find(possible_sol_across_phy_ram
                    ↪ != possible_ram_share.end()) {
                            // If ram sharing
                            unsigned int other_logical_ram_id =
                            ↪ possible_ram_share[possible_sol_across_phy_r
                            LogicalRam other_logical_ram =
                            ↪ circuit.getLogicalRams()[other_logical_ram_i
```

```
                                        MappedRam
                                        ↪ shared_mapped_ram(other_logical_ram_id,
                                        ↪ other_logical_ram.getWidth(),
                                        ↪ other_logical_ram.getDepth(),
                                        ↪ possible_sol_across_phy_ram[0].getID(),
                                        ↪ possible_sol_across_phy_ram[0].getS(),
                                        ↪ possible_sol_across_phy_ram[0].getP(),
                                        ↪ possible_sol_across_phy_ram[0].getType(),
                                        ↪ possible_sol_across_phy_ram[0].getW(),
                                        ↪ possible_sol_across_phy_ram[0].getD(),
                                        ↪ possible_sol_across_phy_ram[0].getMode(),
                                        ↪ possible_sol_across_phy_ram[0].getAddLuts(),
                                        ↪ possible_sol_across_phy_ram[0].getArea(),
                                        ↪ possible_sol_across_phy_ram[0].getArea()-pos
                                        circuit.addMappedRam(shared_mapped_ram);

                                        circuit.getLogicalRams()[other_logical_ram_id].s
                                }

                                circuit.addMappedRam(possible_sol_across_phy_ram[0]);
                                logical_ram.setSolved();
                        }

                } /* End for (auto logical_ram: circuit.getLogicalRams())
                ↪  */

                // At this point we have all solutions for current circuit
                // Carrying on with next circuit ...

        } /* End for (auto circuit: circuits) */
}

void Mapper::mapAll_SingleType_ModifiedCostFunction() {
        for (auto& circuit: circuits) {
                unsigned int counter = 0;

                // Used for cost function
                map<P_RamType, unsigned int> total_physical_ram_used;
                map<P_RamType, unsigned int> available_physical_ram;
                map<P_RamType, double> physical_ram_ratio;
                for (auto physical_ram: architecture.getPhysicalRams()) {
                        total_physical_ram_used[physical_ram.getType()] = 0;
```

```
                // Also defined the available physical ram count
                ↪  for each type
                // based on the current number of logic blocks used
                ↪  for logic
                // Requiring more physical ram of any type above
                ↪  it's limit
                // causes LB count to increase and thus incurs more
                ↪  area than just the ram itself
                available_physical_ram[physical_ram.getType()] =
                ↪  floor(circuit.getLogicBlockNum()/physical_ram.getRatio());
                physical_ram_ratio[physical_ram.getType()] =
                ↪  physical_ram.getRatio();
        }


        for (auto logical_ram: circuit.getLogicalRams()) {
                if (logical_ram.getSolved()) {
                        // Skip if solved
                        continue;
                }

                // Find solution (as long as legal) for every
                ↪  physical ram ...
                vector<MappedRam> possible_sol_across_phy_ram;

                for (auto physical_ram:
                ↪  architecture.getPhysicalRams()) {

                        // Find solution (as long as legal) for
                        ↪  every possible width for a given
                        ↪  physical ram ...
                        vector<MappedRam>
                        ↪  possible_sol_given_phy_ram;

                        /// Legality checks
                        // If TrueDualPort mode, and current
                        ↪  physical ram is LUTRAM, we skip
                        if (logical_ram.getMode() == TrueDualPort &&
                        ↪  physical_ram.getType() == LUTRAM){
                                continue;
                        }
```

```
for (auto current_width:
 ↪  physical_ram.getPossibleWidths()) { //
 ↪  Assuming there is at least one solution
 ↪  where S <= 16

        /// Legality checks
        // If TrueDualPort mode, and
         ↪  current width is max, we skip
        if (logical_ram.getMode() ==
         ↪  TrueDualPort && current_width ==
         ↪  physical_ram.getMaxWidth()){
                continue;
        }

        /// Computations
        unsigned int add_decoder_luts = 0;
        unsigned int add_mux_luts = 0;

        // Initialize accordingly
        unsigned int logical_ram_id =
         ↪  logical_ram.getID();
        unsigned int logical_ram_width =
         ↪  logical_ram.getWidth();
        unsigned int logical_ram_depth =
         ↪  logical_ram.getDepth();
        unsigned int id = counter++;
        unsigned int S = 1;
        unsigned int P = 1;
        P_RamType type =
         ↪  physical_ram.getType();
        unsigned int W = current_width;
        unsigned int D =
         ↪  physical_ram.getWidthToDepths()[W];
        L_RamMode mode =
         ↪  logical_ram.getMode();

        unsigned int add_luts = 0;
        double area = 0.0;

        double cost = 0.0;
```

```
// First we check for width
if (logical_ram.getWidth() > W)
↪  {
        // If physical ram width is
        ↪  not enough, need to go
        ↪  parallel first

        // First find out how many
        ↪  in parallel
        P = (unsigned
        ↪  int)ceil((double)logical_ram.getWidt
}

// Check if depth is enough
if (logical_ram.getDepth() > D) {
        // If depth is not enough,
        ↪  we go series
        S = (unsigned
        ↪  int)ceil((double)logical_ram.getDept
        if (S > 16) {
                // Skip if S > 16
                continue;
        }
        // If some in series,
        ↪  additional luts needed
        if (S > 1) {
                // Additional
                ↪  decoder LUTs
                ↪  calculation
                if (S == 2) {
                        add_decoder_luts
                        ↪  = 1;
                } else {
                        // Since
                        ↪  log2(R):R
                        ↪  decoder
                        ↪  = R
                        ↪  log2(R)-LUTs
                        // With R <=
                        ↪  16,
                        ↪  log2(R)
                        ↪  <= 4
```

```
            // FPGA LUTs
            ↪ are
            ↪ 6-LUTs
            add_decoder_luts
            ↪ = S;
    }

    // Additional mux
    ↪ LUTs
    ↪ calculation
    // One 6-LUTs makes
    ↪ a 4:1 mux at
    ↪ maximum
    if (S > 1 && S <= 4)
    ↪ {
            // One 4:1
            ↪ mux
            ↪ takes 4
            ↪ inputs
            add_mux_luts
            ↪ = 1;
    } else if (S > 4 &&
    ↪ S <= 7) {
            // Two 4:1
            ↪ mux,
            ↪ first
            ↪ one
            ↪ takes 4
            ↪ inputs,
            ↪ second
            ↪ one
            ↪ takes 3
            ↪ more
            add_mux_luts
            ↪ = 2;
    } else if (S > 7 &&
    ↪ S <= 10) {
```

```
            // Three 4:1 mux, first two takes 4 inputs each, second one takes 2 more
            add_mux_luts = 3;
        } else if (S > 10 && S <= 13) {
            // Four 4:1 mux, first three takes 4 inputs each, second one takes 1 more
            add_mux_luts = 4;
        } else if (S > 13 && S <= 16) {
            // Five 4:1 mux, first four takes 4 inputs each, second one takes none
```

```
                                add_mux_luts
                                ↪  = 5;
                        } else {
                                // Exceeding
                                ↪  limit of
                                ↪  S < 16
                                cout <<
                                ↪  "Mapper.cpp:
                                ↪  S
                                ↪  exceeds
                                ↪  limit of
                                ↪  16." <<
                                ↪  endl;
                        }
                        // Make sure to
                        ↪  account for
                        ↪  width
                        add_mux_luts *=
                        ↪  (W*P);
                }
        }

        add_luts = add_decoder_luts +
        ↪  add_mux_luts;
        // If TrueDualPort mode we need to
        ↪  double additional LUTs
        if (logical_ram.getMode() ==
        ↪  TrueDualPort) {
                add_luts *= 2;
        }

        // Area calculation
        area = ((double)S * (double)P *
        ↪  physical_ram.getArea()) +
        ↪  (ceil((double)add_luts /
        ↪  (double)architecture.getN()) *
        ↪  architecture.getLogicBlockArea());

        // Cost calculations
        // Based on what physical ram, how
        ↪  many has been used, how many to
        ↪  be used, we compute a cost
```

```
// If using this ram does not cause more LB to be needed, cost is just area
cost = area;
if ((total_physical_ram_used[physical_ram.getTy + S*P) > available_physical_ram[physical_ram.getType( {
        // If using this ram causes more LB to be needed, add that to cost
        if (physical_ram.getType() == LUTRAM) {
                //physical_ram.getArea() here since LUTRAM blocks are more expensive
                cost += ((double)S*(double)P) * physical_ram.getArea();
                //physical_ram.getRatio() - 1 since a portion is accounted for above
                cost += ((double)S*(double)P) * (physical_ram_ratio[physical - 1) * architecture.getLogicBlockAr
        } else {
                // Assuming all required BRAM blocks are going to require new LB/tiles
```

```
                                        // Assuming all new
                                        ↪ LBs are just
                                        ↪ normal ones to
                                        ↪ simplify
                                        ↪ calculation
                                        cost +=
                                        ↪ ((double)S*(double)P)
                                        ↪ *
                                        ↪ physical_ram_ratio[physical_
                                        ↪ *
                                        ↪ architecture.getLogicBlockAr
                                }
                        }

                        MappedRam
                        ↪ new_mapped_ram(logical_ram_id,
                        ↪ logical_ram_width,
                        ↪ logical_ram_depth, id, S, P,
                        ↪ type, W, D, mode, add_luts,
                        ↪ area, cost);
                        possible_sol_given_phy_ram.push_back(new_mapped_
                } /* End for (auto current_width:
                ↪ physical_ram.getPossibleWidths()) */

                // At this point we have all possible
                ↪ solutions for current physical ram
                // Sort and take least area solution
                if (!possible_sol_given_phy_ram.empty()) {
                        sort(possible_sol_given_phy_ram.begin(),
                        ↪ possible_sol_given_phy_ram.end());
                        possible_sol_across_phy_ram.push_back(possible_s
                }

        } /* End for (auto physical_ram:
        ↪ architecture.getPhysicalRams()) */

        // At this point we have all best solutions for
        ↪ each type of physical ram
        // Sort and take smallest solution
        if (!possible_sol_across_phy_ram.empty()) {
                sort(possible_sol_across_phy_ram.begin(),
                ↪ possible_sol_across_phy_ram.end());
```

```
                        circuit.addMappedRam(possible_sol_across_phy_ram[0]);
                        logical_ram.setSolved();

                        // Update total physical ram used for cost
                        ↪  function
                        P_RamType physical_ram_used_type =
                        ↪  possible_sol_across_phy_ram[0].getType();
                        total_physical_ram_used[physical_ram_used_type]
                        ↪  += possible_sol_across_phy_ram[0].getS()
                        ↪  * possible_sol_across_phy_ram[0].getP();

                        unsigned int extra_physical_ram_required =
                        ↪  total_physical_ram_used[physical_ram_used_type]
                        ↪  -
                        ↪  available_physical_ram[physical_ram_used_type];
                        if (extra_physical_ram_required > 0) {
                                // Reach here if solution's ram
                                ↪  needs exceed currently
                                ↪  available
                                // Update available since more LBs
                                ↪  are now available with this new
                                ↪  ram
                                ↪  requirement
                                unsigned int new_total_lb_count =
                                ↪  (double)total_physical_ram_used[physical_ram
                                ↪  *
                                ↪  physical_ram_ratio[physical_ram_used_type];

                                for (auto physical_ram:
                                ↪  architecture.getPhysicalRams())
                                ↪  {
                                        available_physical_ram[physical_ram.getT
                                        ↪  =
                                        ↪  floor(new_total_lb_count/physical_ra
                                }
                        }
                }
        }
} /* End for (auto logical_ram: circuit.getLogicalRams())
↪  */

// At this point we have all solutions for current circuit
// Carrying on with next circuit ...
```

```
        } /* End for (auto circuit: circuits) */
}

void Mapper::mapAll_SingleType_ModifiedCostFunctionWithSharing() {
        for (auto& circuit: circuits) {
                unsigned int counter = 0;
                map<unsigned int, vector<pair<unsigned int, unsigned int>>>&
                ↪  width_logical_ram_map =
                ↪  circuit.getWidthLogicalRamMap();

                // Used for cost function
                map<P_RamType, unsigned int> total_physical_ram_used;
                map<P_RamType, unsigned int> available_physical_ram;
                map<P_RamType, double> physical_ram_ratio;
                for (auto physical_ram: architecture.getPhysicalRams()) {
                        total_physical_ram_used[physical_ram.getType()] = 0;

                        // Also defined the available physical ram count
                        ↪  for each type
                        // based on the current number of logic blocks used
                        ↪  for logic
                        // Requiring more physical ram of any type above
                        ↪  it's limit
                        // causes LB count to increase and thus incurs more
                        ↪  area than just the ram itself
                        available_physical_ram[physical_ram.getType()] =
                        ↪  floor(circuit.getLogicBlockNum()/physical_ram.getRatio());
                        physical_ram_ratio[physical_ram.getType()] =
                        ↪  physical_ram.getRatio();
                }


                for (auto& logical_ram: circuit.getLogicalRams()) {
                        if (logical_ram.getSolved()) {
                                // Skip if solved
                                continue;
                        }

                        // Find solution (as long as legal) for every
                        ↪  physical ram ...
                        vector<MappedRam> possible_sol_across_phy_ram;
```

```
// Map for <MappedRam ID, LogicalRam ID that's to
↪  share physical ram>
map<unsigned int, unsigned int> possible_ram_share;

for (auto physical_ram:
↪  architecture.getPhysicalRams()) {

        // Find solution (as long as legal) for
        ↪  every possible width for a given
        ↪  physical ram ...
        vector<MappedRam>
        ↪  possible_sol_given_phy_ram;

        /// Legality checks
        // If TrueDualPort mode, and current
        ↪  physical ram is LUTRAM, we skip
        if (logical_ram.getMode() == TrueDualPort &&
        ↪  physical_ram.getType() == LUTRAM){
                continue;
        }

        for (auto current_width:
        ↪  physical_ram.getPossibleWidths()) { //
        ↪  Assuming there is at least one solution
        ↪  where S <= 16

                /// Legality checks
                // If TrueDualPort mode, and
                ↪  current width is max, we skip
                if (logical_ram.getMode() ==
                ↪  TrueDualPort && current_width ==
                ↪  physical_ram.getMaxWidth()){
                        continue;
                }

                /// Computations
                unsigned int add_decoder_luts = 0;
                unsigned int add_mux_luts = 0;

                // Initialize accordingly
                unsigned int logical_ram_id =
                ↪  logical_ram.getID();
```

```
unsigned int logical_ram_width =
↪  logical_ram.getWidth();
unsigned int logical_ram_depth =
↪  logical_ram.getDepth();
unsigned int id = counter++;
unsigned int S = 1;
unsigned int P = 1;
P_RamType type =
↪  physical_ram.getType();
unsigned int W = current_width;
unsigned int D =
↪  physical_ram.getWidthToDepths()[W];
L_RamMode mode =
↪  logical_ram.getMode();

unsigned int add_luts = 0;
double area = 0.0;

double cost = 0.0;

// First we check for width
if (logical_ram.getWidth() > W)
↪  {
        // If physical ram width is
        ↪  not enough, need to go
        ↪  parallel first

        // First find out how many
        ↪  in parallel
        P = (unsigned
        ↪  int)ceil((double)logical_ram.getWidt
}

// Check if depth is enough
if (logical_ram.getDepth() > D) {
        // If depth is not enough,
        ↪  we go series
        S = (unsigned
        ↪  int)ceil((double)logical_ram.getDept
        if (S > 16) {
                // Skip if S > 16
                continue;
```

```
}
// If some in series, additional luts needed
if (S > 1) {
        // Additional decoder LUTs calculation
        if (S == 2) {
                add_decoder_luts = 1;
        } else {
                // Since log2(R):R decoder = R log2(R)-LUTs
                // With R <= 16, log2(R) <= 4
                // FPGA LUTs are 6-LUTs
                add_decoder_luts = S;
        }

        // Additional mux LUTs calculation
        // One 6-LUTs makes a 4:1 mux at maximum
        if (S > 1 && S <= 4) {
                // One 4:1 mux takes 4 inputs
                add_mux_luts = 1;
```

```
} else if (S > 4 &&
↪  S <= 7) {
        // Two 4:1
        ↪  mux,
        ↪  first
        ↪  one
        ↪  takes 4
        ↪  inputs,
        ↪  second
        ↪  one
        ↪  takes 3
        ↪  more
        add_mux_luts
        ↪  = 2;
} else if (S > 7 &&
↪  S <= 10) {
        // Three 4:1
        ↪  mux,
        ↪  first
        ↪  two
        ↪  takes 4
        ↪  inputs
        ↪  each,
        ↪  second
        ↪  one
        ↪  takes 2
        ↪  more
        add_mux_luts
        ↪  = 3;
} else if (S > 10 &&
↪  S <= 13) {
        // Four 4:1
        ↪  mux,
        ↪  first
        ↪  three
        ↪  takes 4
        ↪  inputs
        ↪  each,
        ↪  second
        ↪  one
        ↪  takes 1
        ↪  more
```

```
                                add_mux_luts
                                ↪  = 4;
                            } else if (S > 13 &&
                            ↪  S <= 16) {
                                // Five 4:1
                                ↪  mux,
                                ↪  first
                                ↪  four
                                ↪  takes 4
                                ↪  inputs
                                ↪  each,
                                ↪  second
                                ↪  one
                                ↪  takes
                                ↪  none
                                add_mux_luts
                                ↪  = 5;
                            } else {
                                // Exceeding
                                ↪  limit of
                                ↪  S < 16
                                cout <<
                                ↪  "Mapper.cpp:
                                ↪  S
                                ↪  exceeds
                                ↪  limit of
                                ↪  16." <<
                                ↪  endl;
                            }
                            // Make sure to
                            ↪  account for
                            ↪  width
                            add_mux_luts *=
                            ↪  (W*P);
                        }
                    }

                    add_luts = add_decoder_luts +
                    ↪  add_mux_luts;
                    // If TrueDualPort mode we need to
                    ↪  double additional LUTs
```

```
if (logical_ram.getMode() ==
↪  TrueDualPort) {
        add_luts *= 2;
}

// Area calculation
area = ((double)S * (double)P *
↪  physical_ram.getArea()) +
↪  (ceil((double)add_luts /
↪  (double)architecture.getN()) *
↪  architecture.getLogicBlockArea());

// Cost calculations
// Based on what physical ram, how
↪  many has been used, how many to
↪  be used, we compute a cost
// If using this ram does not cause
↪  more LB to be needed, cost is
↪  just area
cost = area;
if
↪  ((total_physical_ram_used[physical_ram.getTy
↪  + S*P) >
↪  available_physical_ram[physical_ram.getType(
↪  {
        // If using this ram causes
        ↪  more LB to be needed,
        ↪  add that to cost
        if (physical_ram.getType()
        ↪  == LUTRAM) {
                //
                ↪  physical_ram.getArea()
                ↪  here since
                ↪  LUTRAM blocks
                ↪  are more
                ↪  expensive
                cost +=
                ↪  ((double)S*(double)P)
                ↪  *
                ↪  physical_ram.getArea();
```

```
                                //
                                ↪ physical_ram.getRatio()
                                ↪ - 1 since a
                                ↪ portion is
                                ↪ accounted for
                                ↪ above
                                cost +=
                                ↪ ((double)S*(double)P)
                                ↪ *
                                ↪ (physical_ram_ratio[physical
                                ↪ - 1) *
                                ↪ architecture.getLogicBlockAr
                        } else {
                                // Assuming all
                                ↪ required BRAM
                                ↪ blocks are going
                                ↪ to require new
                                ↪ LB/tiles
                                // Assuming all new
                                ↪ LBs are just
                                ↪ normal ones to
                                ↪ simplify
                                ↪ calculation
                                cost +=
                                ↪ ((double)S*(double)P)
                                ↪ *
                                ↪ physical_ram_ratio[physical_
                                ↪ *
                                ↪ architecture.getLogicBlockAr
                        }
                }

                // To optimize mapping, we want to
                ↪ see if we can fit in another
                ↪ logical ram in this
                // if this is not LUTRAM and not
                ↪ already in TrueDualPort mode
                ↪ and not operating at the max
                ↪ width
```

```
if (physical_ram.getType() != LUTRAM
↪  && logical_ram.getMode() !=
↪  SimpleDualPort &&
↪  logical_ram.getMode() !=
↪  TrueDualPort && current_width !=
↪  physical_ram.getMaxWidth()){

    for (unsigned int width_i =
    ↪  current_width;
    ↪  current_width >= 1;
    ↪  current_width /= 2) {

        // Look for other
        ↪  logical rams
        ↪  that has the
        ↪  same (current)
        ↪  width
        vector<pair<unsigned
        ↪  int, unsigned
        ↪  int>>
        ↪  potential_matches
        ↪  =
        ↪  width_logical_ram_map[width_

        unsigned int
        ↪  remaining_depth
        ↪  = D*S -
        ↪  logical_ram_depth;
        unsigned int
        ↪  remaining_space
        ↪  = S*P*W*D -
        ↪  logical_ram_depth*logical_ra
        for (auto
        ↪  potential_match:
        ↪  potential_matches)
        ↪  {
```

```
// For each potential match, we check if the depth can fit
// Note that potential_matches is in sorted order: descending depth size

if (potential_match.fir == logical_ram_id) {
        // Skip if found self
        continue;
}

// Not enough space/depth left, skip
if (potential_match.sec > remaining_depth) {
        continue;
}
```

```
if
 ↪  (width_i*potential_m
 ↪  >
 ↪  remaining_space)
 ↪  {
        continue;
}

// Assuming
 ↪  there's
 ↪  a
 ↪  possible
 ↪  match
// Retrieve
 ↪  from
 ↪  circuit's
 ↪  logical_ram
 ↪  vector
 ↪  based on
 ↪  id
if
 ↪  (circuit.getLogicalR
        //
         ↪  If
         ↪  solved,
         ↪  move
         ↪  on
        continue;
}

// At this
 ↪  point we
 ↪  have a
 ↪  logical
 ↪  ram that
 ↪  can
 ↪  share
 ↪  physical
 ↪  ram
```

```
// We record this possible ram share in our map
// Map is used for look up later after all widths are explored
possible_ram_share[id] = potential_match.firs

// Update cost value
cost *= ((double)logical_ram

// If choose to share, mode changes to either TrueDualPort or ROM
mode = TrueDualPort;
```

```
                                        // Only can
                                         ↪ share
                                         ↪ ram with
                                         ↪ one
                                         ↪ other
                                         ↪ logical
                                         ↪ ram
                                        // Since
                                         ↪ potential_matches
                                         ↪ is
                                         ↪ already
                                         ↪ sorted,
                                         ↪
                                        // the first
                                         ↪ one we
                                         ↪ find is
                                         ↪ the
                                         ↪ largest
                                         ↪ possible
                                         ↪ depth
                                         ↪ that
                                         ↪ fits
                                        goto
                                         ↪ EndOfLoop;
                                }
                                // Note that if no
                                 ↪ potential match
                                 ↪ is found, we
                                 ↪ leave the loop
                                 ↪ and there is no
                                 ↪ difference
                                 ↪ compared
                                // to the simple
                                 ↪ single ram
                                 ↪ mapping function
                                 ↪ above
                        }
                }

                EndOfLoop:
```

```
                        MappedRam
                        ↪ new_mapped_ram(logical_ram_id,
                        ↪ logical_ram_width,
                        ↪ logical_ram_depth, id, S, P,
                        ↪ type, W, D, mode, add_luts,
                        ↪ area, cost);
                        possible_sol_given_phy_ram.push_back(new_mapped_
                } /* End for (auto current_width:
                ↪ physical_ram.getPossibleWidths()) */

                // At this point we have all possible
                ↪ solutions for current physical ram
                // Sort and take least area solution
                if (!possible_sol_given_phy_ram.empty()) {
                        sort(possible_sol_given_phy_ram.begin(),
                        ↪ possible_sol_given_phy_ram.end());
                        possible_sol_across_phy_ram.push_back(possible_s
                }

        } /* End for (auto physical_ram:
        ↪ architecture.getPhysicalRams()) */

        // At this point we have all best solutions for
        ↪ each type of physical ram
        // Sort and take smallest solution
        if (!possible_sol_across_phy_ram.empty()) {
                sort(possible_sol_across_phy_ram.begin(),
                ↪ possible_sol_across_phy_ram.end());

                // Check if chosen solution does any ram
                ↪ sharing
                if
                ↪ (possible_ram_share.find(possible_sol_across_phy_ram
                ↪ != possible_ram_share.end()) {
                        // If ram sharing
                        unsigned int other_logical_ram_id =
                        ↪ possible_ram_share[possible_sol_across_phy_r
                        LogicalRam other_logical_ram =
                        ↪ circuit.getLogicalRams()[other_logical_ram_i
```

```
                MappedRam
                 ↪  shared_mapped_ram(other_logical_ram_id,
                 ↪  other_logical_ram.getWidth(),
                 ↪  other_logical_ram.getDepth(),
                 ↪  possible_sol_across_phy_ram[0].getID(),
                 ↪  possible_sol_across_phy_ram[0].getS(),
                 ↪  possible_sol_across_phy_ram[0].getP(),
                 ↪  possible_sol_across_phy_ram[0].getType(),
                 ↪  possible_sol_across_phy_ram[0].getW(),
                 ↪  possible_sol_across_phy_ram[0].getD(),
                 ↪  possible_sol_across_phy_ram[0].getMode(),
                 ↪  possible_sol_across_phy_ram[0].getAddLuts(),
                 ↪  possible_sol_across_phy_ram[0].getArea(),
                 ↪  possible_sol_across_phy_ram[0].getArea()-pos
                circuit.addMappedRam(shared_mapped_ram);

                circuit.getLogicalRams()[other_logical_ram_id].s
        }

        circuit.addMappedRam(possible_sol_across_phy_ram[0]);
        logical_ram.setSolved();

        // Update total physical ram used for cost
         ↪  function
        P_RamType physical_ram_used_type =
         ↪  possible_sol_across_phy_ram[0].getType();
        total_physical_ram_used[physical_ram_used_type]
         ↪  += possible_sol_across_phy_ram[0].getS()
         ↪  * possible_sol_across_phy_ram[0].getP();

        unsigned int extra_physical_ram_required =
         ↪  total_physical_ram_used[physical_ram_used_type]
         ↪  -
         ↪  available_physical_ram[physical_ram_used_type];
        if (extra_physical_ram_required > 0) {
                // Reach here if solution's ram
                 ↪  needs exceed currently
                 ↪  available
                // Update available since more LBs
                 ↪  are now available with this new
                 ↪  ram
                 ↪  requirement
```

```
                                        unsigned int new_total_lb_count =
                                        ↪  (double)total_physical_ram_used[physical_ram
                                        ↪  *
                                        ↪  physical_ram_ratio[physical_ram_used_type];

                                        for (auto physical_ram:
                                        ↪  architecture.getPhysicalRams())
                                        ↪  {
                                                available_physical_ram[physical_ram.getT
                                                ↪  =
                                                ↪  floor(new_total_lb_count/physical_ra
                                        }
                                }
                        }
                } /* End for (auto logical_ram: circuit.getLogicalRams())
                ↪  */

                // At this point we have all solutions for current circuit
                // Carrying on with next circuit ...

        } /* End for (auto circuit: circuits) */
}

void Mapper::computeCircuitAreas() {
        for (auto& i: circuits) {
                i.computeArea(architecture.getPhysicalRams(),
                ↪  architecture.getN(), architecture.getLogicBlockArea());
        }
}

// Print functions
void Mapper::printCircuits() {
        for (auto i: circuits) {
                i.printInfo_LogicalRam();
        }
}

void Mapper::printCircuitSolutions() {
        for (auto i: circuits) {
                i.printInfo_MappedRam();
        }
}
```

```
void Mapper::printCircuitWidthLogicalRamMap() {
        for (auto i: circuits) {
                i.printInfo_WidthLogicalRamMap();
        }
}
void Mapper::printCircuitAreas() {
        vector<double> data;
        for (auto i: circuits) {
                i.printInfo_Area();
                data.push_back(i.getArea());
        }
        cout << "Geomean: " << geomean(data) << endl;
}

void Mapper::printCircuits_raw() {
        for (auto i: circuits) {
                cout << i.getID() << " " << i.getLogicBlockNum() << endl;
        }
}

void Mapper::printLogicalRam_raw() {
        for (auto i: circuits) {
                for (auto j: i.getLogicalRams()) {
                        cout << i.getID() << " " << j.getID() << " " <<
                        ↪  getLRAMMode_S(j.getMode()) << " " <<
                        ↪  j.getDepth() << " " << j.getWidth() << endl;
                }
        }
}

void Mapper::printCircuitSolutions_raw() {
        for (auto i: circuits) {
                // Assuming stuff comes out in order atm
                for (auto j: i.getMappedRams()) {
                        cout << i.getID() << " ";
                        cout << j.getLogicalRamID() << " ";
                        cout << j.getAddLuts() << " ";
                        cout << "LW " << j.getLogicalRamWidth() << " ";
                        cout << "LD " << j.getLogicalRamDepth() << " ";
                        cout << "ID " << j.getID() << " ";
                        cout << "S " << j.getS() << " ";
```

```
                        cout << "P " << j.getP() << " ";
                        cout << "Type " << j.getType_output() << " ";
                        cout << "Mode " << getLRAMMode_S(j.getMode()) << "
                        ↪  ";
                        cout << "W " << j.getW() << " ";
                        cout << "D " << j.getD();
                        cout << endl;
                }
        }
}
```

```
#ifndef PHYSICALRAM_H_
#define PHYSICALRAM_H_

#include <vector>
#include <map>

#include "LogicalRam.h"

using namespace std;

enum P_RamType {
        LUTRAM = 1,
        BRAM8192,
        BRAM128K,
        BRAM_CUSTOM_1,
        BRAM_CUSTOM_2
};

class PhysicalRam {
private:
        unsigned int max_bits;
        unsigned int max_depth;
        unsigned int max_width;
        P_RamType type;
        double ratio;
        double area;
        vector<unsigned int> possible_widths;
        map<unsigned int, unsigned int> width_to_depths;

public:
        PhysicalRam(unsigned int max_bits_n, unsigned int max_depth_n,
        ↪  unsigned int max_width_n, P_RamType type_n, double ratio);

        // Get functions
        unsigned int getMaxBits() { return max_bits; }
        unsigned int getMaxDepth() { return max_depth; }
        unsigned int getMaxWidth() { return max_width; }
        unsigned int getMaxWidth_mode(L_RamMode mode_n) { return (mode_n ==
        ↪  TrueDualPort) ? max_width/2 : max_width; }
        P_RamType getType() { return type; }
        double getRatio() { return ratio; }
        double getArea() { return area; }
```

```
        vector<unsigned int>& getPossibleWidths() { return possible_widths;
        ↪  }
        map<unsigned int, unsigned int>& getWidthToDepths() { return
        ↪  width_to_depths; }

        // Set functions
        void addPossibleWidth(unsigned int width_n);
        void addToWidthToDepths(unsigned int depth_n, unsigned int width_n);

};


#endif /* PHYSICALRAM_H_ */
```

```cpp
#include <cmath>

#include "PhysicalRam.h"

using namespace std;

PhysicalRam::PhysicalRam(unsigned int max_bits_n, unsigned int max_depth_n,
 ↪  unsigned int max_width_n, P_RamType type_n, double ratio_n) {
        max_bits = max_bits_n;
        max_depth = max_depth_n;
        max_width = max_width_n;
        type = type_n;
        ratio = ratio_n;

        if (type_n == LUTRAM) {
                area = 40000;
        } else {
                area = 9000 + (5 * max_bits_n) + (90 * sqrt(max_bits_n)) +
                 ↪  (600 * 2 * max_width_n);
        }
}

// Get functions

// Set functions
void PhysicalRam::addPossibleWidth(unsigned int width_n) {
        possible_widths.push_back(width_n);
}

void PhysicalRam::addToWidthToDepths(unsigned int depth_n, unsigned int
 ↪  width_n) {
        width_to_depths[width_n] = depth_n;
}
```

```
#include <iostream>
#include <string>
#include <iomanip>
#include <time.h>

#include "Mapper.h"

using namespace std;

enum Architecture_Type {
        Stratix_iv = 1,
        SingleRamType_no_LUTRAM,
        SingleRamType_50_LUTRAM,
        CustomArch_w_LUTRAM
};

void print_help() {
        cout << endl;
        cout << "Usage: ./mapper <architecture_type> <options>" << endl;

        cout << "Architecture type: " << endl;
        cout << "1: Stratix-IV" << endl;
        cout << "2: Single RAM type with no LUTRAM" << endl;
        cout << "3: Single RAM type with 50% LUTRAM" << endl;
        cout << "4: Custom architecture with LUTRAM" << endl;
        cout << endl;

        cout << "Detailed usage:" << endl;
        cout << "./mapper 1" << endl;
        cout << "./mapper 2|3 <max_Kbit> <max_width> <ratio>" << endl;
        cout << "./mapper 4 <lbs_per_lutram> <max_Kbit_1> <max_width_1>
        ↪  <ratio_1> <max_Kbit_2> <max_width_2> <ratio_2>" << endl;
        cout << endl;

        cout << "For 2, 3 and 4, enter details for max_Kbit, max_width and
        ↪  ratio for desired BRAM as well as LUTRAM ratio if applicable."
        ↪  << endl;
        cout << "For 3, LUTRAM specs are same as Stratix-IV" << endl;
        cout << endl;

        cout << "<lbs_per_lutram>: Number of LBs where there's 1 LUTRAM" <<
        ↪  endl;
```

```
        cout << "<max_Kbit>: In number of 1024bits, e.g. 1, 2, 4, = 1K, 2K,
        ↪  4K etc" << endl;
        cout << "<max_width>: In powers of 2 up to <max_Kbit>" << endl;
        cout << "<ratio>: Number of LBs for each BRAM" << endl;
        cout << endl;
        cout << endl;

        exit(-1);
}


int main(int argc, char **argv) {
        /*
                Control panel
        */
        string logical_ram_file = "logical_rams.txt";
        string logical_block_file = "logic_block_count.txt";

        Architecture_Type arch_type = Stratix_iv;

        double ratio_LUT = 1/0.5;
        double ratio_8192 = 10;
        double ratio_128K = 300;

        unsigned int max_bits_custom_1 = 1*1024;
        unsigned int max_depth_custom_1 = 1*1024;
        unsigned int max_width_custom_1 = 32;
        double ratio_custom_1 = 10;
        unsigned int max_bits_custom_2 = 1*1024;
        unsigned int max_depth_custom_2 = 1*1024;
        unsigned int max_width_custom_2 = 32;
        double ratio_custom_2 = 10;

        if (argc < 2) {
                print_help();
        }
        switch (atoi(argv[1])) {
                case 2:
                        arch_type = SingleRamType_no_LUTRAM;

                        if (argc < 5) {
                                print_help();
```

```
                }
                if (atoi(argv[3]) > atoi(argv[2]) * 1024) {
                        print_help();
                }

                max_bits_custom_1 = atoi(argv[2]) * 1024;
                max_depth_custom_1 = max_bits_custom_1;
                max_width_custom_1 = atoi(argv[3]);
                ratio_custom_1 = atoi(argv[4]);

                break;
        case 3:
                arch_type = SingleRamType_50_LUTRAM;

                if (argc < 5) {
                        print_help();
                }
                if (atoi(argv[3]) > atoi(argv[2]) * 1024) {
                        print_help();
                }

                max_bits_custom_1 = atoi(argv[2]) * 1024;
                max_depth_custom_1 = max_bits_custom_1;
                max_width_custom_1 = atoi(argv[3]);
                ratio_custom_1 = (double)atoi(argv[4]);

                break;
        case 4:
                arch_type = CustomArch_w_LUTRAM;

                if (argc < 9) {
                        print_help();
                }
                /* if (atoi(argv[2]) > 100 or atoi(argv[2]) <= 0) {
                        print_help();
                 } */
                if (atoi(argv[4]) > atoi(argv[3]) * 1024) {
                        print_help();
                }
                if (atoi(argv[7]) > atoi(argv[6]) * 1024) {
                        print_help();
                }
```

```
                        // ratio_LUT = 1/((double)atoi(argv[2])/100);
                        ratio_LUT = (double)atoi(argv[2]);

                        max_bits_custom_1 = atoi(argv[3]) * 1024;
                        max_depth_custom_1 = max_bits_custom_1;
                        max_width_custom_1 = atoi(argv[4]);
                        ratio_custom_1 = (double)atoi(argv[5]);

                        max_bits_custom_2 = atoi(argv[6]) * 1024;
                        max_depth_custom_2 = max_bits_custom_1;
                        max_width_custom_2 = atoi(argv[7]);
                        ratio_custom_2 = (double)atoi(argv[8]);

                        break;
                case 1:
                default:
                        break;
        }

        // SingleType_Simple = 1,
        // SingleType_Shared,
        // SingleType_ModifiedCostFunction,
        // SingleType_ModifiedCostFunctionWithSharing
        Solution_Type solution_type =
        ↪  SingleType_ModifiedCostFunctionWithSharing;

        /*
                Switch blocks that controls what physical memory gets
↪  inserted
        */
        bool use_lutram = true;
        bool use_bram8192 = true;
        bool use_bram128k = true;
        bool use_bram_custom_1 = false;
        bool use_bram_custom_2 = false;
        switch (arch_type) {
                case SingleRamType_no_LUTRAM:
                        use_lutram = false;
                        use_bram8192 = false;
                        use_bram128k = false;
                        use_bram_custom_1 = true;
```

```
                break;
        case SingleRamType_50_LUTRAM:
                use_bram8192 = false;
                use_bram128k = false;
                use_bram_custom_1 = true;
                break;
        case CustomArch_w_LUTRAM:
                use_bram8192 = false;
                use_bram128k = false;
                use_bram_custom_1 = true;
                use_bram_custom_2 = true;
                break;
        case Stratix_iv:
        default:
                break;
}

/*
        Contains all the test cases needed
*/
// Base: Stratix-IV
// Rams: LUTRAM, BRAM8192, BRAM128K


unsigned int k = 6;
unsigned int N = 10;
double area = 35000;
Architecture arch(k, N, area);

if (use_lutram) {
        unsigned int max_bits_LUT = 640;
        unsigned int max_depth_LUT = 64;
        unsigned int max_width_LUT = 20;
        PhysicalRam new_physical_ram_LUT(max_bits_LUT,
        ↪  max_depth_LUT, max_width_LUT, LUTRAM, ratio_LUT);
        new_physical_ram_LUT.addPossibleWidth(10);
        new_physical_ram_LUT.addPossibleWidth(20);
        new_physical_ram_LUT.addToWidthToDepths(64, 10);
        new_physical_ram_LUT.addToWidthToDepths(32, 20);
        arch.addPhysicalRam(new_physical_ram_LUT);
}
```

```
if (use_bram8192) {
        unsigned int max_bits_8192 = 8192;
        unsigned int max_depth_8192 = 8192;
        unsigned int max_width_8192 = 32;
        PhysicalRam new_physical_ram_8192(max_bits_8192,
        ↪  max_depth_8192, max_width_8192, BRAM8192, ratio_8192);
        for (unsigned int i_width = 1; i_width <= max_width_8192;
        ↪  i_width *= 2) {
            new_physical_ram_8192.addPossibleWidth(i_width);
            new_physical_ram_8192.addToWidthToDepths(max_bits_8192/i_width,
            ↪  i_width);
        }
        arch.addPhysicalRam(new_physical_ram_8192);
}

if (use_bram128k) {
        unsigned int max_bits_128K = 131072; // 128*1024
        unsigned int max_depth_128K = 131072;
        unsigned int max_width_128K = 128;
        PhysicalRam new_physical_ram_128K(max_bits_128K,
        ↪  max_depth_128K, max_width_128K, BRAM128K, ratio_128K);
        for (unsigned int i_width = 1; i_width <= max_width_128K;
        ↪  i_width *= 2) {
            new_physical_ram_128K.addPossibleWidth(i_width);
            new_physical_ram_128K.addToWidthToDepths(max_bits_128K/i_width,
            ↪  i_width);
        }
        arch.addPhysicalRam(new_physical_ram_128K);
}

if (use_bram_custom_1) {
        PhysicalRam new_physical_ram_custom_1(max_bits_custom_1,
        ↪  max_depth_custom_1, max_width_custom_1, BRAM_CUSTOM_1,
        ↪  ratio_custom_1);
        for (unsigned int i_width = 1; i_width <=
        ↪  max_width_custom_1; i_width *= 2) {
            new_physical_ram_custom_1.addPossibleWidth(i_width);
            new_physical_ram_custom_1.addToWidthToDepths(max_bits_custom_1/i
            ↪  i_width);
        }
        arch.addPhysicalRam(new_physical_ram_custom_1);
}
```

```
        if (use_bram_custom_2) {
                PhysicalRam new_physical_ram_custom_2(max_bits_custom_2,
                ↪  max_depth_custom_2, max_width_custom_2, BRAM_CUSTOM_2,
                ↪  ratio_custom_2);
                for (unsigned int i_width = 1; i_width <=
                ↪  max_width_custom_2; i_width *= 2) {
                        new_physical_ram_custom_2.addPossibleWidth(i_width);
                        new_physical_ram_custom_2.addToWidthToDepths(max_bits_custom_2/i
                        ↪  i_width);
                }
                arch.addPhysicalRam(new_physical_ram_custom_2);
        }
        // arch.printInfo();

        /*
                Mapping computation
        */

    /* clock_t tStart = clock();
        unsigned int run_count = 20;

        for (unsigned int i = 0; i < run_count; i++) {
                Mapper mapper(arch);
                mapper.initCircuits(logical_ram_file, logical_block_file);
                mapper.setSolutionType(solution_type);
                mapper.mapAll_Wrapper();
        }

        printf("Average time taken for %d runs: %.2fs\n", run_count,
↪   (double)(clock() - tStart)/CLOCKS_PER_SEC/run_count); */

        Mapper mapper(arch);
        mapper.initCircuits(logical_ram_file, logical_block_file);
        mapper.setSolutionType(solution_type);
        mapper.mapAll_Wrapper();

        // mapper.computeCircuitAreas();

        /*
                Print/Output
        */
```

```
        // mapper.printCircuits();
        // mapper.printCircuitSolutions();
        // mapper.printCircuitWidthLogicalRamMap();
        // mapper.printCircuitAreas();

        // mapper.printCircuits_raw();
        // mapper.printLogicalRam_raw();
        mapper.printCircuitSolutions_raw();

        return 0;
}
```

```
all:
	g++ -Wall -c -std=c++17 -o main.o main.cpp
	g++ -Wall -c -std=c++17 -o Mapper.o Mapper.cpp
	g++ -Wall -c -std=c++17 -o Circuit.o Circuit.cpp
	g++ -Wall -c -std=c++17 -o LogicalRam.o LogicalRam.cpp
	g++ -Wall -c -std=c++17 -o PhysicalRam.o PhysicalRam.cpp
	g++ -Wall -c -std=c++17 -o MappedRam.o MappedRam.cpp
	g++ -Wall -c -std=c++17 -o Arch.o Arch.cpp
	g++ -Wall -c -std=c++17 -o helper.o helper.cpp
	g++ -o mapper main.o Mapper.o Circuit.o LogicalRam.o PhysicalRam.o
	↪  MappedRam.o Arch.o helper.o

clean:
	rm -f main.o Mapper.o Circuit.o LogicalRam.o PhysicalRam.o
	↪  MappedRam.o Arch.o helper.o
	rm -f mapper
```