Lab 3: Cache Simulation

Introduction

In this lab, you will be simulating cache hit ratios or hit rates for direct mapping, N-way associative, and fully associative with LRU and MRU replacement policies. We will be using Google Colaboratory environment (Colab) which is a Jupyter Notebook-like environment where the default coding language is Python, but supports Linux system command line interface and scripting. So instead of installing Linux distribution on virtual machine or installing Python on your computer, Colab provides a user ready interface to readily start our simulation. Also, this lab will also refer you to an online page to help you review some of the concepts for your understanding of cache simulation. After completing this lab, you are required to do the designated homework.

1. Concepts from Online Page

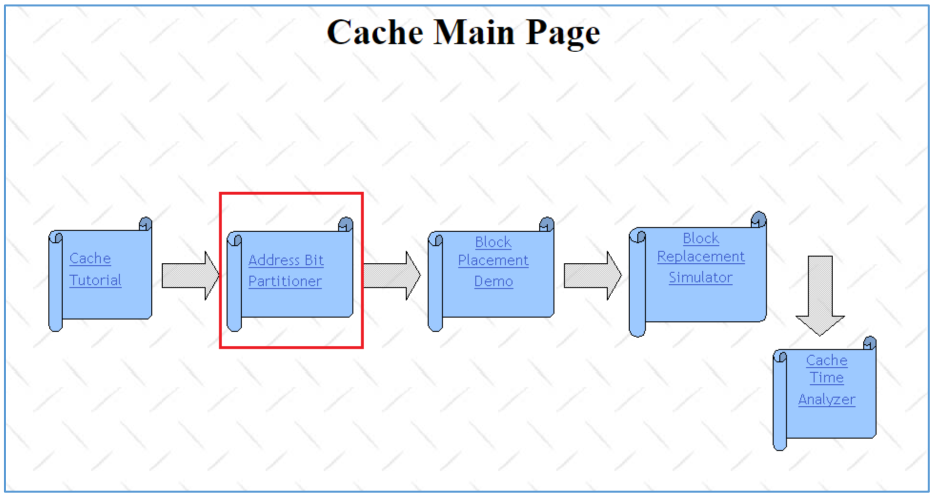
Objectives

The objective in this section is just to help you review and visualize some concepts for cache simulation using an online page.

A. Cache Address Structure and Placement Visualization

>> Go to this page: http://www.ecs.umass.edu/ece/koren/architecture/Cache/frame0.htm

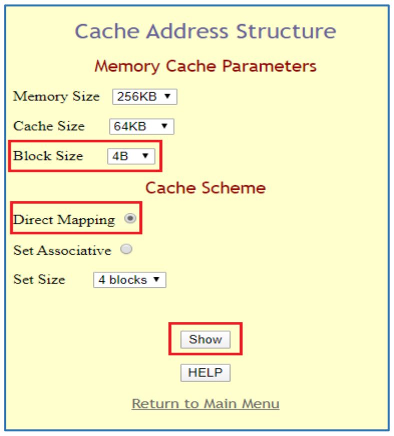
1. Go to the Address Bit Partitioner section.



This part will help you visualize the cache address structure for direct mapping and N-way associative.

Input the settings for **direct mapping**:

The output address partition for 256KB main memory, 64KB cache, and 4Byte word alignment (Block Size)



Cache Address Structure

Memory Cache Parameters

Memory Size 256KB

Cache Size 64KB

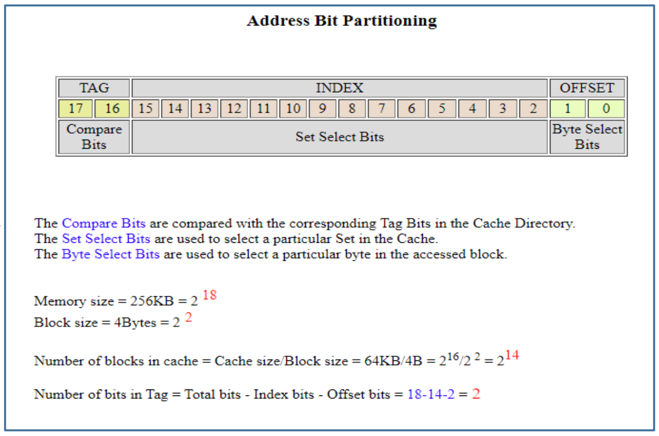
Block Size 4B

Cache Scheme

Direct Mapping ●

Set Associative ○

Set Size 4 blocks



Address Bit Partitioning

TAG		INDEX														OFFSET	
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Compare Bits		Set Select Bits														Byte Select Bits	

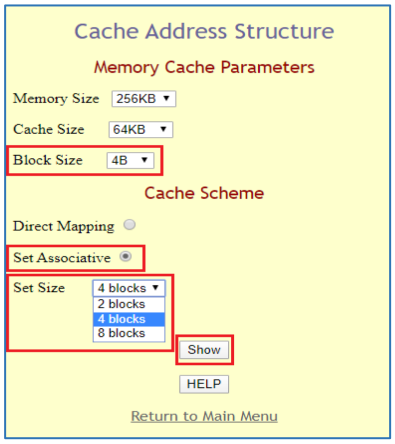
The Compare Bits are compared with the corresponding Tag Bits in the Cache Directory.
The Set Select Bits are used to select a particular Set in the Cache.
The Byte Select Bits are used to select a particular byte in the accessed block.

Memory size = 256KB = 2^{18}
Block size = 4Bytes = 2^{2}

Number of blocks in cache = Cache size/Block size = 64KB/4B = $2^{16}/2^{2} = 2^{14}$

Number of bits in Tag = Total bits - Index bits - Offset bits = 18-14-2 = 2

Settings for **4-way associative mapping**:



Cache Address Structure

Memory Cache Parameters

Memory Size 256KB

Cache Size 64KB

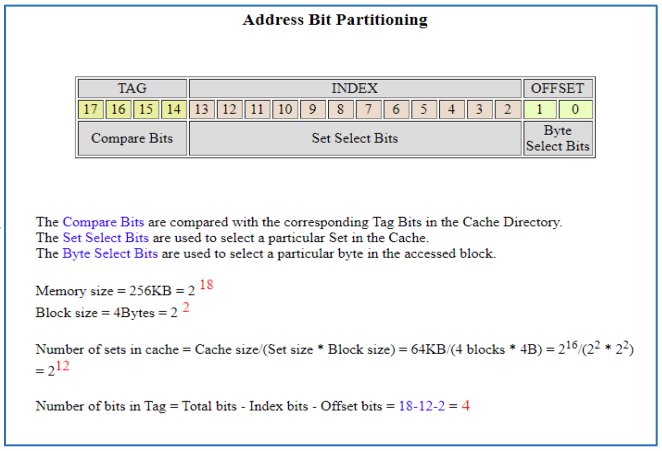
Block Size 4B

Cache Scheme

Direct Mapping ○

Set Associative ●

Set Size 4 blocks



Address Bit Partitioning

TAG				INDEX												OFFSET	
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Compare Bits				Set Select Bits												Byte Select Bits	

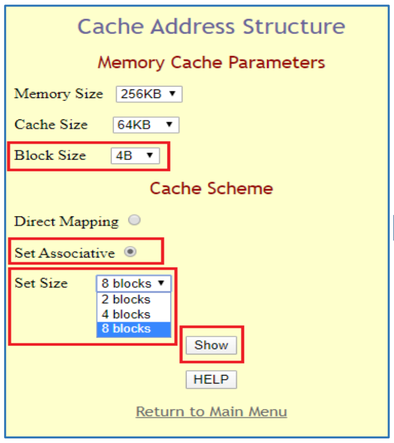
The Compare Bits are compared with the corresponding Tag Bits in the Cache Directory.
The Set Select Bits are used to select a particular Set in the Cache.
The Byte Select Bits are used to select a particular byte in the accessed block.

Memory size = 256KB = 2^{18}
Block size = 4Bytes = 2^{2}

Number of sets in cache = Cache size/(Set size * Block size) = 64KB/(4 blocks * 4B) = $2^{16}/(2^{2} * 2^{2})$
= 2^{12}

Number of bits in Tag = Total bits - Index bits - Offset bits = 18-12-2 = 4

Settings for **8-way associative mapping**:



Cache Address Structure

Memory Cache Parameters

Memory Size 256KB

Cache Size 64KB

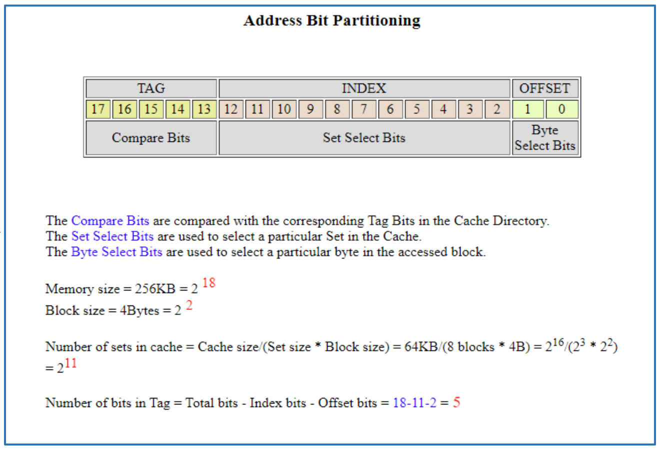
Block Size 4B

Cache Scheme

Direct Mapping ○

Set Associative ●

Set Size 8 blocks



Address Bit Partitioning

TAG					INDEX											OFFSET	
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Compare Bits					Set Select Bits											Byte Select Bits	

The Compare Bits are compared with the corresponding Tag Bits in the Cache Directory.
The Set Select Bits are used to select a particular Set in the Cache.
The Byte Select Bits are used to select a particular byte in the accessed block.

Memory size = 256KB = 2^{18}
Block size = 4Bytes = 2^{2}

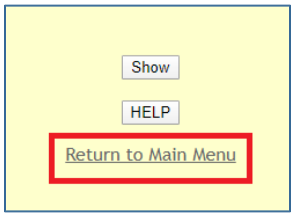
Number of sets in cache = Cache size/(Set size * Block size) = 64KB/(8 blocks * 4B) = $2^{16}/(2^{3} * 2^{2})$
= 2^{11}

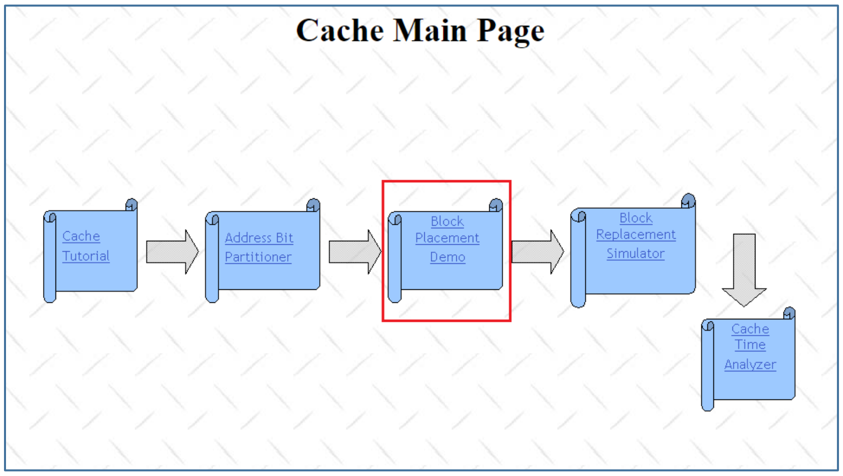
Number of bits in Tag = Total bits - Index bits - Offset bits = 18-11-2 = 5

Please remember that the INDEX or the set bits are used to map the main memory address to the cache address. So the location of the data in the cache locates to the same set bits (INDEX) in the memory.

Then, the tag bits are compared from the cache location. If the tag bits matches to what the processor is searching then it is a HIT else it is a MISS. The offset bits are mostly ignored for the cache access process.

2. Now, return to the main menu and go to the Block Placement Demo.





This part will help you visualize the content of cache from the given binary address.

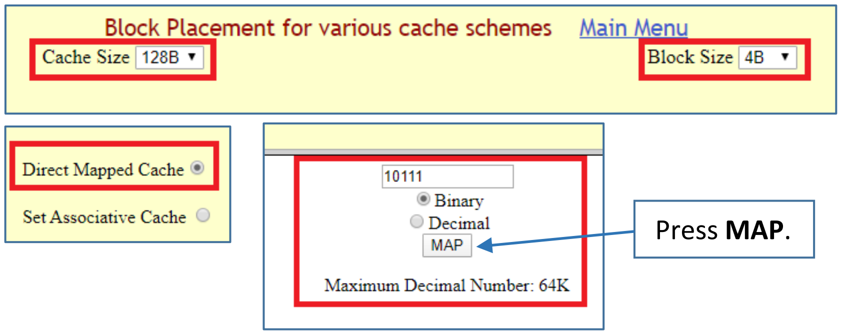
>>Input the following Settings:

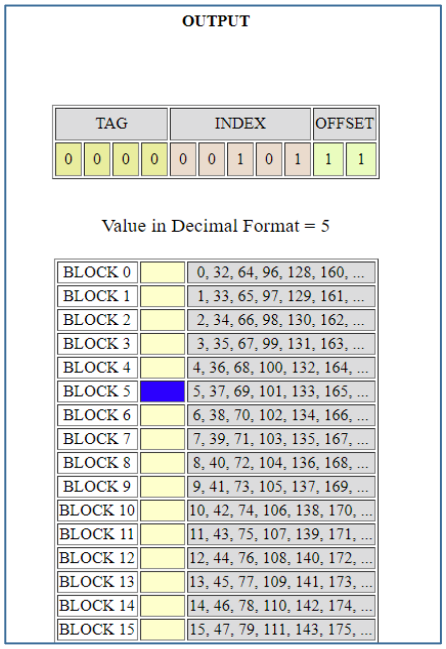
Direct Mapping,

128B Cache Size,

4B Block Size,

Binary Address 10111





OUTPUT

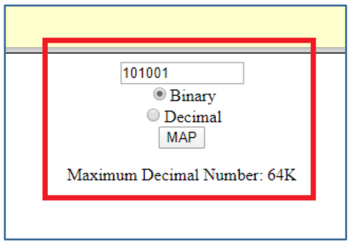
TAG				INDEX					OFFSET	
0	0	0	0	0	0	1	0	1	1	1

Value in Decimal Format = 5

BLOCK 0		0, 32, 64, 96, 128, 160, ...
BLOCK 1		1, 33, 65, 97, 129, 161, ...
BLOCK 2		2, 34, 66, 98, 130, 162, ...
BLOCK 3		3, 35, 67, 99, 131, 163, ...
BLOCK 4		4, 36, 68, 100, 132, 164, ...
BLOCK 5	🟦	5, 37, 69, 101, 133, 165, ...
BLOCK 6		6, 38, 70, 102, 134, 166, ...
BLOCK 7		7, 39, 71, 103, 135, 167, ...
BLOCK 8		8, 40, 72, 104, 136, 168, ...
BLOCK 9		9, 41, 73, 105, 137, 169, ...
BLOCK 10		10, 42, 74, 106, 138, 170, ...
BLOCK 11		11, 43, 75, 107, 139, 171, ...
BLOCK 12		12, 44, 76, 108, 140, 172, ...
BLOCK 13		13, 45, 77, 109, 141, 173, ...
BLOCK 14		14, 46, 78, 110, 142, 174, ...
BLOCK 15		15, 47, 79, 111, 143, 175, ...

>>Now input another address:

Binary Address 101001

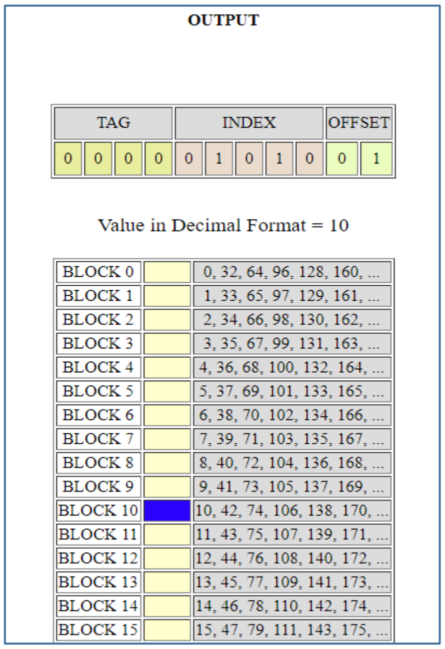


101001

[✓ Binary] Decimal

MAP

Maximum Decimal Number: 64K



OUTPUT

TAG				INDEX					OFFSET	
0	0	0	0	0	1	0	1	0	0	1

Value in Decimal Format = 10

BLOCK 0		0, 32, 64, 96, 128, 160, ...
BLOCK 1		1, 33, 65, 97, 129, 161, ...
BLOCK 2		2, 34, 66, 98, 130, 162, ...
BLOCK 3		3, 35, 67, 99, 131, 163, ...
BLOCK 4		4, 36, 68, 100, 132, 164, ...
BLOCK 5		5, 37, 69, 101, 133, 165, ...
BLOCK 6		6, 38, 70, 102, 134, 166, ...
BLOCK 7		7, 39, 71, 103, 135, 167, ...
BLOCK 8		8, 40, 72, 104, 136, 168, ...
BLOCK 9		9, 41, 73, 105, 137, 169, ...
BLOCK 10	■	10, 42, 74, 106, 138, 170, ...
BLOCK 11		11, 43, 75, 107, 139, 171, ...
BLOCK 12		12, 44, 76, 108, 140, 172, ...
BLOCK 13		13, 45, 77, 109, 141, 173, ...
BLOCK 14		14, 46, 78, 110, 142, 174, ...
BLOCK 15		15, 47, 79, 111, 143, 175, ...

>>Input the following Settings:

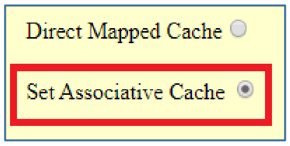
Set Associative Mapping,

4-Way,

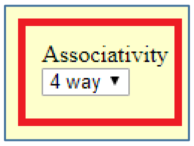
128B Cache Size,

4B Block Size,

Binary Address 10111

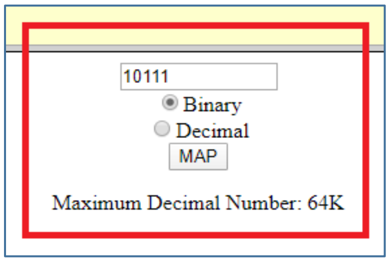


Direct Mapped Cache [✓ Set Associative Cache]



Associativity

4 way



10111

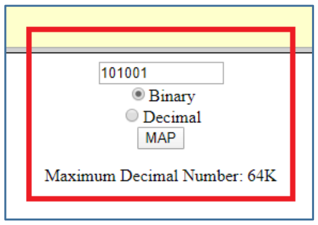
[✓ Binary] Decimal

MAP

Maximum Decimal Number: 64K

>>Now input another address:

Binary Address 101001

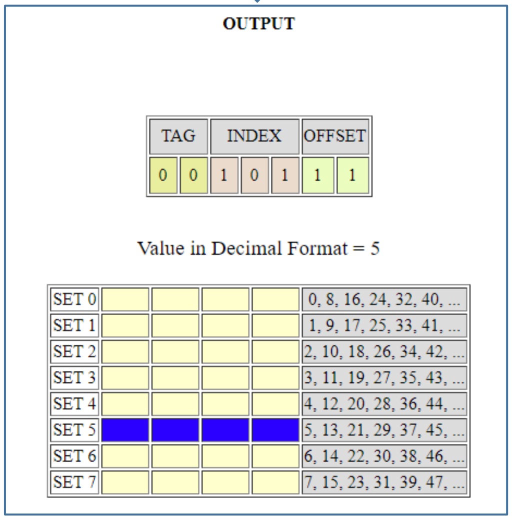


101001

[✓ Binary] Decimal

MAP

Maximum Decimal Number: 64K

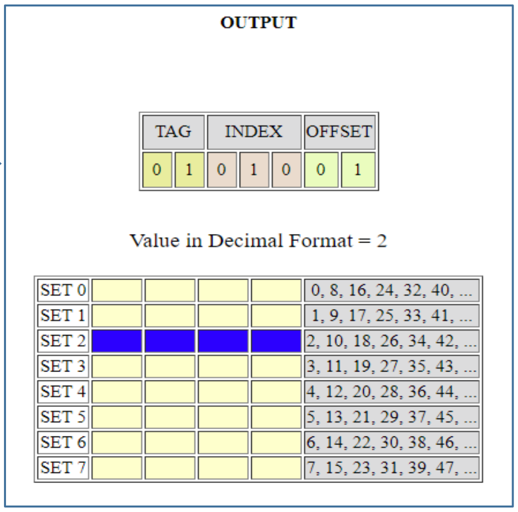


OUTPUT

TAG		INDEX			OFFSET	
0	0	1	0	1	1	1

Value in Decimal Format = 5

SET 0					0, 8, 16, 24, 32, 40, ...
SET 1					1, 9, 17, 25, 33, 41, ...
SET 2					2, 10, 18, 26, 34, 42, ...
SET 3					3, 11, 19, 27, 35, 43, ...
SET 4					4, 12, 20, 28, 36, 44, ...
SET 5	■	■	■	■	5, 13, 21, 29, 37, 45, ...
SET 6					6, 14, 22, 30, 38, 46, ...
SET 7					7, 15, 23, 31, 39, 47, ...



OUTPUT

TAG		INDEX			OFFSET	
0	1	0	1	0	0	1

Value in Decimal Format = 2

SET 0					0, 8, 16, 24, 32, 40, ...
SET 1					1, 9, 17, 25, 33, 41, ...
SET 2	■	■	■	■	2, 10, 18, 26, 34, 42, ...
SET 3					3, 11, 19, 27, 35, 43, ...
SET 4					4, 12, 20, 28, 36, 44, ...
SET 5					5, 13, 21, 29, 37, 45, ...
SET 6					6, 14, 22, 30, 38, 46, ...
SET 7					7, 15, 23, 31, 39, 47, ...

Notice the change in cache content mapping for direct mapping and 4-way associative mapping with 128B cache size x 4B block size.

>> After introducing this interactive online tool, you can explore it by yourself to review or study cache addressing concepts.

2. Cache Simulation in Google Colab

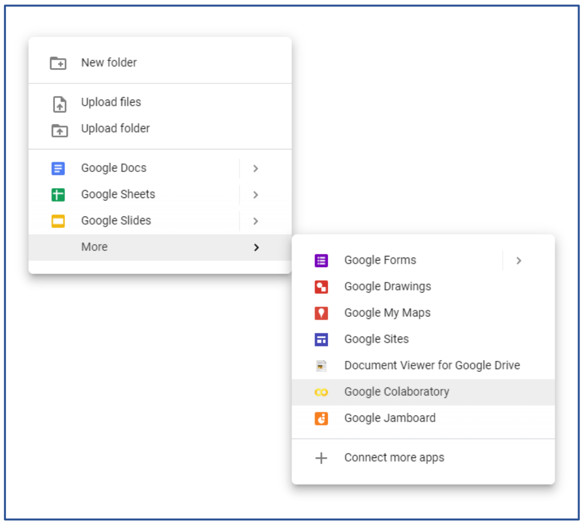
Objectives

The objective in this section is to introduce the Google Colab environment and simulate different cache configurations.

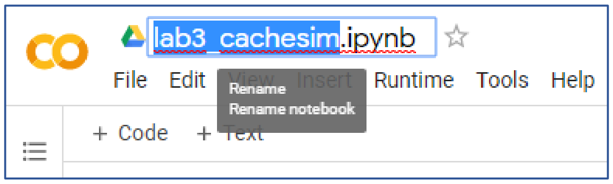
A. Starting Google Colab

>> If you don't have a Google account please make one because this is required to access Google Colab.

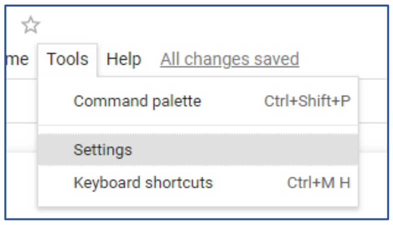
1. Open Google Drive (sign in using your Google account if required or sign in your Chrome browser using your Google account). Then go to your personal drive and right click on any black space then choose Google Colaboratory.

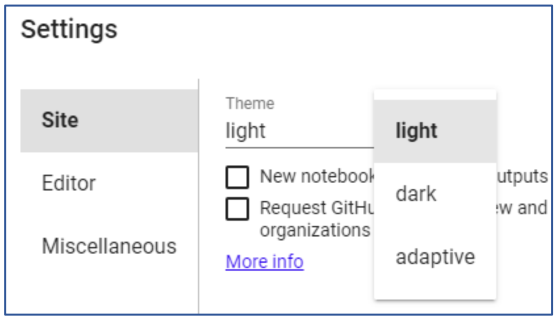


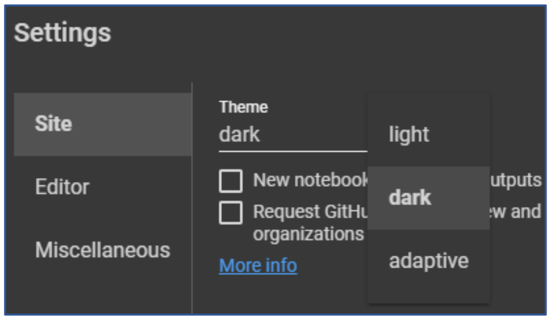
2. To rename your Colab notebook, click the .ipynb filename at the top left part. Then rename it with "lab3_cachesim" as indicated in the figure and press Enter.



3. To customize the settings of your notebook, clock Tools then Settings. You can customize to light or dark theme whichever suits your eyes and enable line numbers that is important for debugging.



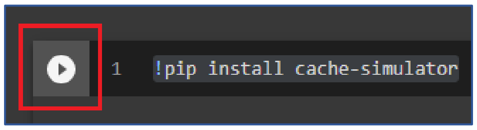




4. Now install the cache-simulator package by inputing the command in the Code Cell. Copy and paste the following command:

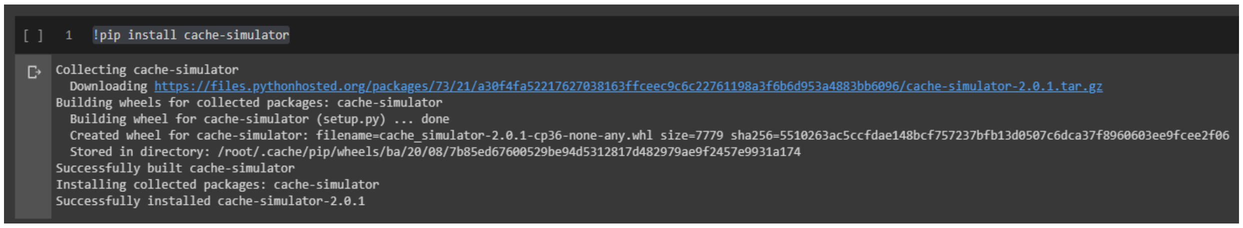
```
!pip install cache-simulator
```

Then to run the command press this button beside the cell.
Or **press Shift + Enter** to **run and insert new cell** below.



```
!pip install cache-simulator
```

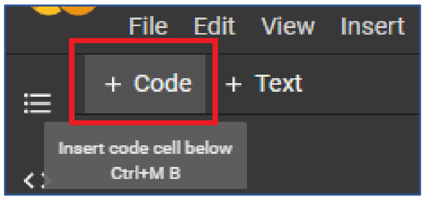
>>After running the command, the package should be successfully installed.



```
[ ]  !pip install cache-simulator

Collecting cache-simulator
  Downloading https://files.pythonhosted.org/packages/73/21/a30f4fa52217627038163ffceec9c6c22761198a3f6b6d953a4883bb6096/cache-simulator-2.0.1.tar.gz
Building wheels for collected packages: cache-simulator
  Building wheel for cache-simulator (setup.py) ... done
  Created wheel for cache-simulator: filename=cache_simulator-2.0.1-cp36-none-any.whl size=7779 sha256=5510263ac5ccfdae148bcf757237bfb13d0507c6dca37f8960603ee9fcee2f06
  Stored in directory: /root/.cache/pip/wheels/ba/20/08/7b85ed67600529be94d5312817d482979ae9f2457e9931a174
Successfully built cache-simulator
Installing collected packages: cache-simulator
Successfully installed cache-simulator-2.0.1
```

>>Some important functions

To manually insert a cell press the +Code at the upper left of the page.

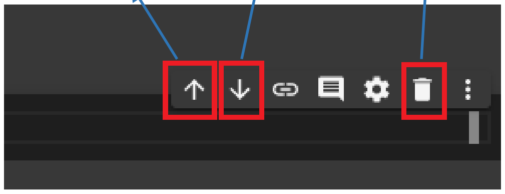


To show cell options, click the cell and the options will show at the right side of the cell.

Shift the cell DOWN

Delete cell

Shift the cell UP

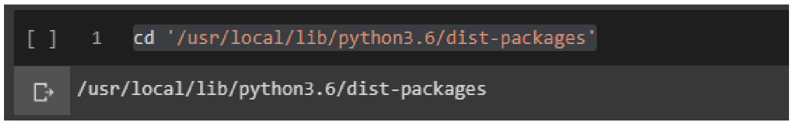


B. Cache Simulation

1. For this package we have to go to its directory to call the function. Copy and paste the command and run the cell.

```
cd '/usr/local/lib/python3.6/dist-packages'
```

>>After running the command, the directory should point to the package directory.



```
[ ]  cd '/usr/local/lib/python3.6/dist-packages'

/usr/local/lib/python3.6/dist-packages
```

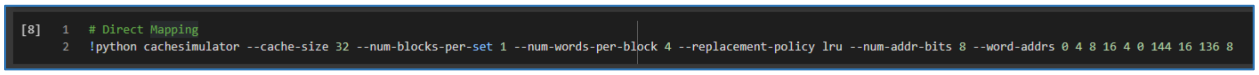
2. Now let us test the cache-simulator package with the following configuration.

Cache Size: **32W**	Blocks per set: **1**	Block Size: **4W**	Replacement Policy: **LRU**	Memory Word Address Size: **2^8**
Word Addresses: **0 4 8 16 4 0 144 16 136 8** (each address is separated by a space)				

Copy and paste the following command to simulate the given configuration and run the cell. Notice that this is direct mapping with blocks per set = 1. **(DO NOT COPY THE GREEN TEXT)**

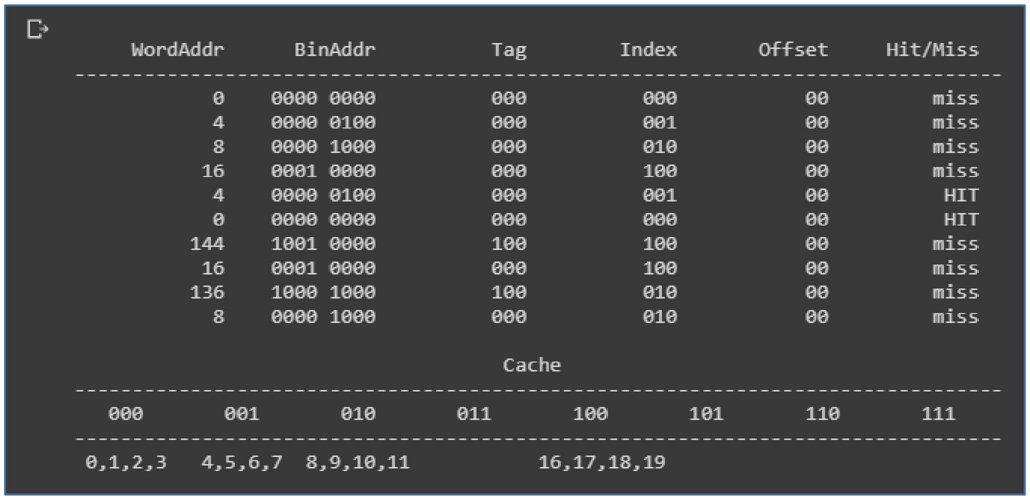
```
# Direct Mapping
!python cachesimulator --cache-size 32 --num-blocks-per-set 1 --num-words-
per-block 4 --replacement-policy lru --num-addr-bits 8 --word-
addrs 0 4 8 16 4 0 144 16 136 8
```

Code in Colab:



```
[8]  # Direct Mapping
     !python cachesimulator --cache-size 32 --num-blocks-per-set 1 --num-words-per-block 4 --replacement-policy lru --num-addr-bits 8 --word-addrs 0 4 8 16 4 0 144 16 136 8
```

Output:



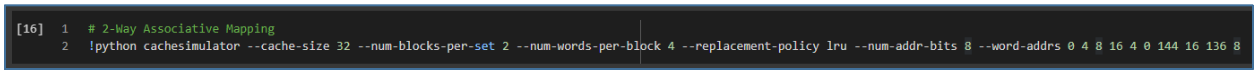
```
     WordAddr      BinAddr          Tag         Index        Offset      Hit/Miss
----------------------------------------------------------------------------------
            0    0000 0000          000           000            00          miss
            4    0000 0100          000           001            00          miss
            8    0000 1000          000           010            00          miss
           16    0001 0000          000           100            00          miss
            4    0000 0100          000           001            00           HIT
            0    0000 0000          000           000            00           HIT
          144    1001 0000          100           100            00          miss
           16    0001 0000          000           100            00          miss
          136    1000 1000          100           010            00          miss
            8    0000 1000          000           010            00          miss

                                       Cache
----------------------------------------------------------------------------------
    000        001        010        011        100        101        110        111
----------------------------------------------------------------------------------
  0,1,2,3    4,5,6,7   8,9,10,11              16,17,18,19
```

Notice how the second 8 and 16 addresses MISS, this is because addresses 144 and 136 occupies the same cache block as address 8 and 16. **Hit Ratio = 2 / 10 = 0.2**. Also, the output will show the final state of the cache on the lower portion.

3. Now, simulate using the same configuration and test addresses as B-2 but **setting block per set = 2**. Notice that this is 2-way associative mapping. Copy and paste the following command and run the cell. **(DO NOT COPY THE GREEN TEXT)**

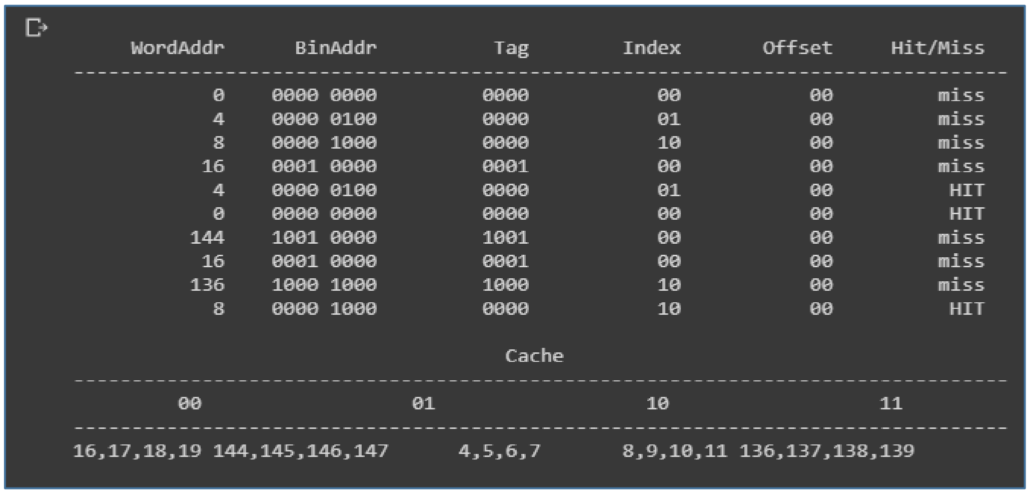
```
# 2-Way Associative Mapping
!python cachesimulator --cache-size 32 --num-blocks-per-set 1 --num-words-
per-block 4 --replacement-policy lru --num-addr-bits 8 --word-
addrs 0 4 8 16 4 0 144 16 136 8
```

Code in Colab:



```
[16]  # 2-Way Associative Mapping
      !python cachesimulator --cache-size 32 --num-blocks-per-set 2 --num-words-per-block 4 --replacement-policy lru --num-addr-bits 8 --word-addrs 0 4 8 16 4 0 144 16 136 8
```

Output:



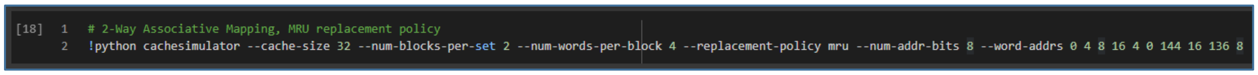
```
   WordAddr       BinAddr            Tag         Index        Offset      Hit/Miss
------------------------------------------------------------------------------------
          0     0000 0000           0000            00            00          miss
          4     0000 0100           0000            01            00          miss
          8     0000 1000           0000            10            00          miss
         16     0001 0000           0001            00            00          miss
          4     0000 0100           0000            01            00           HIT
          0     0000 0000           0000            00            00           HIT
        144     1001 0000           1001            00            00          miss
         16     0001 0000           0001            00            00          miss
        136     1000 1000           1000            10            00          miss
          8     0000 1000           0000            10            00           HIT

                                       Cache
------------------------------------------------------------------------------------
          00                   01                   10                   11
------------------------------------------------------------------------------------
16,17,18,19 144,145,146,147       4,5,6,7         8,9,10,11 136,137,138,139
```

Notice that the second 8 address is a HIT. This is because 2 addresses can now fit using 2-way associative mapping with the given index or set bits. But, the second 16 address is a miss. This is because the replacement policy used is LRU; address 16 is the least recently used after inputting the second address 0. Notice that addresses 0, 16, and 144 have the same index. That means they map to the same set in the cache. This means when address 144 comes, address 16 gets evicted. **Hit Ratio = 3 / 10 = 0.3**.

4. Now, simulate using the same configuration and test addresses as B-3 but setting the **replacement policy as MRU**. Copy and paste the following command and run the cell. **(DO NOT COPY THE GREEN TEXT)**

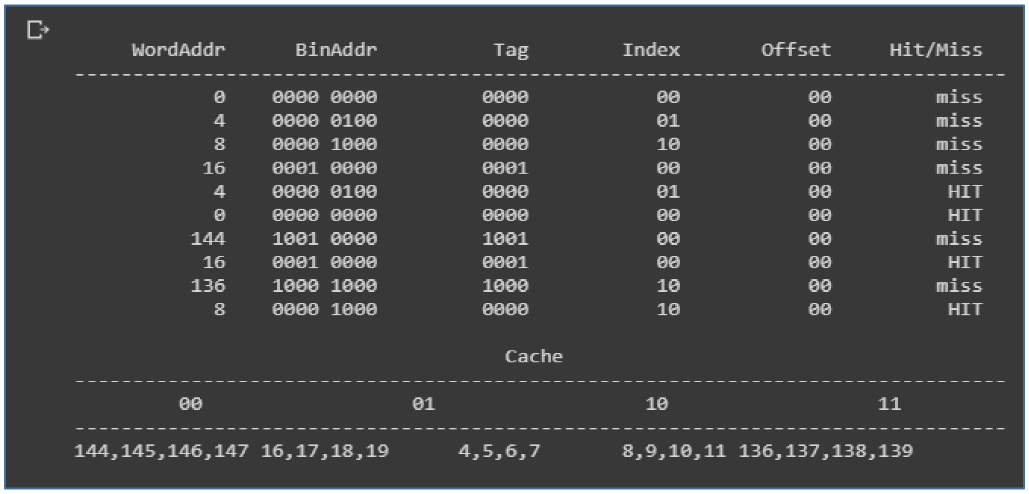
```
# 2-Way Associative Mapping, MRU replacement policy
!python cachesimulator --cache-size 32 --num-blocks-per-set 2 --num-words-
per-block 4 --replacement-policy mru --num-addr-bits 8 --word-
addrs 0 4 8 16 4 0 144 16 136 8
```

Code in Colab:



```
[18]  # 2-Way Associative Mapping, MRU replacement policy
      !python cachesimulator --cache-size 32 --num-blocks-per-set 2 --num-words-per-block 4 --replacement-policy mru --num-addr-bits 8 --word-addrs 0 4 8 16 4 0 144 16 136 8
```

Output:



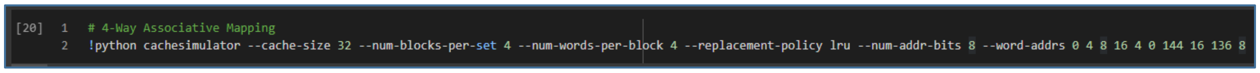
```
     WordAddr       BinAddr          Tag         Index        Offset     Hit/Miss
---------------------------------------------------------------------------------
            0     0000 0000         0000            00            00         miss
            4     0000 0100         0000            01            00         miss
            8     0000 1000         0000            10            00         miss
           16     0001 0000         0001            00            00         miss
            4     0000 0100         0000            01            00          HIT
            0     0000 0000         0000            00            00          HIT
          144     1001 0000         1001            00            00         miss
           16     0001 0000         0001            00            00          HIT
          136     1000 1000         1000            10            00         miss
            8     0000 1000         0000            10            00          HIT

                                       Cache
---------------------------------------------------------------------------------
         00                   01                   10                   11
---------------------------------------------------------------------------------
144,145,146,147 16,17,18,19       4,5,6,7        8,9,10,11 136,137,138,139
```

Using the MRU replacement policy, the second 16 address is now a HIT. Notice that addresses 0, 16, and 144 have the same index 00. With MRU, since the second 0 address hits, when address 144 comes, address 0 will be evicted instead of address 16. **Hit Ratio = 4 / 10 = 0.4**.

5. Now, simulate using the same configuration and test addresses as B-2, but **setting block per set = 4**. Notice that this is 4-way associative mapping. Copy and paste the following command and run the cell. **(DO NOT COPY THE GREEN TEXT)**

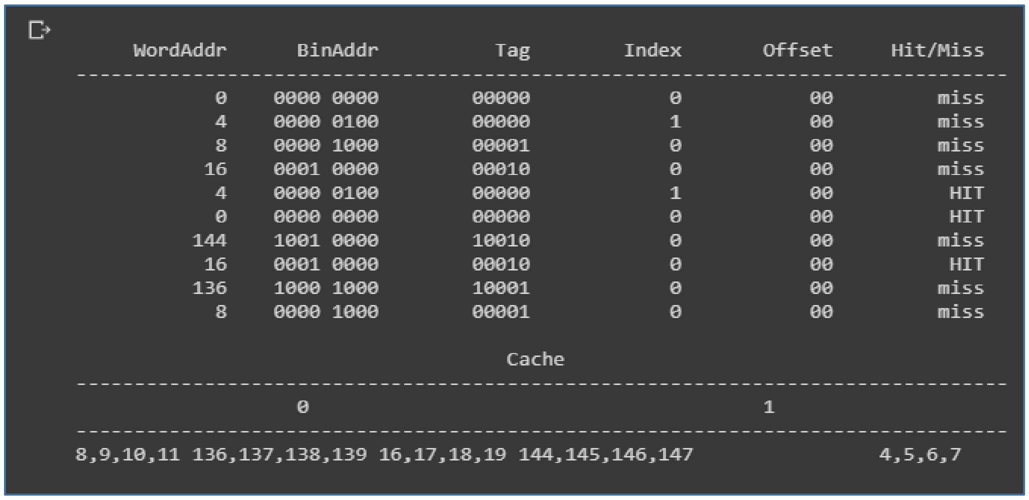
```
# 4-Way Associative Mapping
!python cachesimulator --cache-size 32 --num-blocks-per-set 4 --num-words-
per-block 4 --replacement-policy lru --num-addr-bits 8 --word-
addrs 0 4 8 16 4 0 144 16 136 8
```

Code in Colab:



```
[20]  # 4-Way Associative Mapping
      !python cachesimulator --cache-size 32 --num-blocks-per-set 4 --num-words-per-block 4 --replacement-policy lru --num-addr-bits 8 --word-addrs 0 4 8 16 4 0 144 16 136 8
```

Output:



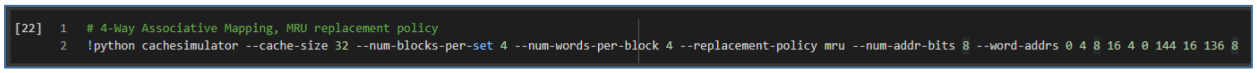
```
     WordAddr       BinAddr          Tag         Index        Offset     Hit/Miss
---------------------------------------------------------------------------------
            0     0000 0000        00000             0            00         miss
            4     0000 0100        00000             1            00         miss
            8     0000 1000        00001             0            00         miss
           16     0001 0000        00010             0            00         miss
            4     0000 0100        00000             1            00          HIT
            0     0000 0000        00000             0            00          HIT
          144     1001 0000        10010             0            00         miss
           16     0001 0000        00010             0            00          HIT
          136     1000 1000        10001             0            00         miss
            8     0000 1000        00001             0            00         miss

                                       Cache
---------------------------------------------------------------------------------
                    0                                       1
---------------------------------------------------------------------------------
8,9,10,11 136,137,138,139 16,17,18,19 144,145,146,147             4,5,6,7
```

Notice that the second 16 address is a HIT while the second 8 address is a MISS. With 4-way associative mapping, addresses 0, 8, 16, 144, and 136 maps to the same set. Before replacement, the set needs to be full, so addresses 0, 8, 16, and 144 will occupy set index 0. Since the second addresses 0, and 16 HITS and address 144 just recently occupies the 4th block in index 0, address 8 is the recently used block. So when address 136 will come, address 8 will be evicted. Then, the second address 8 will be a MISS but will replace address 0 (LRU block in this state) and the final state of the cache is displayed. **Hit Ratio = 3 / 10 = 0.3**.

6. Now, simulate using the same configuration and test addresses as B-5 but setting the **replacement policy as MRU**. Copy and paste the following command and run the cell. **(DO NOT COPY THE GREEN TEXT)**

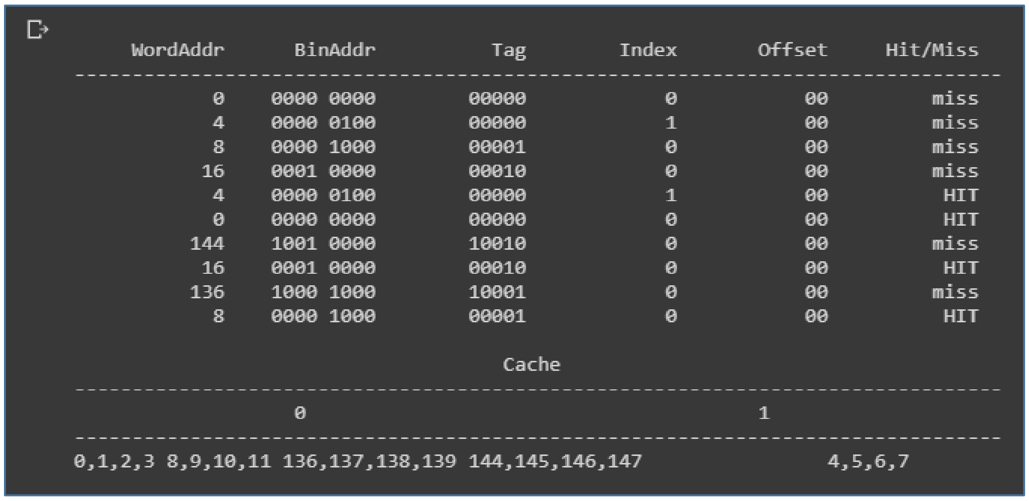
```
# 4-Way Associative Mapping, MRU replacement policy
!python cachesimulator --cache-size 32 --num-blocks-per-set 4 --num-words-per-block 4 --replacement-policy mru --num-addr-bits 8 --word-addrs 0 4 8 16 4 0 144 16 136 8
```

Code in Colab:



```
[22]
# 4-Way Associative Mapping, MRU replacement policy
!python cachesimulator --cache-size 32 --num-blocks-per-set 4 --num-words-per-block 4 --replacement-policy mru --num-addr-bits 8 --word-addrs 0 4 8 16 4 0 144 16 136 8
```

Output:



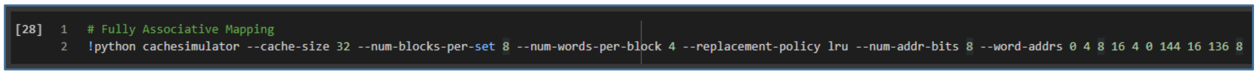
```
     WordAddr       BinAddr            Tag         Index        Offset     Hit/Miss
-------------------------------------------------------------------------------------
            0     0000 0000          00000             0            00         miss
            4     0000 0100          00000             1            00         miss
            8     0000 1000          00001             0            00         miss
           16     0001 0000          00010             0            00         miss
            4     0000 0100          00000             1            00          HIT
            0     0000 0000          00000             0            00          HIT
          144     1001 0000          10010             0            00         miss
           16     0001 0000          00010             0            00          HIT
          136     1000 1000          10001             0            00         miss
            8     0000 1000          00001             0            00          HIT

                                      Cache
-------------------------------------------------------------------------------------
                  0                                        1
-------------------------------------------------------------------------------------
0,1,2,3 8,9,10,11 136,137,138,139 144,145,146,147              4,5,6,7
```

Using the MRU replacement policy, the second 8 address is now a HIT. Since addresses 0, 8, 16, 144, and 136 maps to the same set index 0. After the second address 0 hits, address 144 occupies the 4th block of the set. Then, address 16 hits. From this state, address 136 needs to replace one of the blocks in set index 0. Because address 16 just recently hit (used), following MRU policy, it will be evicted and address 136 will replace it keeping address 8 in the set. **Hit Ratio = 4 / 10 = 0.4**.

7. Now, simulate using the same configuration and test addresses as B-2, but **setting block per set = 8**. Notice that this is fully associative mapping since Set number = Cache Size/block size = N = 8. Copy and paste the following command and run the cell. **(DO NOT COPY THE GREEN TEXT)**

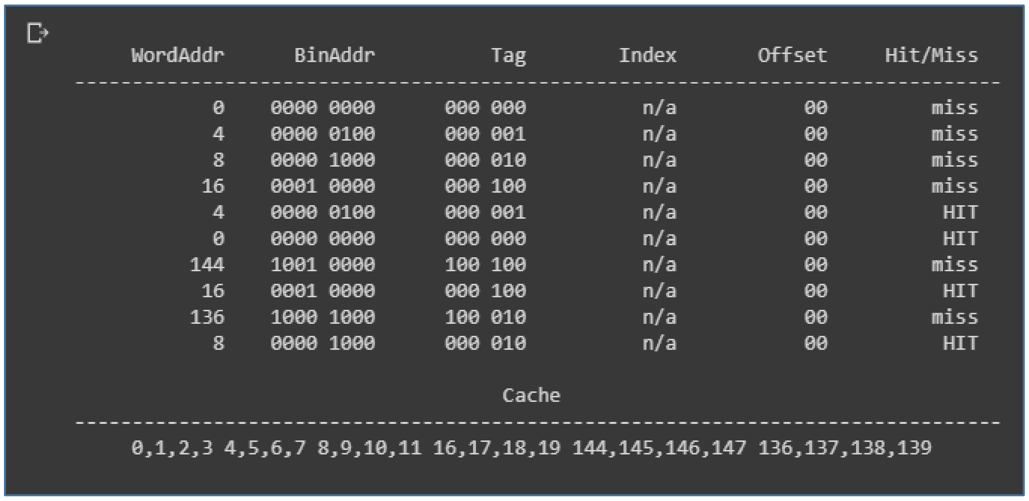
```
# Fully Associative Mapping
!python cachesimulator --cache-size 32 --num-blocks-per-set 8 --num-words-per-block 4 --replacement-policy lru --num-addr-bits 8 --word-addrs 0 4 8 16 4 0 144 16 136 8
```

Code in Colab:



```
[28]
# Fully Associative Mapping
!python cachesimulator --cache-size 32 --num-blocks-per-set 8 --num-words-per-block 4 --replacement-policy lru --num-addr-bits 8 --word-addrs 0 4 8 16 4 0 144 16 136 8
```

Output:

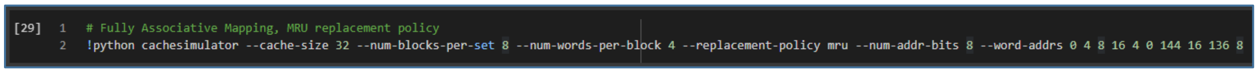


```
     WordAddr      BinAddr          Tag        Index       Offset     Hit/Miss
----------------------------------------------------------------------------
            0    0000 0000      000 000          n/a           00         miss
            4    0000 0100      000 001          n/a           00         miss
            8    0000 1000      000 010          n/a           00         miss
           16    0001 0000      000 100          n/a           00         miss
            4    0000 0100      000 001          n/a           00          HIT
            0    0000 0000      000 000          n/a           00          HIT
          144    1001 0000      100 100          n/a           00         miss
           16    0001 0000      000 100          n/a           00          HIT
          136    1000 1000      100 010          n/a           00         miss
            8    0000 1000      000 010          n/a           00          HIT

                                    Cache
----------------------------------------------------------------------------
     0,1,2,3 4,5,6,7 8,9,10,11 16,17,18,19 144,145,146,147 136,137,138,139
```

Now, let's change the **replacement policy to MRU** and simulate again the fully associative mapped cache. Copy and paste the following command and run the cell. **(DO NOT COPY THE GREEN TEXT)**

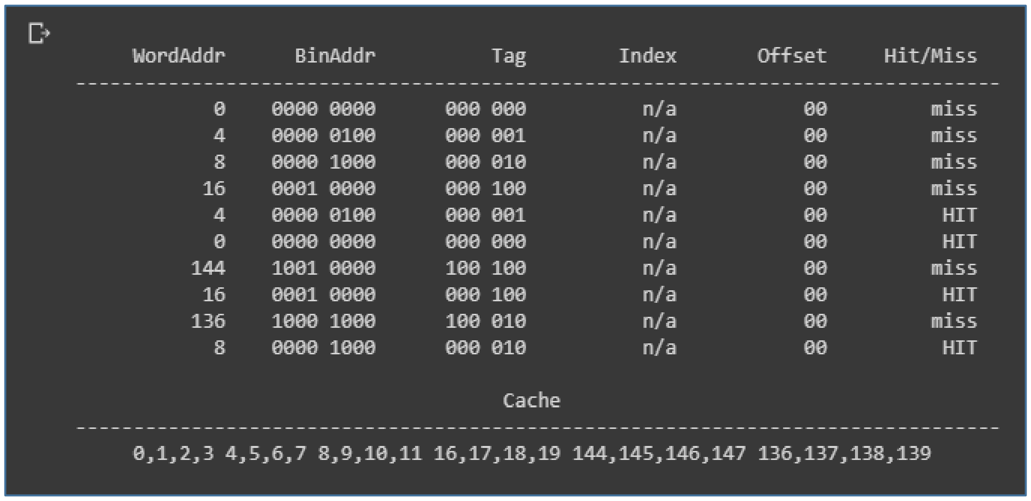
```
# Fully Associative Mapping, MRU replacement policy
!python cachesimulator --cache-size 32 --num-blocks-per-set 8 --num-words-per-block 4 --replacement-policy mru --num-addr-bits 8 --word-addrs 0 4 8 16 4 0 144 16 136 8
```

Code in Colab:



```
[29]  # Fully Associative Mapping, MRU replacement policy
      !python cachesimulator --cache-size 32 --num-blocks-per-set 8 --num-words-per-block 4 --replacement-policy mru --num-addr-bits 8 --word-addrs 0 4 8 16 4 0 144 16 136 8
```

Output:



```
     WordAddr      BinAddr          Tag        Index       Offset     Hit/Miss
----------------------------------------------------------------------------
            0    0000 0000      000 000          n/a           00         miss
            4    0000 0100      000 001          n/a           00         miss
            8    0000 1000      000 010          n/a           00         miss
           16    0001 0000      000 100          n/a           00         miss
            4    0000 0100      000 001          n/a           00          HIT
            0    0000 0000      000 000          n/a           00          HIT
          144    1001 0000      100 100          n/a           00         miss
           16    0001 0000      000 100          n/a           00          HIT
          136    1000 1000      100 010          n/a           00         miss
            8    0000 1000      000 010          n/a           00          HIT

                                    Cache
----------------------------------------------------------------------------
     0,1,2,3 4,5,6,7 8,9,10,11 16,17,18,19 144,145,146,147 136,137,138,139
```

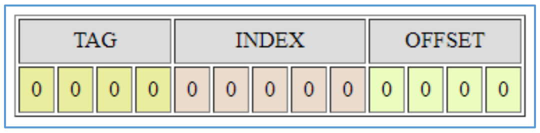
In fully, associative mapping, all blocks in the cache belongs to the same set. Since there are only 6 distinct addresses mapped to 8 cache blocks. All of the 6 distinct addresses will be mapped distinctively in the cache. So for LRU and MRU replacement policies, the same final cache state and hit ratio will be observed for the given test addresses. **Hit Ratio = 4 / 10 = 0.4**.

8. Now let's simulate the following with a different configuration.

Cache Size: **128W**	Blocks per set: **1**	Block Size: **4W**	Replacement Policy: **LRU**	Memory Word Address Size: **2^11**
Word Addresses: **52 56 36 40 44 48 72 76 144 148 152 156 160 164 168 172 176 180 184 220 224 228 184 188 144 148 152 156 160 164 168 172 52 56 36 40 44 48 72 76** (each address is separated by a space)				

*The test addresses are taken from the address sequence of executing the bubble_sort.asm

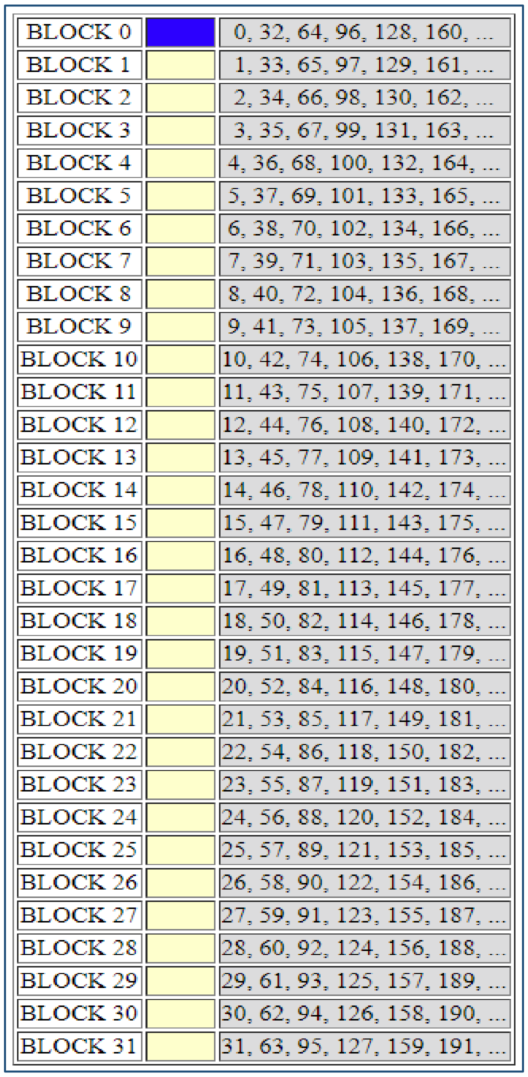
From the Placement Visualization, this is how the address partitioning looks like:



TAG				INDEX					OFFSET			
0	0	0	0	0	0	0	0	0	0	0	0	0

In the online page simulator:
The cache is byte addressable: 1W = 4B
(4B per W) x 4W = 16 offset address,
Then $\log_2 16$ = 4 bits offset

This is how the cache abstraction looks like:



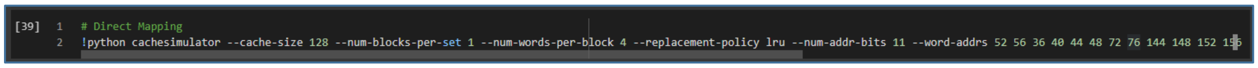
BLOCK 0	🟦	0, 32, 64, 96, 128, 160, ...
BLOCK 1		1, 33, 65, 97, 129, 161, ...
BLOCK 2		2, 34, 66, 98, 130, 162, ...
BLOCK 3		3, 35, 67, 99, 131, 163, ...
BLOCK 4		4, 36, 68, 100, 132, 164, ...
BLOCK 5		5, 37, 69, 101, 133, 165, ...
BLOCK 6		6, 38, 70, 102, 134, 166, ...
BLOCK 7		7, 39, 71, 103, 135, 167, ...
BLOCK 8		8, 40, 72, 104, 136, 168, ...
BLOCK 9		9, 41, 73, 105, 137, 169, ...
BLOCK 10		10, 42, 74, 106, 138, 170, ...
BLOCK 11		11, 43, 75, 107, 139, 171, ...
BLOCK 12		12, 44, 76, 108, 140, 172, ...
BLOCK 13		13, 45, 77, 109, 141, 173, ...
BLOCK 14		14, 46, 78, 110, 142, 174, ...
BLOCK 15		15, 47, 79, 111, 143, 175, ...
BLOCK 16		16, 48, 80, 112, 144, 176, ...
BLOCK 17		17, 49, 81, 113, 145, 177, ...
BLOCK 18		18, 50, 82, 114, 146, 178, ...
BLOCK 19		19, 51, 83, 115, 147, 179, ...
BLOCK 20		20, 52, 84, 116, 148, 180, ...
BLOCK 21		21, 53, 85, 117, 149, 181, ...
BLOCK 22		22, 54, 86, 118, 150, 182, ...
BLOCK 23		23, 55, 87, 119, 151, 183, ...
BLOCK 24		24, 56, 88, 120, 152, 184, ...
BLOCK 25		25, 57, 89, 121, 153, 185, ...
BLOCK 26		26, 58, 90, 122, 154, 186, ...
BLOCK 27		27, 59, 91, 123, 155, 187, ...
BLOCK 28		28, 60, 92, 124, 156, 188, ...
BLOCK 29		29, 61, 93, 125, 157, 189, ...
BLOCK 30		30, 62, 94, 126, 158, 190, ...
BLOCK 31		31, 63, 95, 127, 159, 191, ...

The cache has 32 blocks:
(128W / 4W = 32)

Copy and paste the following command and run the cell. **(DO NOT COPY THE GREEN TEXT)**

```
# Direct Mapping
!python cachesimulator --cache-size 128 --num-blocks-per-set 1 --num-
words-per-block 4 --replacement-policy lru --num-addr-bits 11 --word-
addrs 52 56 36 40 44 48 72 76 144 148 152 156 160 164 168 172 176 180 184
220 224 228 184 188 144 148 152 156 160 164 168 172 52 56 36 40 44 48 72 7
6
```

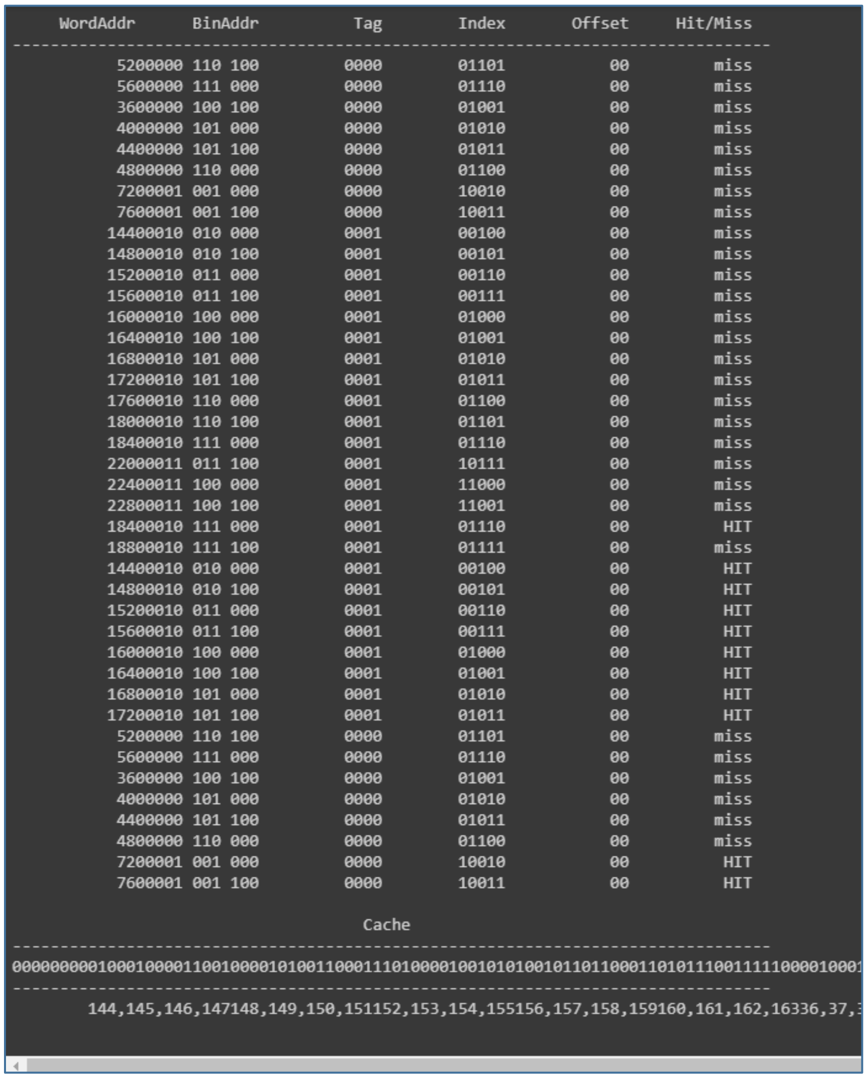
Code in Colab:



```
[39]  # Direct Mapping
      !python cachesimulator --cache-size 128 --num-blocks-per-set 1 --num-words-per-block 4 --replacement-policy lru --num-addr-bits 11 --word-addrs 52 56 36 40 44 48 72 76 144 148 152 156
```

(The command is very long so the side scroll bar will be used to navigate the code in the cell)

Output:



```
   WordAddr       BinAddr            Tag          Index         Offset      Hit/Miss
-------------------------------------------------------------------------------------
          5200000 110 100          0000          01101            00          miss
          5600000 111 000          0000          01110            00          miss
          3600000 100 100          0000          01001            00          miss
          4000000 101 000          0000          01010            00          miss
          4400000 101 100          0000          01011            00          miss
          4800000 110 000          0000          01100            00          miss
          7200001 001 000          0000          10010            00          miss
          7600001 001 100          0000          10011            00          miss
         14400010 010 000          0001          00100            00          miss
         14800010 010 100          0001          00101            00          miss
         15200010 011 000          0001          00110            00          miss
         15600010 011 100          0001          00111            00          miss
         16000010 100 000          0001          01000            00          miss
         16400010 100 100          0001          01001            00          miss
         16800010 101 000          0001          01010            00          miss
         17200010 101 100          0001          01011            00          miss
         17600010 110 000          0001          01100            00          miss
         18000010 110 100          0001          01101            00          miss
         18400010 111 000          0001          01110            00          miss
         22000011 011 100          0001          10111            00          miss
         22400011 100 000          0001          11000            00          miss
         22800011 100 100          0001          11001            00          miss
         18400010 111 000          0001          01110            00           HIT
         18800010 111 100          0001          01111            00          miss
         14400010 010 000          0001          00100            00           HIT
         14800010 010 100          0001          00101            00           HIT
         15200010 011 000          0001          00110            00           HIT
         15600010 011 100          0001          00111            00           HIT
         16000010 100 000          0001          01000            00           HIT
         16400010 100 100          0001          01001            00           HIT
         16800010 101 000          0001          01010            00           HIT
         17200010 101 100          0001          01011            00           HIT
          5200000 110 100          0000          01101            00          miss
          5600000 111 000          0000          01110            00          miss
          3600000 100 100          0000          01001            00          miss
          4000000 101 000          0000          01010            00          miss
          4400000 101 100          0000          01011            00          miss
          4800000 110 000          0000          01100            00          miss
          7200001 001 000          0000          10010            00           HIT
          7600001 001 100          0000          10011            00           HIT

                                      Cache
-------------------------------------------------------------------------------------
00000000010001000011001000010100110001110100001001010100101101100011010111001111100001000
-------------------------------------------------------------------------------------
        144,145,146,147148,149,150,151152,153,154,155156,157,158,159160,161,162,16336,37,
```

Hit Ratio = 11 / 40 = 0.275

(Because the memory address size is long, the output address and final state might be very hard to read because it is all cramped up in a confined spacing.)

HOMEWORK: SIMULATE N-ASSOCIATIVE AND FULLY ASSOCIATIVE

Use the following configuration:

Cache Size: **128W**	Blocks per set: **X**	Block Size: **4W**	Replacement Policy: **LRU**	Memory Word Address Size: **2^11**
Word Addresses: **52 56 36 40 44 48 72 76 144 148 152 156 160 164 168 172 176 180 184 220 224 228 184 188 144 148 152 156 160 164 168 172 52 56 36 40 44 48 72 76** (each address is separated by a space)				

Simulate the following:

1. 2-Way Associative

2. 4-Way Associative

3. Fully Associative

Then, present the following for each 1, 2, and 3 items:

A. Draw the visualization of address partitioning and cache abstraction. Refer to section B-8. Use this online page: http://www.ecs.umass.edu/ece/koren/architecture/Cache/frame0.htm to help you (*no fully associative mapping in the online page*). Please remember to draw the address partitioning based on the given configuration of memory address (11 bits).

B. Show the **simulation command** and also the **output**.

C. Compute the Hit Ratio and **compare** it to the **direct mapping hit ratio** shown in section B-8.

D. Explain the **result** of each simulation.