TUHH
Hamburg University of Technology

HAMBURG UNIVERSITY OF TECHNOLOGY

PROBLEM-BASED LEARNING

Advanced System-on-Chip Design

author
@tuhh.de

Report

Tutor
Dipl.-Ing. Wolfgang BRANDT

March 13, 2017

Contents

List of Figures

List of Tables

Listings

1 Introduction

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

2 Task 1 - Introduction Set Architecture of the MIPS-Processor

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

3 Task 2 - VHDL Introduction

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

4 Task 3 - MIPS Extension

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5 Task 4 - Caches

In the following section we describe an implementation of an instruction cache and a data cache. We develop a *Direct Mapped Cache* using as the instruction cache and a *2-way set associative cache* using as the data cache.
In the following subsection 5.1 we give a short introduction to memories. After that, we simulate the efficiency of a cache with MARS in subsection 5.2.

5.1 Introduction to Memories

TODO Why are there so many different storage types?

A cache is a faster but smaller storage system which is placed nearby the CPU. In the Harvard architecture there is one cache for instructions and one cache for data. There are diverse modes to organize the caches. Primary, the caches are organized by answering the following four questions:

1. *Block Placement* - Where can a block be placed in the cache?
2. *Block Identification* - How is a block found in the cache?
3. *Block Replacement* - Which block is replaced in case of a cache miss?
4. *Write Strategies* - What happens during a write operation?

So, there are different cache organizations. But what are the advantages and disadvantages of the different cache organization forms?
Regarding the *Block Placement* there are three essential block placement strategies: *Direct Mapped*, *Set Associative* and *Fully Associative*. The advandtage of the increasing of the grade of associativity is that the miss rate is reduced and the hit rate is increased. The disadvantage of the associativity is the complex implementation and slower access time.
In view of the *Block Identification*increased With respect to *Block Replacement* either the *LRU* strategy or the *Random* strategy is used.
Regarding the *Write Strategies* we distinguish between the *Write-through* and *Write-back* strategies. The advantages of the strategy Write-back is that single words can be written with the speed of the CPU and not of the main memory. Also, only one single write operation to the main memory is needed for multiple write operations. The advantages of the strategy Write-through are that cache misses can be simpler and cheaper handled. Eventually, the cache blocks do not needed to write back to the main memory in case of a cache miss. Furthermore, the strategy Write-through is easily to implemented.

5.2 Cache Simulation - Results

In the following step we simulate the efficiency of a cache with MARS. On this, we compare the cache performance for different block sizes of a direct mapping cache, a 2-way associative cache and a 4-way associative cache. The two assembler programs *row-major.asm* and *column-major.asm* has been used for the cache simulation. For the simulation we vary the block size and

placemet policy, but we fix the number of cache blocks to 8. Table 1 contains the results regarding the file *column-major.asm* and table 2 illustrates the results of *row-major.asm*. The efficiency of a cache is evaluated by counting the number of cache hits and cache misses during executing the assembler program. Besides, the cache hit rate is determined by the cache hit count and the memory access count. In both assembler programs a 16x16 matrix is fully traversed. Therefore, we get a memory access count with value 256 for each program execution.
The first assembler program *column-major.asm* traverses the 16x16 matrix column by column. At first we traverse the lead column, then the second column and so on. When we traverse the first half of a column, each correspondent block is loaded into the cache. But when we handle the second half of a column, the all data in the cache are replaced because all eight cache blocks are already occupied. In the next colum we also traverse at first the first half and then the second half of the column. When traversing the first half, we must also replace all data in the cache, because all cache blocks are already occupied and have different tag values. Finally, we expect that no cache hit occurs. In fact during each access to an array element causes a cache miss. As you can see in table 1, for all combinations of the placement policy and the cache block size we achieve a cache hit rate of zero.
Contrary to the column major program, we traverse the 16x16 matrix row by row. When we access an array element, the correspondent block is placed into the cache. Directly after accessing this element, we also access the nearby array elements of this block. Thus, we only expect one cache miss for the access of the first block element and cache hits for accessing the remainding elemenets of a block. Depending on the cache block size (i.e. the number of words in a cache block), we achieve expect a diverse number of cache hit count and miss hit count. Moreover, table 2 shows that the results are equal relating to the placement policy.
The above assembler programs contrast traversing the matrix column by column with traversing row by row. Two principles of the cache memory are the *Temporal Locality* and the *Spatial Locality*. The above assembler programs illustrate the spatial locality. Thus, memory accesses whose addresses are adjacent will often be accessed in the near future. Therefore, the matrix is stored in memory row by row. The spatial locality requires to access contiguous data in memory element wise. Hence, it is efficient to traverse the given matrix row by row.

Table 1: Cache Simulation of Column Major

Placement (Policy)	**Block Size (Words)**	**Cache Hit Count**	**Cache Miss Count**	**Cache Hit Rate**
Direct Mapping	2	0	256	0
Direct Mapping	4	0	256	0
Direct Mapping	8	0	256	0
Direct Mapping	16	0	256	0
2-Way Set Associative	2	0	256	0
2-Way Set Associative	4	0	256	0
2-Way Set Associative	8	0	256	0
2-Way Set Associative	16	0	256	0
4-Way Set Associative	2	0	256	0
4-Way Set Associative	4	0	256	0
4-Way Set Associative	8	0	256	0
4-Way Set Associative	16	0	256	0

Table 2: Cache Simulation of Row Major

Placement (Policy)	**Block Size (Words)**	**Cache Hit Count**	**Cache Miss Count**	**Cache Hit Rate**
Direct Mapping	2	128	128	50
Direct Mapping	4	192	64	75
Direct Mapping	8	224	32	88
Direct Mapping	16	240	16	94
2-Way Set Associative	2	128	128	50
2-Way Set Associative	4	192	64	75
2-Way Set Associative	8	224	32	88
2-Way Set Associative	16	240	16	94
4-Way Set Associative	2	128	128	50
4-Way Set Associative	4	192	64	75
4-Way Set Associative	8	224	32	88
4-Way Set Associative	16	240	16	94

5.3 Design a direct mapped cache

To develop an instruction cache and a data cache, we first implement a direct mapped cache. Thus, in figure 5.3 we illustrate the entity of the direct mapped cache with all input and output ports.
Of course, the clock signal *clk* is used to handle the behavior of this entity. The *reset* signal is used to reset the direct mapped cache. When the direct mapped cache is reset, all cache block lines will become invalid. The input port *addrCPU* stores the address given from the CPU to the cache. This address determines the cache block line to be read or to be written. The two inout ports *dataCPU* and *dataMEM* are needed to pass data word between the CPU and the cache as well as between the main memory and the cache. The input signal *newCacheBlockLine* stores the new cache block line to be written into the direct mapped cache. Also, there are several control signals to specify, whether the direct mapped cache should be written or read. The signals *wrCBLine* and *rdCBLine* states whether a whole cache block line should be written or read. Accordingly, the signals *rdWord* and *wrWord* satisfy whether a single word should be written or read in the direct mapped cache. The control signal *wrNewCBLine* says, whether a new cache block line should be written into the direct mapped cache. Furthermore, the input ports *setValid* and *setDirty* controls whether the dirty bit and the valid bit should be reset. The inout port *dirty* stores the new value of the dirty bit or the returns the current value of the dirty bit regarding the current cache block line. Finally, the out port *hit* signalises whether a cache hit or a cache miss is achieved during a read/write operation.

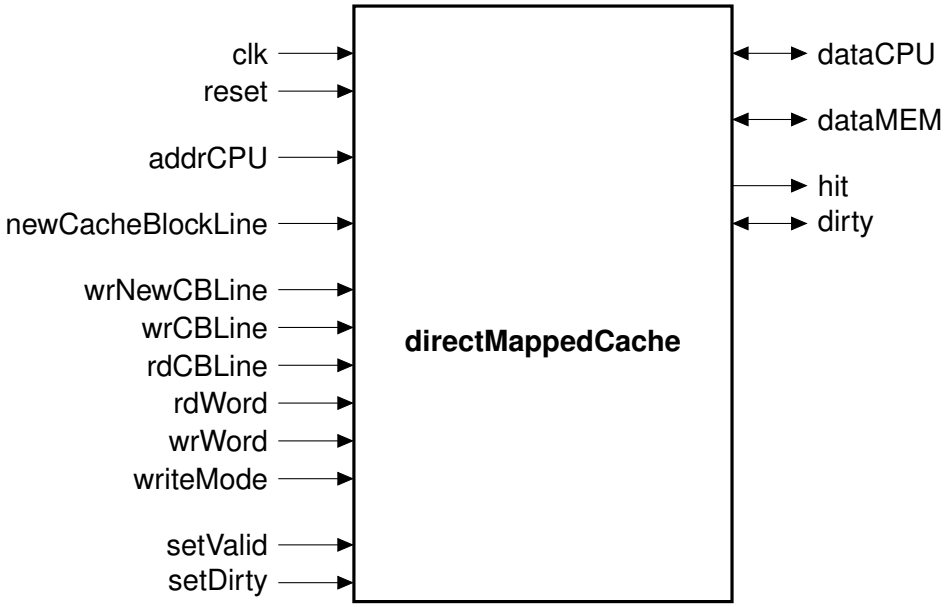


Figure 1: Entity of directMappedCache

5.4 Design a Finite State Machine for the Cache

After we have implemented the direct mapped cache, we have to design a finite state machine for the cache controller. This controller have to count the cache hits and cache misses using counters, which are reset at program start. On the one hand, we implement the write back policy. On the other hand, we implement the write allocate policy.
In figure 2 the state diagram of the cache controller is illustrated. The state diagram represents a Mealy automaton. Besides the state machine inputs are listed in table 3 and the state machine outputs are shown in table 4. A sketch of the state diagram is printed in figure 3.
In the following, we describe the state space of the state machine:

IDLE This state is the initial state of the state machine. When the cache is reset and state machine switches to this state.

CHECK1 In this state, the cache is checking whether write operation will results in a cache hit or not. Thus, the valid bit, dirty bit and the tag values of the correspondent cache block line are relevant for checking cache hit or cache miss.

CHECK2 In this state, the cache is checking whether read operation will results in a cache hit or not. Thus, the valid bit, dirty bit and the tag values of the correspondent cache block line are relevant for checking cache hit or cache miss.

WRITEBACK1 When checking the current cache block line results in a cache miss, and the current cache block line is dirty, this cache block line must be written back to the main memory first. Inside this state, the cache writes the cache block line back to the main memory. The process of writting back requires a specific number of clock cycles. Thus, we have to wait for the main memory. If the main memory signalizes that it is ready, then we can change this state to the next state.

WRITEBACK2 When checking the current cache block line results in a cache miss, and the current cache block line is dirty, this cache block line must be written back to the main memory first. Inside this state, the cache writes the cache block line back to the main memory. This process of writting back requires a specific number of clock cycles. Thus, we have to wait for the main memory. If the main memory signalizes that it is ready, then we can change this state to the next state.

WRITE Before we will write the new data word into the cache, we have to read the cache block line from the main memory into the cache. While the cache is reading from the main memory, we stay in this state. When the main memory is ready and the cache finished reading from the main memory, we switch the current state.

READ In case of a cache miss, we have to read the correspondent cache block line from the main memory into the cache. While the we are reading from the main memory, we are inside this state. The main memory signalizes via a correspondent signal if the read operation is finished.

TOCACHE1 When the write operation has been finished, we achieve this state of the state machine. This state is necessary to increment the miss counter by one.

TOCACHE2 When the read operation has been finished, we achieve this state. This state is necessary to increment the miss counter by one. Also, we transmit the requested data word from the cache to the CPU.

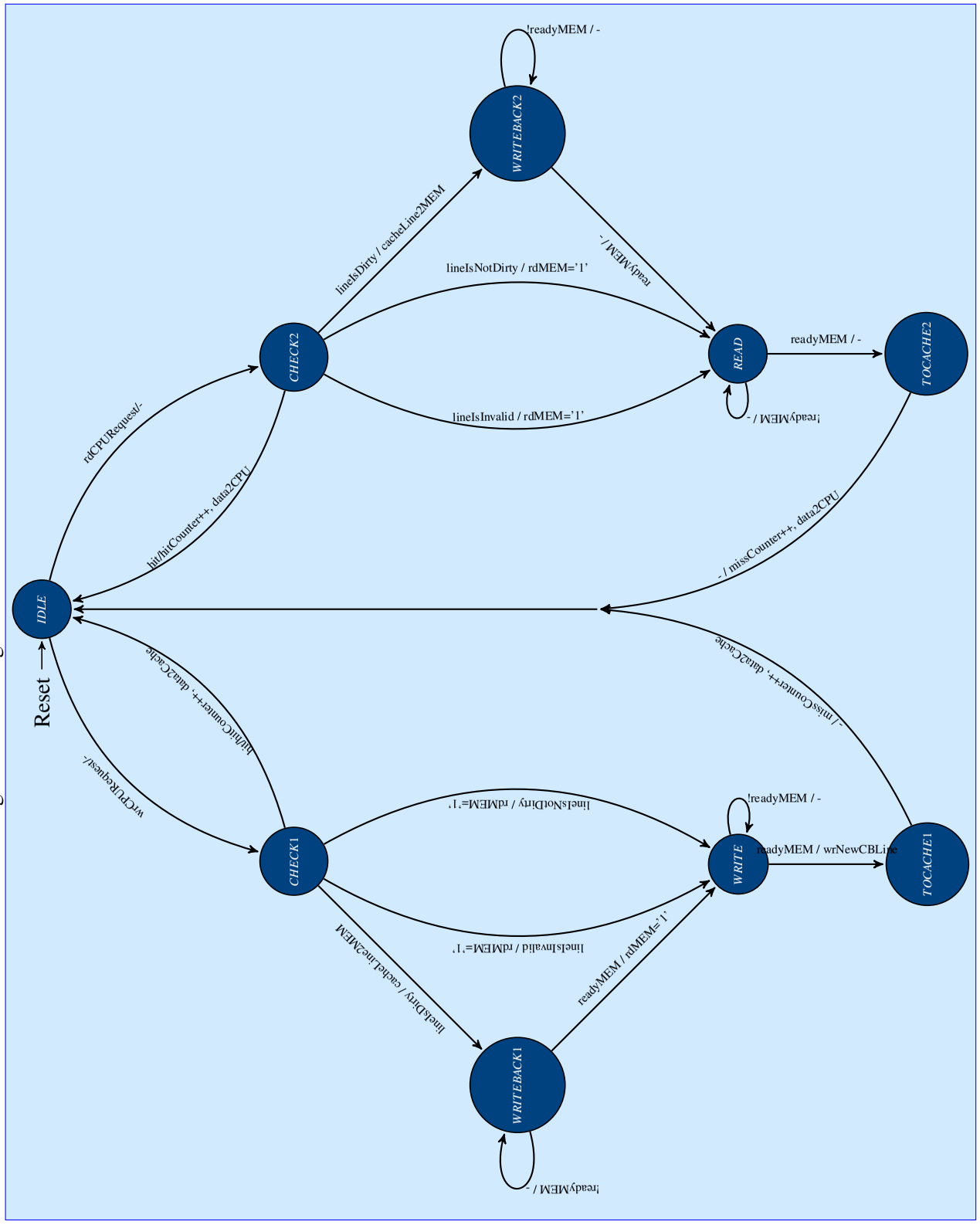


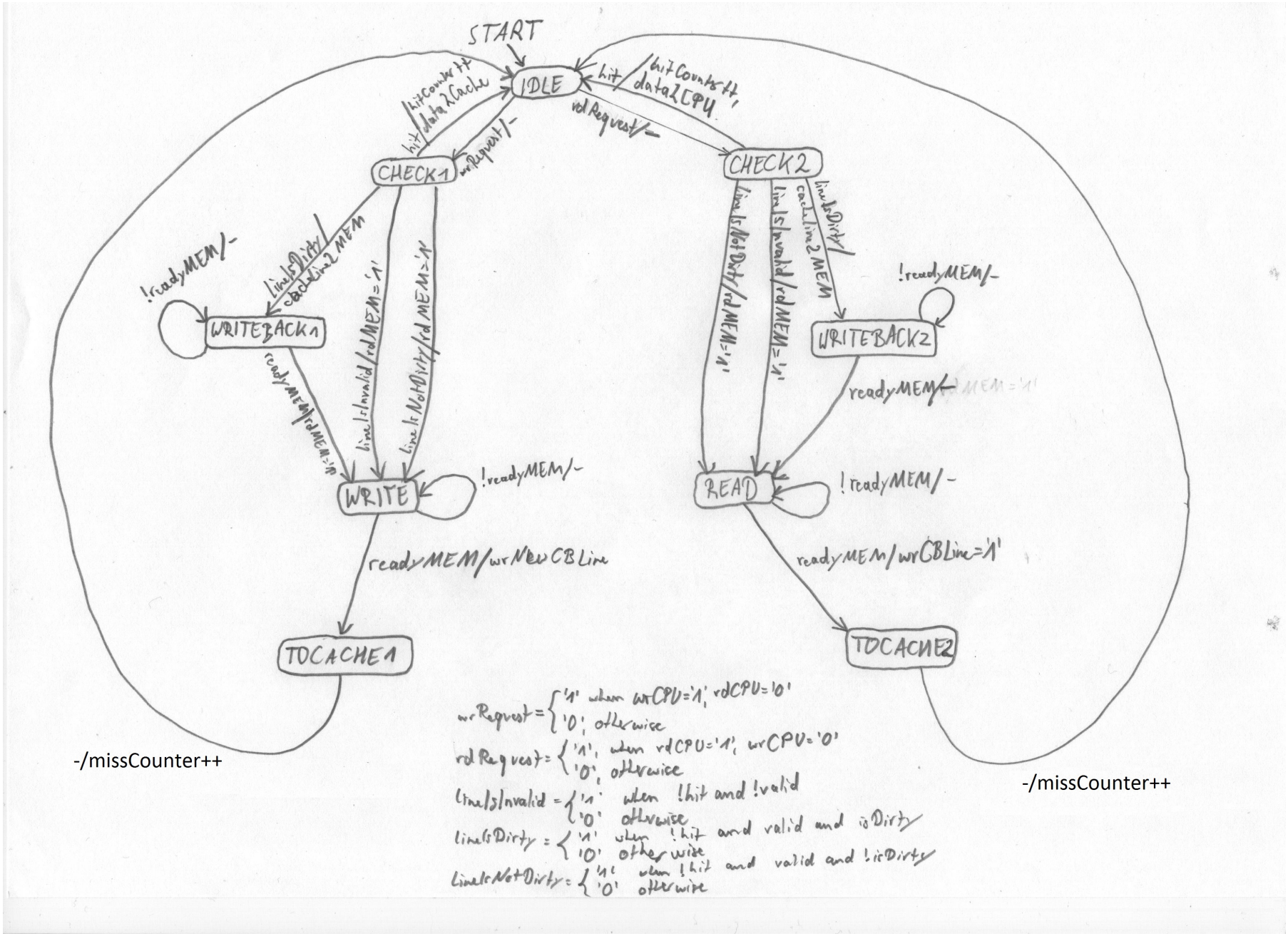
Figure 2: State diagram of the cache controller.

Table 3: Overview - FSM Inputs

Abbreviation	Name	Description
rdCPU	CPU Read Request	-
wrCPU	CPU Write Request	-
cacheMiss	Cache Miss	-
cacheHit	Cache Hit	-
readyMEM	Write-Back is resolved	-
isDirty	Cache Block is dirty	-

Table 4: Overview - FSM Outputs

Abbreviation	Name	Description
stallCPU	Stall Processor	-
setDirty	Set Dirty Bit (Modified) Bit	-
wrMEM	Write To Memory	Write Replaced Block To Memory
dataCPU	Read Data Into CPU	-
rdMEM	Read Cache Block Into Cache From Memory	-
dataCPU2Cache	Write Data Into Cache	-


START
IDLE
CHECK1
CHECK2
WRITEBACK1
WRITEBACK2
WRITE
READ
TOCACHE1
TOCACHE2
hit / hitCounter++, data2CPU
wrRequest = '1' when wrCPU='1', rdCPU='0'; '0' otherwise
rdRequest = '1' when rdCPU='1', wrCPU='0'; '0' otherwise
lineIsInvalid = '1' when !hit and !valid; '0' otherwise
lineIsDirty = '1' when !hit and valid and isDirty; '0' otherwise
lineIsNotDirty = '1' when !hit and valid and !isDirty; '0' otherwise
-/missCounter++
-/missCounter++

5.4.1 Design a Finite State Machine for the Main Memory Controller

The main memory controller has the purpose to either write a given cache block/line to the main memory or to read multiple words from the main memory and return these words as a cache block/line. This main memory controller will be connected with the cache controller. Thus, the main memory controller will send a read cache block/line from the main memory to the cache controller. Also the main memory will get a cache block/line from the cache controller, which should be written into the main memory. Consider that a single data word has a certain wide of bits and a whole cache block/line contains several data words. Furthermore, the main memory could be implemented as a BlockRAM (BRAM). At first, the main memory controller is implemented as a finite state machine of type Mealy. The sketch of the finite state machine is given in figure 4.

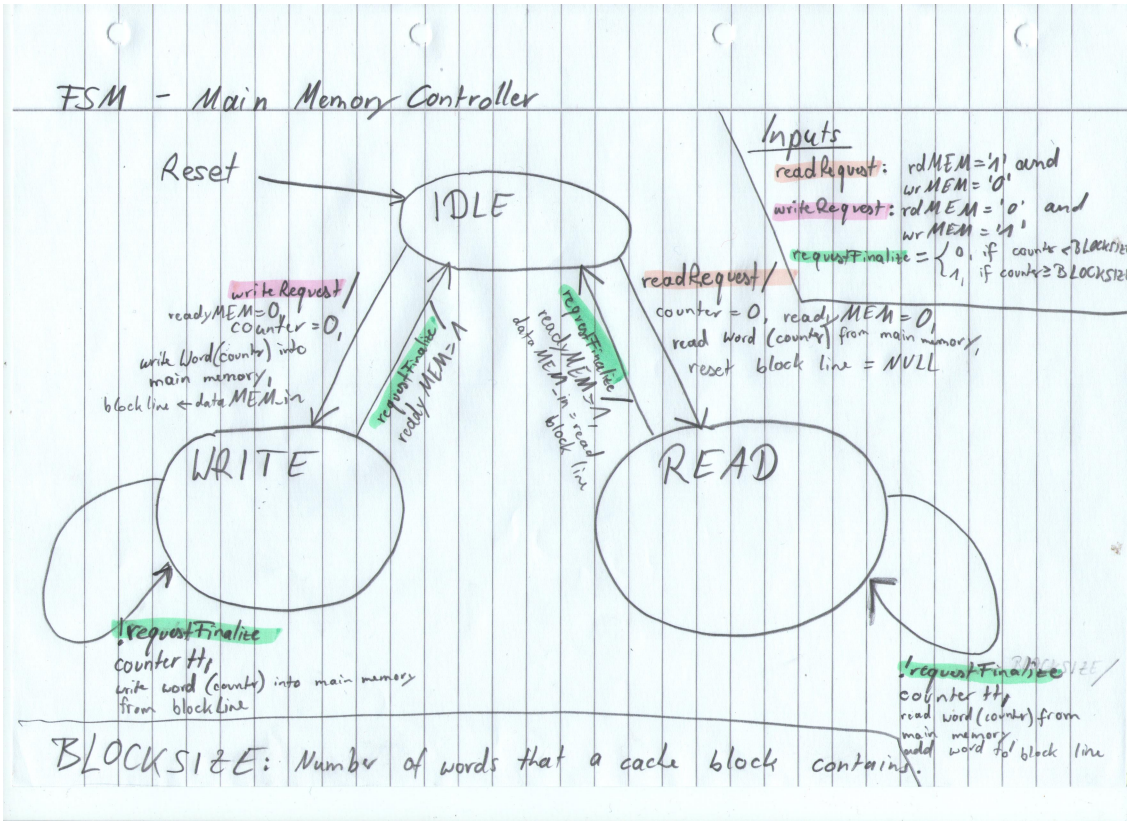


Figure 4: Sketch of Mealy Automata - Main Memory Controller

5.5 Design a testbench and simulate the Cache

After implementation of the Cache with *Write Back Policy* and *Write Allocate Policy* we write a testbench and simulate a system with the following properties:

- Main memory using a BlockRAM with ready signal.
- Direct Mapped Cache with 256 blocks/lines. Each block/line has 4 words. The cache use the write back scheme. Also, byte access is possible.

The testbench should verify the behavior of the cache. Therefore, we look at different test cases. In the following, these test cases are described.

Reset Cache I If the cache is reset, then the miss counter and the hit counter are reset to zero.

Reset Cache II If the cache is reset, then all cache blocks/lines are invalid.

Read Cache, Line is Not Dirty In dem zu lesenden Cacheblock befinden sich bereits gültige Daten. Die Daten sind nicht geändert gegenüber dem Hauptspeicher. Es wird nun erneut gelesen, wobei die Tags unterschiedlich sind. Daher wird aus dem Hauptspeicher in den Cache gelesen. Entsprechend wird das Stall-Signal auf 1 gesetzt und der Miss-Zähler erhöht.

Read Cache - Different Offset Im ersten Lesebefehl wird aus einem Offset-Block aus einem Cacheblock gelesen. Beim nächsten Lesebefehl wird aus dem gleichen Cacheblock aus einem anderen Offset-Block gelesen. Entsprechend wird das Stall-Signal auf 1 gesetzt und der Miss-Zähler erhöht.

Read Cache - Line is Dirty In dem zu lesenden Cacheblock befinden sich bereits gültige Daten. Die Daten sind geändert gegenüber dem Hauptspeicher. Es wird nun erneut gelesen, wobei die Tags unterschiedlich sind. Daher werden die Daten aus dem Cache zuvor in den Hauptspeicher zurückgeschrieben.

Write Cache - Invalid Cacheblocks Zu Beginn sind alle Cacheblöcke invalid. Deshalb wird, wenn ein Cacheblock gelesen wird, aus dem Hauptspeicher gelesen. Entsprechend wird das Stall-Signal auf 1 gesetzt und der Miss-Counter erhöht.

Write Cache – Line is Dirty In dem zu schreibenden Cacheblock befinden sich bereits gültige Daten. Die Daten sind gegenüber dem Hauptspeicher geändert. Es wird nun erneut geschrieben, wobei die Tags unterschiedlich sind. Daher werden die Daten aus dem Cache zuvor in den Hauptspeicher zurückgeschrieben.

Write Cache – Line Is Not Dirty Let's assume that there are already valid, clean data in a cache block/line. If we write new data to this cache block/line and the tags are different, then the valid, clean data will not be written back to the main memory. Instead of that, the correspondent block are read from memory to cache and the relevant offset block is replaced with the new data word. We expect, that the miss counter will be incremented.

Write Cache - Hit Let's assume that there are already valid (clean or invalid) data in the cache block line. If we write new data to this cache block/line and the tags are equal, then then the old data will not be written back to the main memory. Instead of that, the correspondent cache block line is directly rewritten with the new data word. We expect, that the hit counter is incremented.

6 Summary

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

7 Appendix

7.1 Implementation

```
1  ##################################################################
2  #
3  #  Column-major order traversal of 16 x 16 array of words.
4  #  Pete Sanderson
5  #  31 March 2007
6  #
7  #  To easily observe the column-oriented order, run the Memory Reference
8  #  Visualization tool with its default settings over this program.
9  #  You may, at the same time or separately, run the Data Cache Simulator
10 #  over this program to observe caching performance.  Compare the results
11 #  with those of the row-major order traversal algorithm.
12 #
13 #  The C/C++/Java-like equivalent of this MIPS program is:
14 #     int size = 16;
15 #     int[size][size] data;
16 #     int value = 0;
17 #     for (int col = 0; col < size; col++) {
18 #        for (int row = 0; row < size; row++) }
19 #           data[row][col] = value;
20 #           value++;
21 #        }
22 #     }
23 #
24 #  Note: Program is hard-wired for 16 x 16 matrix.  If you want to change
       this,
25 #        three statements need to be changed.
26 #        1. The array storage size declaration at "data:" needs to be changed
        from
27 #           256 (which is 16 * 16) to #columns * #rows.
28 #        2. The "li" to initialize $t0 needs to be changed to the new #rows.
29 #        3. The "li" to initialize $t1 needs to be changed to the new #
       columns.
30 #
31         .data
32 data:   .word     0 : 256       # 16x16 matrix of words
33         .text
34         li       $t0, 16        # $t0 = number of rows
35         li       $t1, 16        # $t1 = number of columns
36         move     $s0, $zero     # $s0 = row counter
37         move     $s1, $zero     # $s1 = column counter
38         move     $t2, $zero     # $t2 = the value to be stored
39 #  Each loop iteration will store incremented $t1 value into next element of
       matrix.
40 #  Offset is calculated at each iteration. offset = 4 * (row*#cols+col)
41 #  Note: no attempt is made to optimize runtime performance!
42 loop:   mult     $s0, $t1       # $s2 = row * #cols  (two-instruction
       sequence)
43         mflo     $s2            # move multiply result from lo register to
             $s2
44         add      $s2, $s2, $s1  # $s2 += col counter
45         sll      $s2, $s2, 2    # $s2 *= 4 (shift left 2 bits) for byte
             offset
46         sw       $t2, data($s2) # store the value in matrix element
```

```
47          addi    $t2, $t2, 1     # increment value to be stored
48  #  Loop control: If we increment past bottom of column, reset row and
        increment column
49  #                If we increment past the last column, we're finished.
50          addi    $s0, $s0, 1     # increment row counter
51          bne     $s0, $t0, loop  # not at bottom of column so loop back
52          move    $s0, $zero      # reset row counter
53          addi    $s1, $s1, 1     # increment column counter
54          bne     $s1, $t1, loop  # loop back if not at end of matrix (past
                the last column)
55  #  We're finished traversing the matrix.
56          li      $v0, 10         # system service 10 is exit
57          syscall                 # we are outta here.
```

Listing 1: column-major.asm

```
1  ###############################################################################
2  #  Row-major order traversal of 16 x 16 array of words.
3  #  Pete Sanderson
4  #  31 March 2007
5  #
6  #  To easily observe the row-oriented order, run the Memory Reference
7  #  Visualization tool with its default settings over this program.
8  #  You may, at the same time or separately, run the Data Cache Simulator
9  #  over this program to observe caching performance.  Compare the results
10 #  with those of the column-major order traversal algorithm.
11 #
12 #  The C/C++/Java-like equivalent of this MIPS program is:
13 #     int size = 16;
14 #     int[size][size] data;
15 #     int value = 0;
16 #     for (int row = 0; col < size; row++) {
17 #        for (int col = 0; col < size; col++) }
18 #           data[row][col] = value;
19 #           value++;
20 #        }
21 #     }
22 #
23 #  Note: Program is hard-wired for 16 x 16 matrix.  If you want to change
       this,
24 #        three statements need to be changed.
25 #        1. The array storage size declaration at "data:" needs to be changed
       from
26 #           256 (which is 16 * 16) to #columns * #rows.
27 #        2. The "li" to initialize $t0 needs to be changed to new #rows.
28 #        3. The "li" to initialize $t1 needs to be changed to new #columns.
29 #
30           .data
31 data:     .word     0 : 256       # storage for 16x16 matrix of words
32           .text
33           li        $t0, 16       # $t0 = number of rows
34           li        $t1, 16       # $t1 = number of columns
35           move      $s0, $zero    # $s0 = row counter
36           move      $s1, $zero    # $s1 = column counter
37           move      $t2, $zero    # $t2 = the value to be stored
38 #  Each loop iteration will store incremented $t1 value into next element of
       matrix.
39 #  Offset is calculated at each iteration. offset = 4 * (row*#cols+col)
40 #  Note: no attempt is made to optimize runtime performance!
41 loop:     mult      $s0, $t1      # $s2 = row * #cols  (two-instruction
       sequence)
42           mflo      $s2           # move multiply result from lo register to
             $s2
43           add       $s2, $s2, $s1 # $s2 += column counter
44           sll       $s2, $s2, 2   # $s2 *= 4 (shift left 2 bits) for byte
             offset
45           sw        $t2, data($s2) # store the value in matrix element
46           addi      $t2, $t2, 1   # increment value to be stored
47 #  Loop control: If we increment past last column, reset column counter and
```

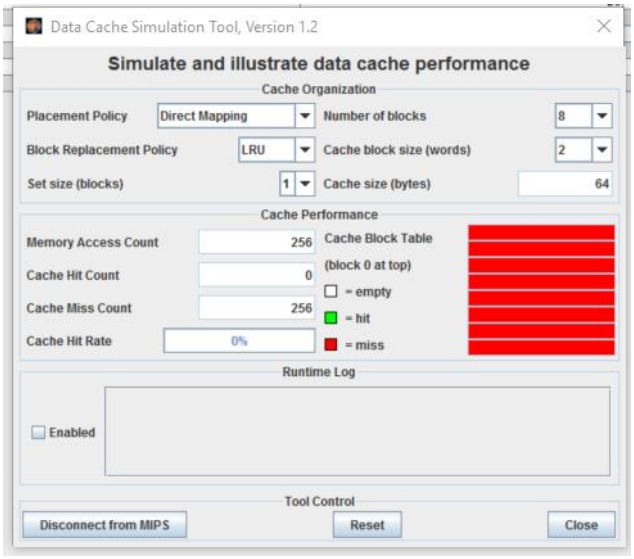


Figure 5: Column Major, Direct Mapping, Cache Block Size 2

```
    increment row counter
48  #                If we increment past last row, we're finished.
49          addi     $s1, $s1, 1     # increment column counter
50          bne      $s1, $t1, loop  # not at end of row so loop back
51          move     $s1, $zero      # reset column counter
52          addi     $s0, $s0, 1     # increment row counter
53          bne      $s0, $t0, loop  # not at end of matrix so loop back
54  #  We're finished traversing the matrix.
55          li       $v0, 10         # system service 10 is exit
56          syscall                  # we are outta here.
```

Listing 2: row-major.asm

7.2 Cache Results - Snapshots

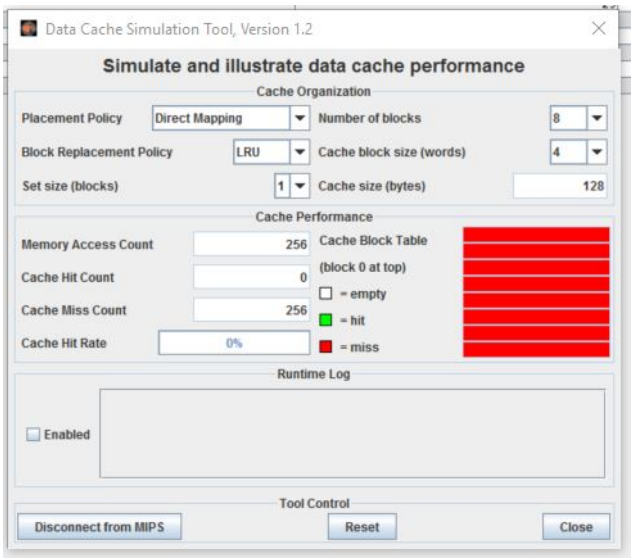


Figure 6: Column Major, Direct Mapping, Cache Block Size 4

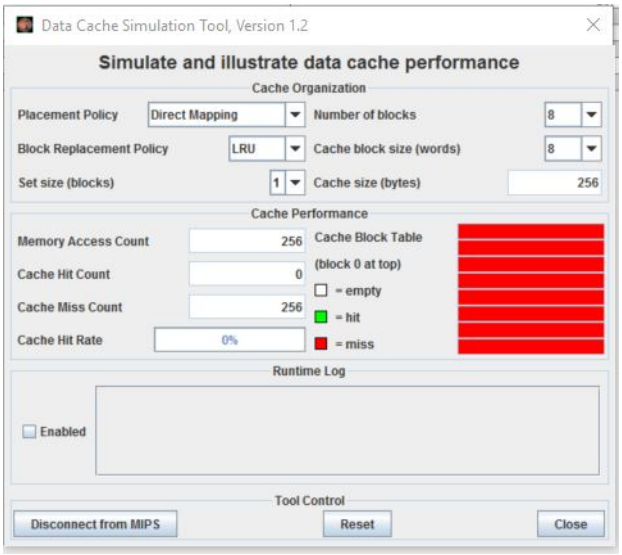


Figure 7: Column Major, Direct Mapping, Cache Block Size 8

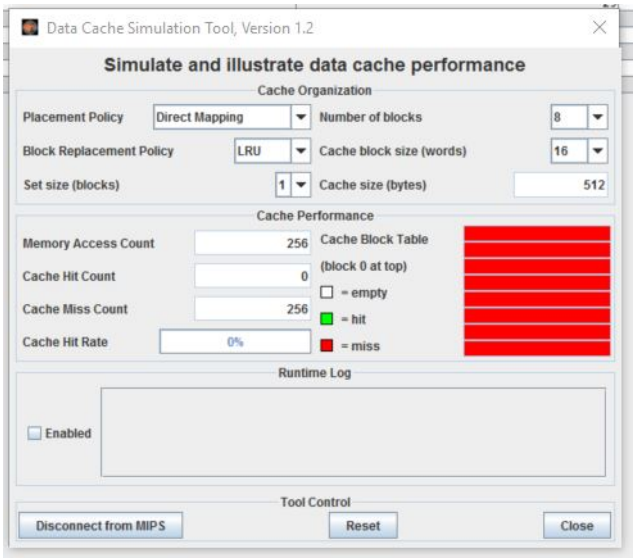


Figure 8: Column Major, Direct Mapping, Cache Block Size 16

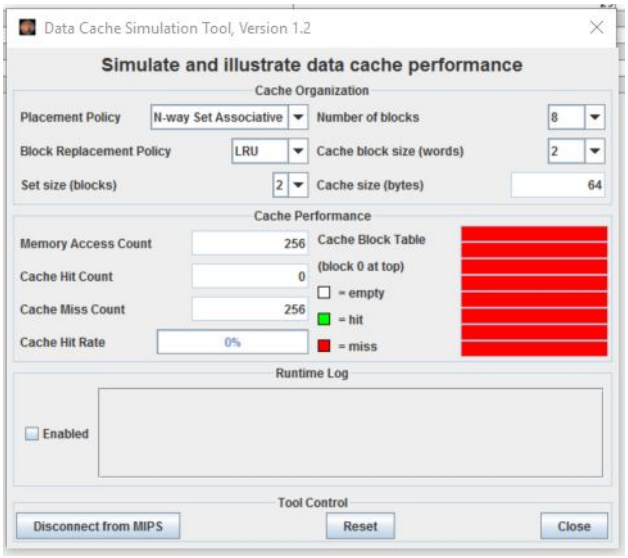


Figure 9: Column Major, 2-Way Associative, Cache Block Size 2

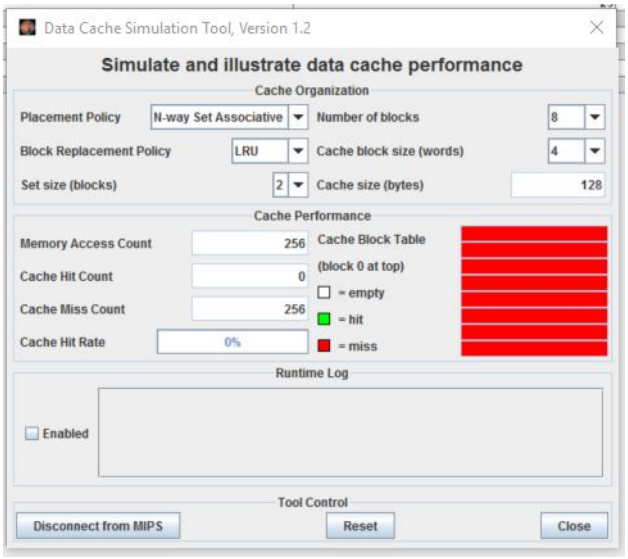


Figure 10: Column Major, 2-Way Associative, Cache Block Size 4

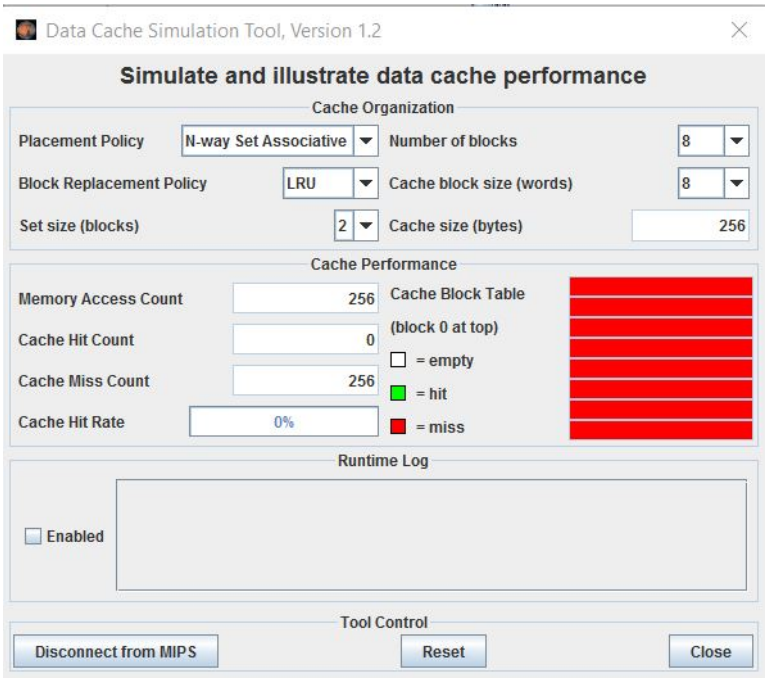


Figure 11: Column Major, 2-Way Associative, Cache Block Size 8

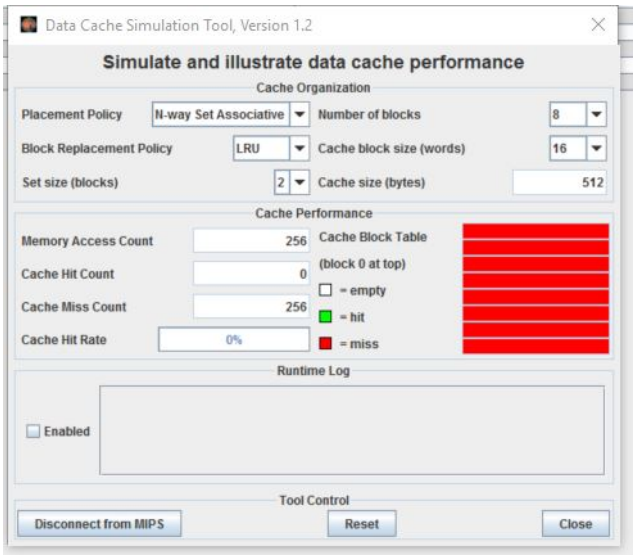


Figure 12: Column Major, 2-Way Associative, Cache Block Size 16

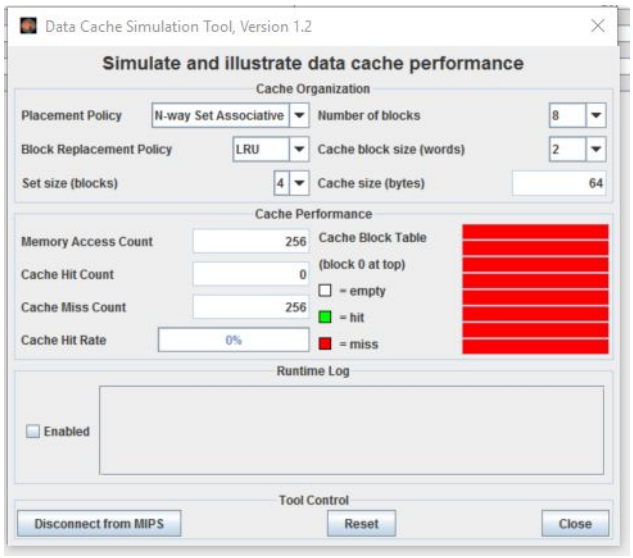


Figure 13: Column Major, 4-Way Associative, Cache Block Size 2

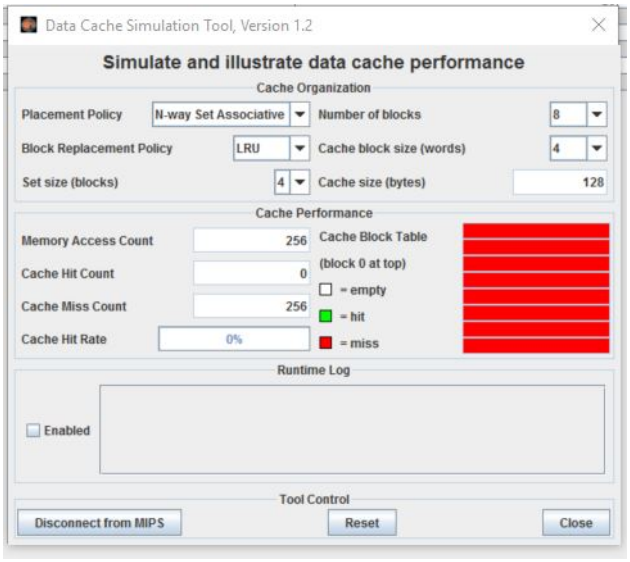


Figure 14: Column Major, 4-Way Associative, Cache Block Size 4

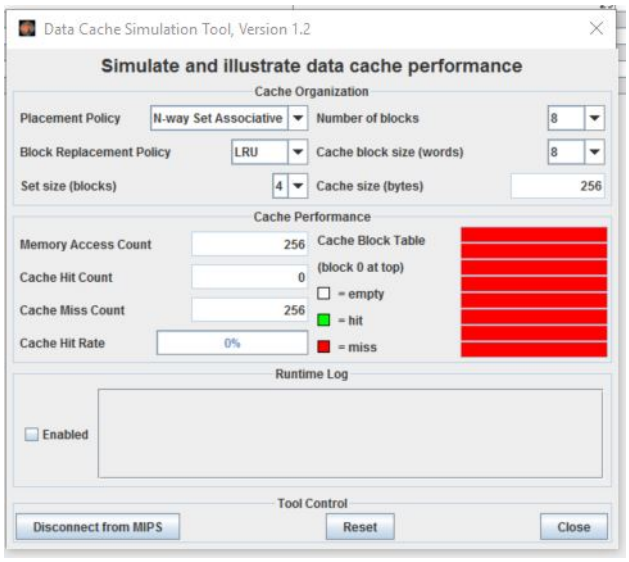


Figure 15: Column Major, 4-Way Associative, Cache Block Size 8

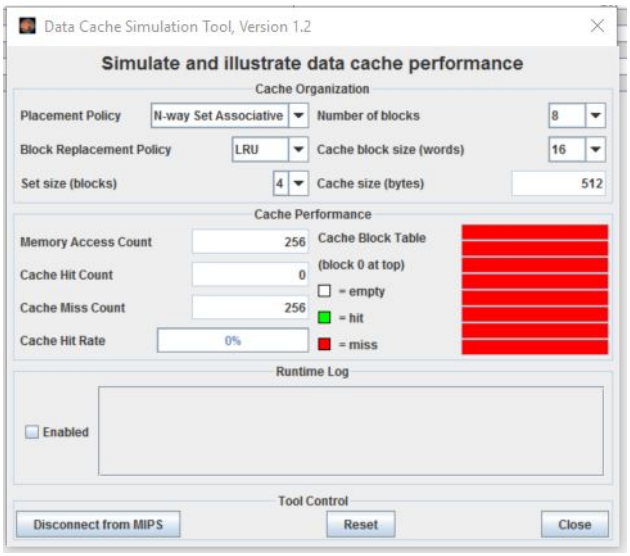


Figure 16: Column Major, 4-Way Associative, Cache Block Size 16

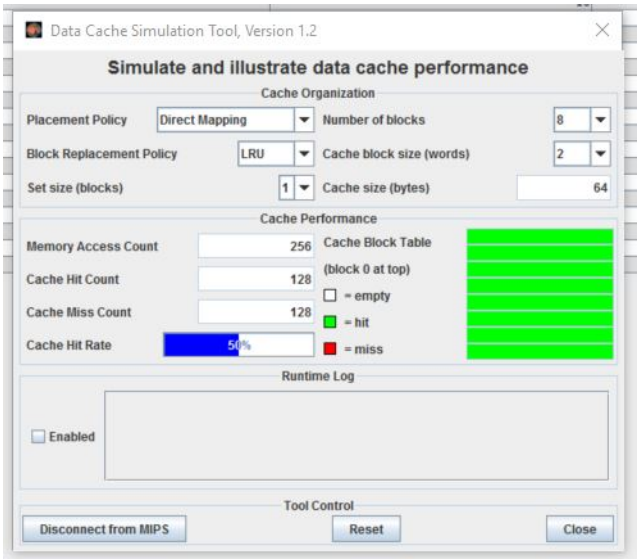


Figure 17: Row Major, Direct Mapping, Cache Block Size 2

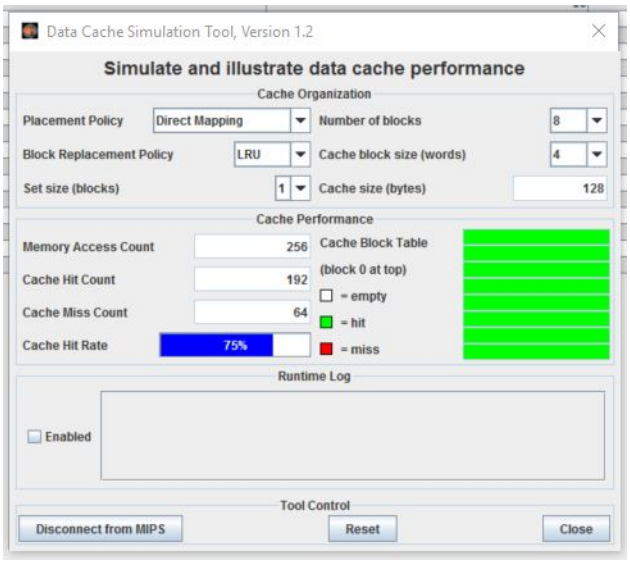


Figure 18: Row Major, Direct Mapping, Cache Block Size 4

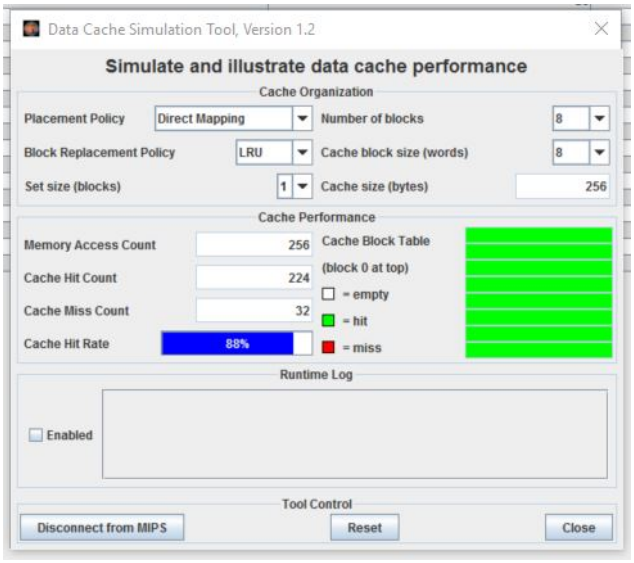


Figure 19: Row Major, Direct Mapping, Cache Block Size 8

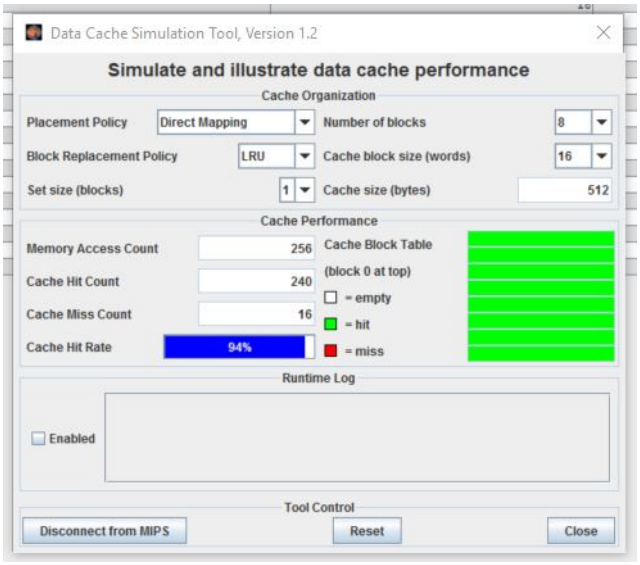


Figure 20: Row Major, Direct Mapping, Cache Block Size 16

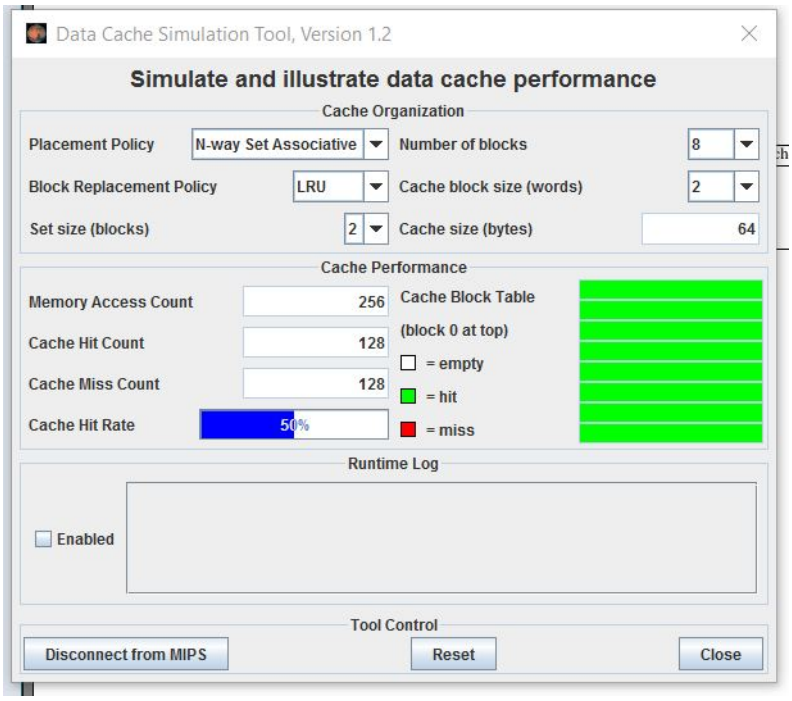


Figure 21: Row Major, 2-Way Associative, Cache Block Size 2

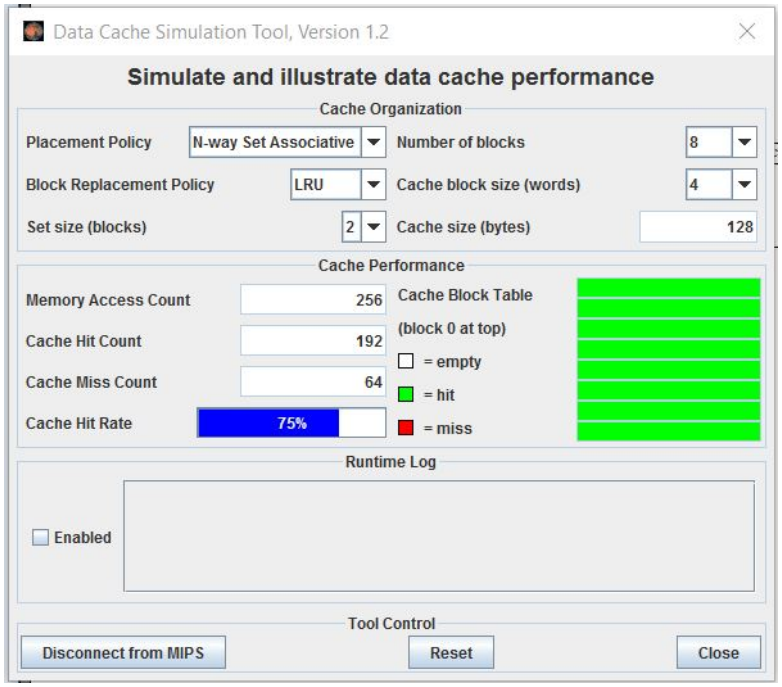


Figure 22: Row Major, 2-Way Associative, Cache Block Size 4

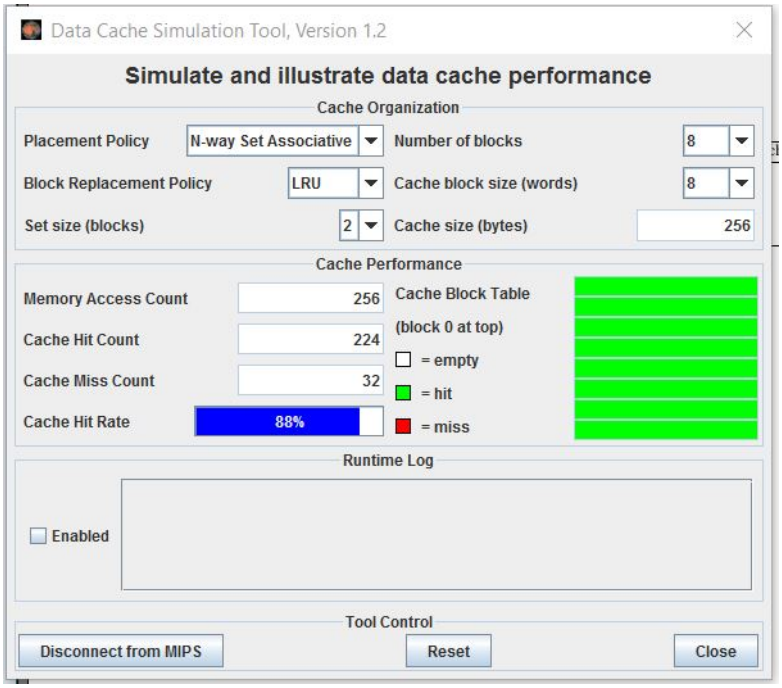


Figure 23: Row Major, 2-Way Associative, Cache Block Size 8

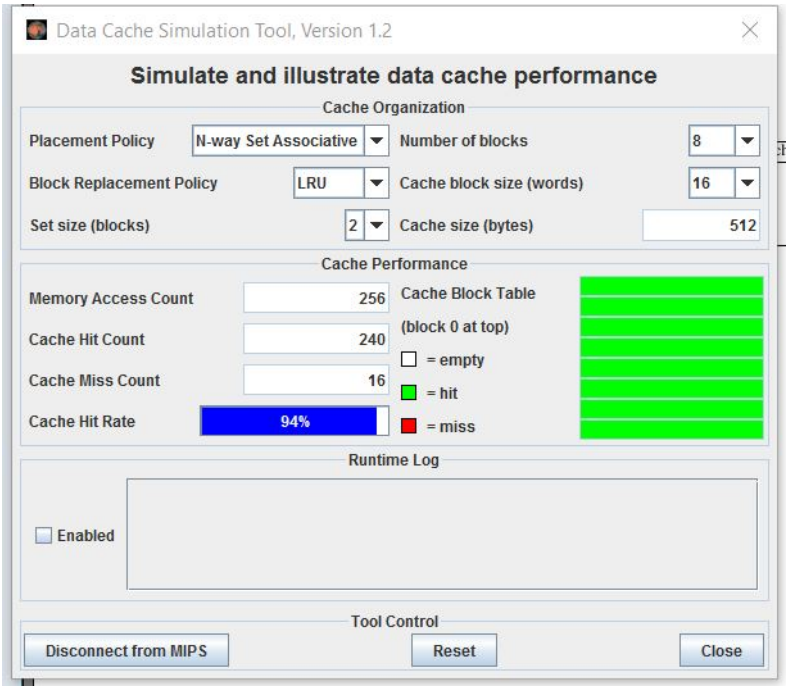


Figure 24: Row Major, 2-Way Associative, Cache Block Size 16

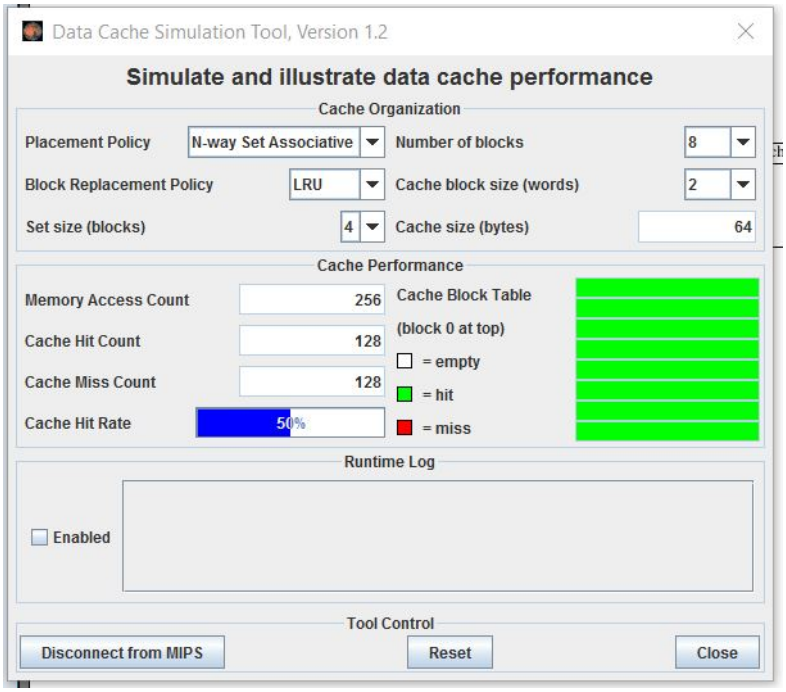


Figure 25: Row Major, 4-Way Associative, Cache Block Size 2

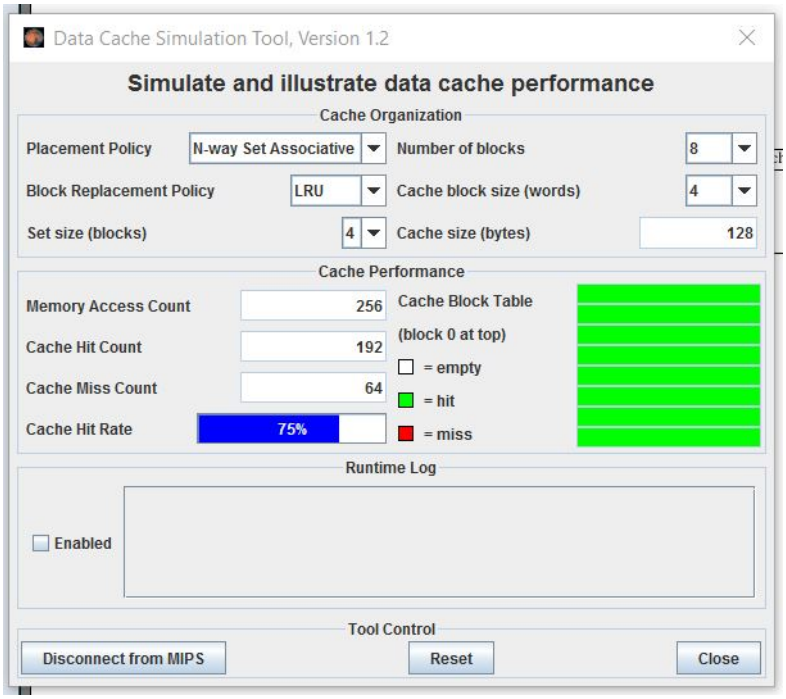


Figure 26: Row Major, 4-Way Associative, Cache Block Size 4

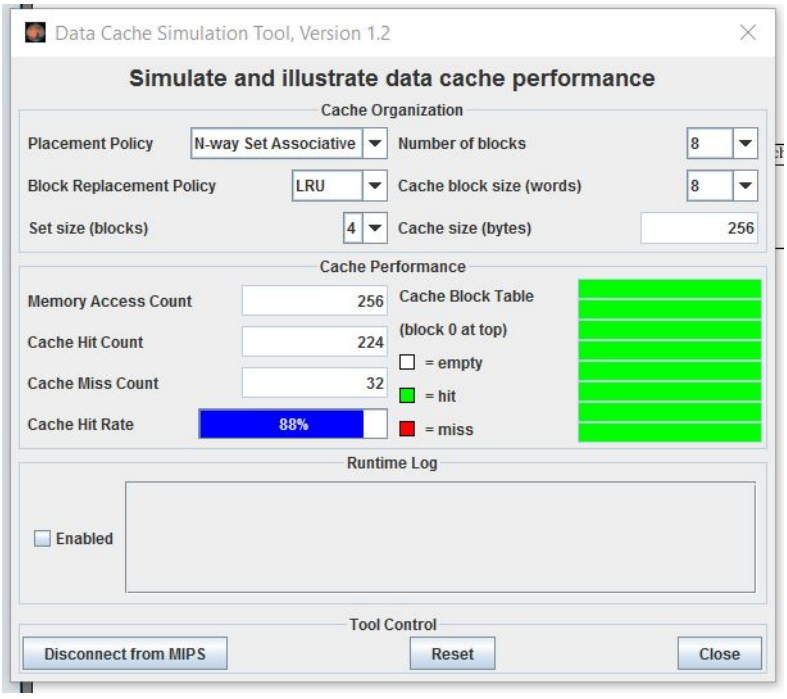


Figure 27: Row Major, 4-Way Associative, Cache Block Size 8

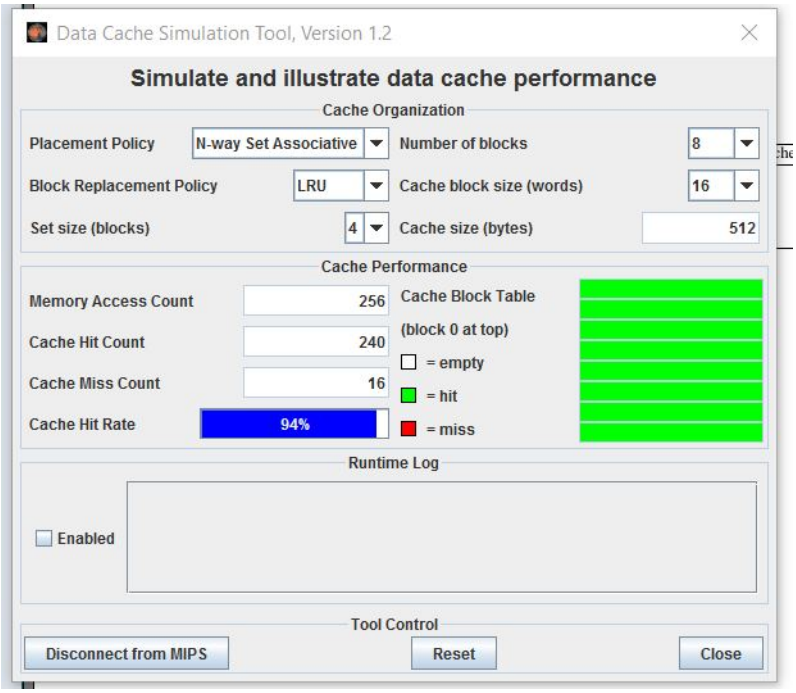


Figure 28: Row Major, 4-Way Associative, Cache Block Size 16