

Assignment 1

Code row-major.asm

```
# Duyệt mảng 16 x 16 theo thứ tự hàng (row-major order).
# Pete Sanderson
# 31 tháng 3 năm 2007
# Để dễ dàng quan sát thứ tự theo hàng, chạy công cụ Hiển thị Tham chiếu Bộ nhớ
# với các cài đặt mặc định trên chương trình này.
# Bạn có thể, cùng lúc hoặc riêng biệt, chạy Bộ mô phỏng Bộ nhớ Cache
# trên chương trình này để quan sát hiệu suất bộ nhớ đệm. So sánh kết quả
# với thuật toán duyệt theo thứ tự cột (column-major order).
# Mã chương trình MIPS tương đương với C/C++/Java như sau:
#   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++;
#       }
#   }
# Lưu ý: Chương trình được cài đặt sẵn cho ma trận 16 x 16. Nếu bạn muốn thay đổi
# kích thước,
#   cần thay đổi ba câu lệnh sau:
#   1. Khai báo kích thước bộ nhớ mảng tại "data:" cần thay đổi từ
#       256 (tương đương 16 * 16) thành #cột * #hàng.
#   2. Lệnh "li" để khởi tạo giá trị cho $t0 cần thay đổi thành số hàng mới.
#   3. Lệnh "li" để khởi tạo giá trị cho $t1 cần thay đổi thành số cột mới.

.data
data: .word    0 : 256    # bộ nhớ lưu trữ cho ma trận 16x16 của các từ
.text
li     t0, 16    # $t0 = số lượng hàng
li     t1, 16    # $t1 = số lượng cột
la     a0, data   # lấy địa chỉ của mảng data
mv     s0, zero   # $s0 = bộ đếm hàng
mv     s1, zero   # $s1 = bộ đếm cột
mv     t2, zero   # $t2 = giá trị sẽ được lưu vào mảng

# Mỗi vòng lặp sẽ lưu giá trị tăng dần của $t2 vào phần tử tiếp theo của ma trận.
# Độ lệch (offset) được tính toán tại mỗi vòng lặp. offset = 4 * (hàng * #cột + cột)
```

Lưu ý: không có tối ưu hóa hiệu suất thời gian trong chương trình này!

```
loop:  mul    s2, s0, t1    # $s2 = hàng * #cột (dãy lệnh gồm hai bước)
      add    s2, s2, s1    # $s2 += bộ đếm cột
      slli   s2, s2, 2     # $s2 *= 4 (dịch trái 2 bit) để tính độ lệch byte
      add    s2, a0, s2    # cộng thêm địa chỉ cơ sở mảng
      sw     t2, 0(s2)     # lưu giá trị vào phần tử của ma trận
      addi   t2, t2, 1     # tăng giá trị cần lưu
```

Điều khiển vòng lặp: Nếu tăng quá cột cuối, reset bộ đếm cột và tăng bộ đếm hàng

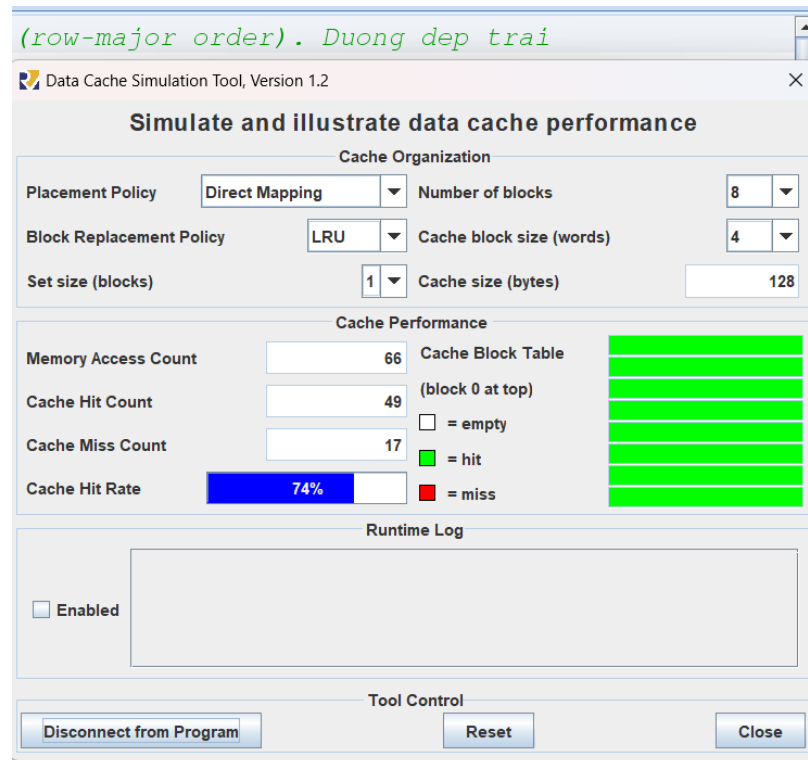
Nếu tăng quá hàng cuối, kết thúc chương trình.

```
      addi   s1, s1, 1     # tăng bộ đếm cột
      bne    s1, t1, loop  # chưa đến cuối hàng, quay lại vòng lặp
      mv     s1, zero      # reset bộ đếm cột
      addi   s0, s0, 1     # tăng bộ đếm hàng
      bne    s0, t0, loop  # chưa đến cuối ma trận, quay lại vòng lặp
```

Chúng ta đã hoàn thành việc duyệt qua ma trận.

```
      li     a7, 10       # dịch vụ hệ thống 10 là thoát chương trình
      ecall              # thoát khỏi chương trình
```

Output:



Cache hit rate cuối cùng là 75%

Mỗi lần xảy ra lỗi (miss), một khối gồm 4 từ sẽ được đưa vào bộ nhớ đệm (cache). Trong quá trình duyệt mảng theo thứ tự hàng (row-major traversal), các phần tử được truy cập theo đúng thứ tự mà chúng được lưu trữ trong bộ nhớ. Do đó, sau mỗi lỗi, sẽ có 3 lần truy cập thành công (hit) vì 3 phần tử tiếp theo sẽ có trong cùng một khối bộ nhớ đệm. Một lỗi khác sẽ xuất hiện khi ánh xạ trực tiếp (Direct Mapping) chuyển sang khối bộ nhớ đệm tiếp theo, và vòng lặp này tiếp tục. Vì vậy, 3 trong tổng số 4 lần truy cập bộ nhớ sẽ được xử lý trong bộ nhớ đệm.

Dự đoán hit rate:

- Khi kích thước khối bộ nhớ đệm tăng từ 4 từ lên 8 từ, tôi dự đoán tỷ lệ hit sẽ đạt khoảng 89%.
- Khi kích thước khối bộ nhớ đệm giảm từ 4 từ xuống 2 từ, tôi dự đoán tỷ lệ hit sẽ giảm xuống khoảng 52%.
- Giải thích: Khi kích thước khối bộ nhớ đệm lớn hơn (8 từ), nhiều phần tử sẽ được tải vào bộ nhớ đệm cùng lúc, giúp tăng số lần truy cập thành công (hit) sau mỗi lỗi. Ngược lại, với khối bộ nhớ đệm nhỏ hơn (2 từ), ít phần tử được tải vào bộ nhớ đệm mỗi lần, làm tăng tần suất lỗi và giảm tỷ lệ hit.

Đổi 4 words thành 8 words:

hàng (row-major order). Dương đẹp trai

Data Cache Simulation Tool, Version 1.2

Simulate and illustrate data cache performance

Cache Organization

Placement Policy: **Direct Mapping** | Number of blocks: **8**

Block Replacement Policy: **LRU** | Cache block size (words): **8**

Set size (blocks): **1** | Cache size (bytes): **256**

Cache Performance

Memory Access Count: **256** | Cache Block Table (block 0 at top)

Cache Hit Count: **224** | ☐ = empty

Cache Miss Count: **32** | ☒ = hit

Cache Hit Rate: **88%** | ☐ = miss

Runtime Log

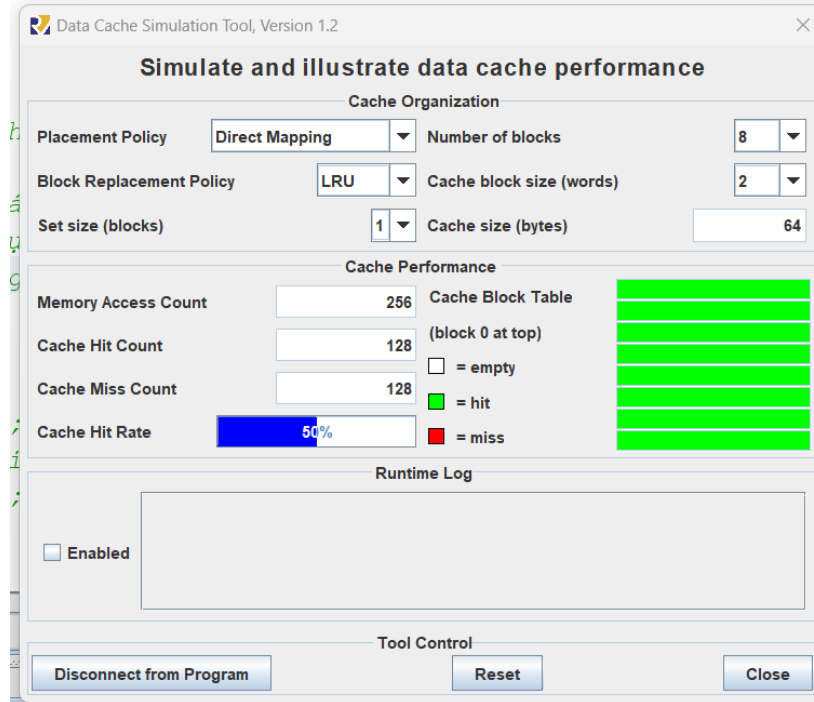
☐ Enabled

Tool Control

Disconnect from Program | **Reset** | **Close**

4 words thành 2 words

hàng (row-major order). Duong dep trai



Code column-major.asm

```
#
# Duyệt mảng 16 x 16 theo thứ tự cột (column-major order).
# Pete Sanderson
# 31 tháng 3 năm 2007
#
# Để dễ dàng quan sát thứ tự theo cột, chạy công cụ Hiển thị Tham chiếu Bộ nhớ
# với các cài đặt mặc định trên chương trình này.
# Bạn có thể, cùng lúc hoặc riêng biệt, chạy Bộ mô phỏng Bộ nhớ Cache
# trên chương trình này để quan sát hiệu suất bộ nhớ đệm. So sánh kết quả
# với thuật toán duyệt theo thứ tự hàng (row-major order).
#
# Mã chương trình MIPS tương đương với C/C++/Java như sau:
# 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++;
#     }
# }
```

```

# }
#
# Lưu ý: Chương trình được cài đặt sẵn cho ma trận 16 x 16. Nếu bạn muốn thay đổi
kích thước,
# cần thay đổi ba câu lệnh sau:
# 1. Khai báo kích thước bộ nhớ mảng tại "data:" cần thay đổi từ
# 256 (tương đương 16 * 16) thành #cột * #hàng.
# 2. Lệnh "li" để khởi tạo giá trị cho $t0 cần thay đổi thành số hàng mới.
# 3. Lệnh "li" để khởi tạo giá trị cho $t1 cần thay đổi thành số cột mới.

.data
data: .word 0 : 256 # ma trận 16x16 của các từ
.text
li t0, 16 # $t0 = số lượng hàng
li t1, 16 # $t1 = số lượng cột
la a0, data # lấy địa chỉ của mảng data
mv s0, zero # $s0 = bộ đếm hàng
mv s1, zero # $s1 = bộ đếm cột
mv t2, zero # $t2 = giá trị sẽ được lưu vào mảng

# Mỗi vòng lặp sẽ lưu giá trị tăng dần của $t2 vào phần tử tiếp theo của ma trận.
# Độ lệch (offset) được tính toán tại mỗi vòng lặp. offset = 4 * (hàng * #cột + cột)
# Lưu ý: không có tối ưu hóa hiệu suất thời gian trong chương trình này!

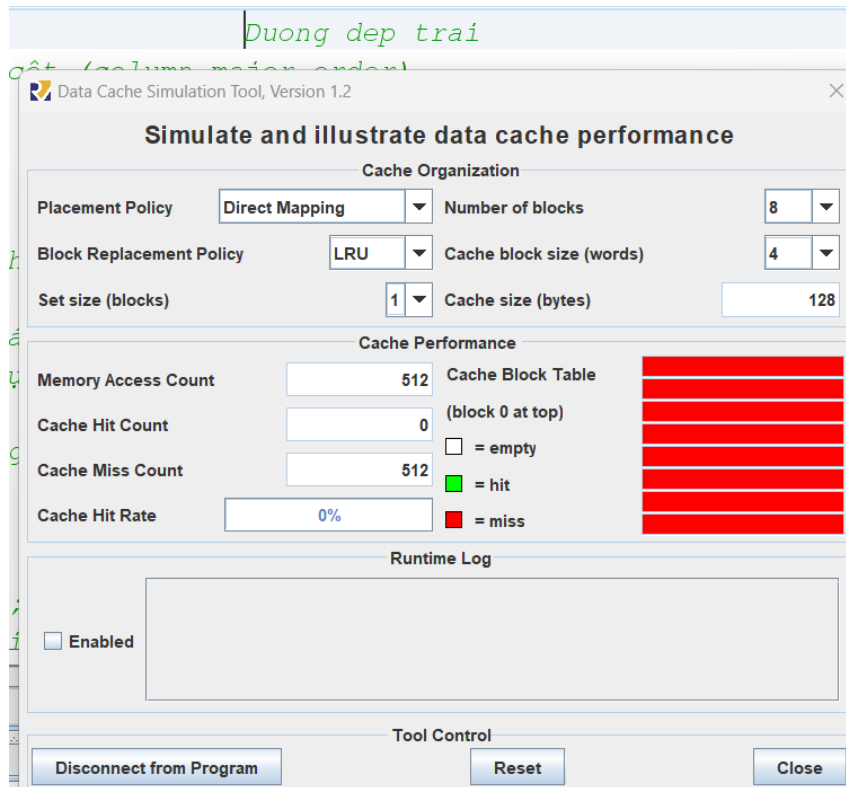
loop: mul s2, s0, t1 # $s2 = hàng * #cột (dãy lệnh gồm hai bước)
add s2, s2, s1 # $s2 += bộ đếm cột
slli s2, s2, 2 # $s2 *= 4 (dịch trái 2 bit) để tính độ lệch byte
add s2, a0, s2 # cộng thêm địa chỉ cơ sở mảng
sw t2, 0(s2) # lưu giá trị vào phần tử của ma trận
addi t2, t2, 1 # tăng giá trị cần lưu

# Điều khiển vòng lặp: Nếu tăng quá cột cuối, reset bộ đếm hàng và tăng bộ đếm cột
# Nếu tăng quá cột cuối cùng, kết thúc chương trình.
addi s0, s0, 1 # tăng bộ đếm hàng
bne s0, t0, loop # chưa đến cuối cột, quay lại vòng lặp
mv s0, zero # reset bộ đếm hàng
addi s1, s1, 1 # tăng bộ đếm cột
bne s1, t1, loop # chưa đến cuối ma trận (chưa qua cột cuối cùng), quay lại
vòng lặp

# Chúng ta đã hoàn thành việc duyệt qua ma trận.
li a7, 10 # dịch vụ hệ thống 10 là thoát chương trình
ecall # thoát khỏi chương trình

```

Output:



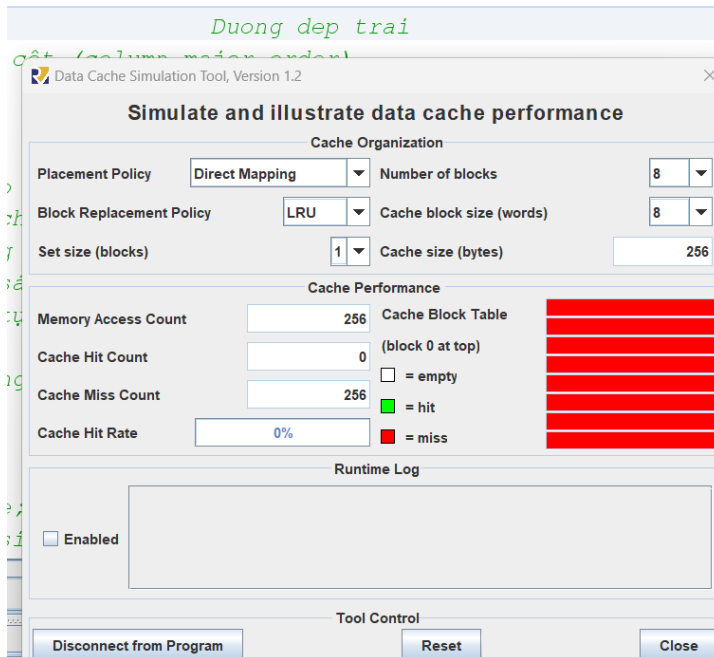
Giải thích: Vấn đề là các vị trí bộ nhớ bây giờ không được truy cập theo thứ tự liên tiếp như trước, mà mỗi lần truy cập cách nhau 16 từ (words) so với lần truy cập trước (theo kiểu vòng tròn). Với các cài đặt chúng ta đã sử dụng, không có hai lần truy cập bộ nhớ liên tiếp nào xảy ra trong cùng một khối bộ nhớ, vì vậy mỗi lần truy cập đều là một lỗi (miss).

Dự đoán tỷ lệ hit:

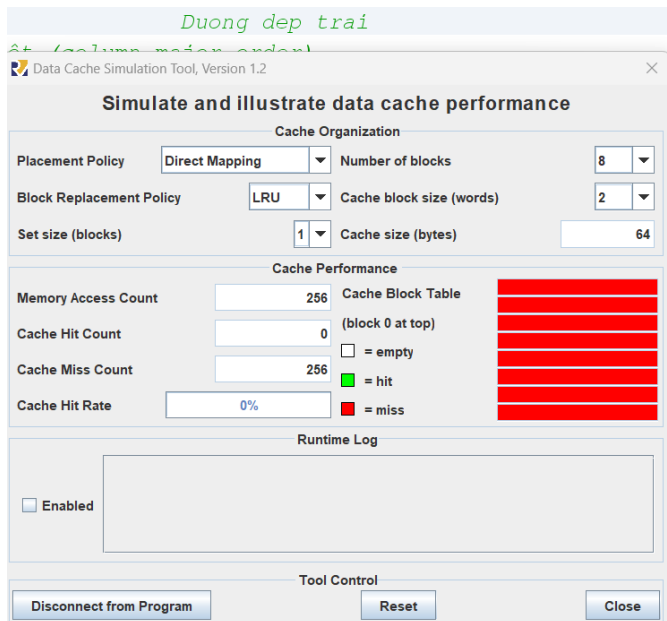
- Nếu kích thước khối bộ nhớ đệm được tăng từ 4 từ lên 8 từ, tôi dự đoán tỷ lệ hit sẽ không thay đổi.
- Nếu kích thước khối bộ nhớ đệm giảm từ 4 từ xuống 2 từ, tôi dự đoán tỷ lệ hit sẽ không thay đổi.
- Giải thích: Việc thay đổi kích thước khối bộ nhớ đệm không có tác dụng vì vẫn chỉ có một lần truy cập vào mỗi khối, lần đầu tiên, trước khi khối đó bị thay thế bởi một khối mới.

Xác minh dự đoán:

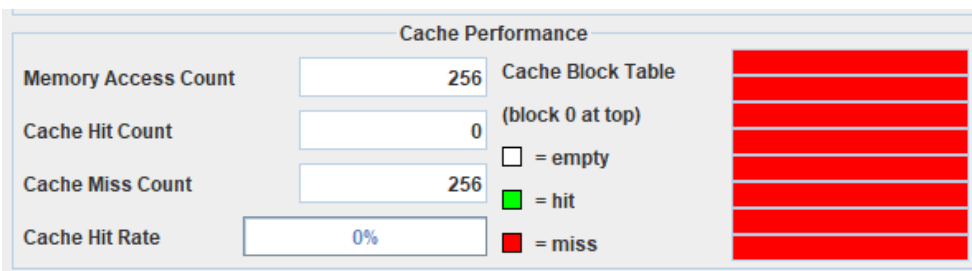
- Từ 4 từ lên 8 từ:



- Từ 4 từ xuống 2 từ:

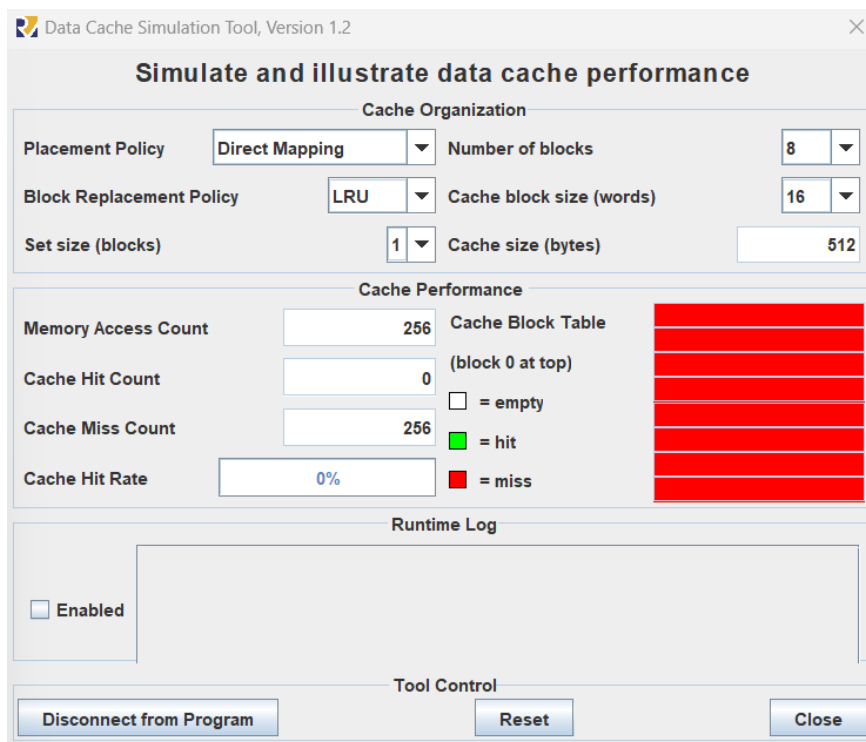
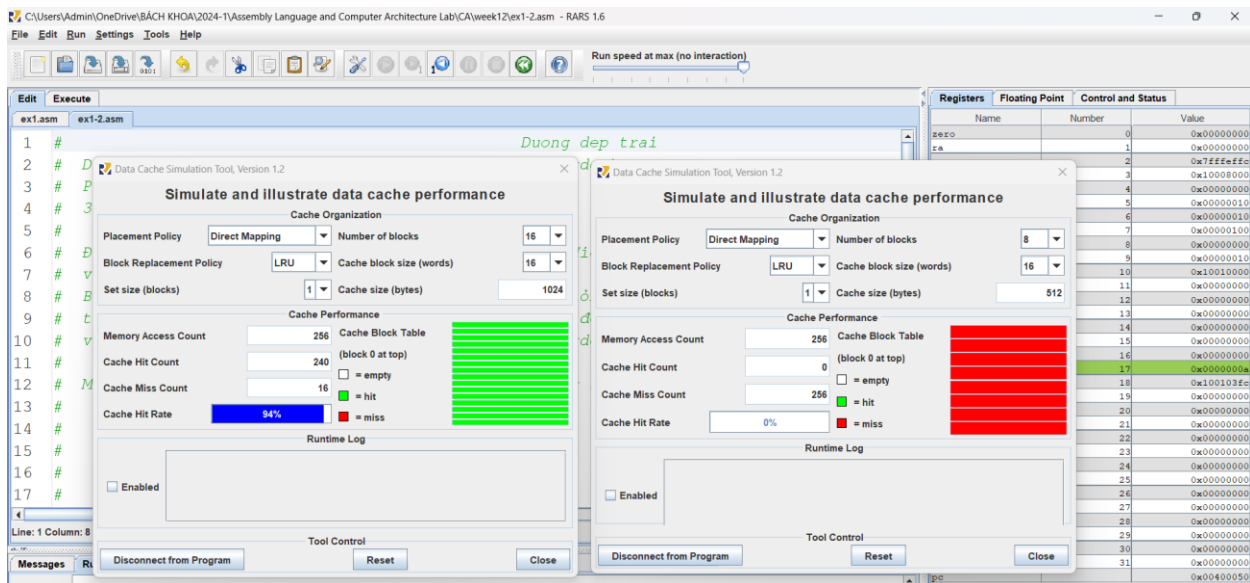


Hiệu suất bộ nhớ đệm (cache performance) của chương trình này là gì?



Trả lời câu hỏi 12 + 13 +14:

Output:



Kích thước khối bộ nhớ đệm là 16 không giúp cải thiện hiệu suất vì vẫn chỉ có một lần truy cập vào mỗi khối, đó là lỗi ban đầu (miss), trước khi khối đó bị thay thế bởi một khối mới.

Data Cache Simulation Tool, Version 1.2

Simulate and illustrate data cache performance

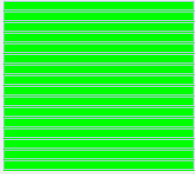
Cache Organization

Placement Policy: Number of blocks:

Block Replacement Policy: Cache block size (words):

Set size (blocks): Cache size (bytes):

Cache Performance

Memory Access Count: Cache Block Table: 

Cache Hit Count: (block 0 at top)

Cache Miss Count: ☐ = empty ☒ = hit ☐ = miss

Cache Hit Rate:

Runtime Log

☐ Enabled

Tool Control

Ở điểm này, toàn bộ ma trận sẽ vừa vặn với bộ nhớ đệm, và khi một khối được đọc vào bộ nhớ đệm, nó sẽ không bao giờ bị thay thế nữa. Chỉ có lần truy cập đầu tiên vào một khối mới gây ra lỗi (miss).

Assignment 2

row-major.asm

```
# Duyệt mảng 16 x 16 theo thứ tự hàng (row-major order).
# Pete Sanderson
# 31 tháng 3 năm 2007
# Dễ dàng quan sát thứ tự theo hàng, chạy công cụ Hiển thị Tham chiếu Bộ nhớ
# với các cài đặt mặc định trên chương trình này.
# Bạn có thể, cùng lúc hoặc riêng biệt, chạy Bộ mô phỏng Bộ nhớ Cache
# trên chương trình này để quan sát hiệu suất bộ nhớ đệm. So sánh kết quả
# với thuật toán duyệt theo thứ tự cột (column-major order).
#
# Mã chương trình MIPS tương đương với C/C++/Java như sau:
# 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++;
#     }
# }
```

```

#   }
#   }
#
# Lưu ý: Chương trình được cài đặt sẵn cho ma trận 16 x 16. Nếu bạn muốn thay đổi
kích thước,
#   cần thay đổi ba câu lệnh sau:
#   1. Khai báo kích thước bộ nhớ mảng tại "data:" cần thay đổi từ
#   256 (tương đương 16 * 16) thành #cột * #hàng.
#   2. Lệnh "li" để khởi tạo giá trị cho $t0 cần thay đổi thành số hàng mới.
#   3. Lệnh "li" để khởi tạo giá trị cho $t1 cần thay đổi thành số cột mới.

.data
data: .word 0 : 256    # bộ nhớ cho ma trận 16x16 của các từ
.text
li    t0, 16          # $t0 = số lượng hàng
li    t1, 16          # $t1 = số lượng cột
la    a0, data         # lấy địa chỉ của mảng data
mv    s0, zero         # $s0 = bộ đếm hàng
mv    s1, zero         # $s1 = bộ đếm cột
mv    t2, zero         # $t2 = giá trị sẽ được lưu vào mảng

# Mỗi vòng lặp sẽ lưu giá trị tăng dần của $t2 vào phần tử tiếp theo của ma trận.
# Độ lệch (offset) được tính toán tại mỗi vòng lặp. offset = 4 * (hàng * #cột + cột)
# Lưu ý: không có tối ưu hóa hiệu suất thời gian trong chương trình này!

loop: mul    s2, s0, t1    # $s2 = hàng * #cột (dãy lệnh gồm hai bước)
      add    s2, s2, s1    # $s2 += bộ đếm cột
      slli   s2, s2, 2     # $s2 *= 4 (dịch trái 2 bit) để tính độ lệch byte
      add    s2, a0, s2    # cộng thêm địa chỉ cơ sở mảng
      sw     t2, 0(s2)     # lưu giá trị vào phần tử của ma trận
      addi   t2, t2, 1     # tăng giá trị cần lưu

# Điều khiển vòng lặp: Nếu tăng quá cột cuối, reset bộ đếm cột và tăng bộ đếm hàng
#   Nếu tăng quá hàng cuối cùng, kết thúc chương trình.
      addi   s1, s1, 1     # tăng bộ đếm cột
      bne    s1, t1, loop  # chưa đến cuối hàng, quay lại vòng lặp
      mv     s1, zero      # reset bộ đếm cột
      addi   s0, s0, 1     # tăng bộ đếm hàng
      bne    s0, t0, loop  # chưa đến cuối ma trận (chưa qua hàng cuối cùng), quay lại
vòng lặp

# Chúng ta đã hoàn thành việc duyệt qua ma trận.
li    a7, 10            # dịch vụ hệ thống 10 là thoát chương trình

```

ecall	# thoát khỏi chương trình
-------	---------------------------

Phần khai báo dữ liệu:

.data

data: .word 0 : 256 # bộ nhớ cho ma trận 16x16 của các từ

- Dòng trên khai báo một vùng nhớ có tên là data dùng để lưu trữ ma trận 16x16 gồm 256 phần tử (mỗi phần tử có kích thước 4 byte, tương ứng với 1 từ).
- Dữ liệu ban đầu của các phần tử trong ma trận là 0.

Phần mã lệnh chính:

.text

li t0, 16 # \$t0 = số lượng hàng

li t1, 16 # \$t1 = số lượng cột

la a0, data # lấy địa chỉ của mảng data

mv s0, zero # \$s0 = bộ đếm hàng

mv s1, zero # \$s1 = bộ đếm cột

mv t2, zero # \$t2 = giá trị sẽ được lưu vào mảng

- Dòng này khởi tạo các giá trị và bộ đếm:
 - \$t0 chứa số lượng hàng của ma trận (16).
 - \$t1 chứa số lượng cột của ma trận (16).
 - \$a0 chứa địa chỉ của mảng data.
 - \$s0 và \$s1 là bộ đếm dùng để duyệt qua hàng và cột của ma trận.
 - \$t2 là biến dùng để lưu trữ giá trị cần ghi vào ma trận, được khởi tạo là 0.

Vòng lặp chính:

*loop: mul s2, s0, t1 # \$s2 = hàng * #cột (dãy lệnh gồm hai bước)*

add s2, s2, s1 # \$s2 += bộ đếm cột

*slli s2, s2, 2 # \$s2 *= 4 (dịch trái 2 bit) để tính độ lệch byte*

add s2, a0, s2 # cộng thêm địa chỉ cơ sở mảng

sw t2, 0(s2) # lưu giá trị vào phần tử của ma trận

addi t2, t2, 1 # tăng giá trị cần lưu

- **Vòng lặp chính** này duyệt qua toàn bộ ma trận theo thứ tự hàng và cột.
- Để tính vị trí (địa chỉ) của phần tử cần truy cập, chương trình thực hiện các bước sau:
 1. **mul s2, s0, t1**: Tính giá trị hàng * số cột, lưu vào \$s2.
 2. **add s2, s2, s1**: Cộng thêm số cột hiện tại vào \$s2, để có chỉ số tổng quát hàng * số cột + cột.
 3. **slli s2, s2, 2**: Dịch trái 2 bit, vì mỗi phần tử ma trận chiếm 4 byte (1 từ), làm cho giá trị trong \$s2 trở thành độ lệch tính theo byte.
 4. **add s2, a0, s2**: Thêm địa chỉ cơ sở của mảng data vào độ lệch tính được, từ đó có được địa chỉ của phần tử cần truy cập.
 5. **sw t2, 0(s2)**: Lưu giá trị \$t2 vào vị trí bộ nhớ mà \$s2 chỉ tới (phần tử ma trận tại hàng và cột hiện tại).
 6. **addi t2, t2, 1**: Tăng giá trị \$t2 lên 1 để chuẩn bị cho lần lưu tiếp theo.

Điều khiển vòng lặp:

addi s1, s1, 1 # tăng bộ đếm cột

bne s1, t1, loop # chưa đến cuối hàng, quay lại vòng lặp

mv s1, zero # reset bộ đếm cột

addi s0, s0, 1 # tăng bộ đếm hàng

bne s0, t0, loop # chưa đến cuối ma trận (chưa qua hàng cuối cùng), quay lại vòng lặp

- **Điều khiển vòng lặp** giúp duyệt qua các hàng và cột của ma trận:
 1. **addi s1, s1, 1**: Tăng bộ đếm cột.
 2. **bne s1, t1, loop**: Nếu chưa đến cột cuối, quay lại vòng lặp.
 3. **mv s1, zero**: Khi đã duyệt hết cột trong một hàng, reset bộ đếm cột.
 4. **addi s0, s0, 1**: Tăng bộ đếm hàng.
 5. **bne s0, t0, loop**: Nếu chưa đến hàng cuối, quay lại vòng lặp.

Kết thúc chương trình:

li a7, 10 # dịch vụ hệ thống 10 là thoát chương trình

ecall

thoát khỏi chương trình

- **Kết thúc chương trình** bằng cách gọi dịch vụ hệ thống ecall với mã dịch vụ là 10 (thoát chương trình).



column-major.asm

Code:

```
# Duyệt qua ma trận 16 x 16 theo thứ tự cột.
# Pete Sanderson
# 31 tháng 3, 2007
#
# Để dễ dàng quan sát thứ tự theo cột, hãy chạy công cụ Visualization
# để theo dõi tham chiếu bộ nhớ với các cài đặt mặc định trên chương trình này.
# Bạn có thể chạy công cụ Mô phỏng Bộ nhớ đệm (Data Cache Simulator)
# cùng lúc hoặc riêng biệt trên chương trình này để quan sát hiệu suất bộ nhớ đệm.
# So sánh kết quả với thuật toán duyệt qua theo thứ tự hàng.
#
# Mã tương đương C/C++/Java với chương trình MIPS này là:
#   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++;
#       }
#   }
#
# Lưu ý: Chương trình này được cấu hình sẵn cho ma trận 16 x 16. Nếu bạn muốn thay
# đổi kích thước,
#   cần thay đổi ba dòng sau:
#   1. Kích thước bộ nhớ lưu trữ mảng tại "data:" cần thay đổi từ
#       256 (tức là 16 * 16) thành #cột * #hàng.
#   2. Lệnh "li" để khởi tạo $t0 cần thay đổi thành số hàng mới.
#   3. Lệnh "li" để khởi tạo $t1 cần thay đổi thành số cột mới.
#
.data
data: .word    0 : 256    # bộ nhớ lưu trữ ma trận 16x16 của các từ
.text
li      t0, 16    # $t0 = số lượng hàng
li      t1, 16    # $t1 = số lượng cột
la      a0, data  # lấy địa chỉ của mảng data
mv      s0, zero  # $s0 = bộ đếm hàng
mv      s1, zero  # $s1 = bộ đếm cột
mv      t2, zero  # $t2 = giá trị sẽ được lưu vào mảng
# Mỗi lần lặp trong vòng lặp sẽ lưu giá trị tăng dần vào phần tử tiếp theo trong ma
# trận.
# Độ lệch bộ nhớ được tính ở mỗi lần lặp. offset = 4 * (row*#cols+col)
```

```

# Lưu ý: không cố gắng tối ưu hiệu suất thời gian chạy!
loop:  mul    s2, s0, t1    # $s2 = row * #cols (dãy lệnh gồm hai bước)
        add    s2, s2, s1    # $s2 += bộ đếm cột
        slli   s2, s2, 2    # $s2 *= 4 (dịch trái 2 bit) để tính độ lệch byte
        add    s2, a0, s2    # cộng thêm địa chỉ cơ sở mảng
        sw     t2, 0(s2)    # lưu giá trị vào phần tử của ma trận
        addi   t2, t2, 1    # tăng giá trị cần lưu
# Điều khiển vòng lặp: Nếu bộ đếm hàng vượt quá cuối cột, reset bộ đếm hàng và tăng
# bộ đếm cột.
# Nếu vượt quá cột cuối cùng, kết thúc chương trình.
        addi   s0, s0, 1    # tăng bộ đếm hàng
        bne    s0, t0, loop  # chưa đến cuối cột nên quay lại vòng lặp
        mv     s0, zero     # reset bộ đếm hàng
        addi   s1, s1, 1    # tăng bộ đếm cột
        bne    s1, t1, loop  # quay lại vòng lặp nếu chưa đến cuối ma trận (chưa qua cột
# cuối cùng)
# Chương trình đã duyệt xong ma trận.
        li     a7, 10       # dịch vụ hệ thống 10 là thoát chương trình
        ecall              # kết thúc chương trình.

```

Duyệt qua ma trận theo thứ tự cột: Chương trình sẽ đi qua từng phần tử trong ma trận theo thứ tự cột, thay vì thứ tự hàng như trong chương trình duyệt theo hàng (row-major order).

Khởi tạo dữ liệu: Các biến được khởi tạo như sau:

- \$t0: số hàng của ma trận (16).
- \$t1: số cột của ma trận (16).
- \$a0: chứa địa chỉ cơ sở của mảng data.
- \$s0: bộ đếm hàng.
- \$s1: bộ đếm cột.
- \$t2: giá trị sẽ được lưu vào ma trận, khởi tạo là 0.

Tính toán offset: Để tính toán vị trí của từng phần tử trong ma trận, chương trình thực hiện các phép toán để tính độ lệch và lưu trữ giá trị vào đúng phần tử trong mảng.

Điều khiển vòng lặp: Mỗi vòng lặp sẽ duyệt qua một cột và sau khi duyệt hết cột, chương trình sẽ reset bộ đếm hàng và chuyển sang cột tiếp theo. Vòng lặp tiếp tục cho đến khi duyệt hết tất cả các cột và hàng trong ma trận.

Thoát chương trình: Sau khi đã duyệt hết ma trận, chương trình kết thúc bằng cách gọi dịch vụ hệ thống với mã 10, yêu cầu thoát khỏi chương trình.

Output:



fibonacci.asm

Code:

Tính 12 số Fibonacci đầu tiên và lưu vào mảng, sau đó in ra màn hình
--


```

.data
fibs: .word 0 : 16      # "mảng" chứa 12 giá trị Fibonacci
size: .word 16          # kích thước của mảng
.text
la t0, fibs            # tải địa chỉ của mảng fibs
la t5, size            # tải địa chỉ của biến size
lw t5, 0(t5)           # tải kích thước của mảng
li t2, 1               # 1 là số Fibonacci đầu tiên và thứ hai
sw t2, 0(t0)           # F[0] = 1
sw t2, 4(t0)           # F[1] = F[0] = 1
addi t1, t5, -2        # Bộ đếm vòng lặp, sẽ lặp (size-2) lần
loop: lw t3, 0(t0)      # Lấy giá trị từ mảng F[n]
lw t4, 4(t0)           # Lấy giá trị từ mảng F[n+1]
add t2, t3, t4         # $t2 = F[n] + F[n+1]
sw t2, 8(t0)           # Lưu F[n+2] = F[n] + F[n+1] vào mảng
addi t0, t0, 4         # tăng địa chỉ mảng Fibonacci
addi t1, t1, -1        # giảm bộ đếm vòng lặp
bgtz t1, loop          # tiếp tục nếu chưa kết thúc
la a0, fibs            # tham số đầu tiên cho hàm in (mảng)
add a1, zero, t5       # tham số thứ hai cho hàm in (kích thước)
jal print              # gọi hàm in ra màn hình
li a7, 10              # gọi hệ thống để thoát
ecall                  # thoát chương trình

```

hàm in các số trên cùng một dòng.

```

.data
space:.asciz " "        # ký tự khoảng trắng để tách giữa các số
head: .asciz "Các số Fibonacci là:\n"
.text
print:add t0, zero, a0  # địa chỉ bắt đầu của mảng
add t1, zero, a1       # khởi tạo bộ đếm vòng lặp bằng kích thước mảng
la a0, head            # tải địa chỉ của dòng tiêu đề
li a7, 4               # dịch vụ in chuỗi
ecall                  # in tiêu đề
out: lw a0, 0(t0)       # tải số Fibonacci từ mảng để in
li a7, 1               # dịch vụ in số nguyên
ecall                  # in số Fibonacci
la a0, space           # tải địa chỉ của ký tự khoảng trắng
li a7, 4               # dịch vụ in chuỗi
ecall                  # in ký tự khoảng trắng
addi t0, t0, 4         # tăng địa chỉ mảng
addi t1, t1, -1        # giảm bộ đếm vòng lặp

```

bgtz t1, out	# tiếp tục vòng lặp nếu chưa hết
jr ra	# quay lại

1. Khởi tạo Mảng Fibonacci:

- Chương trình tính toán 12 số Fibonacci đầu tiên và lưu chúng vào một mảng fibs. Các số Fibonacci đầu tiên là $F[0] = 1$ và $F[1] = 1$.
- Vòng lặp tính toán tiếp các giá trị Fibonacci từ $F[2]$ đến $F[11]$ bằng cách cộng hai số trước đó trong mảng ($F[n] + F[n+1]$).

2. In ra kết quả:

- Sau khi tính xong mảng Fibonacci, chương trình gọi một hàm in để in các số Fibonacci ra màn hình theo một dòng, cách nhau bởi dấu cách (" ").

3. Các chỉ thị chính:

- la được sử dụng để tải địa chỉ của mảng hoặc các biến vào các thanh ghi.
- lw và sw dùng để đọc và ghi giá trị từ mảng.
- addi được dùng để thay đổi giá trị của bộ đếm hoặc địa chỉ.
- bgtz kiểm tra xem bộ đếm vòng lặp còn lớn hơn 0 không, nếu có thì tiếp tục lặp lại.
- Hàm print in các số Fibonacci ra màn hình, mỗi số được in kèm với khoảng trắng.

Tóm tắt chi tiết về các phần chính:

1. Khởi tạo số Fibonacci đầu tiên:

- $F[0] = 1$ và $F[1] = 1$ được khởi tạo ngay từ đầu.
- Các số tiếp theo được tính bằng cách cộng hai số trước đó.

2. Vòng lặp tính toán Fibonacci:

- Sử dụng vòng lặp để tính toán tiếp các số Fibonacci từ $F[2]$ đến $F[11]$. Mỗi lần lặp sẽ lấy hai giá trị trước đó trong mảng, tính tổng và lưu kết quả vào mảng.

3. In kết quả:

- Chương trình in ra chuỗi "Các số Fibonacci là:".
- Sau đó in từng số Fibonacci, mỗi số được in cách nhau bởi một dấu cách.

4. Kết thúc chương trình:

- Sau khi in xong, chương trình gọi dịch vụ hệ thống để thoát chương trình (ecall với a7 = 10).

Memory Reference Visualization, Version 1.0

Visualizing memory reference patterns

Show unit boundaries (grid marks) ☒

Memory Words per Unit ▼


Unit Width in Pixels ▼

Unit Height in Pixels ▼

Display Width in Pixels ▼

Display Height in Pixels ▼

Base address for display ▼



Counter value 10

Tool Control

Các s? Fibonacci là:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
-- program is finished running (0) --