



HAMBURG UNIVERSITY OF TECHNOLOGY

PROBLEM-BASED LEARNING

Advanced System-on-Chip Design

author
@tuhh.de

Report

Tutor
Dipl.-Ing. Wolfgang BRANDT

March 12, 2017

Contents

1	Introduction	5
2	Task 1 - Introduction Set Architecture of the MIPS-Processor	6
3	Task 2 - VHDL Introduction	7
4	Task 3 - MIPS Extension	8
5	Task 4 - Caches	9
5.1	Memories	9
5.2	Cache Simulation - Results	9
5.3	Design a Finite State Machine for the Cache	11
5.3.1	Design a Finite State Machine for the Main Memory Controller	17
5.3.2	Design a testbench and simulate the Cache	18
6	Summary	19
7	Appendix	20
7.1	Implementation	20
7.2	Cache Results - Snapshots	24

List of Figures

1	State diagram of the cache controller.	13
2	State diagram of the cache controller.	15
3	Sketch of Mealy Automata - Cache Controller	16
4	Sketch of Mealy Automata - Main Memory Controller	17
5	Column Major, Direct Mapping, Cache Block Size 2	24
6	Column Major, Direct Mapping, Cache Block Size 4	25
7	Column Major, Direct Mapping, Cache Block Size 8	25
8	Column Major, Direct Mapping, Cache Block Size 16	26
9	Column Major, 2-Way Associative, Cache Block Size 2	26
10	Column Major, 2-Way Associative, Cache Block Size 4	27
11	Column Major, 2-Way Associative, Cache Block Size 8	27
12	Column Major, 2-Way Associative, Cache Block Size 16	28
13	Column Major, 4-Way Associative, Cache Block Size 2	28
14	Column Major, 4-Way Associative, Cache Block Size 4	29
15	Column Major, 4-Way Associative, Cache Block Size 8	29
16	Column Major, 4-Way Associative, Cache Block Size 16	30
17	Row Major, Direct Mapping, Cache Block Size 2	30
18	Row Major, Direct Mapping, Cache Block Size 4	31
19	Row Major, Direct Mapping, Cache Block Size 8	31
20	Row Major, Direct Mapping, Cache Block Size 16	32
21	Row Major, 2-Way Associative, Cache Block Size 2	32
22	Row Major, 2-Way Associative, Cache Block Size 4	33
23	Row Major, 2-Way Associative, Cache Block Size 8	33
24	Row Major, 2-Way Associative, Cache Block Size 16	34
25	Row Major, 4-Way Associative, Cache Block Size 2	34
26	Row Major, 4-Way Associative, Cache Block Size 4	35
27	Row Major, 4-Way Associative, Cache Block Size 8	35
28	Row Major, 4-Way Associative, Cache Block Size 16	36

List of Tables

1	Cache Simulation of Column Major	10
2	Cache Simulation of Row Major	11
3	Overview - FSM States	14
4	Overview - FSM Inputs	14
5	Overview - FSM Outputs	14

Listings

1	column-major.asm	21
2	row-major.asm	23

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.

5.1 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 advantage 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

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

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 column 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 remaining elements 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 2: Cache Simulation of Row Major

1	2	3	4	5
1	2	3	4	5
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 Finite State Machine for the Cache

In figure 2 the state diagram of the cache controller is illustrated. The state diagram represents a Mealy automaton. The state space of the state machine is given in table 3. Besides the state machine inputs are listed in table 4 and the state machine outputs are shown in table 5. A sketch of the state diagram is printed in figure 3.

Figure 1: State diagram of the cache controller.

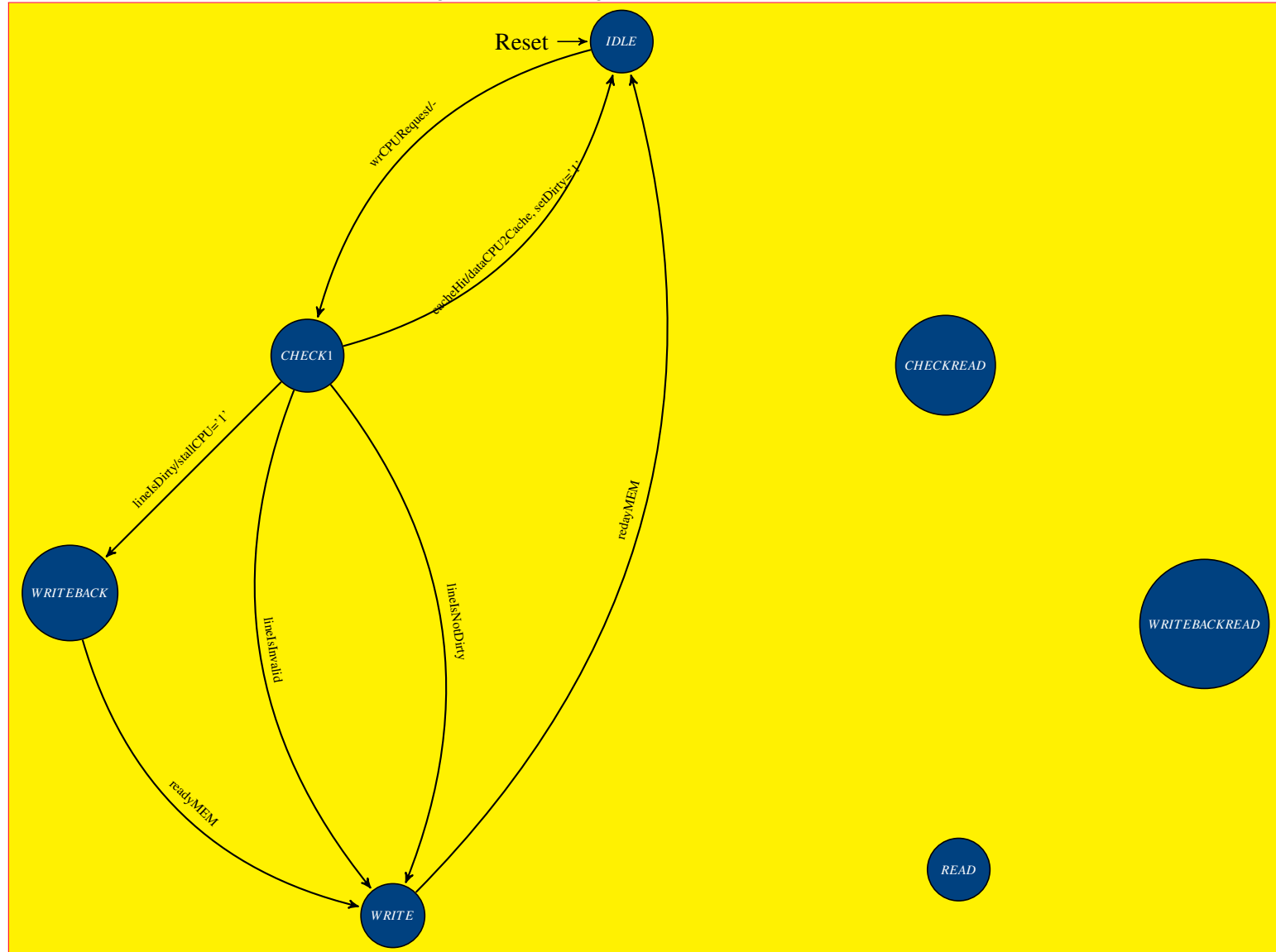


Table 3: Overview - FSM States

Abbreviation	Name	CPU Request Mode	Description
IDLE	-	-	-
CW	COMPARE WRITE	Write Request	-
CMW	CACHE MISS WRITE	Write Request	-
WBW	WRITE BACK WRITE	Write Request	-
WCW	WRITE CACHE WRITE	Write Request	-
CR	COMPARE READ	Read Request	-
CMR	CACHE MISS READ	Read Request	-
WBR	WRITE BACK READ	Read Request	-
WCR	WRITE CACHE READ	Read Request	-

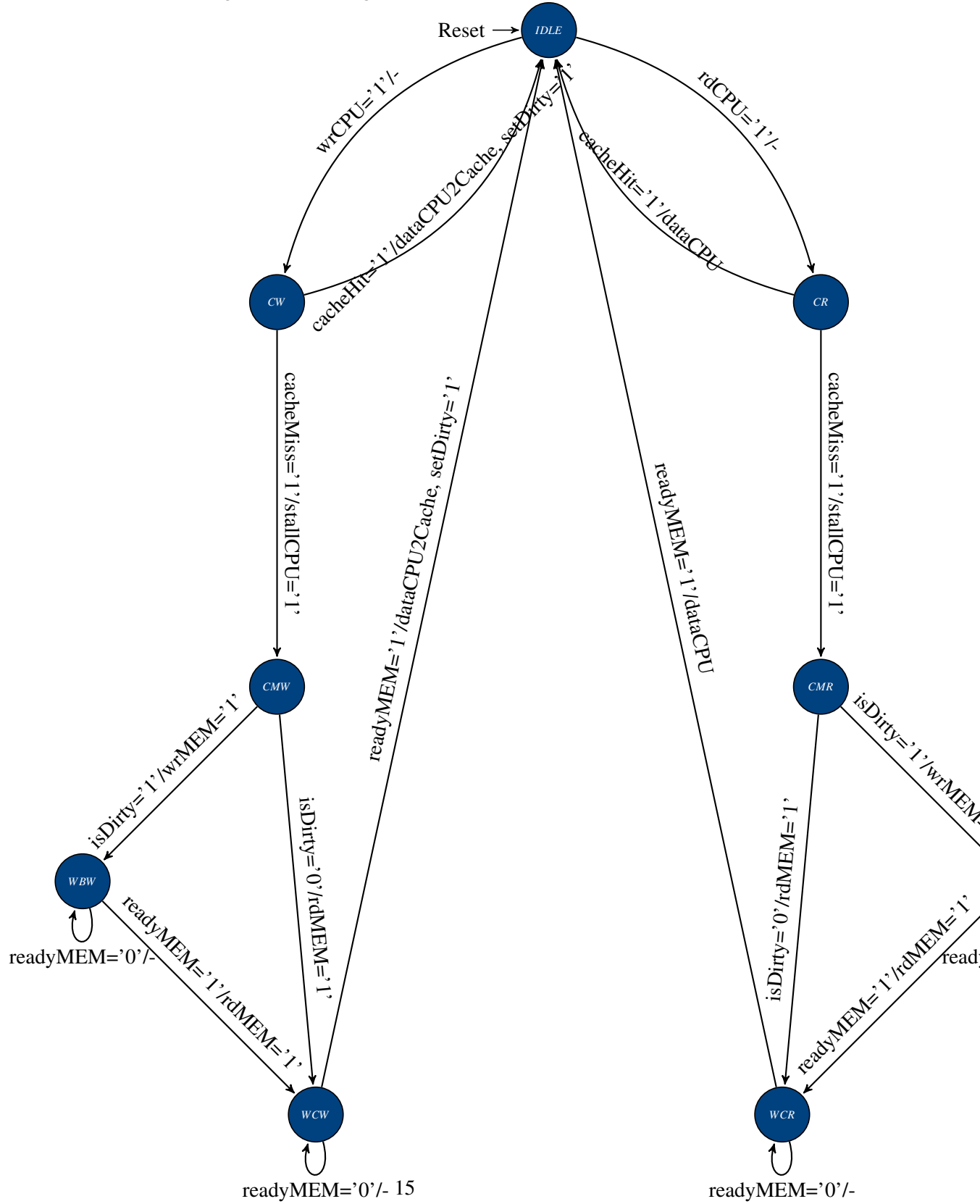
Table 4: Overview - FSM Inputs

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 5: 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	-

Figure 2: State diagram of the cache controller.



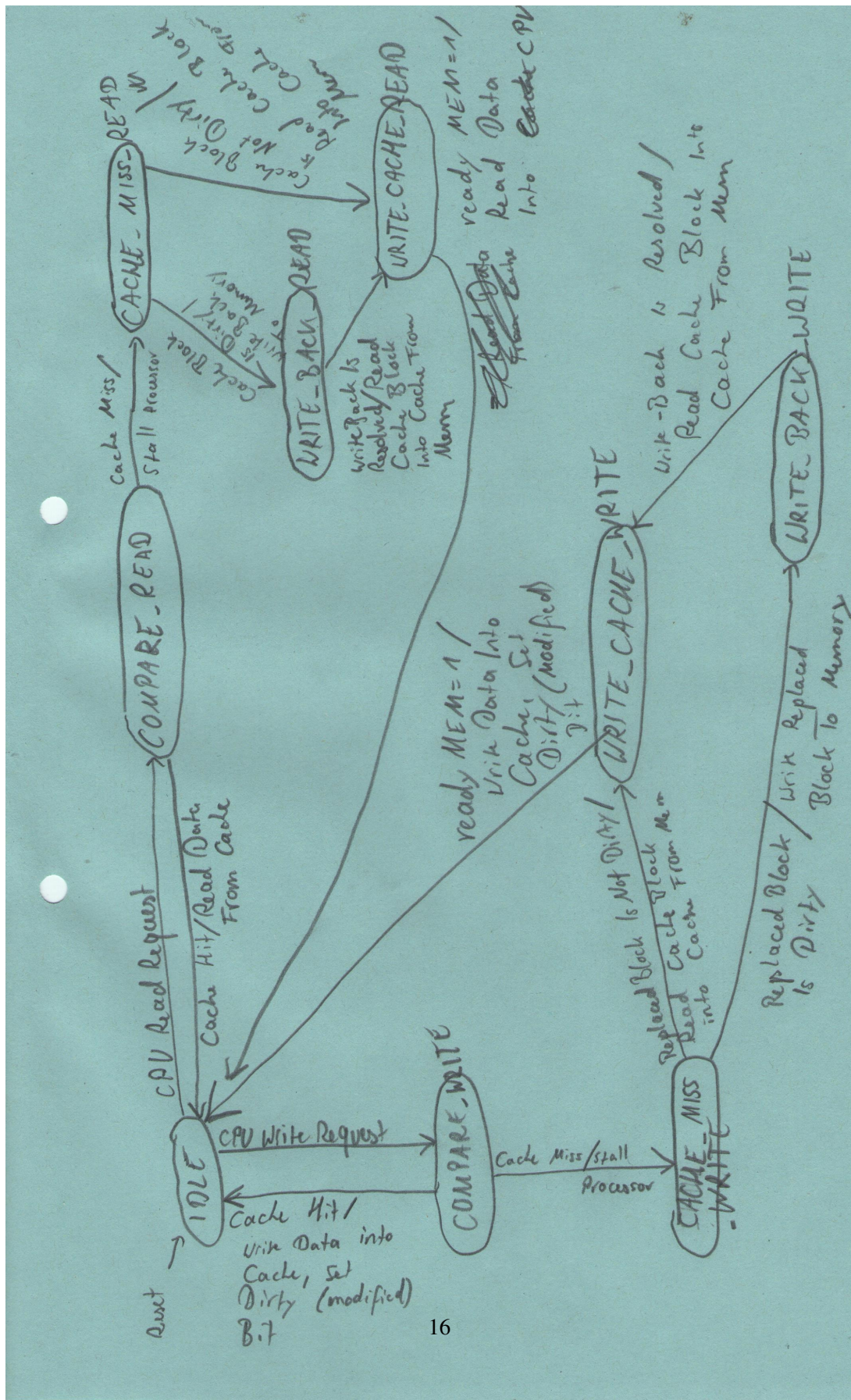


Figure 3: Sketch of Mealy Automata - Cache Controller

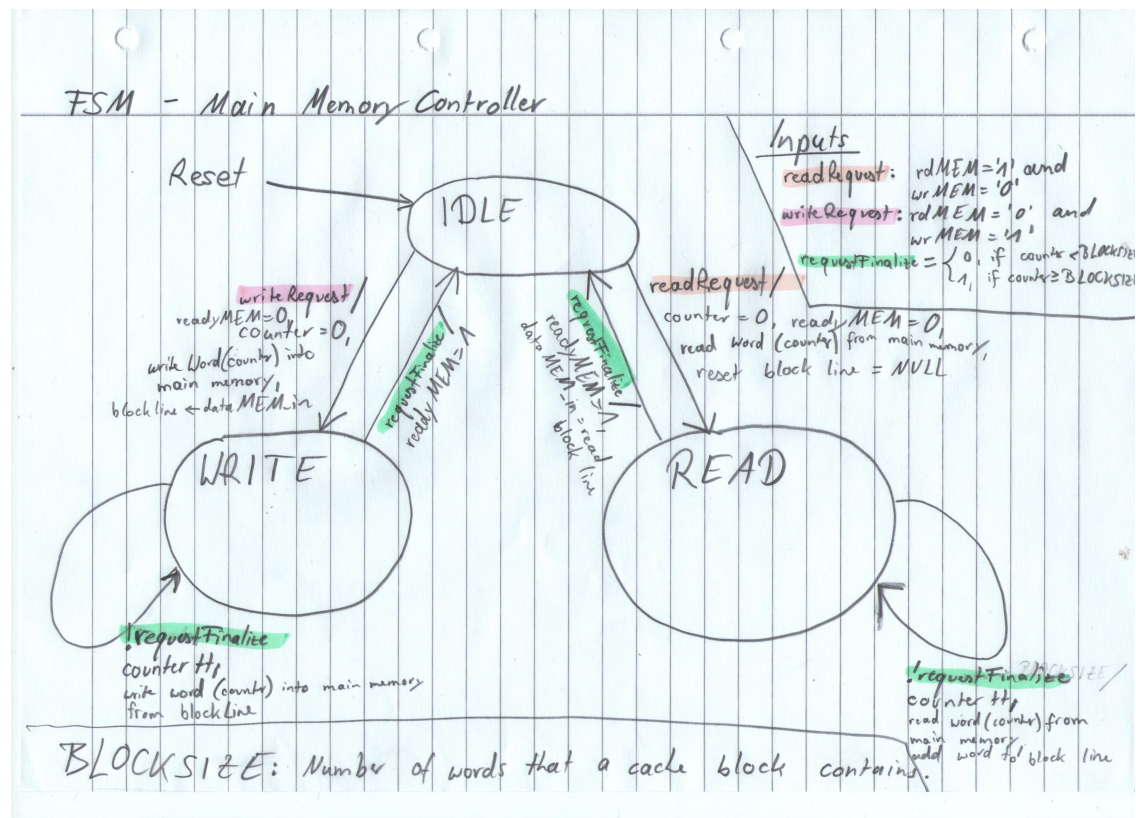


Figure 4: Sketch of Mealy Automata - Main Memory Controller

5.3.1 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.

5.3.2 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.

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
25 # this,
26 #     three statements need to be changed.
27 #     1. The array storage size declaration at "data:" needs to be changed
28 #     from
29 #         256 (which is 16 * 16) to #columns * #rows.
30 #     2. The "li" to initialize $t0 needs to be changed to the new #rows.
31 #     3. The "li" to initialize $t1 needs to be changed to the new #
32 #     columns.
33 #
34 # .data
35 data: .word 0 : 256 # 16x16 matrix of words
36 .text
37 li $t0, 16 # $t0 = number of rows
38 li $t1, 16 # $t1 = number of columns
39 move $s0, $zero # $s0 = row counter
40 move $s1, $zero # $s1 = column counter
41 move $t2, $zero # $t2 = the value to be stored
42 # Each loop iteration will store incremented $t1 value into next element of
43 # matrix.
44 # Offset is calculated at each iteration. offset = 4 * (row*#cols+col)
45 # Note: no attempt is made to optimize runtime performance!
46 loop: mult $s0, $t1 # $s2 = row * #cols (two-instruction
sequence)
mflo $s2 # move multiply result from lo register to
$s2
add $s2, $s2, $s1 # $s2 += col counter
sll $s2, $s2, 2 # $s2 *= 4 (shift left 2 bits) for byte
offset
sw $t2, data($s2) # store the value in matrix element

```

```
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; row < 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
24 # this,
25 #     three statements need to be changed.
26 #     1. The array storage size declaration at "data:" needs to be changed
27 #     from
28 #         256 (which is 16 * 16) to #columns * #rows.
29 #     2. The "li" to initialize $t0 needs to be changed to new #rows.
30 #     3. The "li" to initialize $t1 needs to be changed to new #columns.
31 #
32 .data
33 data: .word 0 : 256      # storage for 16x16 matrix of words
34 .text
35 li $t0, 16              # $t0 = number of rows
36 li $t1, 16              # $t1 = number of columns
37 move $s0, $zero          # $s0 = row counter
38 move $s1, $zero          # $s1 = column counter
39 move $t2, $zero          # $t2 = the value to be stored
40 # Each loop iteration will store incremented $t1 value into next element of
41 # matrix.
42 # Offset is calculated at each iteration. offset = 4 * (row*#cols+col)
43 # Note: no attempt is made to optimize runtime performance!
44 loop: mult $s0, $t1       # $s2 = row * #cols (two-instruction
45      sequence)
46      mflo $s2             # move multiply result from lo register to
47      $s2
48      add $s2, $s2, $s1     # $s2 += column counter
49      sll $s2, $s2, 2       # $s2 *= 4 (shift left 2 bits) for byte
50      offset
51      sw $t2, data($s2)    # store the value in matrix element
52      addi $t2, $t2, 1     # increment value to be stored
53 # Loop control: If we increment past last column, reset column counter and

```

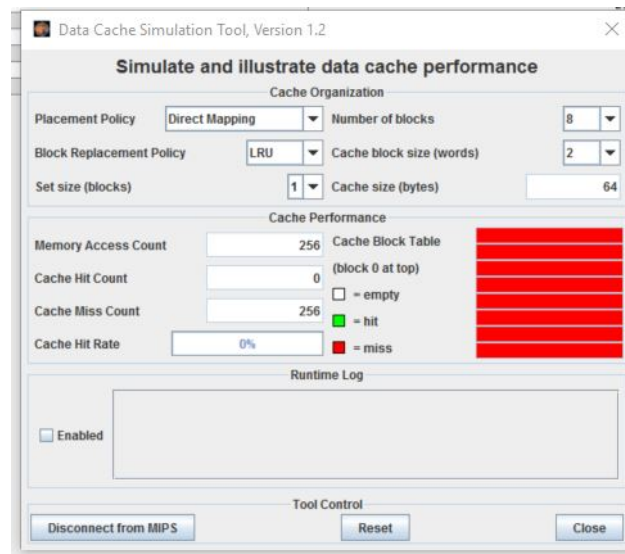



Figure 5: Column Major, Direct Mapping, Cache Block Size 2

```

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

```

Listing 2: row-major.asm

7.2 Cache Results - Snapshots

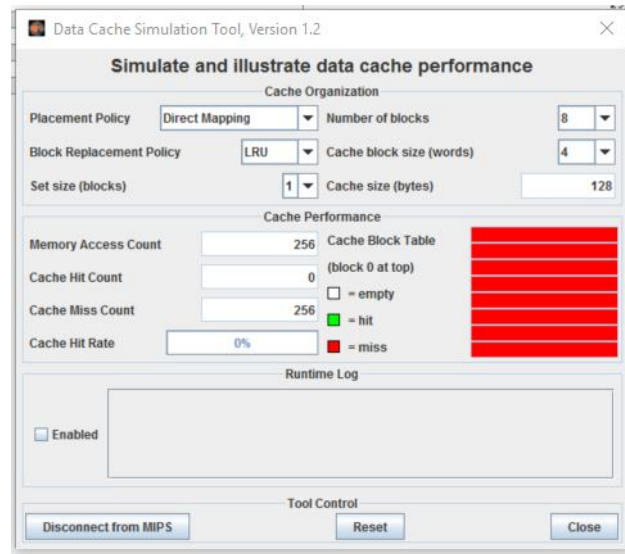


Figure 6: Column Major, Direct Mapping, Cache Block Size 4

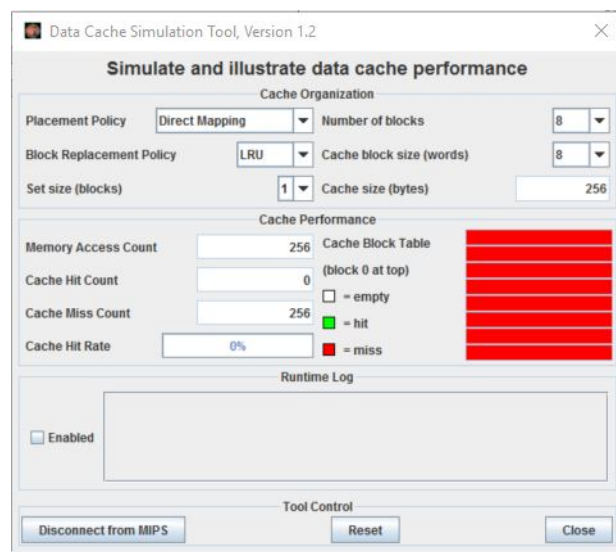


Figure 7: Column Major, Direct Mapping, Cache Block Size 8

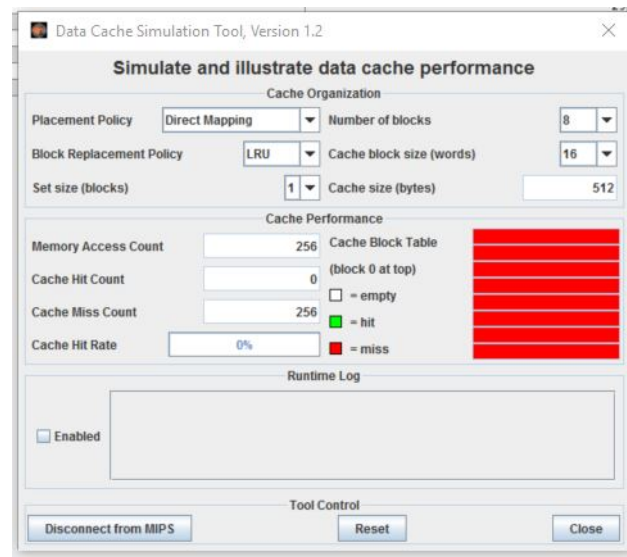


Figure 8: Column Major, Direct Mapping, Cache Block Size 16

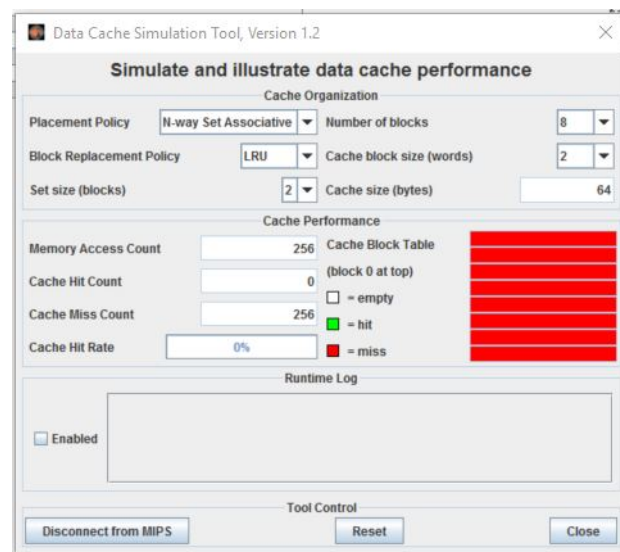


Figure 9: Column Major, 2-Way Associative, Cache Block Size 2

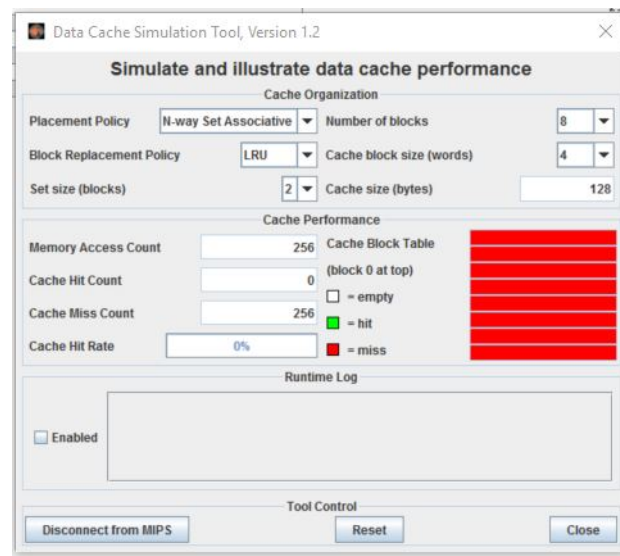


Figure 10: Column Major, 2-Way Associative, Cache Block Size 4

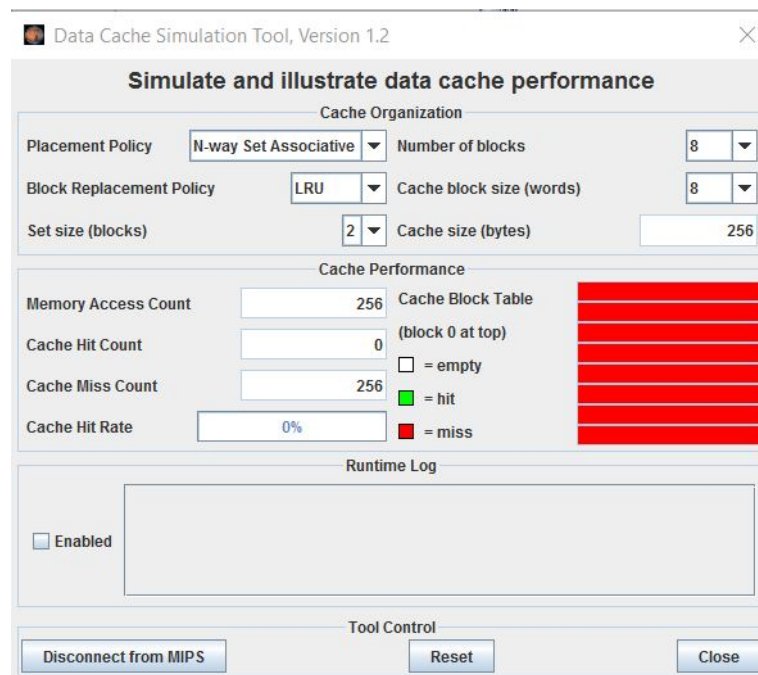


Figure 11: Column Major, 2-Way Associative, Cache Block Size 8

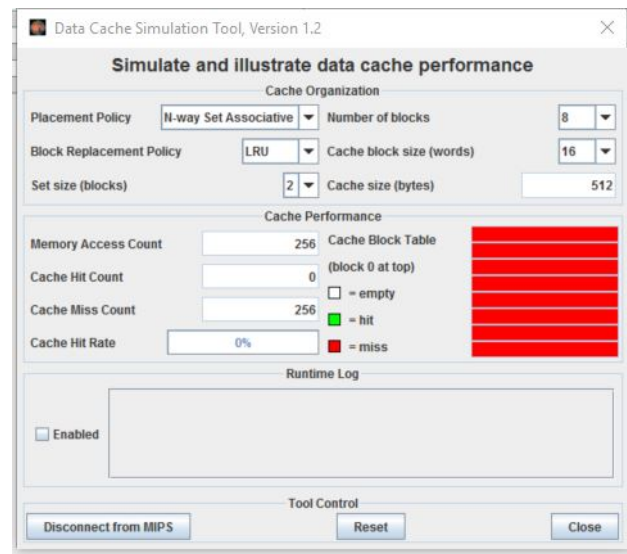


Figure 12: Column Major, 2-Way Associative, Cache Block Size 16

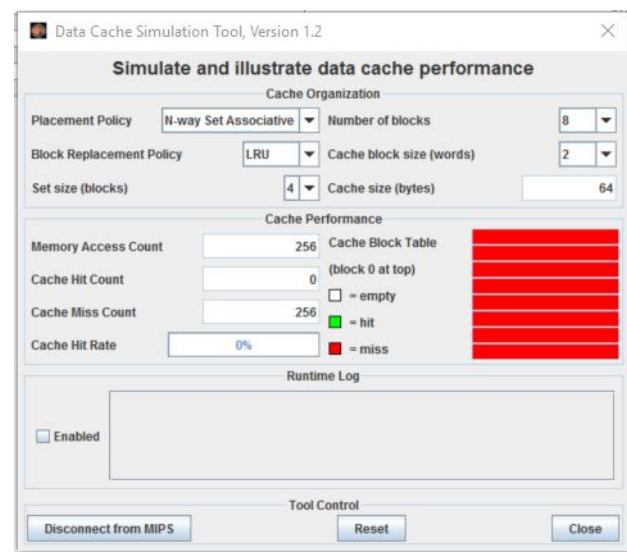


Figure 13: Column Major, 4-Way Associative, Cache Block Size 2

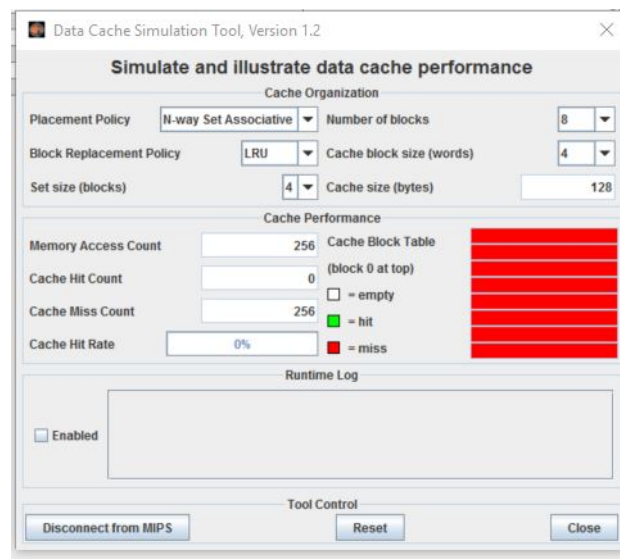


Figure 14: Column Major, 4-Way Associative, Cache Block Size 4

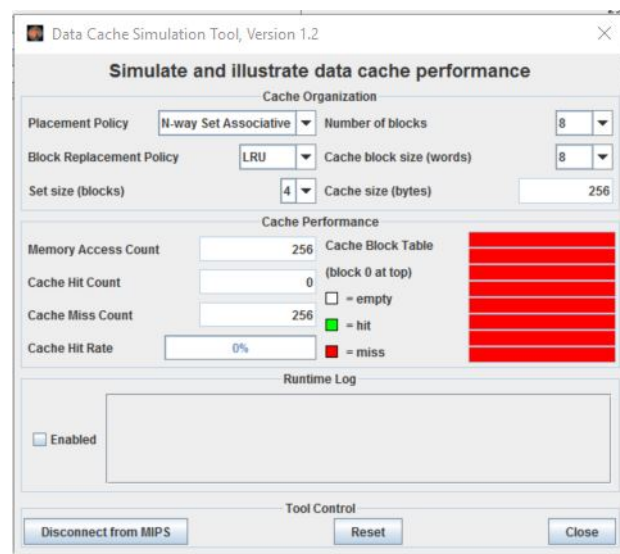


Figure 15: Column Major, 4-Way Associative, Cache Block Size 8

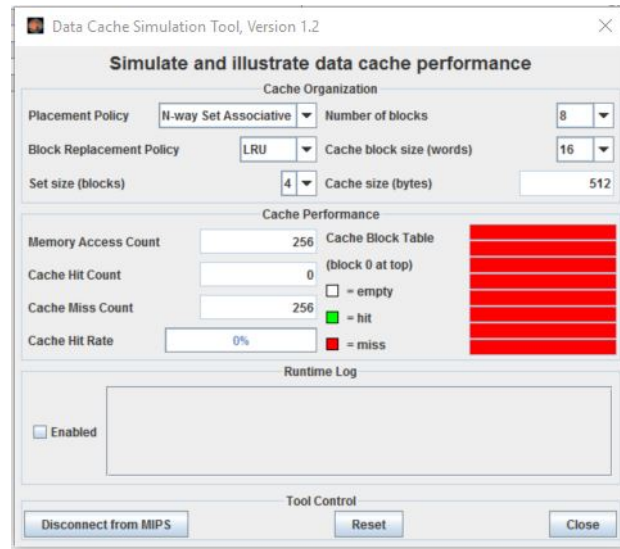


Figure 16: Column Major, 4-Way Associative, Cache Block Size 16

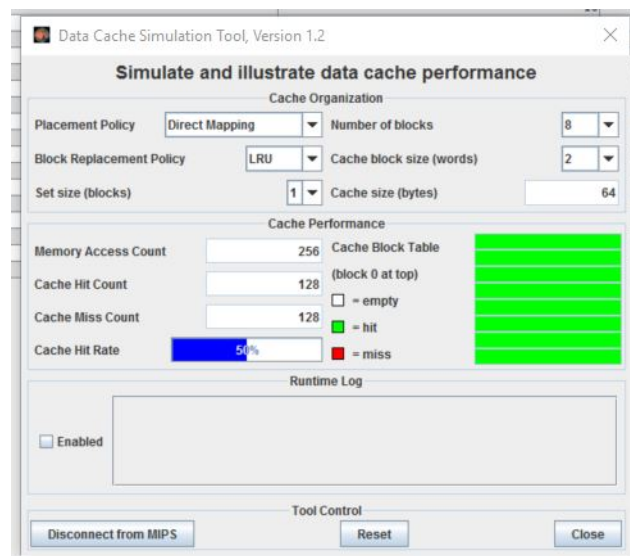


Figure 17: Row Major, Direct Mapping, Cache Block Size 2

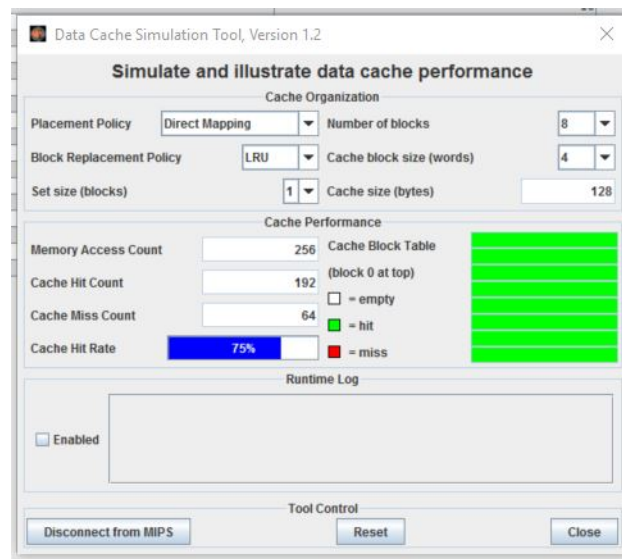


Figure 18: Row Major, Direct Mapping, Cache Block Size 4

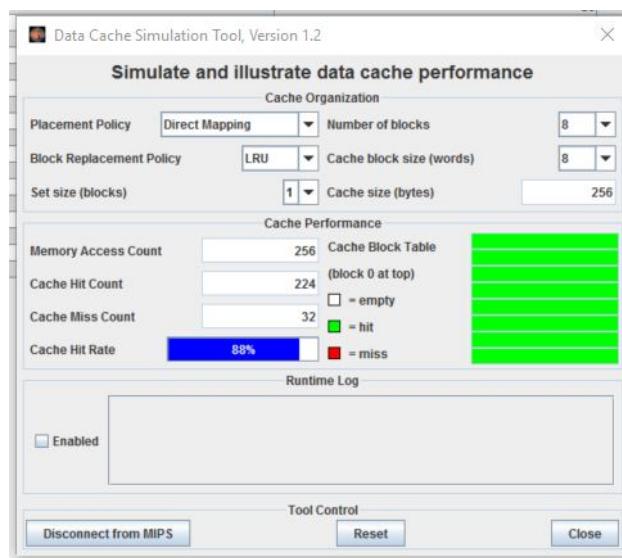


Figure 19: Row Major, Direct Mapping, Cache Block Size 8

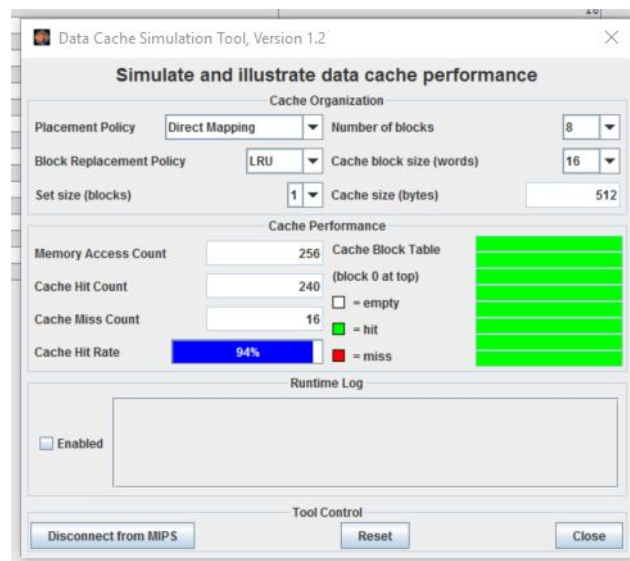


Figure 20: Row Major, Direct Mapping, Cache Block Size 16

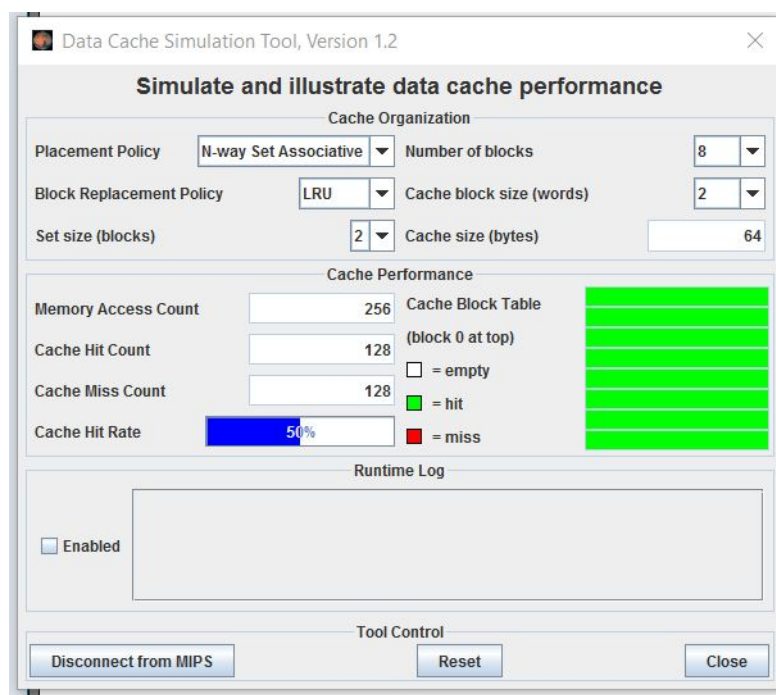


Figure 21: Row Major, 2-Way Associative, Cache Block Size 2

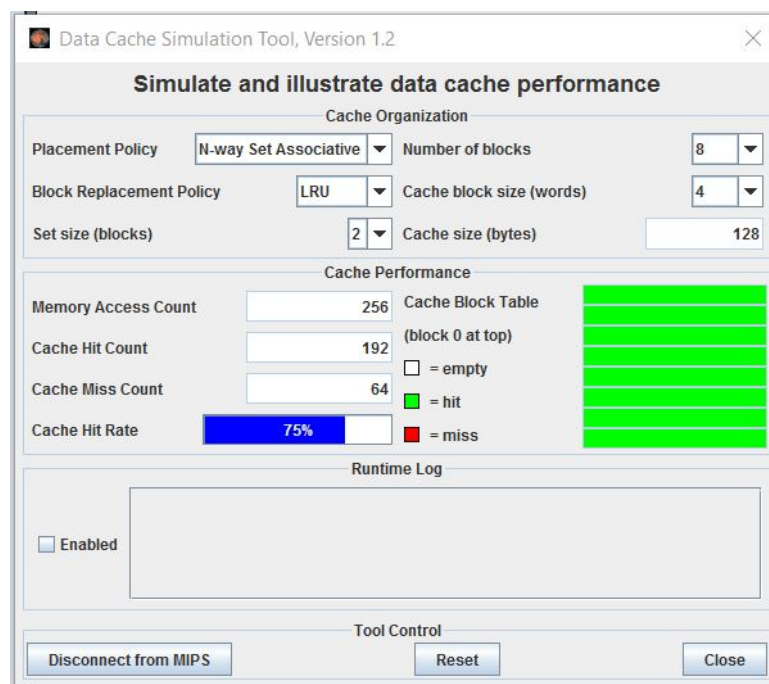


Figure 22: Row Major, 2-Way Associative, Cache Block Size 4

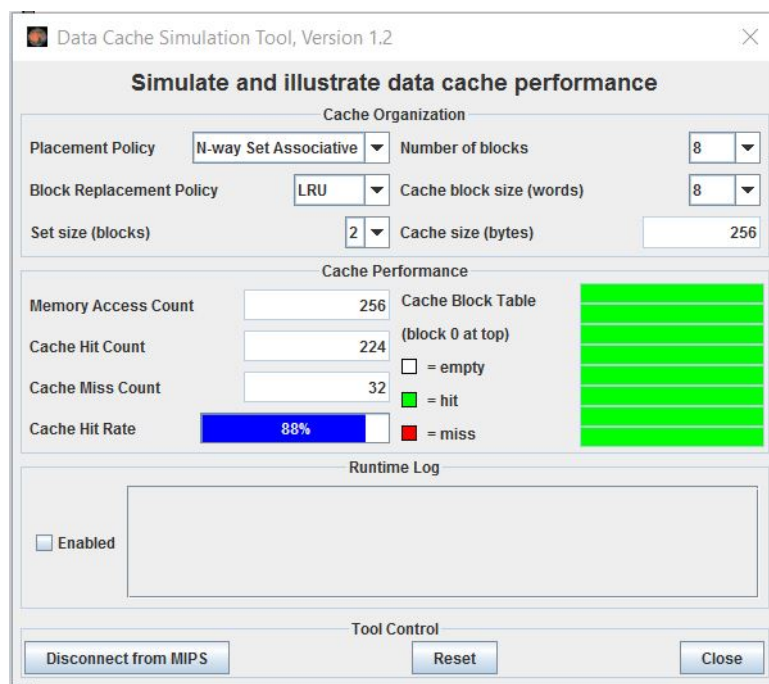


Figure 23: Row Major, 2-Way Associative, Cache Block Size 8

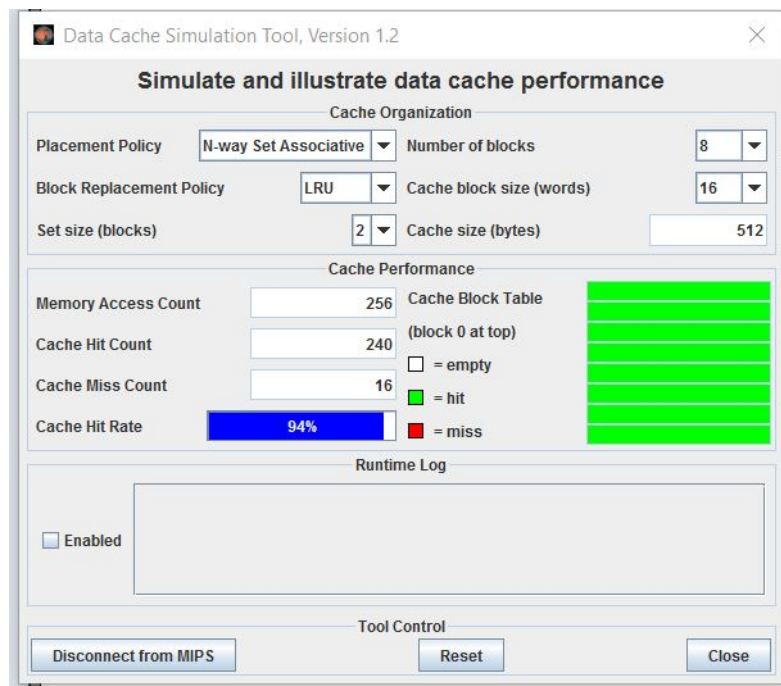


Figure 24: Row Major, 2-Way Associative, Cache Block Size 16

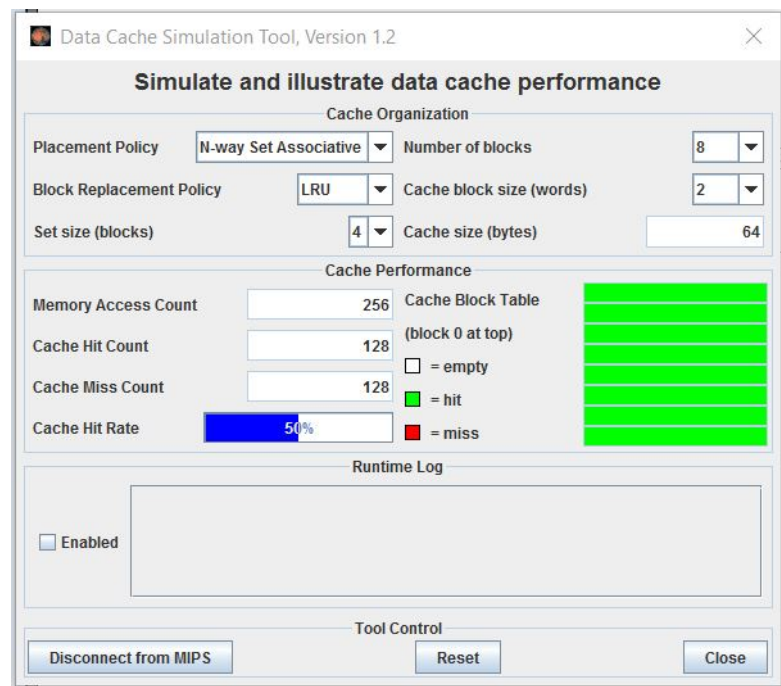


Figure 25: Row Major, 4-Way Associative, Cache Block Size 2

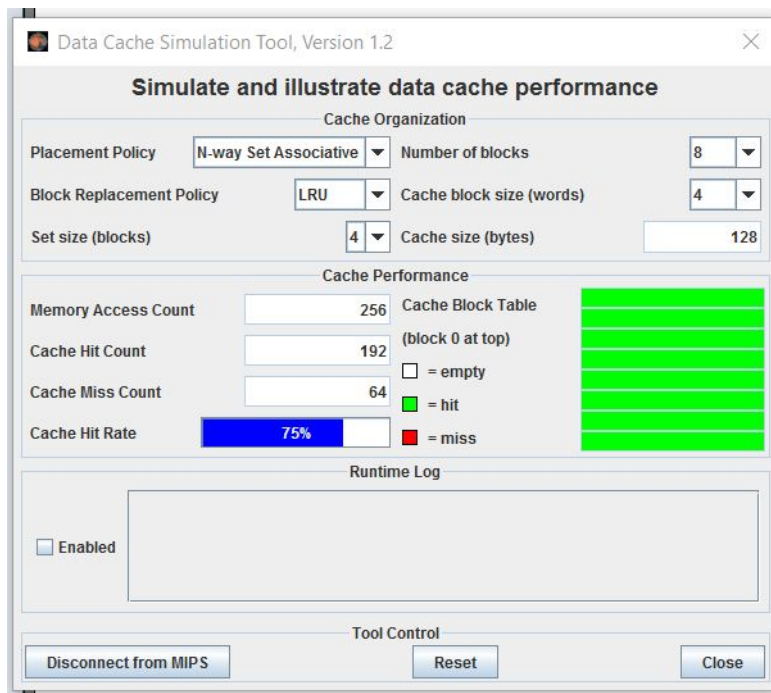


Figure 26: Row Major, 4-Way Associative, Cache Block Size 4

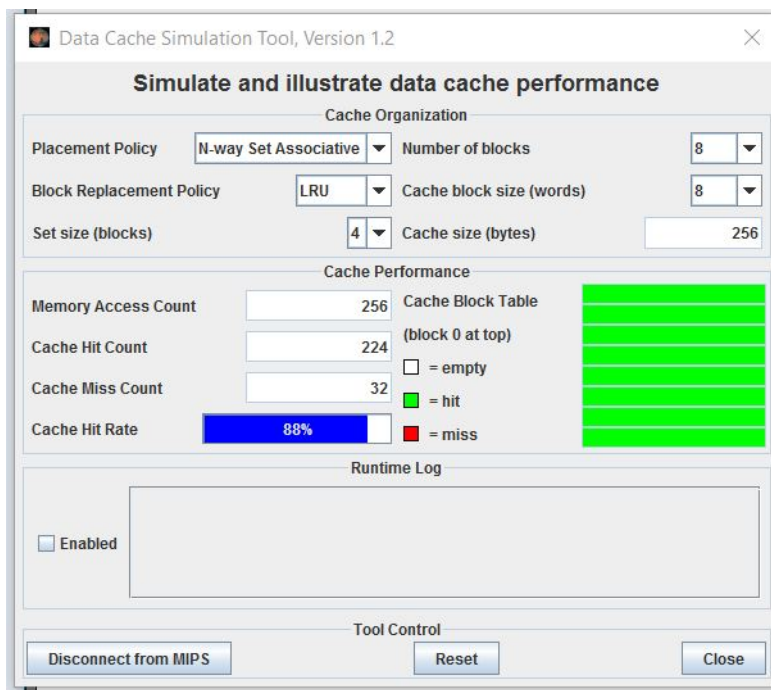


Figure 27: Row Major, 4-Way Associative, Cache Block Size 8

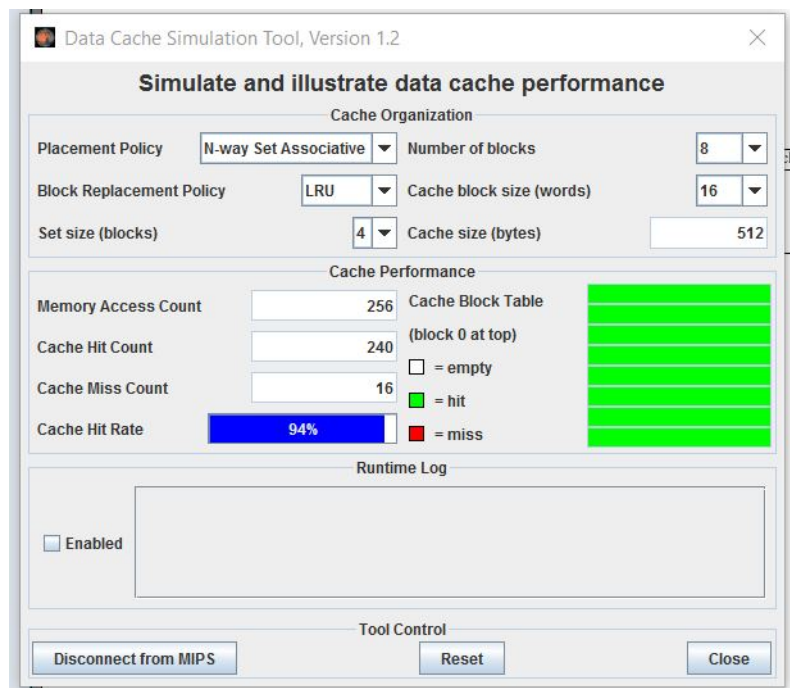


Figure 28: Row Major, 4-Way Associative, Cache Block Size 16