Matrix Manipulation Algorithms for Hasse Processor Implementation

Hahanov Vladimir, Dahiri Farid
Kharkov National University of Radioelectronics, Ukraine

***Abstract* – The processor is implemented in software-hardware modules, which are based on the use of programming languages: C ++, Verilog, Python 2.7 and platforms: Microsoft Windows, X Window (in Unix and Linux) and Macintosh OS X. HDL-code generator makes it possible to automatically synthesize HDL-code of the processor structure from 1 to 16 bits for parallel processing corresponding number of input vectors or words.**

I. Introduction

The binary coverage consists of finding the minimum lines combination in a matrix to make full one-units coverage. This task is very simple and doesn't require particular human effort for a small matrix. However this task becomes very difficult for a big matrix, for example considering a matrix of 1000 lines of 100 elements. In this case obviously we need a machine to do the job.

This paper will cover 2 methods that perform binary coverage with a reasonable speed performance for a matrix with up to 50000 x 50000 elements.

Those algorithms are coded in python for convenience reasons. This language is the optimal choice for algorithm prototyping. However, the final implementation of the algorithms will be coded in C++ for memory performance and multithreading.

II. Formation of the Minimum Coverage for Two-Dimensional Matrix

2.1 Matrix to sub-matrices horizontal split

Solving practical problem for generating a binary coverage for unit matrix is to find the minimum combination of rows and columns, covering all unit values. Searching coverage for a small matrix is realized using simple techniques [1]. Searching binary coverage for a large matrix containing thousands of rows and columns requires a lot of time and money.

Horizontal partition of a matrix into submatrices is focused to divide the matrix M into a set of submatrices with a predetermined number of rows in accordance with the expression:

$$M(R, S^R) = \{M'_i, M'_{i+1}, \dots M'_{i+n}\},$$

where n=R/S^R, M is the original matrix; R is number of rows; S^R is number S of rows per sub-matrix; M'_i is the derivative matrix, resulting from horizontal split; i represents index of the matrix. For example, the matrix M is:

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	1	1	1	1
2	1	0	1	0	0	0	0	1	1	1
3	1	0	1	0	0	0	0	0	0	0
4	1	1	1	1	1	0	0	0	0	0
5	0	0	0	0	1	0	0	1	0	0
6	1	0	0	1	1	1	1	1	1	0
7	0	0	0	0	0	1	1	1	1	0
8	1	1	1	1	1	0	0	0	0	0
9	1	1	1	1	0	0	0	1	1	1
10	0	0	0	0	0	1	1	1	1	1

Applying the method with horizontal split parameter $S^R = 5$ will result on 2 matrixes:

Matrix M'_1

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	1	1	1	1
2	1	0	1	0	0	0	0	1	1	1
3	1	0	1	0	0	0	0	0	0	0
4	1	1	1	1	1	0	0	0	0	0
5	0	0	0	0	1	0	0	1	0	0

Manuscript received June 19, 2014.
Vladimir Hahanov is with the Kharkov National University of Radioelectronics, Computer Engineering Faculty, Kharkov, Ukraine, e-mail: hahanov@kture.kharkov.ua
Farid Dahiri is with the Kharkov National University of Radioelectronics, Computer Engineering Faculty, Design Automation Department, Kharkov, Ukraine.

Matrix M'_2

	1	2	3	4	5	6	7	8	9	10
6	1	0	0	1	1	1	1	1	1	0
7	0	0	0	0	0	1	1	1	1	0
8	1	1	1	1	1	0	0	0	0	0
9	1	1	1	1	0	0	0	1	1	1
10	0	0	0	0	0	1	1	1	1	1

$$M = \{M'_1, M'_2\}$$

Applying this method we can considerably reduce binary coverage search time. In fact, when having two separate matrices M_1' and M_2', we can parallelize binary coverage search.

2.2. Matrix to sub-matrices horizontal vertical split

When solving the problem of binary coverage generation the most effective method is splitting M both horizontally and vertically into a set of submatrices with the specified number of rows and columns according to the expression:

$$M(R, C, S^R, S^C) = \{M'_{i,j}, M'_{i,j+1}, M'_{i,j+2}, M'_{i+1,j}, \dots M'_{i+n,j+m}\},$$
$$j = C/C^R,$$

where C is number of matrix columns; C^R is number C of columns per sub-matrix.

An example of horizontal-vertical partitioning of the matrix M when split-parameters have the following values S^R=5 и S^C=5 is represented below:

Matrix $M'_{1,1}$

	1	2	3	4	5
1	0	0	0	0	0
2	1	0	1	0	0
3	1	0	1	0	0
4	1	1	1	1	1
5	0	0	0	0	1

Matrix $M'_{2,2}$

	6	7	8	9	10
1	0	1	1	1	1
2	0	0	1	1	1
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	1	0	0

Matrix $M'_{2,1}$

	1	2	3	4	5
6	1	0	0	1	1
7	0	0	0	0	0
8	1	1	1	1	1
9	1	1	1	1	0
10	0	0	0	0	0

Matrix $M'_{1,2}$

	6	7	8	9	10
6	1	1	1	1	0
7	1	1	1	1	0
8	0	0	0	0	0
9	0	0	1	1	1
10	1	1	1	1	1

Partition of the matrix into submatrices can significantly reduce the search time of a binary coverage due to parallel processing submatrices.

2.3. Python matrix representation

In Python [2-4] bi-dimensional arrays (matrices) are not implemented; thereby a matrix is represented by an array of arrays. The class that implements the array interface in Python is List, so a matrix is represented by a list of lists. Let's consider the following matrix:

	6	7	8	9	10
6	1	1	1	1	0
7	1	1	1	1	0
8	0	0	0	0	0
9	0	0	1	1	1
10	1	1	1	1	1

A simplified python representation of this matrix is shown in Listing 1.

Listing 1
```
matrix = [[1, 1, 1, 1, 0],
          [1, 1, 1, 1, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 1, 1, 1],
          [1, 1, 1, 1, 1]]
```

In fact, a matrix is composed of a list of "Line" objects. A line object is a python matrix row representation. Bellow goes an example of matrix row and its python representation (Listing 2):

1	1	1	1	1

Listing 2
```
print line

index      :0
indexes    :[]
values     :[1, 1, 1, 1]
ones count :[1, 1, 1, 1]
```

The "Line" class members with the unique constructor are represented in Listing 3:

Listing 3
```
class Line(object):
…
    def __init__(self, index = 0, indexes = [], values = [], ones_count = []):
        self.index      = index
        self.indexes    = indexes
        self.values     = values
        self.ones_count = ones_count
        if len(self.values) != len(self.ones_count):
            self.ones_count.extend(values)
…
end
```

Listing 4 involves some methods of the class Line:

Listing 4
```
class Line(object):
…
    def ones_qnty(self):
        qnty = 0
        for value in self.values:
            if value:
                qnty += 1
        return qnty

    def binary_or(self, line):
        before_ones_qnty = self.ones_qnty()
        for key, value in enumerate(line.values):
            self_value = self.values[key]
            self.values[key] = self_value | value
            self.ones_count[key] += value
        if self.ones_qnty() > before_ones_qnty:
            for index in line.indexes:
                self.indexes.append(index)
            self.indexes.append(line.index)
            return True
        return False

    def binary_xor(self, line):
        for key, value in enumerate(line.values):
            self.ones_count[key]    -= value
            if self.ones_count[key] <= 0:
                self.ones_count[key] = 0
                self.values[key] = 0
            else:
                self.values[key]        = 1
        self.indexes.remove(line.index)
…
end
```

The method “ones_qnty” returns the quantity of one-unit in the line.

The method “binary_or” applies binary or transformation with the given line. A Boolean variable is returned:

- true if the binary or modified the line;
- false if the line was not modified.

The “ones_count” instance variable keeps ones count implied in binary or operation.

The method “binary_xor” performs a custom xor operation with a given line. For each element the “value” is subtracted from “ones_qnty”. If the “ones_qnty” is equal to zero, the value field of the concerned line is set to zero too, otherwise it is set to one-unit.

A matrix set is represented as a container, which simplifies the manipulation of complicated matrices (Listing 5).

Listing 5
```
class ComplexMatrix(object):

    def __init__(self, matrices):
        self.matrices   = matrices
        self.columns    = len(self.matrices[0].lines)
        self.rows       = len(self.matrices) / self.columns

    def matrixAtRowColumn(self, row, column):
        return self.matrices[row * self.columns + column]
```

2.4. Matrix to sub-matrices horizontal split implementation

Software implementation of vertical-horizontal partitioning matrix of arbitrary size into a set of submatrices with user-defined number of rows is shown in the Listing 6.

Listing 6
```
class MatrixUtils(object):
…
    @staticmethod
    def split_array(array, divisions):
        return [array[i:i+divisions] for i in range(0, len(array), divisions)]

    @staticmethod
    def linearSplittedMatricesFromValues(matrix, max_lines = 25):
        splited_lines = MatrixUtils.split_array(matrix.lines, max_lines)
        matrixes = []
        for lines in splited_lines:
            matrixes.append(Matrix(lines))

        return matrixes

…
end
```

The method “split_matrix” splits a list into a set of lists by the following manner (Listing 7):

Listing 7
```
matrix_to_split = [[0, 0], [0, 1], [1, 1], [1, 0]]
matrices = split_array(matrix_to_split, 2)
>>>Print matrices
[[[0, 0], [0, 1]], [[1, 1], [1, 0]]]
```

Then the list is rearranged in order to obtain 2 matrices (Listing 8):

Listing 8
```
 [[0, 0],
 [0, 1]]

[[1, 1],
 [1, 0]]
```

Matrix to sub-matrices horizontal vertical split implementation “ComplexMatrixFromValues” is a utility method; it returns a complex matrix from a set of matrices. The original matrix is split horizontally and vertically by calling the internal method “rectSplitedMatricesFromValues” (Listing 9):

Listing 9

```
class MatrixUtils(object):
…
  @staticmethod
  def complexMatrixFromValues(values,
                              horiz_max_lines = 10,
                              vert_max_lines = 10):
    matrices =
MatrixUtils.rectSplitedMatricesFromValues(values,
                                          horiz_max_lines,
                                          vert_max_lines)

    return ComplexMatrix(matrices)
…
end
```

The method "RectSplitedMatricesFromValues" returns a set of matrices (Listing 10):

Listing 10

```
 @staticmethod
  def rectSplitedMatricesFromValues(values,
horiz_max_lines = 10, vert_max_lines = 10):
    horiz_splitted_lines = []
    for hrz_line in values:

horiz_splitted_lines.append(MatrixUtils.split_array(hrz_line
, horiz_max_lines))

    result = []
    final_results = []
    elements_count = len(horiz_splitted_lines[0])
    print 'elements count: ' + str(elements_count)

    for _ in range(elements_count):
      result.append([])

    line_count = 0
    for line in horiz_splitted_lines:
      element_count = 0
      for element in line:
        result[element_count].append(element)
        element_count += 1

      line_count += 1

      if (not (line_count % horiz_max_lines) and
line_count):
        final_results.append(result)
        result = []
        for _ in range(elements_count):
          result.append([])

    print 'result: ' + str(result)
    matrices = []
    mat_counter = 0
    for res in final_results:
      for val in res:
        mat_index = [mat_counter / horiz_max_lines,
mat_counter % vert_max_lines]
        print mat_index

matrices.append(Matrix(MatrixUtils.matrixLinesFromListV
alues(val), mat_index))
        mat_counter += 1

    return matrices
```

2.5. Matrix's binary coverage signature

The matrix's binary coverage signature represents a high level optimization, particularly for horizontal-vertical split. This method is not yet implemented.

The idea is about representing a matrix by a hash unique to matrix one-unit quantity and repartition. The goal is to avoid recalculating binary coverage for a same matrix. Bellow follows matrix signature representation:

$$H_{i,j} = \{\{R, C\}, \{M_{i,j}\}\}$$

Matrix signature associated value:

$$V_{i,j} = \{\{R_i\}, \{ M_{i,j}\}\}$$

Let's consider the following example:

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	1	1	1	1
2	1	0	1	0	0	0	0	1	1	1
3	1	0	1	0	0	0	0	0	0	0
4	1	1	1	1	1	0	0	0	0	0
5	0	0	0	0	1	0	0	1	0	0
6	0	0	0	0	0	0	1	1	1	1
7	1	0	1	0	0	0	0	1	1	1
8	1	0	1	0	0	0	0	0	0	0
9	1	1	1	1	1	0	0	0	0	0
10	0	0	0	0	1	0	0	1	0	0

Applying the method with horizontal split parameter S^R=5 and vertical split parameter S^C=5 will result on 4 matrices:

Matrix $M'_{1,1}$

	1	2	3	4	5
1	0	0	0	0	0
2	1	0	1	0	0
3	1	0	1	0	0
4	1	1	1	1	1
5	0	0	0	0	1

Matrix $M'_{1,2}$

	6	7	8	9	10
1	0	1	1	1	1
2	0	0	1	1	1
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	1	0	0

Matrix $M'_{2,1}$

	1	2	3	4	5
6	0	0	0	0	0
7	1	0	1	0	0
8	1	0	1	0	0
9	1	1	1	1	1
10	0	0	0	0	1

Matrix $M'_{2,2}$

	6	7	8	9	10
6	0	1	1	1	1
7	0	0	1	1	1
8	0	0	0	0	0
9	0	0	0	0	0
10	0	0	1	0	0

Now we are going to calculate the hash for each matrix:

$$H'_{1,1} = \{\{5,5\},\{0000010100101001111100001\}\}$$
$$H'_{1,2} = \{\{5,5\},\{0111100111000000000000100\}\}$$
$$H'_{2,1} = \{\{5,5\},\{0000010100101001111100001\}\}$$
$$H'_{2,2} = \{\{5,5\},\{0111100111000000000000100\}\}$$

The values for those hashes:

$$V'_{1,1} = \{\{4\},\{11111\}\}$$
$$V'_{1,2} = \{\{1\},\{01111\}\}$$
$$V'_{2,1} = \{\{9\},\{11111\}\}$$
$$V'_{2,2} = \{\{6\},\{01111\}\}$$

The hashes $H'_{1,1}$, $H'_{1,2}$ are respectively equal to $H'_{2,1}, H'_{2,2}$. This means that if we already know the binary coverage of $M'_{1,1}$ we can deduce the binary coverage result of $M'_{2,1}$ by simply calculating it's hash.

2.6. Methods comparison

Both methods goals are to parallelize binary coverage. The horizontal split is simple to realize. However, it does not consider the matrix columns count. For large matrices, where the columns count is considerably bigger than the rows count, this method can create time overhead because of linear complexity of binary coverage method.

The horizontal-vertical method is much harder to realize and the time to split the matrix into sub-matrices is considerably higher than for simple horizontal split. Also should be considered the time for reconstructing the matrices into the original matrix because of vertical split. The advantage of this method is high parallelizing. In fact, a large matrix can be split into matrices with smaller columns number. Moreover, a custom logic can be realized in order to processes particular matrices to speed up the coverage.

Profile test for both methods is presented in Listing 11, the result – in Listing 12.

Listing 11

```
'''
Created on 17.04.2013

@author: _______
'''
from models.matrix_utils import MatrixUtils
from test_main import test_table_4
import cProfile

def linear_split_test():
    for _ in range(100):
MatrixUtils.linearSplittedMatricesFromValues(test_table_4, 5)
def rect_split_test():
    for _ in range(100):
MatrixUtils.rectSplitedMatricesFromValues(test_table_4, 5, 5)
if __name__ == '__main__':
    cProfile.run('rect_split_test()')
    cProfile.run('linear_split_test()')
```

Listing 12

```
306104 function calls in 1.576 seconds
Ordered by: standard name
ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
 1   0.000   0.000   1.576   1.576 <string>:1(<module>)
 56000   0.281   0.000   0.399   0.000 line.py:9(__init__)
 1 0.051  0.051   1.576   1.576 main.py:17(rect_split_test)
 11200  0.029  0.000  0.029  0.000 matrix.py:11(__init__)
 8100   0.078   0.000   0.112   0.000
matrix_utils.py:15(split_array)
 100   0.300   0.003   1.525   0.015
matrix_utils.py:43(rectSplitedMatricesFromValues)
 11200   0.404   0.000   0.910   0.000
matrix_utils.py:83(matrixLinesFromListValues)
 8200   0.015   0.000   0.015   0.000 {len}
 145500   0.276   0.000   0.276   0.000 {method
'append' of 'list' objects}
 1   0.000   0.000   0.000   0.000 {method 'disable' of
'_lsprof.Profiler' objects}
 56000   0.118   0.000   0.118   0.000 {method 'extend'
of 'list' objects}
 9801   0.023   0.000   0.023   0.000 {range}

     188604 function calls in 1.069 seconds

Ordered by: standard name
ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
 1   0.000   0.000   1.069   1.069 <string>:1(<module>)
 8100   0.035   0.000   0.054   0.000 line.py:9(__init__)
 1   0.047   0.047   1.069   1.069
main.py:12(linear_split_test)
 81950   0.256   0.000   0.256   0.000
matrix.py:11(__init__)
 100   0.059   0.001   0.062   0.001
matrix_utils.py:15(split_array)
 100   0.430   0.004   1.022   0.010
matrix_utils.py:31(linearSplittedMatricesFromValues)
 100   0.053   0.001   0.124   0.001
matrix_utils.py:93(matrixFromListValues)
 100   0.000   0.000   0.000   0.000 {len}
 89950   0.167   0.000   0.167   0.000 {method 'append'
of 'list' objects}
 1   0.000   0.000   0.000   0.000 {method 'disable' of
'_lsprof.Profiler' objects}
 8100   0.019   0.000   0.019   0.000 {method 'extend'
of 'list' objects}
 101   0.002   0.000   0.002   0.000 {range}
```

The linear method splits the original matrix on 17 sub-matrices. The rect-split method splits the matrix into 112 matrices but is 50% slower.

III. Matrix binary coverage algorithms

3.1 Binary OR for first line algorithm

We take the first matrix line which contains at least 1 one-unit and apply binary or operation with the rest of the lines. If the first line is full of zeros (contains only zeros), we skip it and iterate to the next line. The operation is repeated until a non empty line is found. If the 5 lines of the sub-matrix are full of zeros, the entire sub-matrix is skipped and will not participate in further calculations. Each time we apply OR, we check if the operation changed the line, if yes we keep the result line and save the indexes of both lines.

For example, let us consider the following matrices M1, M2, and M3.

Matrix M1:

	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	0	0
3	0	1	0	0	0
4	0	1	0	0	0
5	0	0	1	0	0

$\boldsymbol{l1\ empty \rightarrow l1\ skipped}$

$\boldsymbol{l2\ empty \rightarrow l2\ skipped}$

$\boldsymbol{l3^{□} \cup l4 \rightarrow l3^{□} = 01000 \rightarrow l4\ rejected}$

$\boldsymbol{l3 \cup l5 \rightarrow l3^{1}\ saved}$

We have skipped $\{l_1,l_2\}$ because they contain only zeros. Those lines are empty.

Matrix M2:

	1	2	3	4	5
1	0	1	0	0	1
2	0	0	0	0	0
3	1	1	0	0	0
4	0	0	1	1	0
5	1	1	0	0	0

$l1 \cup l2 \rightarrow l1^{□} = 01001\ not\ changed \rightarrow l2\ rejected$

$l1^{□} \cup l3 \rightarrow l1^{1} = 11001\ changed \rightarrow l3\ saved$

$l1^{1} \cup l4 \rightarrow l1^{2} = 11111\ changed \rightarrow l4\ saved$

Matrix M3:

	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0

$l1\ empty \rightarrow l1\ skipped$

$l2\ empty \rightarrow l2\ skipped$

$l3\ empty \rightarrow l3\ skipped$

$l4\ empty \rightarrow l4\ skipped$

$l5\ empty \rightarrow l5\ skipped$

Each OR operation we check if the binary coverage for 1 is full, in other words if the resulting line is full of one-units. At state $l_1$2 the line is full of one-units we no more perform OR operation. Let's make it clear with the following example:

Matrix M4:

	1	2	3	4	5	6	7	8	9	10
1	0	1	0	1	0	0	0	0	0	0
2	1	0	0	1	0	0	0	0	0	0
3	0	0	0	0	1	0	0	0	0	0
4	1	1	1	1	1	0	0	0	0	0
5	0	0	1	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	1	1
7	1	0	0	0	1	1	0	1	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0

$l1 \cup l2 \rightarrow l1^{1} = 1101000000 \rightarrow l2\ saved$

$l1^{1} \cup l3 \rightarrow l1^{2} = 1101100000 \rightarrow l3\ saved$

$l1^{2} \cup l4 \rightarrow l1^{2} = 1101100000 \rightarrow l4\ rejected$

$l1^{2} \cup l5 \rightarrow l1^{3} = 1111100000 \rightarrow l5\ saved$

$l1^{3} \cup l6 \rightarrow l1^{4} = 1111101011 \rightarrow l6\ saved$

$l1^{4} \cup l7 \rightarrow l1^{5} = 1111111111 \rightarrow l7\ saved$

$l1^{5} = \{l1, l2, l3, l5, l6, l7\}$

3.2 Binary OR for first line with reverse algorithm

This method consists of 3 steps:

1. First the matrix lines are sorted by one-units quantity descending. Consider the following example of the matrix M1:

	1	2	3	4	5	6	7	8	9	10
1	1	1	0	0	0	0	0	0	0	0
2	0	0	1	1	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0
4	0	1	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	1	1	1	1
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0

Bellow the sorted matrix M_{1S}:

	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	0	0	1	1	1	1
2	0	0	1	1	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0
4	0	1	0	0	0	0	1	0	0	0
1	1	1	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0

2. Then we perform binary or for first line until we reach full coverage or the end of the matrix (if we reach the last line without full coverage the calculation is stopped):

$$\begin{aligned}
&l5 \cup l2 \rightarrow l5^1 = 0011001111 \rightarrow l2\ saved\\
&l5^1 \cup l3 \rightarrow l5^2 = 0011111111 \rightarrow l3\ saved\\
&l5^2 \cup l4 \rightarrow l5^3 = 0111111111 \rightarrow l4\ saved\\
&l5^3 \cup l1 \rightarrow l5^4 = 1111111111 \rightarrow l1\ saved\\
&l5^4 = \{l5, l2, l3, l4, l1\}
\end{aligned}$$

3. After that we begin deleting one line from $l5^4$each time. The lines are deleted starting from the penultimate line. Each time a line is deleted, we check if the binary coverage is still total. It is important to note that the algorithm is applied recursively. We can optimize binary coverage by deleting redundant lines. The following example demonstrates that fact:

$$\begin{aligned}
&l5 \cup l2 \rightarrow l5^1 = 0011001111 \rightarrow l2\ saved\\
&l5^1 \cup l3 \rightarrow l5^2 = 0011111111 \rightarrow l3\ saved\\
&l5^2 \cup l4 \rightarrow l5^3 = 0111111111 \rightarrow l4\ saved\ (line\ to\ delete)\\
&l5^3 \cup l1 \rightarrow l5^4 = 1111111111 \rightarrow l1\ saved\\
&l5^4 = \{l5, l2, l3, l4, l1\}
\end{aligned}$$

The result is represented below:

$$\begin{aligned}
&l5 \cup l2 \rightarrow l5^1 = 0011001111 \rightarrow l2\ saved\\
&l5^1 \cup l3 \rightarrow l5^2 = 0011111111 \rightarrow l3\ saved\\
&l5^3 \cup l1 \rightarrow l5^4 = 1111111111 \rightarrow l1\ saved\\
&l5^4 = \{l5, l2, l3, l1\}
\end{aligned}$$

The binary coverage without row *l*4 is still total, *l*4 will be excluded from the final coverage. The row *l*4 is marked as primary redundant. Now, when we have found one redundant line, we begin applying this algorithm recursively, until we delete all redundant lines (rows). After there is no more redundant rows for the primary row *l*4, the same algorithm is applied for the next primary row. In this case the next primary row is *l*3. After the algorithm applied for all primary rows, the quantity of redundant rows is compared and the best result is considered.

Applying the algorithm for primary row *l*3:

$$\begin{aligned}
&l5 \cup l2 \rightarrow l5^1 = 0011001111 \rightarrow l2\ saved\\
&l5^1 \cup l3 \rightarrow l5^2 = 0011111111 \rightarrow l3\ saved(line\ to\ delete)\\
&l5^3 \cup l1 \rightarrow l5^4 = 1111111111 \rightarrow l1\ saved\\
&l5^4 = \{l5, l2, l3, l1\}
\end{aligned}$$

The result is represented below:

$$\begin{aligned}
&l5 \cup l2 \rightarrow l5^1 = 0011001111 \rightarrow l2\ saved\\
&l5^1 \cup l3 \rightarrow l5^2 = 0011111111 \rightarrow l3\ saved(line\ to\ delete)\\
&l5^3 \cup l1 \rightarrow l5^4 = 1111111111 \rightarrow l1\ saved\\
&l5^4 = \{l5, l2, l3, l1\}
\end{aligned}$$

Deleting the row l_3 makes the binary coverage not full. The row l_3 cannot be deleted.

Applying for row l_2:

$$\begin{aligned}
&\mathbf{l5 \cup l2 \rightarrow l5^1 = 0011001111 \rightarrow l2\ saved}(line\ to\ delete)\\
&l5^1 \cup l3 \rightarrow l5^2 = 0011111111 \rightarrow l3\ saved\\
&l5^3 \cup l1 \rightarrow l5^4 = 1111111111 \rightarrow l1\ saved\\
&l5^4 = \{l5, l3, l1\}
\end{aligned}$$

The result is represented below:

$$\begin{aligned}
&l5^1 \cup l3 \rightarrow l5^2 = 0000001111 \rightarrow l3\ saved\\
&l5^3 \cup l1 \rightarrow l5^4 = 1111001111 \rightarrow l1\ saved\\
&l5^4 = \{l5, l1\}
\end{aligned}$$

Deleting the row l_2 makes the binary coverage not full. The row l_2 cannot be deleted.

3.3 Binary OR for first line algorithm implementation

The following method implements the binary or for first line algorithm. First, a primary line is chosen. The primary contains must not be full of zeros (empty line). If the matrix contains only empty lines, the entire matrix is skipped and an empty line is returned (Listing 13).

Listing 13

```
class Matrix(object):
…
   def binary_or_for_first_line_with_full_check(self):
      if not len(self.lines):
         return Line()
      first_line = self.lines[0].deepcopy()
      iter_lines = iter(self.lines)
      next(iter_lines)

      try:
         passedLoop = False
         while first_line.is_full_of_zeros():
            first_line = next(iter_lines)
            passedLoop = True
         if passedLoop:
            first_line = first_line.deepcopy()
      except StopIteration:
         return first_line

      for line in iter_lines:
         first_line.binary_or(line)
         if first_line.is_full_of_ones():
            break

      return first_line
…
end
```

3.4 Binary OR for first line with reverse algorithm implementation

The static method “applyMatrixRecurtion” from “Algorithm” class takes as argument a matrix object of type “Matrix”. The returned value is a list of lines objects of type “Line” (Listing 14).

Listing 14

```
class Algorithm(object):
…
    @staticmethod
    def applyMatrixRecurtion(matrix):
        matrix.sort_by_ones_qnty()
        matrix.updateLinesMapping()

        return matrix.binary_full_check_reverse()
…
end
```

Matrix class implements the algorithm (Listing 15).

Listing 15

```
class Matrix(object):
…
    def binary_full_check_reverse(self):
        line = 
self.binary_or_for_first_line_with_full_check()
        result_lines = []

        for value in line.indexes:
            newLine = line.deepcopy()
            newLine.binary_xor(self.lineForIndex(value))
            result_lines.append(newLine)

        return result_lines
…
end
```

Bellow follows an example of input matrix and output values.
Input matrix is represented in Listing 16.

Listing 16

```
test_table = [
[0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0], #0
[0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], #1
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0], #2
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], #3
[0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], #4
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0], #5
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], #6
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], #7
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0], #8
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], #9
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0], #10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], #11
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0], #12
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0], #13
[0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0], #14
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], #15
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], #16
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], #17
[0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1], #18
        ]
```

Now goes the call that generates the output (Listing 17).

Listing 17

```
if __name__ == '__main__':
    lines = Algorithm.applyMatrixRecurtion(
MatrixUtils.matrixFromListValues(test_table))
    for line in lines:
        print line
```

And finally the output (Listing 18):

Listing 18

```
index      :0
indexes    :[18, 1, 14, 2, 12, 13, 6, 8, 10, 15, 17]
values     :[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
ones count :[1, 3, 3, 2, 3, 3, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1]
index      :0
indexes    :[1, 14, 2, 12, 13, 6, 8, 10, 15, 17]
values     :[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0]
ones count :[1, 2, 3, 1, 3, 3, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 0, 0]
index      :0
indexes    :[18, 14, 2, 12, 13, 6, 8, 10, 15, 17]
values     :[1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
ones count :[1, 3, 3, 2, 2, 3, 0, 1, 0, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1]
index      :0
indexes    :[18, 1, 2, 12, 13, 6, 8, 10, 15, 17]
values     :[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
ones count :[1, 2, 3, 2, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
index      :0
indexes    :[18, 1, 14, 12, 13, 6, 8, 10, 15, 17]
values     :[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1]
ones count :[1, 3, 2, 2, 3, 3, 1, 1, 1, 1, 0, 2, 1, 1, 1, 1, 1, 1, 1]
index      :0
indexes    :[18, 1, 14, 2, 13, 6, 8, 10, 15, 17]
values     :[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1]
ones count :[1, 3, 3, 1, 3, 3, 1, 1, 1, 1, 1, 2, 1, 1, 0, 1, 1, 1, 1]
index      :0
indexes    :[18, 1, 14, 2, 12, 6, 8, 10, 15, 17]
values     :[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1]
ones count :[1, 3, 2, 2, 3, 3, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 0, 1, 1]
index      :0
indexes    :[18, 1, 14, 2, 12, 13, 8, 10, 15, 17]
values     :[1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
ones count :[1, 3, 3, 2, 3, 3, 1, 0, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1]
index      :0
indexes    :[18, 1, 14, 2, 12, 13, 6, 10, 15, 17]
values     :[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
ones count :[1, 3, 3, 2, 3, 3, 1, 1, 1, 1, 1, 2, 0, 1, 1, 1, 1, 1, 1]
index      :0
indexes    :[18, 1, 14, 2, 12, 13, 6, 8, 15, 17]
values     :[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1]
ones count :[1, 3, 3, 2, 3, 3, 1, 1, 1, 1, 1, 2, 1, 0, 1, 1, 1, 1, 1]
index      :0
indexes    :[18, 1, 14, 2, 12, 13, 6, 8, 10, 17]
values     :[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1]
ones count :[1, 3, 3, 2, 3, 3, 1, 1, 1, 1, 1, 2, 1, 1, 1, 0, 1, 1, 1]
index      :0
indexes    :[18, 1, 14, 2, 12, 13, 6, 8, 10, 15]
values     :[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
ones count :[0, 3, 3, 2, 3, 3, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1]
```

3.5 Methods comparison

Methods called to profile algorithms performance (Listing 19):

Listing 19

```
def recurtion_test():
    for _ in range(100):

Algorithm.applyMatrixRecurtion(MatrixUtils.matrixFromL
istValues(test_table))

def first_line_or_test():
    for _ in range(100):

Algorithm.applyMatrixFirstLineOR(MatrixUtils.matrixFro
mListValues(test_table))

if __name__ == '__main__':
    cProfile.run('first_line_or_test()')
    cProfile.run('recurtion_test()')
```

Profile results for first line or test are represented in Listing 20.

Listing 20

```
21105 function calls in 0.146 seconds
Ordered by: standard name
  ncalls    tottime    percall    cumtime    percall
filename:lineno(function)
1    0.000    0.000    0.146    0.146 <string>:1(<module>)
100    0.001    0.000    0.103    0.001
algorithm.py:15(applyMatrixFirstLineOR)
5500    0.026    0.000    0.026    0.000
line.py:19(ones_qnty)
1800    0.011    0.000    0.021    0.000
line.py:29(is_full_of_ones)
100    0.001    0.000    0.001    0.000
line.py:34(is_full_of_zeros)
1800    0.041    0.000    0.063    0.000
line.py:39(binary_or)
100    0.002    0.000    0.005    0.000
line.py:78(deepcopy)
2000    0.014    0.000    0.022    0.000 line.py:9(__init__)
1    0.002    0.002    0.146    0.146
main.py:27(first_line_or_test)
100    0.000    0.000    0.000    0.000
matrix.py:11(__init__)
100    0.011    0.000    0.102    0.001
matrix.py:73(binary_or_for_first_line_with_full_check)
100    0.017    0.000    0.042    0.000
matrix_utils.py:93(matrixFromListValues)
100    0.000    0.000    0.000    0.000 {iter}
5900    0.012    0.000    0.012    0.000 {len}
3000    0.006    0.000    0.006    0.000 {method 'append'
of 'list' objects}
1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
301    0.002    0.000    0.002    0.000 {method 'extend' of
'list' objects}
100    0.000    0.000    0.000    0.000 {next}
1    0.000    0.000    0.000    0.000 {range}
```

Profile results for recursion method are represented in Listing 21.

Listing 21

```
596943 function calls in 3.720 seconds
Ordered by: standard name
ncalls    tottime    percall    cumtime    percall
filename:lineno(function)
1    0.000    0.000    3.720    3.720 <string>:1(<module>)
100    0.002    0.000    3.678    0.037
algorithm.py:52(applyMatrixRecurtion)
341792    1.177    0.000    1.177    0.000
line.py:19(ones_qnty)
41000    0.257    0.000    0.493    0.000
line.py:29(is_full_of_ones)
100    0.000    0.000    0.001    0.000
line.py:34(is_full_of_zeros)
41000    0.845    0.000    1.160    0.000
line.py:39(binary_or)
1100    0.021    0.000    0.024    0.000
line.py:57(binary_xor)
1200    0.020    0.000    0.043    0.000
line.py:78(deepcopy)
3100    0.024    0.000    0.036    0.000 line.py:9(__init__)
1    0.003    0.003    3.720    3.720
main.py:23(recurtion_test)
100    0.012    0.000    1.949    0.019
matrix.py:102(binary_full_check_reverse)
100    0.000    0.000    0.000    0.000
matrix.py:11(__init__)
100    0.005    0.000    1.688    0.017
matrix.py:18(sort_by_ones_qnty)
100    0.039    0.000    0.039    0.000
matrix.py:39(updateLinesMapping)
1100    0.003    0.000    0.003    0.000
matrix.py:45(lineForIndex)
100    0.210    0.002    1.869    0.019
matrix.py:73(binary_or_for_first_line_with_full_check)
109346    0.720    0.000    1.426    0.000
matrix_utils.py:110(reverseLines)
100    0.013    0.000    0.038    0.000
matrix_utils.py:93(matrixFromListValues)
100    0.000    0.000    0.000    0.000 {iter}
47300    0.089    0.000    0.089    0.000 {len}
4200    0.009    0.000    0.009    0.000 {method 'append'
of 'list' objects}
1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
3601    0.008    0.000    0.008    0.000 {method 'extend'
of 'list' objects}
1100    0.003    0.000    0.003    0.000 {method 'remove'
of 'list' objects}
100    0.257    0.003    1.683    0.017 {method 'sort' of
'list' objects}
```

```
100    0.000    0.000    0.000    0.000 {next}
1    0.000    0.000    0.000    0.000 {range}
```

The second method is much slower, however more accurate because of binary xor recursion. Both methods are still subjects to optimization.

III CONCLUSION

The processor is implemented in software-hardware modules, which are based on the use of programming languages: C++, Verilog, Python 2.7 and platforms: Microsoft Windows, X Window (in Unix and Linux) and Macintosh OS X. HDL-code generator makes it possible to automatically synthesize HDL-code of the processor structure from 1 to 16 bits for parallel processing corresponding number of input vectors or words.

Verification of HDL-processor code is executed on test examples of coverage problem using two optimization strategies: reversible algorithm to eliminate redundancy and partitioning the coverage matrix for the purpose of further parallel processing by Hasse processors.

The performance of the proposed two methods depends on the distribution of unit elements in the matrix; this information is important for large matrices. However, most of this information is not available. In this case, both the approaches can be used for logic optimization and increase the speed of the second method.

The best solution is a combination of two proposed methods: sorting matrix and its subsequent partitioning into submatrices; use of reverse algorithm of recursive method for submatrices containing more than one unit element, and a simple horizontal partitioning algorithm for the rest of the matrix; going to step 2 of the first recursive method.

REFERENCES

[1] *Gorbatov V.A.* Fundamentals of Discrete Mathematics / V.A. Gorbatov.– M.: Vysshaya Shkola.– 1986. – 311 p.
[2] http://www.python.org/
[3] http://www.eclipse.org/
[4] http://pydev.org/