ECE1756 Assignment 3 Written Report

Louis Chen – 1000303502

This report outlines the architecture overview, algorithms used, and evaluations of benchmark circuits using the developed RAM Mapper application. The report uses "RAM Mapper", "the application", and "the program" interchangeably, and so as "logic blocks per RAM block" and "respect ratio".

Section I – Algorithm Description

This section illustrates the 4 main algorithms deployed to achieve minimization on total area and number of RAMs used.

1) Minimum Area Estimate (or Area Underestimation)

The first algorithm to optimize area usage is to estimate, given a logical RAM block, using what physical RAM block can result in minimum area consumption before doing the actual mapping stage. The procedures to do so are shown as follows.

a. Iterate over every logical RAM in the input file "logical_rams.txt"
b. Compute RAM size by multiplying its depth by width
c. Take the computed RAM size, calculate the number of physical RAMs needed to map the logical RAM, for all available physical RAM types (such as LUTRAMs, M8KRAMs, M128KRAMs, etc)
d. Calculate the area consumptions for all physical RAM types using the equations provided in the handout
e. Rank the areas in ascending order, and try to map the logical RAM using the first physical RAM type without allocating additional spare logic blocks; if there are not enough type 1 physical RAMs available, then try with the next in the sequence, and so forth
f. If the numbers of available physical RAMs for all physical RAM types are insufficient, then recalculate area consumptions for all physical RAM types by considering areas of additional added spare logic blocks
g. Rank the areas in ascending order, and try to map the logical RAM using the first physical RAM type by allocating additional spare logic blocks; if using type 1 physical RAM fails to allocate the block, try with the next type, and so forth

If the steps appear unclear, consider the following scenario. For instance, there exists a logical RAM of dimension depth 1024 x width 64. The total RAM size will be 1024 x 64 = 64 kbits. Say available physical RAM types are M8KRAM (8192 bits) and M128KRAM (131072 bits). The application first calculates number of physical blocks needed: it needs 8 M8KRAMs or 1 M128KRAM to map. Now calculate the areas: mapping with M8KRAMs consumes X unit areas, and mapping with M128KRAMs consumes Y unit areas. Say X < Y, then the application will first map this logical RAM with M8KRAM while not allocating additional logic blocks; if it finds resources run out, it will then try mapping with M128KRAM while not allocating additional logic blocks. If both trials fail to map, the application will recalculate area consumptions by taking the area caused by additional logic blocks into account. Say M8KRAM has logic block ratio 10:1, and M128KRAM has 300:1. Area for M8KRAM will be 8 x 10 * (35000 + 40000) / 2 + X, denoted as F. Area for M128KRAM will be 1 * 300 * (35000 + 40000) / 2 + Y, denoted as G. Say F < G, then

the application will try to allocate the block using M8KRAM by allowing additional logic blocks to be allocated. If for any reason this step fails, the application will do the same using M128KRAM.

Moreover, this algorithm only gives the RAM Mapper an estimate of which block to use to minimize area. However, the actual mapped RAM may consume more areas due to restricted dimensions of the physical RAM. For example, the algorithm figures that, for the logical RAM of dimension 10 words by 32 bits, it is best to map it with 1 LUTRAM which has RAM size of 640 bits. However in reality, it will need 2 LUTRAMs in parallel, where each LUTRAM is of dimension 32 words by 20 bits, and thus will result in more area consumption. Nevertheless, if one relates this "underestimate" of area to the heuristic cost (underestimate cost from current node to target node) in the A* algorithm, he/she will accept this approach since the underestimation of area indeed tunes the search space towards optimality.

2) RAM Mapping Algorithm

For every logical RAM block with a determined physical RAM type to map with, the application first checks whether the logical RAM can be mapped using maximum physical RAM depth in less than, or equal to, 16 blocks. If not, then it will try the next physical RAM type in the efficiency list. If yes, then there exist two cases to investigate. Case 1 is when logical RAM depth is larger than maximum physical RAM depth. Case 2 is the opposite.

For case 1, the application will always map the logical RAM using the minimum physical RAM depth that is larger than the logical RAM depth. The reason is to elongate the width per physical RAM block (i.e. to minimize the number of columns of physical RAMs needed). By doing so, the logical RAM will always require 0 additional LUTs for decoder and mux, since number of rows is always 1. The trade-off here is the potential waste of memory due to inefficient RAM dimension sizing. However, the application obtains good results with this approach, and therefore will stick with it.

For case 2 where the logical RAM must be partitioned into multiple rows, the application iterates all possible physical RAM depths. Throughout the loop, it keeps track of the physical depth which results in the largest "logical RAM depth mod physical RAM depth" (i.e. the physical depth which results in the smallest waste of memory in the vertical direction) while not violating the greater-than-16-blocks-in-series mapping constraint. It then uses the candidate physical depth to calculate corresponding physical width and number of rows/columns and thus maps the logical RAM using these variables. The intuition here is that by minimizing the waste in the vertical direction, the algorithm can hopefully minimize the overall waste of memory, given physical RAM depth is always order of magnitude larger than physical RAM width; physical depth spans from 8192 to 256, while physical width spans from 1 to 32, respectively. Thus, by prioritizing waste minimization in the vertical direction, the application seeks to reduce the overall waste of mapping, since horizontal (number of columns in parallel) mapping can be done in way smaller chunks which lead to smaller waste of unused memory.

3) Dynamic Allocation of RAM Blocks

The dynamic allocation algorithm is to dynamically modify the maximum number of allowed LUTRAMs, M8KRAMs, M128KRAMs, or other custom-defined physical RAMs regarding newly added decoders, multiplexers, and extra spare logic blocks allocated in resource run-out scenario.

By doing so, it is ensured that every logical RAM mapping iteration is exposed to the maximum available physical RAMs, assuming the sequence of mapping all logical RAMs is fixed; in other words, every mapping iteration is an optimal subproblem, assuming the application is given a fixed order of logical RAM mapping sequence.

An add-on feature to this algorithm is: for every logical RAM mapping iteration, if the algorithm in part 1 shows that using M8KRAM optimizes area consumption but there are not enough M8KRAMs, the application will not naively allocate 10 times the number of M8KRAMs required (call this X) plus the M8KRAMs themselves. Instead, the application will first subtract X by the number of currently available M8KRAMs (call the result Y), and calculate (Y - 1) * 10 + (10 – current_total_used_logic_tiles mod 10). For example, it is best to map this current logical RAM with 2 M8KRAMs. So far there are 24 logic tiles being allocated (these include regular logic block + LUTRAMs allocated so far + logic blocks of decoders/multiplexers allocated so far), and 1 M8KRAM occupied. This entails that there is 1 M8KRAM available. The application will first subtract 2 (required number of M8KRAMs) by 1 (number of currently available M8KRAMs) which results in 1, and calculates (1 - 1) * 10 + (10 – 24 mod 10) = 6. This entails only 6 more logic blocks are needed to be allocated to tolerate 2 M8KRAMs. Again, same algorithm applies to M128KRAMs and other custom-defined physical RAMs.

4) Input Logical RAM List Mapping Order

The last notable algorithm used in the application for optimizing area consumption is the mapping order of input logical RAMs. Experiments have shown that mapping logical RAMs in descending RAM size order optimizes area consumption. With the default order, the application yields a geometric average area of 2.09882e8. With increasing RAM size mapping order, the application yields 2.19092e8. With decreasing RAM size mapping order, the application yields 2.03860e8. The differences between ascending/default and default/descending mapping orders are 0.09e8 and 0.06e8, which are 9 million and 6 million unit areas (noticeably large). These numbers are results using Section IV part 4's approach.

Section II – Computational Complexity

For the computational complexity of the RAM Mapper, the expected time complexity is O(N*MlogM), where N is the number of circuits, and M is the maximum number of logical RAMs per circuit. One can think of the application as 2 nested for loops. The outer loop iterates over all N circuits, while the inner loop iterates over at most M logical RAMs to be mapped in that circuit. Before iterating the inner loop which consumes O(M) time, a sort (mentioned in Section I part 4) is performed on at most M logical RAMs. Assuming C++ standard template library uses randomized quicksort for sorting, the expected run-time will be O(MlogM), with the worst case being $O(M^2)$. For each inner loop iteration (i.e. for each logical RAM to be mapped), it takes O(1) time to determine which physical RAM type to map with, O(1) time to calculate number of rows/columns and depth/width needed to map, and O(1) time to update the total number of used logic blocks (if reallocation of logic blocks happen), the maximum number of available LUTRAMs, M8KRAMs, M128KRAMs, other custom-defined RAMs, and other variables.

Thus, time complexity is O(N*MlogM). Code snippets are shown in the next section for the audience to have a better understanding of the application structure.

Section III – Application Architecture Overview & Code Snippet

This section illustrates the architecture overview of the RAM-Mapper, multiple custom classes created to ease the design, and code snippet screenshots for each class. There screenshots may make the application seem to be tailored just for RAM mapping using LUTRAM, M8KRAM and M128KRAM. However, the actual code in the zipped source package is modified to behave more generally and can adapt custom-defined RAM dimension, as specified from the input arguments to the program.

1) Application Architecture Overview

The application itself is structured as a class object called *Mapper*. *Mapper* includes top-level functional methods such as *initMapper()* which parses the input files and initializes data members, *mapBRAM()* which runs the mapper, and *genFile()* which generates the mapping file for the checker. It also records a vector of pointers of circuits to map. The following is the header definition of class *Mapper*.

```
class Mapper {
public:
    Mapper();
    ~Mapper();

    // main methods
    void initMapper(const char* logicalRAMTxt, const char* logicBlockCountTxt);
    void mapBRAM(long int size1, int maxWidth1, int ratio1,
            long int size2, int maxWidth2, int ratio2);
    void genFile(const char* mappingFileTxt);

    // debugging methods
    void printCircuitList(bool printLogicalRAM, bool printPhysicalRAM);
    void printArea();

private:
    std::vector<Circuit*> m_circuitList;
};

#endif /* MAPPER_H_ */
```

The second notable class is called *Circuit*. As its name describes, *Circuit* contains functional/helper methods and data members for mapping all logical RAM within one circuit. Important methods are *mapSingleRAM()* and *mapDualRAM()* which map all logical RAMs given single or dual RAM types within that circuit, *sanityCheck()* which checks whether any mapping constraint (such as number of rows must not exceed 16, and some RAM dimension/type restrictions) is violated, *computeArea()* which calculates total number of RAMs used for each type of RAM, total required logic block tiles, and total area (including unused logic blocks), and *genFile()* which generates lines of strings consisting of mapping info for every logical RAMs in that circuit for the checker. The following is the header definition of class *Circuit*.

```
class Circuit {
public:
    Circuit();
    ~Circuit();

    // functional methods
    void setCircuitInfo(int index, int num);
    void insertLogicalRAM(int id, const char *mode, int depth, int width);
    void mapSingleBRAM(long int size, int maxWidth, int ratio);
    void mapDualBRAM(long int size1, int maxWidth1, int ratio1,
           long int size2, int maxWidth2, int ratio2);
    void updateUsedRAMs(int mode, int numUsedRAMs);
    void updateLogicCount(int inputDecodeLogic, int outputMuxLogic);
    void updateAddSpareLogic(int addSpareLogic);
    void sanityCheck();
    void computeArea();
    void genFile(std::ofstream &file);

    // helper methods
    int getCircuitId();
    int getAvailLUTRAM();
    int getAvailBRAM();
    int getAvailBRAM2();
    int getUsedLUTRAM();
    int getUsedBRAM();
    int getUsedBRAM2();
    int getTotalLB();
    unsigned long long getTotalArea();
    static bool logicalRAMCompare(LogicalRAM* &a, LogicalRAM* &b);
    unsigned long long calculateSRAMArea(unsigned long bits, unsigned int maxWidth);

    // debugging methods
    void printCircuit(bool printLogicalRAM, bool printPhysicalRAM);
    void printAreaUsage();
private:
    // basic circuit info
    int m_id;
    int m_logicCount;
    int m_decodeLUT;
    int m_muxLUT;
    int m_decodeMuxCount;
    int m_addSpareLogic;
    int m_addSpareUsed;

    // RAM info
    std::vector<LogicalRAM*> m_logicalRAMList;
    int m_usedLUTRAM;
    long m_sizeBRAM1;
    int m_maxWidthBRAM1;
    int m_ratioBRAM1;
    int m_usedBRAM1;
    long m_sizeBRAM2;
    int m_maxWidthBRAM2;
    int m_ratioBRAM2;
    int m_usedBRAM2;

    // Area info
    int m_finalUsedLUTRAM;
    int m_finalUsedBRAM;
    int m_finalUsedBRAM2;
    int m_finalUsedRegularLB;
    int m_finalRequiredLBTile;
    unsigned long long m_finalArea;
};
```

The last class is called *LogicalRAM*. As denoted in its name, *LogicalRAM* takes care of all mapping tasks related to one logical RAM. It contains methods that map itself to one of the physical RAM types, and data members which bookkeeps the information about the physical RAM it is mapped to, such as depth, width, number of rows and columns, and number of additional decoders and multiplexers needed. It also has sanity check function which checks if any mapping constraint is violated, and the function to generate mapping file information for the logical RAM. Below is the header definition of class *LogicalRAM*.

```
class LogicalRAM {
public:
    LogicalRAM();
    ~LogicalRAM();

    // functional methods
    void initLogicalRAM(int id, Circuit *circuit, const char *mode, int depth, int width);
    int getRAMMode();
    int getRAMSize();
    int getPhysicalRAMMode();
    void mapSingleBRAM(long int size, int maxWidth, int ratio);
    void mapDualBRAM(long int size1, int maxWidth1, int ratio1,
            long int size2, int maxWidth2, int ratio2);

    // helper methods
    bool mapAsLUTRAM(bool allocate);
    bool mapAsBRAM1(bool allocate);
    bool mapAsBRAM2(bool allocate);
    void mapWithRealloc();
    unsigned long long calculateSRAMArea(unsigned long bits, unsigned int maxWidth);
    void updateNumDecoderMux();
    void sanityCheck();
    void genFile(std::ofstream &file);

    // debugging methods
    void printLogicalRAM();
    void printPhysicalRAM();

    // enum for logical/physical RAM mode
    enum {
        ROM,
        SinglePort,
        SimpleDualPort,
        TrueDualPort,
        LUTRAM,
        BRAM1,
        BRAM2
    };
```

```
private:
    // logical RAM data members
    int m_id;
    Circuit *m_circuit;
    int m_mode;
    int m_depth;
    int m_width;
    int m_size;

    // physical RAM data members
    int m_physicalMode;
    int m_physicalDepth;
    int m_physicalWidth;
    int m_physicalNumRow;
    int m_physicalNumCol;
    int m_physicalNumDecoder;
    int m_physicalNumMux;
    int m_addSpareLBAlloc;

    // single/dual BRAM mapping
    long int m_sizeBRAM1;
    int m_maxWidthBRAM1;
    int m_minDepthBRAM1;
    int m_ratioBRAM1;
    long int m_sizeBRAM2;
    int m_maxWidthBRAM2;
    int m_minDepthBRAM2;
    int m_ratioBRAM2;
};
```

2) Code Snippet

Due to the size of the RAM-Mapper application, only the most important functions are pasted in this section. The screenshots here only show the functional flow of the application; they may not necessarily reflect the final look of the code, since the application has been revised while report is being constructed. Therefore, for code clarity marking, please refer to the actual zipped code package. The first screenshot contains the application's main function and *Mapper* instantiation. It expects <logical_rams.txt> <logic_block_count.txt> <path_for_output_mapping_file.txt> <type 1 RAM size> <type 1 max width> <type 1 respect ratio> as input arguments. It also allows user to add a flag "-t" after all input arguments to enable printing area information of all circuits. For more detailed commands to run the application, please refer to the README file inside the zipped package, or Section IV part 1.

```
int main(int argc, char **argv) {

    bool printArea = false;
    long int size1 = 0;
    long int size2 = 0;
    int maxWidth1 = 0;
    int maxWidth2 = 0;
    int ratio1 = 0;
    int ratio2 = 0;

    if(argc < 7) {
        std::cout << "[ERROR main] Too few input arguments" << std::endl;
        exit(-1);
    }
    else if(argc < 9) {
        size1 = atol(argv[4]);
        maxWidth1 = atoi(argv[5]);
        ratio1 = atoi(argv[6]);
        if(argc == 8 && strcmp(argv[7], "-t") == 0)
            printArea = true;
    }
    else if(argc < 10) {
        std::cout << "[ERROR main] Too few input arguments" << std::endl;
        exit(-1);
    }
    else if(argc < 12) {
        size1 = atol(argv[4]);
        maxWidth1 = atoi(argv[5]);
        ratio1 = atoi(argv[6]);
        size2 = atol(argv[7]);
        maxWidth2 = atoi(argv[8]);
        ratio2 = atoi(argv[9]);
        if(argc == 11 && strcmp(argv[10], "-t") == 0)
            printArea = true;
    }
    else {
        std::cout << "[ERROR main] Too many input arguments" << std::endl;
        exit(-1);
    }

    Mapper mapper;
    mapper.initMapper(argv[1], argv[2]);
    mapper.mapBRAM(size1, maxWidth1, ratio1, size2, maxWidth2, ratio2);
    mapper.genFile(argv[3]);
    if(printArea)
        mapper.printArea();

    return 0;
}
```

The following is the parsing input file and class initialization function.

```
void Mapper::initMapper(const char *logicalRAMTxt, const char * logicBlockCountTxt) {

    std::ifstream file;
    char s[32];

    // parse logic_block_count.txt
    file.open(logicBlockCountTxt, std::ifstream::in);
    while(file >> s)
        if(strcmp(s, "0") == 0)
            break;
    do {
        int logicCount;
        file >> s;
        logicCount = atoi(s);
        Circuit *circuit = new Circuit();
        circuit->setCircuitInfo(0, m_circuitList.size());
        circuit->setCircuitInfo(1, logicCount);
        m_circuitList.push_back(circuit);
    } while(file >> s);
    file.close();

    // parse logical_RAM.txt
    file.open(logicalRAMTxt, std::ifstream::in);
    while(file >> s)
        if(strcmp(s, "0") == 0)
            break;
    do {
        int circuitId;
        int RAMId;
        char mode[32];
        int depth;
        int width;

        circuitId = atoi(s);
        file >> s;
        RAMId = atoi(s);
        file >> s;
        memset(mode, '\0', sizeof(mode));
        strncpy(mode, s, sizeof(mode));
        file >> s;
        depth = atoi(s);
        file >> s;
        width = atoi(s);

        if(circuitId < 0 || circuitId >= (int)m_circuitList.size()) {
            std::cout << "[ERROR Mapper::init] Circuit ID "
                    << circuitId << " is not valid" << std::endl;
            exit(-1);
        }
        m_circuitList[circuitId]->insertLogicalRAM(RAMId, mode, depth, width);
    } while(file >> s);
    file.close();
```

The followings are the two nested loops for mapping. Clearly, the time complexity is O(N) times (O(MlogM) + O(M)) = O(N*MlogM), given N is the number of circuits, and M is the maximum number of logical RAMs per circuit. Note that the *m_logicalRAMList* vector which stores all logical RAMs to be mapped in every circuit is sorted in descending order of RAM size before mapping takes place (as explained in Section I part 4). Methods to perform sanity check and area computation are followed by every circuit mapping.

```
void Mapper::mapBRAM(long int size1, int maxWidth1, int ratio1,
        long int size2, int maxWidth2, int ratio2) {

    for(unsigned int i=0; i<m_circuitList.size(); i++)
        if(size2 == 0)
            m_circuitList[i]->mapSingleBRAM(size1, maxWidth1, ratio1);
        else
            m_circuitList[i]->mapDualBRAM(size1, maxWidth1, ratio1,
                    size2, maxWidth2, ratio2);
}
```

```
void Circuit::mapDualBRAM(long int size1, int maxWidth1, int ratio1,
        long int size2, int maxWidth2, int ratio2) {

    m_sizeBRAM1 = size1;
    m_maxWidthBRAM1 = maxWidth1;
    m_ratioBRAM1 = ratio1;
    m_sizeBRAM2 = size2;
    m_maxWidthBRAM2 = maxWidth2;
    m_ratioBRAM2 = ratio2;

    m_usedLUTRAM = 0;
    m_usedBRAM1 = 0;
    m_decodeMuxCount = 0;

    // sort logical RAMs with descending RAM size to achieve minimum area
    sort(m_logicalRAMList.begin(), m_logicalRAMList.end(), Circuit::logicalRAMCompare);

    // loop through all logical RAMs and map each of them
    for(unsigned int i=0; i<m_logicalRAMList.size(); i++)
        m_logicalRAMList[i]->mapDualBRAM(size1, maxWidth1, ratio1,
                size2, maxWidth2, ratio2);

    sanityCheck();
    computeArea();
}
```

The following is a portion of the actual mapping function. One can see that there are multiple IF statements which consists of different allocation trial order. There is no loop in this function, and thus time complexity is O(1), as claimed above.

```
void LogicalRAM::mapDualBRAM(long int size1, int maxWidth1, int ratio1,
        long int size2, int maxWidth2, int ratio2) {

    m_sizeBRAM1 = size1;
    m_maxWidthBRAM1 = maxWidth1;
    m_minDepthBRAM1 = size1 / maxWidth1;
    m_ratioBRAM1 = ratio1;

    m_sizeBRAM2 = size2;
    m_maxWidthBRAM2 = maxWidth2;
    m_minDepthBRAM2 = size2 / maxWidth2;
    m_ratioBRAM2 = ratio2;

    int numLUTRAM = (int)ceil(m_size/640.0);
    int numBRAM = (int)ceil(m_size/(1.0*m_sizeBRAM1));
    int numBRAM2 = (int)ceil(m_size/(1.0*m_sizeBRAM2));
    long long areaLUTRAM = numLUTRAM * 40000;
    long long areaBRAM = calculateSRAMArea(m_sizeBRAM1, m_maxWidthBRAM1) * numBRAM;
    long long areaBRAM2 = calculateSRAMArea(m_sizeBRAM2, m_maxWidthBRAM2) * numBRAM2;

    // Non TrueDualPort
    if(m_mode <= LogicalRAM::SimpleDualPort) {
        // LUTRAM < BRAM < BRAM2
        if(areaLUTRAM <= areaBRAM && areaBRAM <= areaBRAM2) {
            if(!mapAsLUTRAM(false))
                if(!mapAsBRAM1(false))
                    if(!mapAsBRAM2(false))
                        mapWithRealloc();
        }
        // LUTRAM < BRAM2 < BRAM
        else if(areaLUTRAM <= areaBRAM2 && areaBRAM2 <= areaBRAM) {
            if(!mapAsLUTRAM(false))
                if(!mapAsBRAM2(false))
                    if(!mapAsBRAM1(false))
                        mapWithRealloc();
        }
        // BRAM < LUTRAM < BRAM2
        else if(areaBRAM <= areaLUTRAM && areaLUTRAM <= areaBRAM2) {
            if(!mapAsBRAM1(false))
                if(!mapAsLUTRAM(false))
                    if(!mapAsBRAM2(false))
                        mapWithRealloc();
        }
        // BRAM < BRAM2 < LUTRAM
        else if(areaBRAM <= areaBRAM2 && areaBRAM2 <= areaLUTRAM) {
            if(!mapAsBRAM1(false))
                if(!mapAsBRAM2(false))
                    if(!mapAsLUTRAM(false))
                        mapWithRealloc();
        }
        // BRAM2 < LUTRAM < BRAM
        else if(areaBRAM2 <= areaLUTRAM && areaLUTRAM <= areaBRAM) {
            if(!mapAsBRAM2(false))
```

The next screenshot is the actual mapping function. As explained in Section I part 2, there are multiple IF statements that determine the number of rows and columns, and size of depth and width for each logical RAM. The following is the function that maps type 1 block RAM. The ones that map LUTRAM and type 2 block RAM are very similar.

```
bool LogicalRAM::mapAsBRAM1(bool allocate) {

    // return false immediately if exceeds 16 rows using
    // largest available physical depth - m_sizeBRAM1
    if((int)ceil(m_depth/(1.0*m_sizeBRAM1)) > 16)
        return false;

    m_physicalMode = LogicalRAM::BRAM1;

    bool set = false;
    for(int depth=m_minDepthBRAM1; depth<=m_sizeBRAM1; depth*=2) {
        if(m_depth <= m_minDepthBRAM1 && m_mode != LogicalRAM::TrueDualPort) {
            m_physicalDepth = depth;
            m_physicalWidth = m_sizeBRAM1/m_physicalDepth;
            m_physicalNumRow = 1;
            m_physicalNumCol = (int)ceil(m_width/(m_physicalWidth*1.0));
            set = true;
            break;
        }
        else if(depth != m_minDepthBRAM1 && m_depth <= depth) {
            m_physicalDepth = depth;
            m_physicalWidth = m_sizeBRAM1/m_physicalDepth;
            m_physicalNumRow = 1;
            m_physicalNumCol = (int)ceil(m_width/(m_physicalWidth*1.0));
            set = true;
            break;
        }
    }
    if(!set) {
        int mod = -1;
        int depth = m_sizeBRAM1;
        int stop = (m_mode == LogicalRAM::TrueDualPort) ?
                m_minDepthBRAM1 * 2 : m_minDepthBRAM1;
        for(int i=depth; i>=stop; i/=2) {
            int quotient = (int)ceil(m_depth/(i*1.0));
            int remainder = m_depth % i;
            if(remainder > mod && quotient <= 16) {
                mod = remainder;
                depth = i;
            }
        }
        m_physicalDepth = depth;
        m_physicalWidth = m_sizeBRAM1/m_physicalDepth;
        m_physicalNumRow = (int)ceil(m_depth/(m_physicalDepth*1.0));
        m_physicalNumCol = (int)ceil(m_width/(m_physicalWidth*1.0));
        set = true;
    }
```

```
    int availBRAM = m_circuit->getAvailBRAM() - m_circuit->getUsedBRAM();
    if(!allocate) {
        if(m_physicalNumRow * m_physicalNumCol > availBRAM)
            return false;
    }
    else {
        // allocate enough logic blocks if insufficient
        int numBRAMsNeeded = m_physicalNumRow * m_physicalNumCol;
        numBRAMsNeeded -= (m_circuit->getAvailBRAM() - m_circuit->getUsedBRAM());
        int numAddSpareLogicNeeded = m_ratioBRAM1 * numBRAMsNeeded -
                (m_circuit->getTotalLB() % m_ratioBRAM1);
        if(numAddSpareLogicNeeded < 0)
            numAddSpareLogicNeeded = 0;
        m_circuit->updateAddSpareLogic(numAddSpareLogicNeeded);
        m_addSpareLBAlloc = numAddSpareLogicNeeded;

        // if number of BRAMs needed required is greater than available BRAM1
        availBRAM = m_circuit->getAvailBRAM() - m_circuit->getUsedBRAM();
        if(m_physicalNumRow * m_physicalNumCol > availBRAM) {
            std::cout << "[ERROR LogicalRAM::mapAsBRAM1] Allocation of "
                    << numAddSpareLogicNeeded << " spare LBs not enough" << std::endl;
            m_circuit->updateAddSpareLogic(-1*numAddSpareLogicNeeded);
            exit(-1);
        }
    }

    // update # of rows/columns of physical RAMs
    updateNumDecoderMux();
    // update circuit used RAM
    m_circuit->updateUsedRAMs(LogicalRAM::BRAM1, m_physicalNumRow*m_physicalNumCol);
    // update circuit logic count
    m_circuit->updateLogicCount(m_physicalNumDecoder, m_physicalNumMux);
    return true;
}
```

For the latest complete source code, please refer to the zipped source folder.

Section IV – Mapper Results

This section illustrates table and numerical results generate by the RAM-Mapper application.

1) RAM Mapping Results for Example Stratix-IV Like Architecture

The following table shows the RAM usage information and total FPGA area consumption.

Circuit #	LUTRAM Blocks Used	8192 bit RAM Blocks Used	128k bit RAM Blocks Used	Regular Logic Blocks Used	Required Logic Block Tiles in Chip	Total FPGA Area (incl. unused blocks)
0	1118	209	2	2941	4059	2.02E+08
1	1536	386	12	3116	4652	2.32E+08
2	45	24	0	1836	1881	9.38E+07
3	37	56	1	2808	2845	1.42E+08
4	294	801	20	7932	8226	4.11E+08
5	31	252	4	3692	3723	1.86E+08
6	76	160	0	1853	1929	9.60E+07

7	420	431	14	3990	4410	2.20E+08
8	134	558	18	5342	5580	2.78E+08
9	1	32	0	1636	1637	8.14E+07
10	584	200	6	1473	2057	1.02E+08
11	120	135	1	1337	1457	7.20E+07
12	11	4	2	1632	1643	8.17E+07
13	6	20	0	4491	4497	2.24E+08
14	133	194	6	1828	1961	9.76E+07
15	43	63	4	1956	1999	9.93E+07
16	10	48	2	2182	2192	1.09E+08
17	2	59	0	1165	1167	5.75E+07
18	176	44	6	2040	2216	1.10E+08
19	370	256	7	2267	2637	1.31E+08
20	48	265	7	2679	2727	1.36E+08
21	24	44	1	5102	5126	2.56E+08
22	612	286	2	2454	3066	1.53E+08
23	4	104	11	5231	5235	2.61E+08
24	87	385	14	4325	4412	2.20E+08
25	55	81	0	4517	4572	2.28E+08
26	414	260	8	1449	2600	1.29E+08
27	0	20	0	1496	1496	7.39E+07
28	149	265	8	2004	2650	1.32E+08
29	277	201	9	3029	3306	1.65E+08
30	241	4	0	5419	5660	2.82E+08
31	2	79	0	4347	4349	2.17E+08
32	614	510	17	3633	5100	2.55E+08
33	30	400	10	4006	4036	2.01E+08
34	51	432	13	1705	4320	2.16E+08
35	64	144	0	1380	1444	7.15E+07
36	585	610	20	1697	6100	3.05E+08
37	0	0	12	14969	14969	7.47E+08
38	29	263	9	3206	3235	1.61E+08
39	413	215	6	1891	2304	1.15E+08
40	21	155	1	3064	3085	1.54E+08
41	597	264	7	2105	2702	1.35E+08
42	36	65	0	1337	1373	6.81E+07
43	146	128	0	1212	1358	6.74E+07
44	174	86	6	2118	2292	1.14E+08
45	2	12	1	2782	2784	1.39E+08
46	852	397	11	3490	4342	2.17E+08
47	39	23	0	1439	1478	7.30E+07
48	92	496	20	6875	6967	3.48E+08
49	1475	1324	39	11883	13358	6.67E+08
50	115	502	0	11884	11999	5.99E+08
51	14	420	4	4206	4220	2.11E+08

52	426	355	0	9603	10029	5.01E+08
53	0	816	0	10817	10817	5.41E+08
54	761	128	0	10903	11664	5.82E+08
55	1539	16	0	10341	11880	5.93E+08
56	63	235	6	4578	4641	2.32E+08
57	192	370	0	7145	7337	3.66E+08
58	65	752	2	7700	7765	3.87E+08
59	4576	1828	0	13719	18295	9.14E+08
60	10	552	0	20371	20381	1.02E+09
61	0	1890	62	15079	18900	9.45E+08
62	917	347	14	4999	5916	2.95E+08
63	0	391	15	4846	4846	2.42E+08
64	1973	1309	34	11151	13124	6.55E+08
65	11	350	0	12721	12732	6.36E+08
66	84	535	20	6312	6396	3.19E+08
67	1658	496	16	2928	4960	2.47E+08
68	192	0	0	4850	5042	2.51E+08

Total CPU run time for the RAM-Mapper application to run ranges between 3.6 to 7 milliseconds, with the most frequently resulted run time being approximately 4 milliseconds. Geometric average of the total FPGA area is 2.08043e8 unit areas. RAM Mapper can be run in two modes: LUTRAM + 1 BRAM, and LUTRAM + 2 BRAMs. Commands are shown as follows

For the LUTRAM + 1 BRAM mode, run

*./RAM-Mapper <logical_rams.txt> <logic_block_count.txt> <output_mapping_file.txt> *
<BRAM size> <BRAM max width> <BRAM respect ratio> <-t>

For the LUTRAM + 2 BRAMs mode, run

*./RAM-Mapper <logical_rams.txt> <logic_block_count.txt> <output_mapping_file.txt> *
*<BRAM 1 size> <BRAM 1 max width> <BRAM 1 respect ratio> *
<BRAM 2 size> <BRAM 2 max width> <BRAM 2 respect ratio> <-t>

The last flag <-*t*> is optional; the application prints the above table if this flag is set. Please note that <-*t*> needs to be the last input argument if used.

2) RAM Mapping Results with Single BRAM

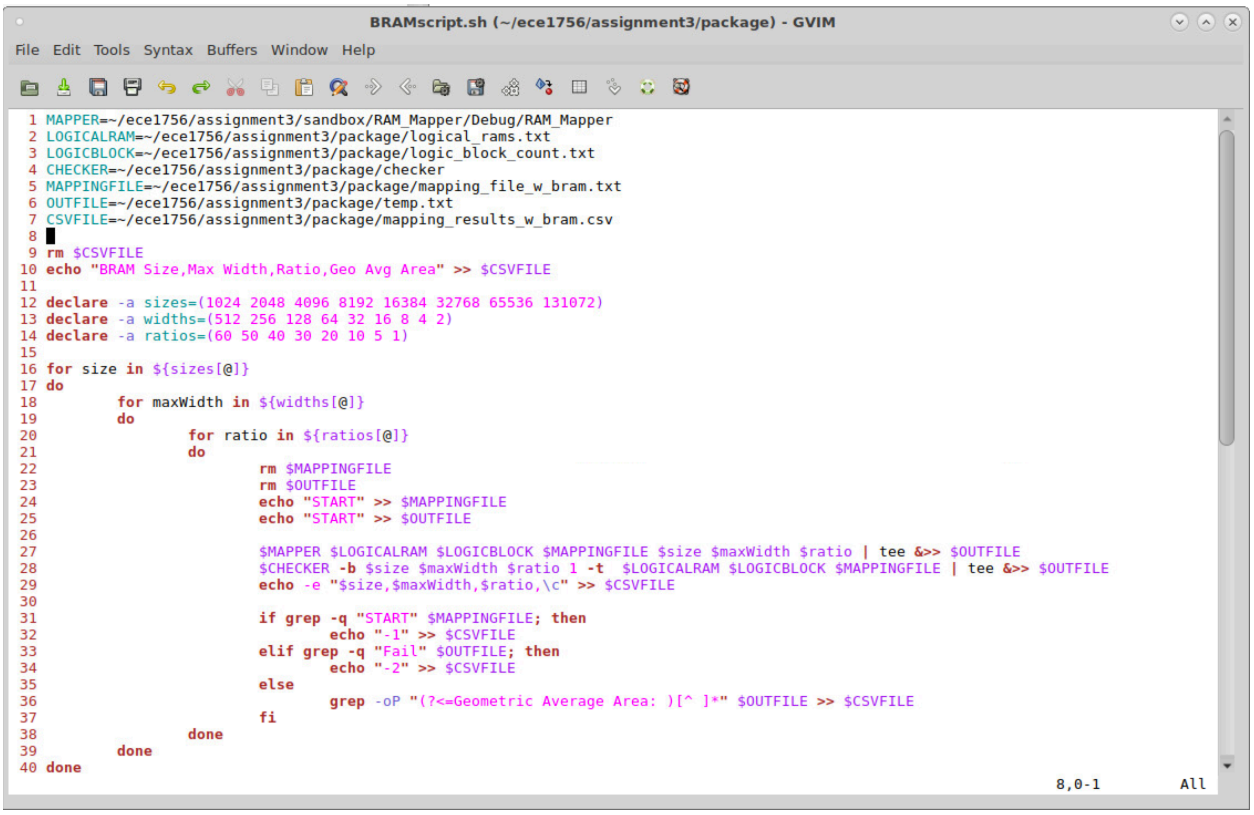
To explore the any potential trend as BRAM size, max width, and respect ratio (logic blocks/RAM block) change, 8 values of BRAM size, 9 values of max width, and 8 values of respect ratio are swept, which totals to 8 x 9 x 8 = 576 sweeps being performed. The sweep values are denoted as follows.

a. BRAM size: {1024, 2048, 4096, 8192, 16384, 32768, 65535, 131072}
    i. The set is basically 1k, 2k, … , 128k, as required by the handout
b. Max width: {512, 256, 128, 64, 32, 16, 8, 4, 2}

i. The idea to start from max width of 512 is because 1024 is the minimum number in the BRAM size sweep set, and setting max width to half of it leaves some room for number of rows (room for vertical spanning)
ii. Note max width of 1 is not used, since this won't implement the *TrueDualPort* logical RAM

c. Respect ratio: {60, 50, 40, 30, 20, 10, 5, 1}

Of course, a short shell script is created to perform the sweeps. RAM-Mapper is also configured such that it can take the three more parameters as input arguments. The following is the shell script that performs the parameter sweep.



Since showing all 576 geometric average areas in a table takes up too much space, instead the report will show 2 tables: top 20 sweep combinations that result in minimum geometric average area, and top 20 for maximum geometric average area. The following is the table containing top 20 trials for minimum geometric average area.

BRAM Size	Max Width	Ratio	Geo Avg Area
8192	32	5	2.17E+08
8192	16	5	2.18E+08
16384	64	10	2.26E+08
16384	32	10	2.26E+08
4096	32	5	2.27E+08

4096	16	5	2.28E+08
8192	32	10	2.36E+08
8192	64	10	2.37E+08
16384	32	5	2.38E+08
16384	16	5	2.41E+08
8192	64	5	2.43E+08
32768	64	10	2.47E+08
32768	32	10	2.50E+08
1024	4	1	2.50E+08
2048	4	1	2.51E+08
16384	128	10	2.54E+08
4096	64	5	2.56E+08
1024	8	1	2.57E+08
2048	8	1	2.58E+08
32768	64	20	2.58E+08

At a first glance, the best BRAM size seems to mostly reside in the middle of its sweeping set (from 4096 to 16384), the best max width seems to mostly reside again in the middle of its sweeping set (from 16 to 64), but the best respect ratio seems to reside mostly in the lower end of its sweeping set (from 5 to 10). This implicitly shows that it is better for both BRAM size and its max width to be some "average" number amongst their sweep sets. Respect ratio being neither maximum not minimum, but close to the lower bound implicitly explains that smaller number of logic blocks required to place a BRAM should be small, as LUTRAM cannot be used in this case. Conversely, the following table shows the maximum geometric average area results.

BRAM Size	Max Width	Ratio	Geo Avg Area
1024	2	60	6.79E+09
131072	2	60	6.22E+09
2048	2	60	5.72E+09
1024	2	50	5.67E+09
65536	2	60	5.47E+09
131072	2	50	5.44E+09
4096	2	60	5.15E+09
32768	2	60	5.09E+09
16384	2	60	4.92E+09
8192	2	60	4.90E+09
131072	512	1	4.90E+09
2048	2	50	4.77E+09
65536	2	50	4.69E+09
131072	2	40	4.66E+09

1024	2	40	4.55E+09
1024	4	60	4.53E+09
32768	2	50	4.31E+09
4096	2	50	4.30E+09
16384	2	50	4.14E+09
8192	2	50	4.11E+09

Taking the same approach, one sees that the worst BRAM size seems to be very close to the max/min value in its sweep set. The worst max width and respect ratio tend to be 2 and 50/60, respectively. This shows an opposite trend from the minimum area case, which makes sense. An exception case of BRAM size being 131072, max width being 512, and respect ratio being 1 does not follow the trend, however. Reasoning through this exception case more carefully, it eventually makes sense since when BRAM size is at its maximum, and allocation of the BRAM exists (even if unused) between every logic block, which consumes a lot of area. Conclusions drawn from this experiment are as follows.

a. Best BRAM size is in between 4096 and 16384. As it deviates from this range, geometric average area starts to increase.
b. Best max width is in between 16 to 64. As it deviates from this range, geometric average area starts to increase.
c. Best respect ratio is in between 5 to 10. As it deviates from this range, geometric average area starts to increase.
d. RAM mapping using single BRAM sizing from 1k to 128k has worse geometric average area than RAM mapping using Stratix-IV like architecture.

3) RAM Mapping Results with Single BRAM + LUTRAM

It is interesting to observe the geometric average area change when logic blocks are configurable as LUTRAMs. This part enables the LUTRAM configurability, and delivers results using the same sweep parameter sets as part 2. Below table shows the top 20 sweep combinations that result in smallest geometric average area.

BRAM Size	Max Width	Ratio	Geo Avg Area
16384	32	10	2.17E+08
8192	16	5	2.20E+08
8192	32	5	2.24E+08
32768	64	20	2.26E+08
8192	32	10	2.28E+08
16384	64	10	2.29E+08
4096	16	5	2.29E+08
32768	128	30	2.33E+08
4096	32	5	2.35E+08
16384	64	20	2.36E+08
8192	64	10	2.36E+08

16384	16	10	2.37E+08
65536	128	40	2.38E+08
32768	128	20	2.39E+08
65536	64	30	2.39E+08
32768	64	30	2.41E+08
32768	32	20	2.41E+08
65536	128	50	2.42E+08
65536	128	30	2.42E+08
16384	16	5	2.42E+08

At a quick glance, BRAM size seems to reside mostly in between 4096 and 65536. Max width seems to reside mostly in between 16 and 128. Respect ratio seems to reside mostly in between 5 and 40. These observations are very similar in terms of their spanning directions in the parameter set comparing to part 2 results. For instance, part 2 had the best BRAM size in between 4096 and 16384, and this part has that in between 4096 and 65535, spanning towards maximum size. Part 2 had the best max width in between 16 and 64, and this part has that in between 16 and 128 (again, spanning towards maximum max width). Part 2 had best respect ratio in between 5 and 10, and this part has that in between 5 and 40 (again, spanning towards maximum respect ratio). These observations implicitly explain that, since half of logic blocks become configurable as LUTRAMs, there becomes more room for the RAM Mapper to compare area efficiency between different physical RAM types (in this case single BRAM plus LUTRAM). Another important factor is the respect ratio. Now the application can map RAMs using LUTRAMs, it will use less BRAMs, and thus respect ratio can be increased to save more area (allocated unused logic blocks can be used as LUTRAMs). Conversely, the following table shows the top 20 trials which result in the largest geometric average area.

BRAM Size	Max Width	Ratio	Geo Avg Area
1024	2	60	7.27E+09
2048	2	60	6.12E+09
1024	2	50	6.07E+09
131072	512	1	6.02E+09
4096	2	60	5.51E+09
2048	2	50	5.11E+09
1024	2	40	4.87E+09
1024	4	60	4.85E+09
4096	2	50	4.61E+09
131072	256	1	4.57E+09
65536	512	1	4.43E+09
2048	2	40	4.10E+09
8192	2	60	4.06E+09
1024	4	50	4.05E+09

131072	128	1	3.86E+09
2048	4	60	3.72E+09
4096	2	40	3.70E+09
1024	2	30	3.67E+09
32768	512	1	3.65E+09
1024	8	60	3.62E+09

Similar to the conclusion drawn in part 2, max width and respect ratio tend to be in the two extremes in their parameter sweep sets. Reasons for these trends are explained in part 2. However, BRAM size being any number except 16384 can result in large geometric average area. Perhaps, in this worst case, dimensions of the BRAM (max width) and its respect ratio have larger impact. The BRAM size becomes less impactful. Therefore, the conclusions drawn in this part are stated as follows.

a. With LUTRAM's availability, there becomes more room for BRAM size, max width, and respect ratio to reside in order to optimize area.
b. With LUTRAM's availability, the spanning behaviors in the parameter sets of all three parameters all share the same direction (i.e. their maximum best numbers all increase).
c. With LUTRAM's availability, in the worst case, geometric average area becomes less dependent on BRAM size comparing to max width and respect ratio.

4) RAM Mapping Results with LUTRAM and 2 Types of SRAM-Based BRAMs

For this part, the application uses the same parameter sweep technique as parts 2 and 3, but on 2 block RAMs. To reduce total number of sweeps, elements that resulted in high geometric average area in parts 2 and 3 are removed, with some new elements added. Elements in parameter sets are shown as follows.

a. BRAM size: {4096, 8192, 16384, 32768, 65535, 131072}
b. Max width: {256, 128, 64, 32, 16, 8, 4}
c. Respect ratio: {300, 100, 50, 20, 10, 5}

There are (6 x 7 x 6) x (6 x 7 x 6) = 63503 sweep combinations in total. Note that the 2 block RAMs share the same parameter sets. The application could have constructed the sets separately for the 2 BRAMs (having 1 relatively large BRAM, and 1 relatively small BRAM), but using the same sets should be sufficient for this assignment because the set still contains relatively large and small numbers, if doubling the run-time is not of an issue. The following table shows the 10 best RAM block sizing which yields the minimum geometric average area.

Type 1 BRAM Size	Type1 Max Width	Type 1 Ratio	Type 2 BRAM Size	Type 2 Max Width	Type 2 Ratio	Geometric Average Area
4096	16	10	16384	16	20	2.03860E+08
8192	32	20	16384	16	20	2.04969E+08
4096	16	10	32768	32	50	2.05214E+08

8192	16	10	16384	16	50	2.05424E+08
4096	16	10	16384	8	20	2.05457E+08
8192	16	10	16384	32	50	2.05546E+08
4096	16	20	8192	16	10	2.05606E+08
4096	16	10	32768	16	50	2.05685E+08
8192	16	10	32768	32	100	2.05830E+08
4096	16	10	16384	32	20	2.05922e+08

The geometric average area for the best RAM architecture is 2.0386e8 unit areas, which wins out amongst parts 1, 2, and 3. The parameters of the architecture are shown in the table below.

Type	**RAM Size**	**Minimum Width (scalable)**	**Maximum Width (physical)**	**Respect Ratio**	**Compliant Logical Block Type**
LUTRAM	640 bits	10 bit	20 bit	1	ROM SinglePort SimpleDualPort
M4KRAM	4096 bits	1 bit	16 bit	10	ROM SinglePort SimpleDualPort TrueDualPort
M16KRAM	16384 bits	1 bit	16 bit	20	ROM SinglePort SimpleDualPort TrueDualPort

This RAM architecture appears to be the most area efficient amongst all sweep combinations, since it has the smallest geometric average area, 2.0386e8. Although this is a huge improvement from previous experiments, the behavior is still somewhat expected. First, for both M4KRAM and M16KRAM, their RAM sizes, max widths, and respect ratios (logic blocks per RAM block) agree with the conclusions found in parts 2 and 3; they are not in the extremes. Second, their RAM size ratio is 4:1, which provides relatively large and relatively small blocks mapping. Third, their maximum widths are close to the low end of the parameter set, which is expected since area is calculated based on size and max width; therefore, minimizing max width optimizes area consumption. Fourth, their respect ratios are not too big to cause a waste of spare logic blocks while they are allocated under resource run-out scenario. Thus, this RAM architecture optimizes area consumption the most, so far.

5) RAM Mapping Results with LUTRAM and 2 Types of MJT-Based BRAMs

This part is very similar to part 4, with an exception of SRAMs being replaced with MJTs. The difference between SRAM and MJT architectures is the area model; area of an SRAM is approximately 4 times larger than that of an MJT given the same size and max width. Due to this, one should expect the geometric average area decreases in general. Again, the same three parameters (BRAM size, max width, respect ratio) are swept with the same numbers, which totals

to 63503 sweeps. The following table illustrates the top 10 sweep combinations that result in the lowest geometric average area.

Type 1 BRAM Size	Type1 Max Width	Type 1 Ratio	Type 2 BRAM Size	Type 2 Max Width	Type 2 Ratio	Geometric Average Area
4096	16	10	32768	16	20	1.79053E+08
4096	16	10	65536	32	50	1.79548E+08
4096	16	10	32768	32	20	1.80317E+08
4096	16	20	32768	32	20	1.80968E+08
8192	32	20	32768	16	20	1.81071E+08
4096	16	20	16384	16	10	1.81092E+08
8192	16	10	65536	32	50	1.81116E+08
4096	16	10	65536	64	50	1.81199E+08
8192	16	10	65536	16	50	1.81294E+08
16384	32	20	32768	16	20	1.81471E+08

The geometric average area for the best RAM architecture using MJTs is 1.79053e8 unit areas, which is almost 25 million (3.0386e8 – 1.79053e8 = 2.4807e7) unit areas less than that using SRAMs in part 4. The parameters of this MJT architecture are the same as the SRAM approach, with the only difference being the RAM size for the second RAM type being 32k bits. Detailed architecture information is shown in the following table. The difference is marked in RED.

Type	RAM Size	Minimum Width (scalable)	Maximum Width (physical)	Respect Ratio	Compliant Logical Block Type
LUTRAM	640 bits	10 bit	20 bit	1	ROM SinglePort SimpleDualPort
M4KRAM	4096 bits	1 bit	16 bit	10	ROM SinglePort SimpleDualPort TrueDualPort
M32KRAM	32768 bits	1 bit	16 bit	20	ROM SinglePort SimpleDualPort TrueDualPort

Indeed, geometric average area is decreased by order of magnitude as expected. However, it is not reduced to a fourth. This is due to existence of the required logic blocks plus some logical RAMs being mapped as LUTRAMs. In addition, parameters for this best MJT-based architecture share high similarity to that for the best SRAM-based architecture, with an exception of using 32kbit size RAM rather than 16kbit. Same max widths, respect ratio for all RAM types are used. A few

hypotheses can be drawn to explain why the best architecture does not change by a lot between SRAM-based and MJT-based mechanisms, shown as follows.

a. The algorithms used (mentioned in Section I) does not depend on area model.
b. Area models (equations) between SRAM-based and MJT-based mechanisms depend on the same variables (RAM size, max width). The application uses the same RAM size and max width to calculate the two areas for every sweep combination. The difference is the constant that gets multiplied with RAM size. Both approaches share the same area minimization problem.

Overall, this assignment is very challenging in terms of understanding the background of the content, constructing an adaptive application that can take in various RAM types, and optimizing geometric average area with smart algorithms. Although the advanced mapping technique (concatenating physical RAMs with different dimensions) is not implemented, area performance with naïve mapping technique is acceptable (recall 2.0386e8 unit areas for SRAM-based, and 1.79053e8 for MJT-based). For future work, it is worth to try the advanced approach out.