ASSIGNMENT LAB 12

-Họ tên: Hồ Tuấn Huy

-MSSV: 20225856

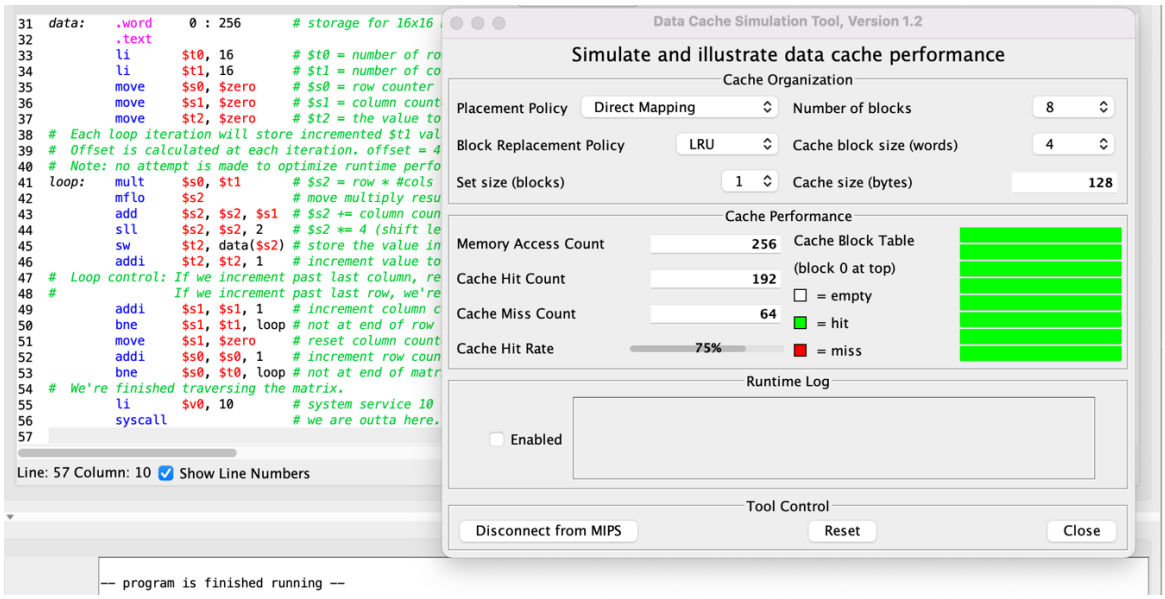
Assignment 1: Running the Data Cache Simulator tool.

CODE: row-major:

```
################################################################################
#####
#  Row-major order traversal of 16 x 16 array of words.
#  Pete Sanderson
#  31 March 2007
#
#  To easily observe the row-oriented order, run the Memory Reference
#  Visualization tool with its default settings over this program.
#  You may, at the same time or separately, run the Data Cache Simulator
#  over this program to observe caching performance.  Compare the results
#  with those of the column-major order traversal algorithm.
#
#  The C/C++/Java-like equivalent of this MIPS program is:
#    int size = 16;
#    int[size][size] data;
#    int value = 0;
#    for (int row = 0; row < size; row++) {
#      for (int col = 0; col < size; col++) }
#        data[row][col] = value;
#        value++;
#      }
#    }
#
#  Note: Program is hard-wired for 16 x 16 matrix.  If you want to change this,
#       three statements need to be changed.
#       1. The array storage size declaration at "data:" needs to be changed from
#          256 (which is 16 * 16) to #columns * #rows.
#       2. The "li" to initialize $t0 needs to be changed to new #rows.
#       3. The "li" to initialize $t1 needs to be changed to new #columns.
#
       .data
data:    .word     0 : 256      # storage for 16x16 matrix of words
       .text
       li      $t0, 16       # $t0 = number of rows
       li      $t1, 16       # $t1 = number of columns
       move    $s0, $zero    # $s0 = row counter
       move    $s1, $zero    # $s1 = column counter
```

```
        move    $t2, $zero     # $t2 = the value to be stored
#  Each loop iteration will store incremented $t1 value into next element of matrix.
#  Offset is calculated at each iteration. offset = 4 * (row*#cols+col)
#  Note: no attempt is made to optimize runtime performance!
loop:   mult    $s0, $t1       # $s2 = row * #cols  (two-instruction sequence)
        mflo    $s2            # move multiply result from lo register to $s2
        add     $s2, $s2, $s1  # $s2 += column counter
        sll     $s2, $s2, 2    # $s2 *= 4 (shift left 2 bits) for byte offset
        sw      $t2, data($s2) # store the value in matrix element
        addi    $t2, $t2, 1    # increment value to be stored
#  Loop control: If we increment past last column, reset column counter and increment
row counter
#               If we increment past last row, we're finished.
        addi    $s1, $s1, 1    # increment column counter
        bne     $s1, $t1, loop # not at end of row so loop back
        move    $s1, $zero     # reset column counter
        addi    $s0, $s0, 1    # increment row counter
        bne     $s0, $t0, loop # not at end of matrix so loop back
#  We're finished traversing the matrix.
        li      $v0, 10        # system service 10 is exit
        syscall                # we are outta here.
```

⇨ Ta thu được final cache hit rate là 75%:



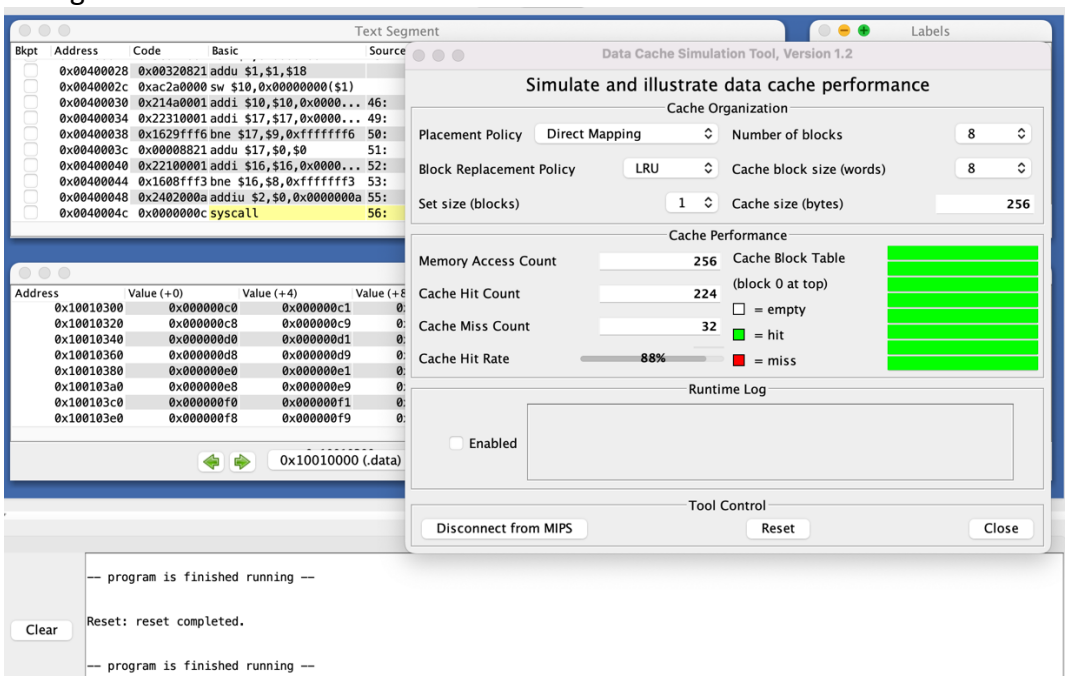
Giải thích: Vì với mỗi lần bỏ lỡ, một khối 4-word được ghi vào bộ đệm. Trong một đường truyền chính theo hàng, các phần tử của ma trận được truy cập theo cùng thứ tự chúng được lưu trữ trong bộ nhớ => Mỗi lần bỏ lỡ bộ đệm thì theo sau là 3 lần truy cập vì 3 phần tử tiếp theo được tìm thấy trong cùng một khối bộ đệm. Tiếp

theo là một lỗi khác khi ma trận trực tiếp ánh xạ tới khối bộ đệm tiếp theo và sau đó lặp lại chính nó => Cứ 4 lần truy cập bộ nhớ thì 3 lần được giải quyết trong bộ đệm.

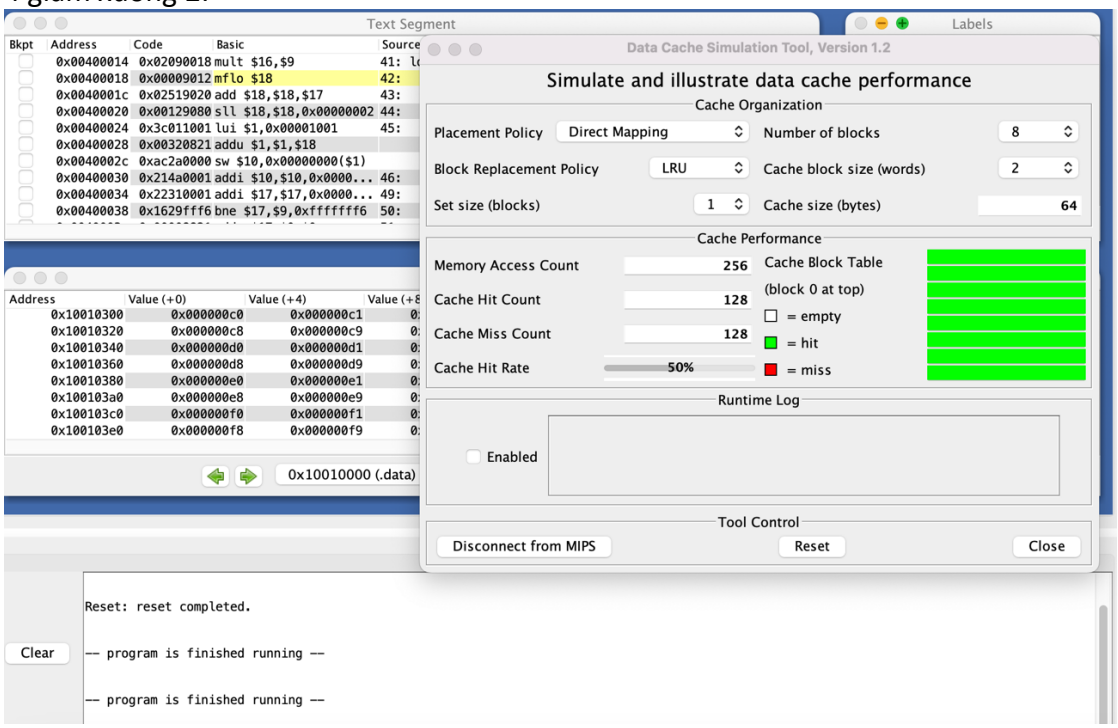
⇨ Từ đó ta có thể dự đoán khi tăng block size từ 4 lên 8 words thì hit rate sẽ là 87.5%, còn nếu giảm từ 4 xuống 2 thì sẽ là 50%.

<u>Kiểm chứng dự đoán:</u>

- 4 tăng lên 8:



- 4 giảm xuống 2:

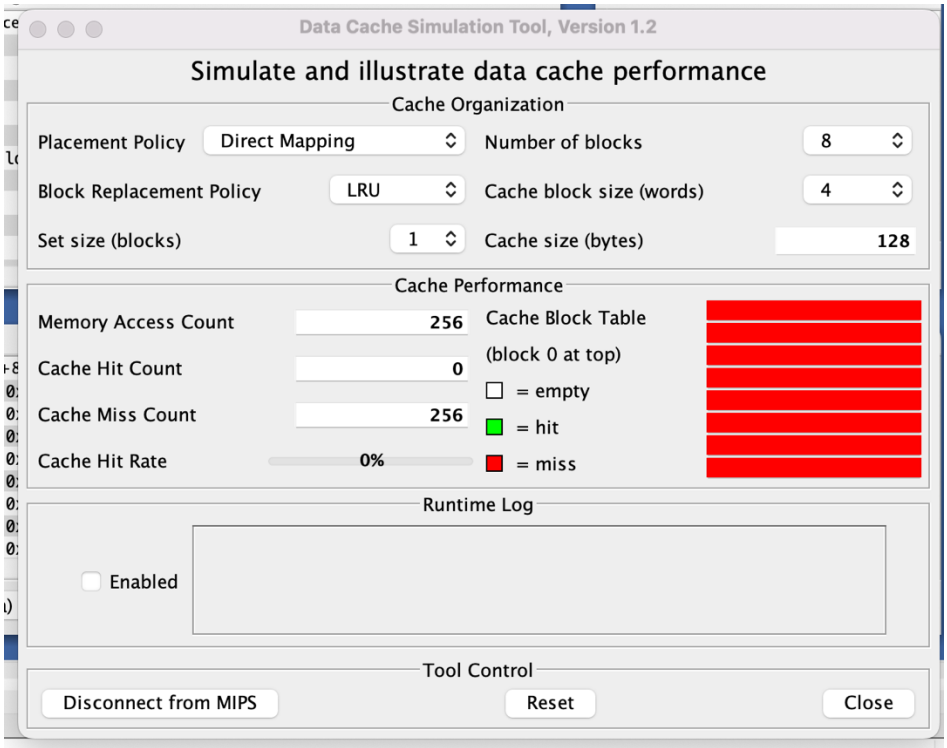


CODE: column major:

```
#################################################################
#
#  Column-major order traversal of 16 x 16 array of words.
#  Pete Sanderson
#  31 March 2007
#
#  To easily observe the column-oriented order, run the Memory Reference
#  Visualization tool with its default settings over this program.
#  You may, at the same time or separately, run the Data Cache Simulator
#  over this program to observe caching performance.  Compare the results
#  with those of the row-major order traversal algorithm.
#
#  The C/C++/Java-like equivalent of this MIPS program is:
#    int size = 16;
#    int[size][size] data;
#    int value = 0;
#    for (int col = 0; col < size; col++) {
#      for (int row = 0; row < size; row++) }
#        data[row][col] = value;
#        value++;
#      }
#    }
#
#  Note: Program is hard-wired for 16 x 16 matrix.  If you want to change this,
#       three statements need to be changed.
#       1. The array storage size declaration at "data:" needs to be changed from
#          256 (which is 16 * 16) to #columns * #rows.
#       2. The "li" to initialize $t0 needs to be changed to the new #rows.
#       3. The "li" to initialize $t1 needs to be changed to the new #columns.
#
       .data
data:    .word     0 : 256      # 16x16 matrix of words
       .text
       li      $t0, 16       # $t0 = number of rows
       li      $t1, 16       # $t1 = number of columns
       move    $s0, $zero    # $s0 = row counter
       move    $s1, $zero    # $s1 = column counter
       move    $t2, $zero    # $t2 = the value to be stored
#  Each loop iteration will store incremented $t1 value into next element of
matrix.
#  Offset is calculated at each iteration. offset = 4 * (row*#cols+col)
#  Note: no attempt is made to optimize runtime performance!
loop:    mult    $s0, $t1      # $s2 = row * #cols  (two-instruction sequence)
```

```
        mflo    $s2         # move multiply result from lo register to $s2
        add     $s2, $s2, $s1  # $s2 += col counter
        sll     $s2, $s2, 2    # $s2 *= 4 (shift left 2 bits) for byte offset
        sw      $t2, data($s2) # store the value in matrix element
        addi    $t2, $t2, 1    # increment value to be stored
#  Loop control: If we increment past bottom of column, reset row and
increment column
#               If we increment past the last column, we're finished.
        addi    $s0, $s0, 1    # increment row counter
        bne     $s0, $t0, loop # not at bottom of column so loop back
        move    $s0, $zero     # reset row counter
        addi    $s1, $s1, 1    # increment column counter
        bne     $s1, $t1, loop # loop back if not at end of matrix (past the last
column)
#  We're finished traversing the matrix.
        li      $v0, 10        # system service 10 is exit
        syscall                # we are outta here.
```

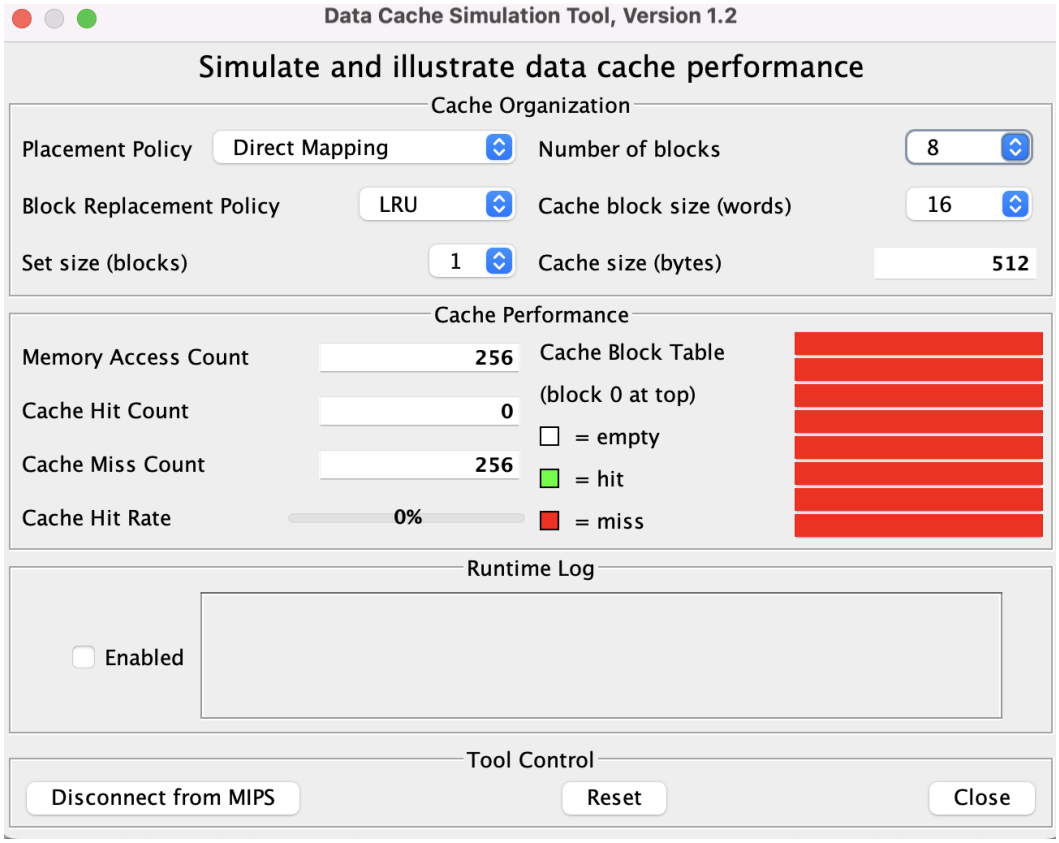
Kết quả:



⇨ Khi chạy chương trình column thì cache hit rate=0 nên hiệu suất bộ nhớ đệm là 0%. Vì khi giá trị của ma trận được điền theo từng cột dọc thì vị trí các ô nhớ không tuần

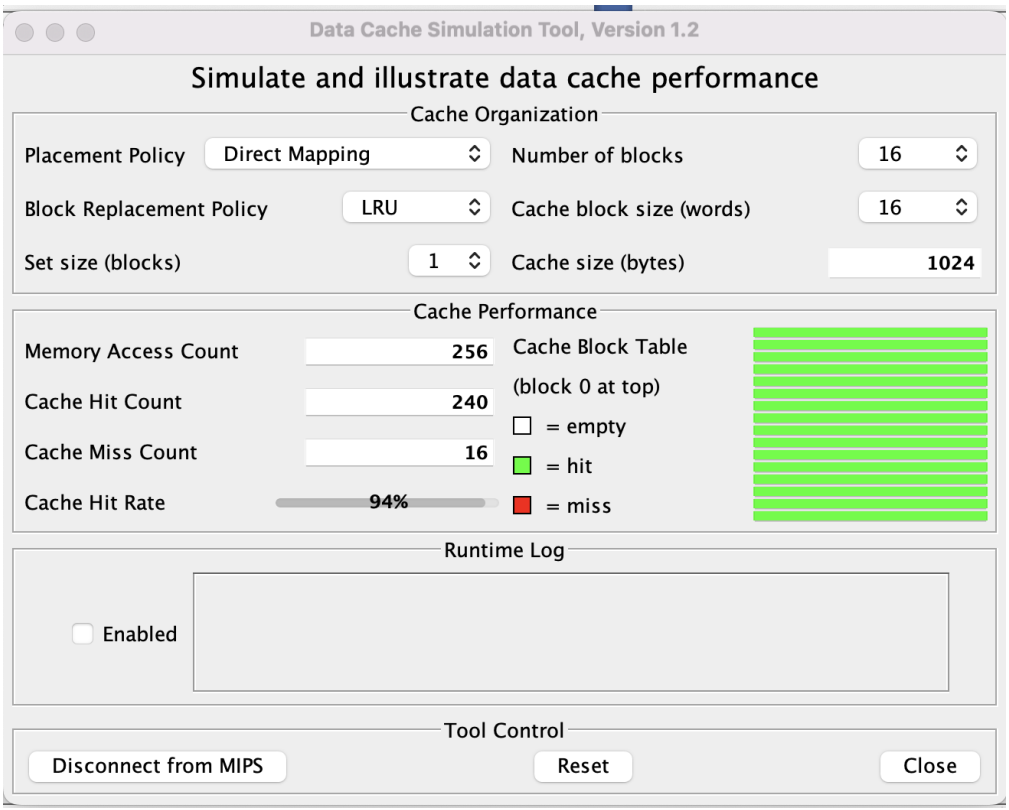
tự như theo hàng mà với mỗi phần tử của ma trận thì phải sau 16 words mới là phần tử kế tiếp của nó.

- Khi chuyển block size sang 16 và giữ nguyên số lượng block là 8:



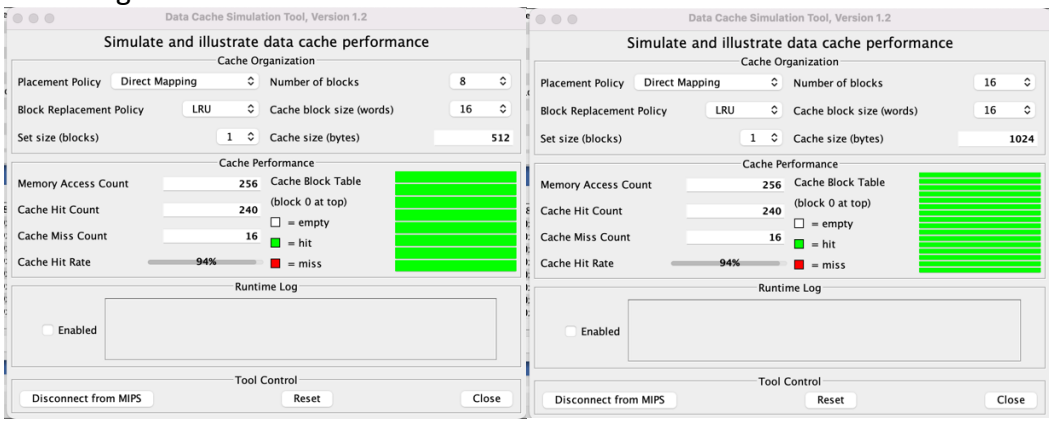
⇨ Cache hit rate là 0% vì mặc dù số word mỗi block là 16 đủ để lưu trữ cả 1 hàng của ma trận nhưng chỉ có 8 block nhớ nên với 8 lần miss đầu thì mỗi lần miss sẽ có 16 words được ghi vào cache và 8 block sẽ đầy sau 8 lần miss nhưng 8 lần ghi sau đó cũng vẫn là những lần miss bởi vì ma trận được duyệt theo cột và nhảy cóc 16 words mỗi lần ghi chứ không tuần tự => 8 lần miss sau, mỗi lần vẫn sẽ có 16 words được ghi vào cache và do 8 block đã đầy hết nên các block sẽ bị ghi đè bắt đầu từ block số 0. Quá trình này lặp lại với mỗi lần cột được duyệt nên tất cả đều là cache miss.

- Khi tăng số lượng block lên 16:

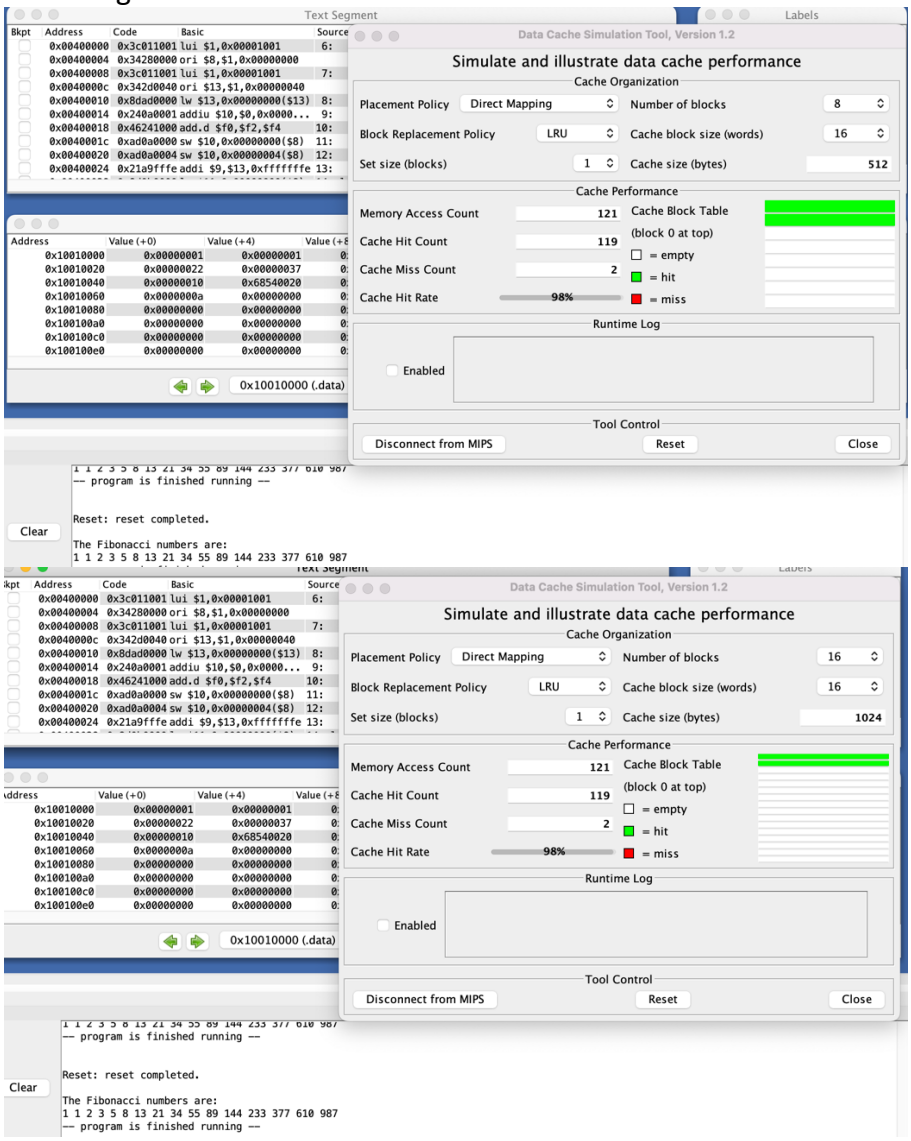


⇨ Hit rate là 94% => Bộ nhớ của cache đúng bằng dung lượng của ma trận. Do đó 16 lần ghi đầu vẫn sẽ là cache miss, tuy nhiên đối với mỗi lượt cache miss thì 16 words sẽ được ghi vào các ô kế tiếp và các lần ghi tiếp theo sẽ gặp các ô này và xảy ra cache hit => Trừ 16 lượt miss ở đầu thì mọi lượt sau đều là cache hit => cache hit rate=94%.

- Thực hiện tương tự bước 13-15 như của chương trình column đối với chương trình row và Fibonacci:
  + Chương trình row:



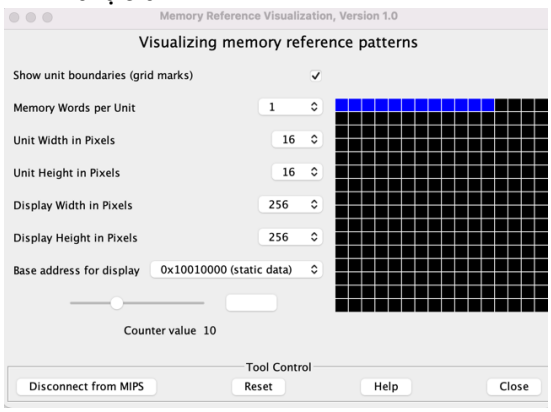
+ Chương trình Fibonacci:



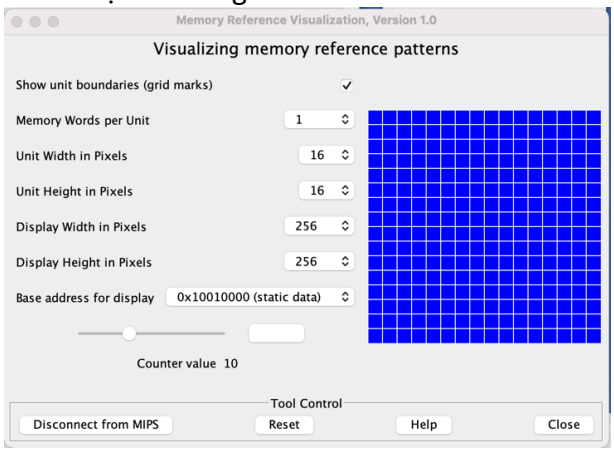
Assignment 2: The Memory Reference Visualization tool:

- Với chương trình row:

  + Khi thực thi:

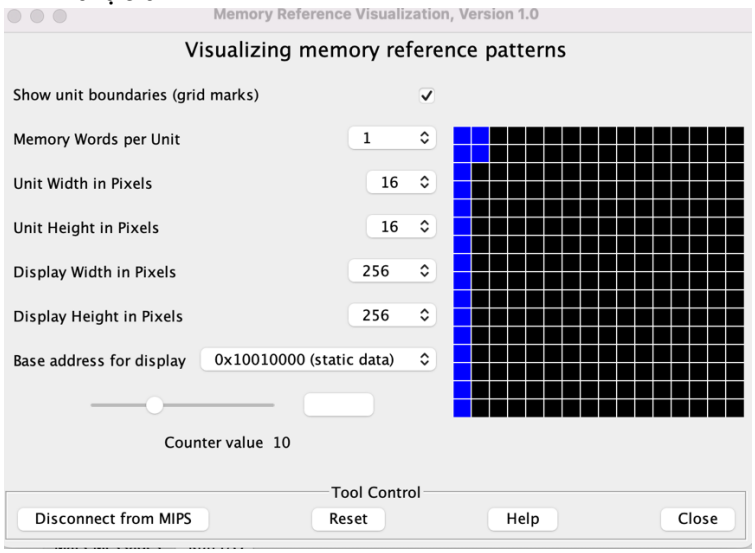


+ Khi thực thi xong:

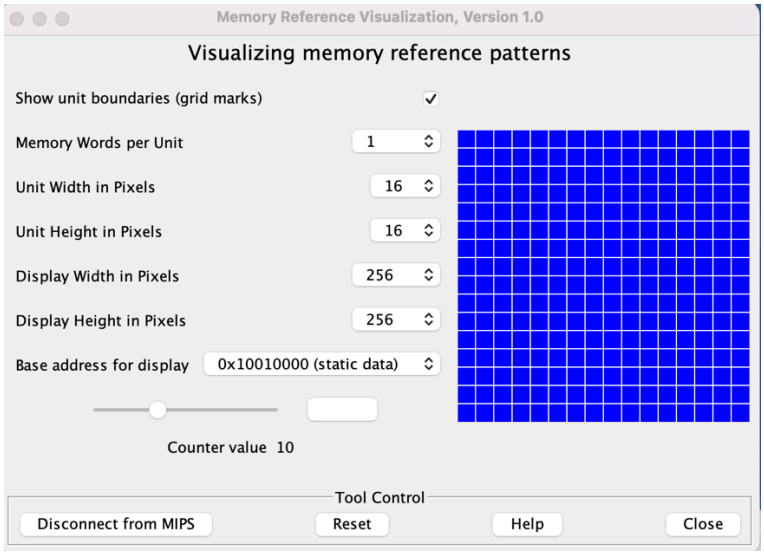


- Với chương trình column:

+ Khi thực thi:



+ Khi thực thi xong:



- Với chương trình Fibonacci khi thực thi xong: