RAM Mapping and RAM Architecture

Algorithm Description

For this assignment, I formulated the RAM Mapping problem into a Integer linear Problem (ILP) and solve the ILP with a python library call PuLP. PuLP is an open-source solver and can be easily installed through "pip3 install PuLP" command.

In terms of creating the ILP, I started with the objective function and decided to only implement the basic version of RAM Mapping, which is only one type of physical RAM is allowed for a given logical RAM. According the lab manual, the objective function can be defined as:

$$Area = \bar{A}_{LB} N_{LB} + \sum_{i=0}^{N_{b-type}} A^{i}_{bram} N^{i*}_{bram}$$

Where $\bar{A}_{LB}$ is the average logic block (LB) size. This is determined by the ratio of LB with no LUTRAM support and LB with LUTRAM support. For the Stratix-IV-like architecture, this value should be (35000 + 40000)x50% = 37500. N_{LB} is the final number of logic blocks required by the limiting resource. Then, the summation symbol implies we need to sum areas of all BRAM types. Each BRAM area is calculated by multiplying a single BRAM's area of type i (A^{i}_{bram}) with number of BRAMs of type i (N^{i*}_{bram}). One thing to note, the number of N^{i*}_{bram} is purely based on N_{LB} and its availability, which is the number of LBs required for a type i BRAM to be available.

N_{LB} value is determined by the limiting resource. This means N_{LB} can only take the maximum logic blocks required by total logic blocks, LUTRAM blocks and BRAM blocks. These can be easily expressed as constraints in ILP:

$$N_{LB} >= N^{circuit}_{LB} + N^{extra}_{LB} + N_{lutram}$$

$$N_{LB} >= C^{lutram} N_{lutram}$$

$$N_{LB} >= \{C_0 N^{0}_{bram}, C_1 N^{1}_{bram}, ..., C_N N^{N}_{bram}\}$$

$N^{circuit}_{LB}$ is a constant given by the logic_block_count.txt file. N^{extra}_{LB} and N_{lutram} are number of extra logic blocks and number of LUTRAMs. For a given circuit, the value of N^{extra}_{LB} and N_{lutram} is calculated by summing extra LBs and LUTRAMs used by each logic RAM. C^{lutram} and C_i are defined as the number of LBs required for one LUTRAM or BRAM available. For the Stratix-IV-like architecture, C^{lutram} is 2, the reciprocal of 50%. C_0 is 10 for 8K BRAM and C_1 is 300 for 128K BRAM. N^{i}_{bram} is the total number of type i BRAM used for a given circuit.

For the second half of the objective function, we should focus on finding the number of BRAMs since area of each BRAM type can be easily pre-computed according to the BRAM size and maximum width. Finding the number of BRAMs is also a summation process, where we sum BRAMs used by each logical RAM based on their types.

Then, the problem becomes how do we compute the physical rams used by each logical RAM in a circuit? In the ILP, each logical RAM is represented by a set of boolean variables. Those variables represent whether the logic RAM chooses one type of physical RAM or not. Because only one type of physical RAM is allowed, we also need to add the constraint that sum of three variables equals to 1 to our ILP solver. The mathematical formulas can be expressed as:

$$LogicRAM = \{P_{lutram}, P_{bram-0}, P_{bram-1}, ..., P_{bram-N}\}, P = \{1, 0\}$$

$$\sum LogicRAM = 1$$

After that, we can compute the best width and depth configuration of each physical RAM among all available configurations. With best configuration confirmed, we can easily calculate the number of physical RAM in parallel, in series and extra LUTs required. Then, N_{lutram}, N^i_{bram} and N^{extra}_{lut} can be written as:

$$N_{lutram} = \sum_{k=0}^{logicRAMs} P^k_{lutram}$$

$$N^i_{bram} = \sum_{k=0}^{logicRAMs} P^k_{bram-i}$$

$$N^{extra}_{lut} = \sum_{k=0}^{logicRAMs} N^{extra-k}_{lut}$$

Where $N^{extra-k}_{lut}$ is the extra LUTs used for a logical RAM in a given circuit and N^{extra}_{lut} is the total number of extra LUTs used in a given circuit.

One last step to take care of is the conversion between LUTs to LBs, and N_{LB} to N^{i*}_{bram}. For the given LB structure, there are 10 LUTs in one LB. Therefore, a constraint should be added for this conversion:

$$\frac{N^{extra}_{lut}}{10} + 10 >= N^{extra}_{LB} >= \frac{N^{extra}_{lut}}{10}$$

For a known N_{LB}, N^{i*}_{bram} value is based on it's availability:

$$\frac{N_{LB}}{availability^i} >= N^{i*}_{bram} >= \frac{N_{LB}}{availability^i} - availability^i$$

Now, the RAM Mapping problem is successfully converted into an ILP problem.

Algorithm Complexity

The PuLP library uses an algorithm called COIN-OR Branch and Cut (CBC), which is implemented based on the Simplex algorithm. Although I can not find any information about CBC complexity, a rough complexity description of the Simplex algorithm can be found online [2]. In average cases, the Simplex algorithm complexity is in polynomial growth. In big-O notation, it can be written as $O(n^k)$, where n is the number of variables and k is a constant value. In worst cases, the complexity becomes exponential in time, which is $O(k^n)$. However, we can assume the complexity is always $O(n^k)$ because CBC applied heuristics and branch-cut methods [1] to reduce complexity and possibility of worst cases.

Results for the Stratix-IV-like Architecture

Command line to run the program: **python3 main.py**

*Make sure logic_block_count.txt, logical_rams.txt, and ram_config.txt exist. And the RAM configurations in ram_config.txt match your expectation.

Table 1: RAM mapping results for example Stratix-IV-like architecture

Circuit #	LUTRAM	8K BRAM	128K BRAM	Regular LBs	Required LBs	Total Area
Circuit 0	753	369	12	2941	3694	1.8436 e8
Circuit 1	1136	413	13	2988	4130	2.0581 e8
Circuit 2	0	62	0	1836	1836	0.9162 e8
Circuit 3	0	90	0	2808	2808	1.4000 e8
Circuit 4	154	806	26	7907	8061	4.0222 e8
Circuit 5	0	312	0	3692	3692	1.8429 e8
Circuit 6	0	185	6	1853	1853	0.9245 e8
Circuit 7	271	422	14	3947	4220	2.1090 e8
Circuit 8	92	543	18	5342	5434	2.7151 e8
Circuit 9	0	33	0	1636	1636	0.8134 e8
Circuit 10	560	203	6	1467	2030	1.0083 e8
Circuit 11	76	140	4	1329	1405	0.6961 e8
Circuit 12	0	43	0	1632	1632	8.1191 e8
Circuit 13	0	4	10	4491	4491	2.2367 e8
Circuit 14	125	195	6	1826	1951	0.9709 e8
Circuit 15	0	154	0	1956	1956	0.9728 e8
Circuit 16	0	88	0	2181	2181	1.0879 e8
Circuit 17	0	61	0	1165	1165	0.5744 e8
Circuit 18	0	156	6	2036	2036	1.0105 e8
Circuit 19	243	248	8	2236	2480	1.2375 e8
Circuit 20	12	270	9	2679	2700	1.3497 e8
Circuit 21	0	76	0	5100	5100	2.5495 e8
Circuit 22	536	296	9	2429	2965	1.4724 e8
Circuit 23	0	249	2	5230	5230	2.6108 e8
Circuit 24	30	435	14	4325	4355	2.1722 e8
Circuit 25	0	112	0	4517	4517	2.2569 e8
Circuit 26	330	228	7	1453	2280	1.1347 e8
Circuit 27	0	20	0	1496	1496	0.7389 e8
Circuit 28	301	236	7	2063	2364	1.1739 e8
Circuit 29	66	309	10	3025	3091	1.5425 e8
Circuit 30	0	215	0	5419	5419	2.7076 e8
Circuit 31	0	80	0	4347	4347	2.1682 e8
Circuit 32	812	454	15	3716	4540	2.2684 e8
Circuit 33	26	403	10	4006	4032	2.0117 e8
Circuit 34	51	416	14	1705	4200	2.0996 e8

Circuit #	LUTRAM	8K BRAM	128K BRAM	Regular LBs	Required LBs	Total Area
Circuit 35	52	143	4	1376	1430	0.7083 e8
Circuit 36	685	458	18	1761	5400	2.6995 e8
Circuit 37	0	48	0	14969	14969	7.4746 e8
Circuit 38	0	275	10	3202	3202	1.5948 e8
Circuit 39	264	211	7	1845	2110	1.0545 e8
Circuit 40	0	179	0	3060	3060	1.5280 e8
Circuit 41	374	241	8	2035	2410	1.2045 e8
Circuit 42	0	90	0	1337	1337	0.6638 e8
Circuit 43	111	132	4	1212	1323	0.6576 e8
Circuit 44	0	196	6	2114	2114	1.0560 e8
Circuit 45	0	1	3	2782	2782	1.3882 e8
Circuit 46	564	398	13	3421	3985	1.9892 e8
Circuit 47	0	55	0	1439	1439	0.7117 e8
Circuit 48	0	558	22	6875	6875	3.4286 e8
Circuit 49	1237	1312	43	11883	13120	6.5525 e8
Circuit 50	0	580	0	11884	11884	5.9353 e8
Circuit 51	0	393	14	4204	4204	2.1011 e8
Circuit 52	0	641	0	9603	9603	4.8002 e8
Circuit 53	0	816	0	10817	10817	5.4063 e8
Circuit 54	0	885	0	10903	10903	5.4473 e8
Circuit 55	157	1049	34	10341	10498	5.2388 e8
Circuit 56	0	342	4	4578	4578	2.2856 e8
Circuit 57	0	466	0	7145	7145	3.5644 e8
Circuit 58	0	762	10	7700	7700	3.8436 e8
Circuit 59	4344	1798	59	13626	17980	8.9803 e8
Circuit 60	0	558	0	20371	20371	10.1758 e8
Circuit 61	0	1887	63	15079	18900	9.4482 e8
Circuit 62	43	493	16	4888	4931	2.4612 e8
Circuit 63	0	455	11	4846	4846	2.4207 e8
Circuit 64	843	1120	37	10451	11294	5.6400 e8
Circuit 65	0	357	0	12721	12721	6.3558 e8
Circuit 66	0	597	20	6310	6310	3.1541 e8
Circuit 67	694	505	17	2609	5100	2.5495 e8
Circuit 68	0	192	0	4850	4850	2.4231 e8
CPU runtime on UG machine					38.30 Sec	
Geometric average of area					2.00726 e8	

Results and Analysis without LUTRAM

Table 2: Results without LUTRAM

BRAM size	Max Width	LBs/BRAM	Geo. Avg. Area
1 kbit	8	1.4	2.3445 e8
2 kbit	8	1.9	2.2720 e8
4 kbit	16	3.4	2.1575 e8
8 kbit	32	6.3	2.1403 e8
16 kbit	32	7.5	2.2202 e8
32 kbit	64	12	2.4255 e8
64 kbit	64	16	2.8389 e8
128 kbit	128	26	3.5130 e8

At first glance of table 2, we can notice the the max width and LBs/BRAM generally increases with increasing BRAM size. However, the areas don't follow the consistent decreasing trend. In the beginning, areas are dropping, reaching a minimal value for the 8k BRAM and increasing afterward. The trend in areas suggests a trade-off is occurring. Small BRAM size comes with small areas and flexibility, but it also brings high area per bit cost and more extra circuitry to implement large logical RAMs. On the other hand, large BRAM size has low area per bit cost and requires less extra circuitry for large logical RAMs, but it costs more areas and introduces lots of bit wasting for small logical RAMs. From the table 2, 8k BRAM balances the trade-off well and implements the design with smallest area.

Results and Analysis with LUTRAM

Table 3: Results with LUTRAM

BRAM size	Max Width	LBs/BRAM	Geo. Avg. Area
1 kbit	8	1.8	2.2817 e8
2 kbit	8	2.3	2.1591 e8
4 kbit	8	4	2.0409 e8
8 kbit	16	7.2	1.9908 e8
16 kbit	32	13.5	2.0032 e8
32 kbit	64	24	2.0693 e8
64 kbit	64	35	2.1927 e8
128 kbit	128	58	2.4096 e8

With LUTRAM becomes available, the major difference compares to table 2 is all areas are smaller in table 3. This is because the RAM Mapper now has two options to implement each logical RAM to reduce area. In addition, we can see LBs/BRAM values increase a lot. With LUTRAM taking some part of logical RAMs, it's reasonable that BRAMs can be less dense

on the FPGA. 8k Brams still has the best area, but we can notice the 16k BRAM's area only lose by a small difference. If LUTRAM has larger size or more width options, 16k BRAM may outperform 8k BRAM.

Find a Better Organization of RAM Blocks

For the custom achitecture, I used three types of RAM blocks with one of them is LUTRAM.
LUTRAM: 640 bits size, 20 bits max width, 5.5 LBs/LUTRAM.

First BRAM: 8192 bits size, 16 bits max width, 7.5 LBs/BRAM.

Second BRAM: 32768 bits size, 16 bits max width, 250 LBs/BRAM.

The geometric average area is 1.9582e+08.

The original 50% LUTRAM ratio is definitely too dense for the test circuits. Many LUTRAMs are used but taking lots of area. By cuting the ratio, the final area should decrease. 8k BRAM is kept unchanged because it gives the best area in table 2 and 3. The area also decrease slightly by replacing the 128k BRAM in the original design with 32k BRAM. The reason may due to 32k BRAM is more flexible and the flexibility is a better trade-off than the low area per bit in 128 BRAM. Finally, I tuned the ratio of LUTRAM and BRAMs a little bit to further reduce the area. With all these efforts, the area only dropped by a small percent compared to the original architecture. I do believe the design is reaching a diminishing marginal range.

References

[1] Robin Lougee-Heimer John Forrest. Cbc user guide. Accessed: 2023-11-26.

[2] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.

Appendix A: RAM Mapper Source Code

main.py

```
import configparser
from math import sqrt
from pulp import *
import numpy as np
from LogicRamType import LogicRamType
import os, time

mode_str2int = {
    'ROM': 0,
    'SinglePort': 1,
    'SimpleDualPort': 2,
    'TrueDualPort': 3,
}
mode_int2str = ['ROM', 'SinglePort', 'SimpleDualPort' ,'TrueDualPort']


def read_circuits():
    path = "logical_rams.txt"
    f = open(path)
    Num_Circuits = int(f.readline().split()[1])
    f.readline() # skip categories

    circuits = [[] for _ in range(Num_Circuits)]
    for line in f.readlines():
        circuitID, RamID, Mode, Depth, Width = line.split()
        circuits[int(circuitID)].append({'mode': mode_str2int[Mode], 'depth':
            int(Depth), 'width': int(Width)})
    f.close()

    path = "logic_block_count.txt"
    f = open(path)
    f.readline() #skip

    block_count = [0 for _ in range(Num_Circuits)]
    for line in f.readlines():
        circuitID, blocks = line.split()
        block_count[int(circuitID)] = int(blocks)

    f.close()
    return circuits, block_count

def read_ram_config():
    config = configparser.ConfigParser(inline_comment_prefixes="#")
    config.read('ram_config.txt')

    # read lutram configuration
    lutram_config = {}
```

```
lutram_config['type'] = config['LUTRAM']['type']
if lutram_config['type'] == 'LUTRAM':
    # achitecture with LUTRAM
    lutram_config['bit_size'] = int(config['LUTRAM']['bit_size'])
    lutram_config['availability'] = float(config['LUTRAM']['availability'])
    lutram_config['mode'] = np.array([int(val) for val in config['LUTRAM']['mode'].split(',')])
    lutram_config['width'] = np.array([int(val) for val in config['LUTRAM']['width'].split(',')])
    lutram_config['depth'] = lutram_config['bit_size'] // lutram_config['width']
    avg_logic_block_area = (35000 * (lutram_config['availability'] -1) + 40000) / lutram_config['availability']
    # bit size, width, depth check
    for val in lutram_config['width']*lutram_config['depth']:
        if val != lutram_config['bit_size']:
            print("LURAM config: Error in bit_size or width. Please double check")
            exit()

else:
    # achitecture with no LUTRAM
    lutram_config['bit_size'] = -1
    lutram_config['availability'] = -1
    lutram_config['mode'] = np.array([-1])
    lutram_config['width'] = np.array([-1])
    lutram_config['depth'] = np.array([-1])
    avg_logic_block_area = 35000

# read bram configuration, there are multiple bram configurations
bram_configs = []
for section in config.sections():
    if section != 'LUTRAM':
        if(config[section]['type'] != 'BRAM'):
            continue
        else:
            bram_config = {}
            bram_config['type'] = config[section]['type']
            bram_config['bit_size'] = int(config[section]['bit_size'])
            bram_config['availability'] = float(config[section]['availability'])
            bram_config['mode'] = np.array([int(val) for val in config[section]['mode'].split(',')])
            bram_config['width'] = np.array([int(val) for val in config[section]['width'].split(',')])
            bram_config['depth'] = bram_config['bit_size'] // bram_config['width']
            area = 9000 + 5*bram_config['bit_size'] + 90*sqrt(bram_config['bit_size']) + 1200*bram_config['width'][-1]
```

```
                bram_config['area'] = area
                bram_configs.append(bram_config)

                # bit size, width, depth check
                for val in bram_config['width']*bram_config['depth']:
                    if val != bram_config['bit_size']:
                        print("BRAM config: Error in bit_size or width. Please
                            double check")
                        exit()

    return lutram_config, bram_configs, avg_logic_block_area

def solve_circuit(circuit_id, circuit, block_count, lutram_config,
    bram_configs, avg_logic_block_area):
    LB_size = 10

    problem = LpProblem("RamMappingProblem", LpMinimize)

    # Objective: minimize total area
    # total area is given by : total logic blocks * area + total brams A *
        area + total brams B * area + ...
    # Note about limiting resource, constraints needed
    final_logic_block = LpVariable('Final_logic_block', 0, None, LpInteger)
    final_brams = [LpVariable(f"Final_bram_{i}", 0, None, LpInteger) for i in
        range(len(bram_configs))]
    problem += avg_logic_block_area*final_logic_block + lpSum([bram_configs[i
        ]['area']*final_brams[i] for i in range(len(bram_configs))]), "
        MinimizeArea"

    # Start adding constraints
    # three types of factors, the maximum one is the limiting factor
    # logic blocks required by circuit LB + extra LB + LUTRAM, logic blocks
        required by LUTRAM, logic blocks required by each BRAM

    # add constraint for each logic ram
    logic_rams = [LogicRamType(circuit_id, logic_ram_id, circuit[logic_ram_id
        ]['width'], circuit[logic_ram_id]['depth'], \
            circuit[logic_ram_id]['mode'], lutram_config, bram_configs,
                problem) for logic_ram_id in range(len(circuit))]

    total_lutram_count = lpSum([logic_ram.lutram_count for logic_ram in
        logic_rams])
    total_bram_counts = []
    for i in range(len(bram_configs)):
        total_bram_counts.append(lpSum([logic_ram.bram_counts[i] for logic_ram
            in logic_rams]))
    total_extra_lut = lpSum([logic_ram.extra_lut for logic_ram in logic_rams])

    # add constraint for [logic blocks required by circuit LB + extra LB +
        LUTRAM]
```

```
    final_extra_LB = LpVariable('Final_extra_LB', 0, None, LpInteger) #
        convert extra_lut to extra LB
    problem += final_extra_LB >= total_extra_lut/LB_size # - LB_size
    problem += final_extra_LB <= total_extra_lut/LB_size + LB_size

    problem += final_logic_block >= block_count + final_extra_LB +
        total_lutram_count

    # add constraint for [logic blocks required by LUTRAM]
    problem += final_logic_block >= total_lutram_count*lutram_config['
        availability']

    # add constraint for [logic blocks required by each BRAM]
    for i in range(len(bram_configs)):
        problem += final_logic_block >= total_bram_counts[i]*bram_configs[i]['
            availability']
        # calculate the final brams based on the final logic blocks
        problem += final_brams[i]*bram_configs[i]['availability'] >=
            final_logic_block - bram_configs[i]['availability']
        problem += final_brams[i]*bram_configs[i]['availability'] <=
            final_logic_block # + bram_configs[i]['availability']

    problem.solve(PULP_CBC_CMD(msg=0, timeLimit=7))
    # print("Status:", LpStatus[problem.status])
    # for v in problem.variables():
    #     if("Final" in v.name):
    #         print(v.name, "=", v.varValue)
    # print("Final_lutram = ", value(total_lutram_count))
    area = avg_logic_block_area*final_logic_block.varValue + np.sum([
        bram_configs[i]['area']*final_brams[i].varValue for i in range(len(
        bram_configs))])
    return area, logic_rams

def generate_mapping_file(f, circuit_id, logic_rams):
    f.write(f"//----------Circuit {circuit_id
        }----------\n")
    for logic_ram_id, logic_ram in enumerate(logic_rams):
        S, P, extra_lut, TYPE, W, D = logic_ram.final_config()
        f.write(f'{circuit_id} {logic_ram_id} {extra_lut} LW {logic_ram.width}
            LD {logic_ram.depth} ID {logic_ram_id} S {S} P {P} Type {TYPE}
            Mode {mode_int2str[logic_ram.mode]} W {W} D {D} \n')


def main():
    circuits, block_count = read_circuits()
    lutram_config, bram_configs, avg_logic_block_area = read_ram_config()

    if os.path.exists('mapping.txt'):
        os.remove('mapping.txt')
    f = open('mapping.txt', 'w')
```

```
    areas = []

    time_start = time.time()

    # iterate through all 69 circuits
    for i in range(len(circuits)):
        area, logic_rams = solve_circuit(i, circuits[i], block_count[i],
            lutram_config, bram_configs, avg_logic_block_area)
        print(f"Circuit {i} finished with area: {area}")
        areas.append(area)
        generate_mapping_file(f, i, logic_rams)

    print("Geometric Average:", '{:.4e}'.format(np.exp(np.log(areas).mean()))
        )
    print("Total time spent:", round(time.time() - time_start, 2), 'Sec')

    f.close()

main()
```

LogicRamType.py

```
from pulp import *
import numpy as np

class LogicRamType:
    def __init__(self, circuit_id, logic_ram_id, width, depth, mode,
        lutram_config, bram_configs, problem):
        self.circuit_id = circuit_id
        self.logic_ram_id = logic_ram_id
        self.width = width
        self.depth = depth
        self.mode = mode

        self.lutram_best_config = self.optimal_configuration(lutram_config)
        self.bram_best_configs = [self.optimal_configuration(b_config) for
            b_config in bram_configs]

        # binary variables to select which type of RAM is used
        if(self.lutram_best_config[0]):
            self.logical_lutram = LpVariable(f"logical_lutram_{circuit_id}_{
                logic_ram_id}", cat='Binary')
        else:
            self.logical_lutram = 0 # prun invalid variable

        self.logical_brams = []
        for i in range(len(bram_configs)):
            if(self.bram_best_configs[0]):
                self.logical_brams.append(LpVariable(f"logical_bram_{
```

```
                circuit_id}_{logic_ram_id}_{i}", cat='Binary'))
            else:
                self.logical_brams.append(0) # prun invalid variable

        # only one type of ram can be used
        problem += (self.logical_lutram + lpSum(self.logical_brams)) == 1

        # calculate number of physical lutram, brams, extra LUTs based on
            which RAM is used.
        self.lutram_count = self.logical_lutram * self.lutram_best_config[0]
        self.bram_counts = [self.logical_brams[i] * self.bram_best_configs[i
            ][0] for i in range(len(self.bram_best_configs))]
        self.extra_lut = self.logical_lutram * self.lutram_best_config[1] +
            lpSum([self.logical_brams[i] * self.bram_best_configs[i][1] for i
            in range(len(bram_configs))])


    # find best configuration (i.e. width, depth, parallel_count, series_count
        )for a given ram
    # keep in mind, connect in parallel requires no extra LUT
    # connect in series requires extra LUI
    # And maximum number of connection in series is 16
    # return parallel_count, series_count, optimal width, optimal depth,
        number of extra LUT
    def optimal_configuration(self, ram_config):
        if self.mode not in ram_config['mode']:
            return 0, 0, 0, 0, 0, 0
        range = len(ram_config['width']) if self.mode != 3 else len(ram_config
            ['width']) - 1
        parallel_list = -(-self.width // np.array(ram_config['width'][:range])
            ) # round up
        series_list = -(-self.depth // np.array(ram_config['depth'][:range]))
            # round up
        idx = np.argmin(parallel_list*series_list)
        parallel_count = parallel_list[idx]
        series_count = series_list[idx]
        if series_count >= 16:
            return 0, 0, 0, 0, 0, 0   # no optimal configuration

        extra_lut = self.compute_extra_lut(series_count, self.width)
        if(self.mode == 3):
            extra_lut *= 2 # double for TrueDualPort mode

        return parallel_count*series_count, extra_lut, parallel_count,
            series_count, ram_config['width'][idx], ram_config['depth'][idx]

    # connect in series requires extra LUTs
    # one is log2(R) : R decoder
    # one is R : 1 MUX with [width] number of copies
    def compute_extra_lut(self, series_count, width):
```

```
        if(series_count == 1):
            return 0 # no extra lut needed

        if series_count == 2:
            decoder_count = 1 # special case
        else:
            decoder_count = series_count

        if(series_count <= 4):
            R1mux_count = 1
        elif(series_count <= 8):
            R1mux_count = 3
        elif(series_count <= 12):
            R1mux_count = 4
        elif(series_count <= 16):
            R1mux_count = 5

        mux_count = width * R1mux_count
        return decoder_count + mux_count

    # return final configuration in format: S P extra_lut TYPE W D
    def final_config(self):
        if(value(self.logical_lutram)):
            return self.lutram_best_config[3], self.lutram_best_config[2],
                self.lutram_best_config[1], 1, self.lutram_best_config[4], self
                .lutram_best_config[5]
        else:
            for i in range(len(self.logical_brams)):
                if(value(self.logical_brams[i])):
                    return self.bram_best_configs[i][3], self.
                        bram_best_configs[i][2], self.bram_best_configs[i][1],
                        i+2, self.bram_best_configs[i][4], self.
                        bram_best_configs[i][5]

        return 0 # shouldn't get here
```