



HAMBURG UNIVERSITY OF TECHNOLOGY

PROBLEM-BASED LEARNING

Advanced System-on-Chip Design

author
@tuhh.de

Report

Tutor
Dipl.-Ing. Wolfgang BRANDT

March 24, 2017

Contents

1	Introduction	5
2	Task 3 - MIPS Extension	6
2.1	Introduction to the MIPS Enhancements	6
2.1.1	Activating Pipelining	6
2.1.2	Installing Forwarding	7
2.1.3	Implementing Stalling	8
2.2	Activating BRAM	8
3	Task 4 - Caches	9
3.1	Introduction to Memories	9
3.2	Cache Simulation - Results	10
3.3	Design a direct mapped cache	12
3.4	Design a Finite State Machine for the Cache	13
3.4.1	Design a Finite State Machine for the Main Memory Controller	18
3.5	Design a testbench and simulate the Cache	19
4	Task 5 - Branch Prediction	22
5	Summary	23
6	Appendix	24
6.1	Implementation	24
6.2	Cache Results - Snapshots	28

List of Figures

1	FEC Slides, Diagram of Forwarding Logic	7
2	Entity of directMappedCache	12
3	State diagram of the cache controller.	15
4	Sketch of Mealy Automata - Cache Controller, Version 2	17
5	Sketch of Mealy Automata - Main Memory Controller	19
6	Result Simulation - Cache	20
7	Result Simulation Gtkwave - Cache Hit	22
8	Column Major, Direct Mapping, Cache Block Size 2	28
9	Column Major, Direct Mapping, Cache Block Size 4	29
10	Column Major, Direct Mapping, Cache Block Size 8	29
11	Column Major, Direct Mapping, Cache Block Size 16	30
12	Column Major, 2-Way Associative, Cache Block Size 2	30
13	Column Major, 2-Way Associative, Cache Block Size 4	31
14	Column Major, 2-Way Associative, Cache Block Size 8	31
15	Column Major, 2-Way Associative, Cache Block Size 16	32
16	Column Major, 4-Way Associative, Cache Block Size 2	32
17	Column Major, 4-Way Associative, Cache Block Size 4	33
18	Column Major, 4-Way Associative, Cache Block Size 8	33
19	Column Major, 4-Way Associative, Cache Block Size 16	34
20	Row Major, Direct Mapping, Cache Block Size 2	34
21	Row Major, Direct Mapping, Cache Block Size 4	35
22	Row Major, Direct Mapping, Cache Block Size 8	35
23	Row Major, Direct Mapping, Cache Block Size 16	36
24	Row Major, 2-Way Associative, Cache Block Size 2	36
25	Row Major, 2-Way Associative, Cache Block Size 4	37
26	Row Major, 2-Way Associative, Cache Block Size 8	37
27	Row Major, 2-Way Associative, Cache Block Size 16	38
28	Row Major, 4-Way Associative, Cache Block Size 2	38
29	Row Major, 4-Way Associative, Cache Block Size 4	39
30	Row Major, 4-Way Associative, Cache Block Size 8	39
31	Row Major, 4-Way Associative, Cache Block Size 16	40

List of Tables

1	Cache Simulation of Column Major	11
2	Cache Simulation of Row Major	11
3	Overview - FSM Inputs	16
4	Overview - FSM Outputs	16
5	Test Cases for Simulation	21

Listings

1	Change1_WBAtFallingEdge.txt	6
2	Change2_activatingPipelineRegisters.txt	6
3	Change3_hazardDetectionForwardLogic.txt	7
4	column-major.asm	25
5	row-major.asm	27

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 3 - MIPS Extension

2.1 Introduction to the MIPS Enhancements

The aim of this exercise was to add multiple functionalities to a given CPU called „MIPS“. In the beginning there was no pipelining activated and several adjustments had to be made. In this chapter each of the enhancements are introduced in principle and afterwards its concrete implementation is explained in detail.

For each enhancement a corresponding assemble instruction file was adapted and its execution was carefully observed. Only one enhancement at a time was changed at a time to reduce the space of errors that need to be handled at once.

2.1.1 Activating Pipelining

The first wanted improvement of the mips was the activation of pipelining. The following adjustments were made:

Exercise 3-3.3 gave a hint about the writing of data back into a register at a negative clock edge “The register file must be read in decode stage and written in write back stage within a single cycle.” TODO Footnote

```
1      _____ MA/WB Pipeline Register —  
2 from   WB  <= (MA.c , MA.wa, MA.pc4 , aout) when rising_edge(clk);  
3 to     WB  <= (MA.c , MA.wa, MA.pc4 , aout) when falling_edge(clk);
```

Listing 1: WBatFallingEdge.txt

Afterwards conveniently predefined registers which could save and pass on memory instructions at each phase of the pipeline could be inserted by changing the code belonging to each of the registers at the ID (Instruction / Decode) phase, at the EX (Execution) phase, the MA (Memory Access) phase, as well as the WB (Write Back) phase.

```
1 from :  
2 ID  <= (IF_ir , pc4);                                — when rising_edge(clk);  
3 EX  <= (c , i , wa , a , b , signext , ID.pc4 , rd2); — when rising_edge(clk);  
4 MA  <= (EX.c , EX.i , EX.wa, EX.a, EX.imm, EX.pc4 , EX.rd2 ,  
5      pcbranch , pcjump , aluout , zero , lez , ltz , gtz);  
6                               — when rising_edge(clk);  
7 WB  <= (MA.c , MA.wa, MA.pc4 , aout);              — when rising_edge(clk);  
8  
9 to :  
10 ID <= (IF_ir , pc4)                                when rising_edge(clk);  
11 EX <= (c , i , wa , a , b , signext , ID.pc4 , rd2) when rising_edge(clk);  
12 MA <= (EX.c , EX.i , EX.wa, EX.a, EX.imm, EX.pc4 , EX.rd2 ,  
13      pcbranch , pcjump , aluout , zero , lez , ltz , gtz)  
14                                         when rising_edge(clk);  
15 WB <= (MA.c , MA.wa, MA.pc4 , aout)                when rising_edge(clk);
```

Listing 2: ActivatingPipelineRegisters.txt

In order to make sure that no mistakes occurred and all steps were taken according to the modification plan precise testing had to take place. Small simple programs were executed and

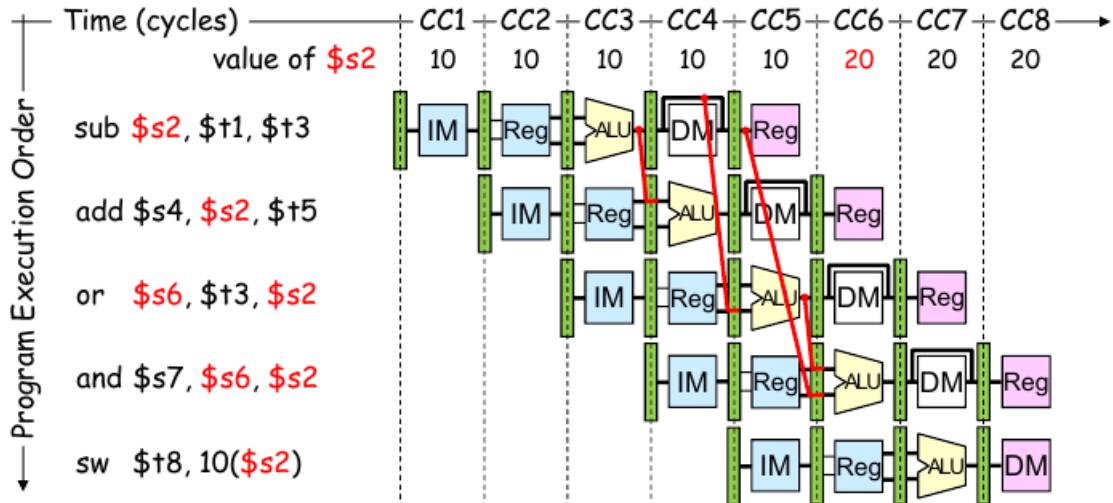


Figure 1: FEC Slides, Diagram of Forwading Logic

examined using GTKWAVE, a visualization tool for vhdl code executions. First, an assembler program which was perfectly executed by the mips before the adaptions was modified with additional nop commands after each execution command in the code. In fact we needed exactly three nop commands because we were using a 4 stage pipeline and the longest number of clock cycles that one command needed to wait was three clock cycles at this point. The reason is we wanted to mitigate data and control hazards and needed to make sure that the registers were running as expected without taking care of the issues of forwarding and stalling yet.

Of course it was not the aim to add three nop operations after each instruction every time. But in the beginning this was the approach needed. Later on the goal was to let the mips add nops wherever needed automatically by sending a stalling signal.

2.1.2 Installing Forwarding

After making sure that pipelining was running the next step at hand was to install a forwarding logic which implements the following behaviour:

```

1      _____ Hazard Detection and Forward Logic _____
2  ForwardA <=
3  fromALUE when ( i.Rs /= "00000" and i.Rs = EX.wa and EX.c.regwr = '1' ) else
4  fromALUM when ( i.Rs /= "00000" and i.Rs = MA.wa and MA.c.regwr = '1' ) else
5  fromMEM when ( i.Rs /= "00000" and i.Rs = WB.wa and WB.c.regwr = '1' ) else
6  fromReg;
7  ForwardB <=
8  fromALUE when ( i.Rt /= "00000" and i.Rt = EX.wa and EX.c.regwr = '1' ) else
9  fromALUM when ( i.Rt /= "00000" and i.Rt = MA.wa and MA.c.regwr = '1' ) else
10 fromMEM when ( i.Rt /= "00000" and i.Rt = WB.wa and WB.c.regwr = '1' ) else
11 fromReg;
```

Listing 3: HazardDetectionForwardLogic.txt

The aim was to let instructions that were using registers that were written to in one instruction step earlier use the calculated information directly out of the ALU instead of having to wait until the calculation was written back into a register at WB phase. The same procedure applies to instructions that needed a calculated result 2 or 3 clock cycles later.

2.1.3 Implementing Stalling

We needed to implement the functionality of the processor to „freeze“ certain processing steps. If an access of a cache or a command like „load word“ takes more than one clock cycle then consequent instructions needed to wait for the corresponding time delay. Our implementation basically uses a new signal called „TODO Insert StallingSignal name which is to be called whenever such a delay occurs. It has three consequences:

The PC (Program Counter) is kept the same. As long as the stalling signal is set to one no new commands will be passed on to the instruction registers of the EX phase. If a stalling signal occurs the instruction registers in EX phase are overwritten with a nop command. This nop will be passed on to next instruction registers like MA and WB.

```

1      _____ ID/EX Pipeline Register with Multiplexer Stalling —
2 from :EX    <= (c, i, wa, a, b, signext, ID.pc4, rd2) when rising_edge(clk);
3 to : EX     <= Bubble when Stall_disablePC = '1' and rising_edge(clk) else
4             (c, i, wa, a, b, signext, ID.pc4, rd2) when rising_edge(clk);
```

Listing 4: BubbleInsertionExPhase.txt

The stalling signal will remain activated until the command which called the stalling signal reaches the EX, MA, or WB phase. The phase in which the signal is deactivated again depends on the command. Load word for instance only needs a delay of one stalling command. Whereas other commands like conditional branches where the result of the instruction has an impact on the stalling time need more cycles.

Before finding this optimal solution a dead one was chosen before. We overengineered a bit by inserting a counter which simple counted down from 3 to 0. Until it was clear that the pipeline itself could be used as a counter to set the stalling signal back to one. Our lesson was: Program as few things as possible, use all the available things that you can!

Again at this point we wrote small programs that activated behaviour that needed to be checked and varied the numbers of auxiliary nops inbetween critical instructions until the system was running quite well.

2.2 Activating BRAM

The next big task was to use BRAM (Block RAM) instead of DRAM (Distributed RAM). DRAM was more suitable as memory for the single cycle CPU, because BRAM has clocked outputs. Using BRAM introduces another pipelining step and would have been a disadvantage for a single cycle CPU. But in our case, with pipelining activated, can be used in an efficient manner. If the mips were running on a FPGA then of course BRAM has the advantage that normally more resources are available.

The DRAM was turned into BRAM by commenting out the corresponding rising_edge(clk) and falling_edge(clk) commands in the bram.vhd file and started working on the issues of having to deal with an additional pipelining step. It turned out that all commands were read into the IF/ID phase one clock cycle later. This behaviour is shown in figure ???. The code that was executed was consisting of three nop operations followed by a JAL command. The PC of the 4th command, JAL, is 12. The three nops have PCs 0, 4 and 8. The upper red marking demonstrates the delay of 2 clock cycles after the PC was set to the value of 12. Before within the single cycle mips without BRAM it only took one clock cycle for an instruction to appear in the ID phase. The old delay is represented by the green marking below the first one.



Figure 2: Delay of PC instruction after 3 nops one JAL command

Our control logic has to be adapted to the fact that now 4 nops had to be inserted instead of just three and also the program counter had to be frozen accordingly. The PC freezing logic had to receive the instruction commands as early as possible. Therefore the logic needed to check the opcode of the instruction even before it was written into the first ID register. This way the system could understand when the program counter needs to be kept the same. Inserting nops after a branch or jump command occurred in the ID register was not that much of a deal. The only big adaption was the addition of one stalling cycle more. Before the functionality basically was: If a jump or branch command occurs, stall until it reoccurs in MA phase, then set it back to '0' again. Now we needed one more stalling cycle and needed to let the stalling signal be set to '0' if the branch or jump command occurred in the WB phase. Unfortunately the control instruction registers were not passed onto the WB phase from the MA phase. Therefore we had to adjust the system and add additional instruction forwarding in the mips_pkg.vhd file.

TODO insert Mips pkg instruction forward information.

```

1 nextpc <=
2 MA.pcjump    when MA.c.jump  = '1' else --- j / jal jump addr
3 MA.pcbranch   when branch     = '1' else --- branch (bne , beq) addr
4 MA.a          when MA.c.jr   = '1' else --- jr addr
5 --- The conditions below freeze the PC
6 pc when (IF_ir(31 downto 26) = "100011")   else ---LW
7 pc when (IF_ir(31 downto 26) = "000011") or (i.mnem = JAL) or (EX.i.mnem =
      JAL) or (MA.i.mnem = JAL) else ---JAL
8 pc when (IF_ir(31 downto 26) = "000101") or (i.mnem = BNE) or (EX.i.mnem =
      BNE) or (MA.i.mnem = BNE) else ---BNE
9 pc when (IF_ir(31 downto 26) = "000100") or (i.mnem = BEQ) or (EX.i.mnem =
      BEQ) or (MA.i.mnem = BEQ) else ---BEQ
10 pc when (IF_ir(31 downto 26) = "000010") or (i.mnem = J)   or (EX.i.mnem = J)
      or (MA.i.mnem = J) else ---J
11 pc when ((IF_ir(5 downto 0) = "001000") and (IF_ir(31 downto 26) = "000000"
      )) or

```

```
12 | ( i .mnem = JR ) or (EX. i .mnem = JR) or (MA. i .mnem = JR) else --JR  
13 | pc4 ; -- standard case: pc + 4, take following instruction;
```

Listing 5: FreezingPC.txt

2.3 Additional Details

2.4 Lessons Learned so Far

Besides the normal lessons like

- “keeping code as simple as possible”
- “commenting code pays off”
- “divide and conquer problems”
- “do testing, testing and testing”

another very important lesson was to take into consideration that it takes more time to get things running if team members are using different versions of an operating system or even different operating systems themselves. For instance we had to adapt scripts for windows and linux computers and during some time had different simulation results on our computers, which later on dissolved after cleaning up temporary files and folders. Still this was quite annoying and took time to dissolve. Reducing the realm of possible error can be a big advantage.

3 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.

At first we give a short introduction to memories 3.1. After that, we simulate the efficiency of a cache with MARS in subsection 3.2 and compare different kinds of cache organizations. Afterwards, we design and implement a direct mapped cache in subsection 3.3. This direct mapped cache will be used to develop a complete cache used as an instruction cache. Therefore, we design a finite state machine representing the behavior of this cache in 3.4 and finally we write a testbench to verify the implementation in 3.5.

3.1 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 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 need to write back to the main memory in case of a cache miss. Furthermore, the strategy Write-through is easily to implemented.

3.2 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 placement 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 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 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

3.3 Design a direct mapped cache

To develop an instruction cache and a data cache, we first implement a direct mapped cache. Thus, in figure 3.3 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 *wrCBLLine* and *rdCBLLine* 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 *wrNewCBLLine* 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.

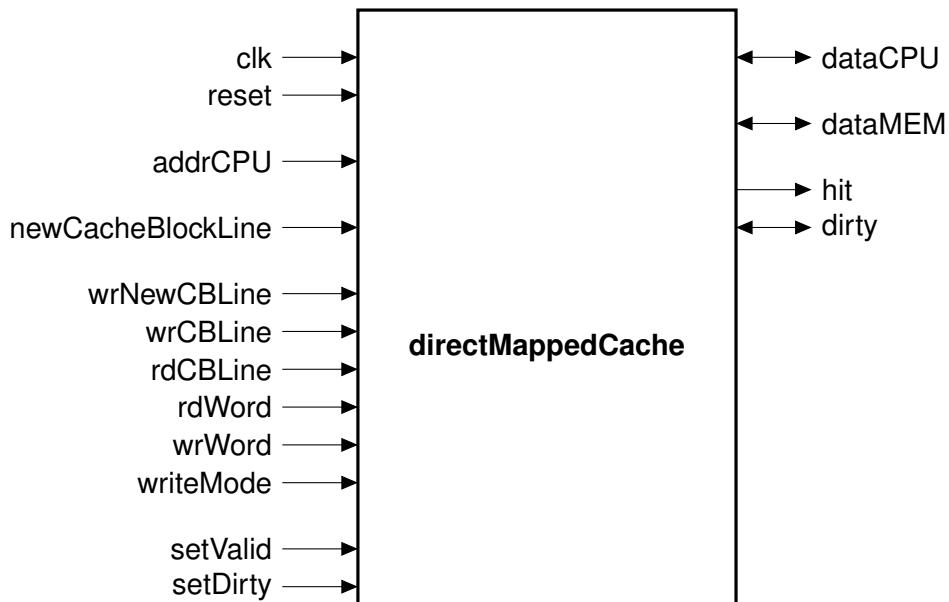


Figure 3: Entity of directMappedCache

3.4 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 3 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 4.

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 4: State diagram of the cache controller.

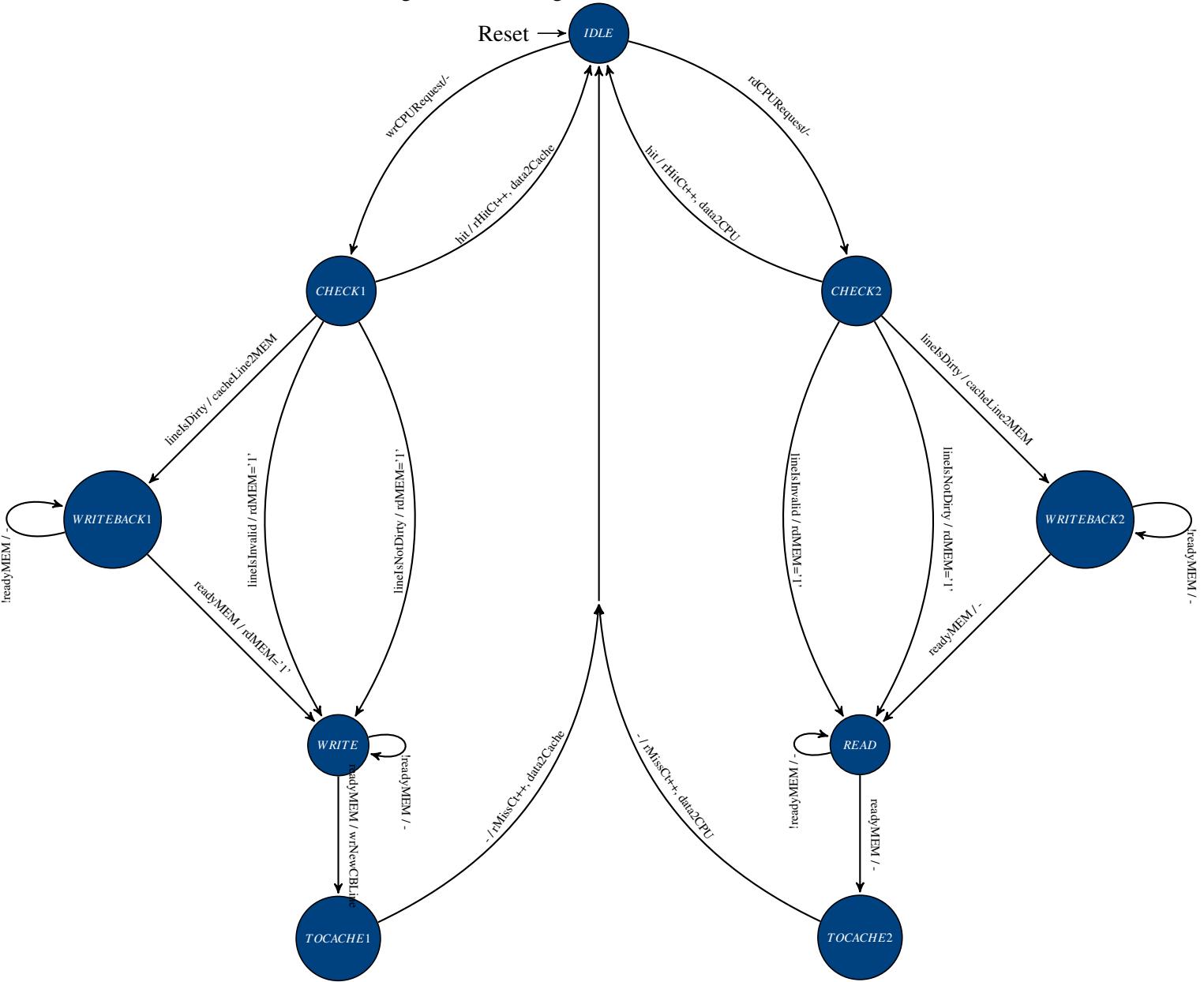
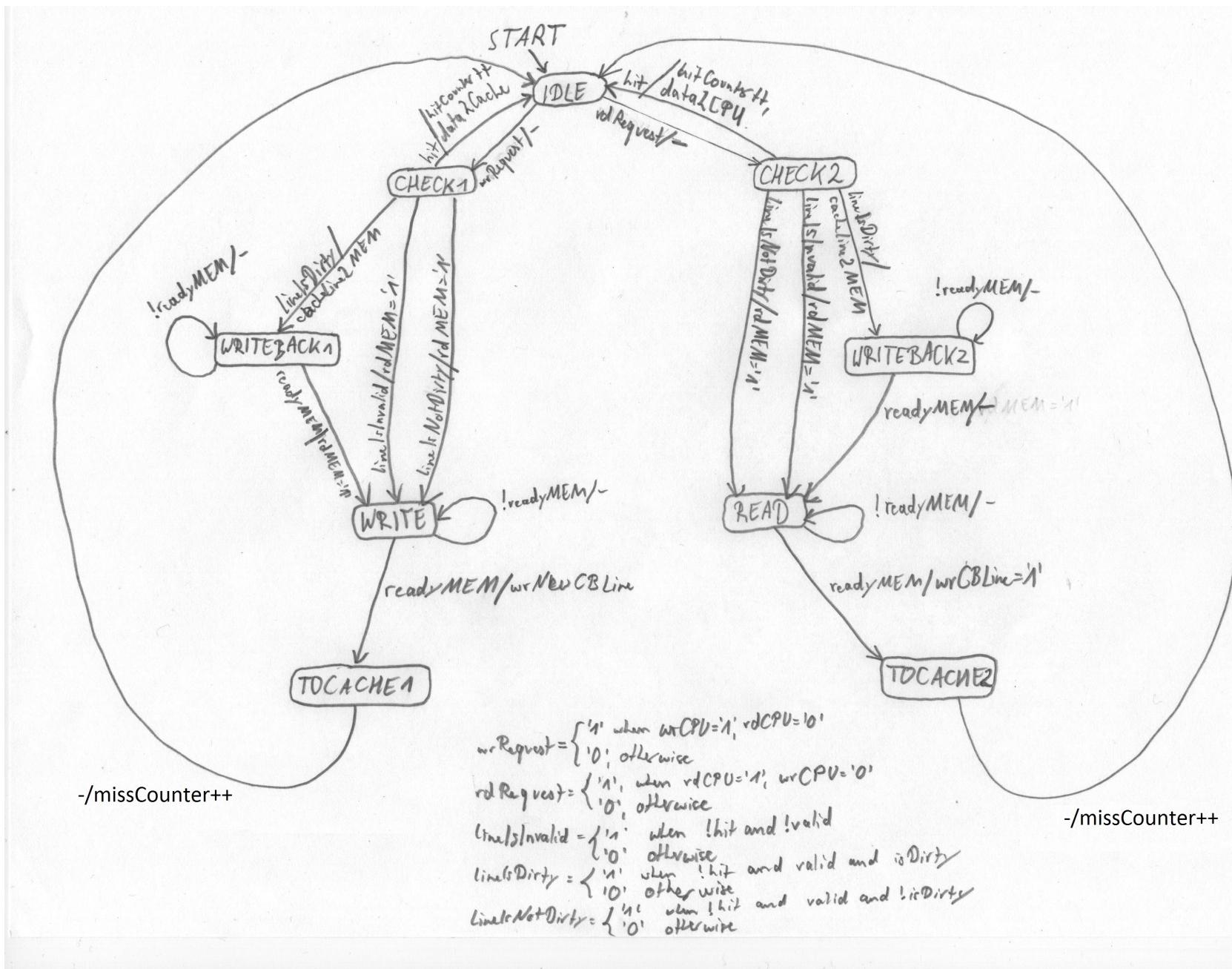


Table 3: Overview - FSM Inputs

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

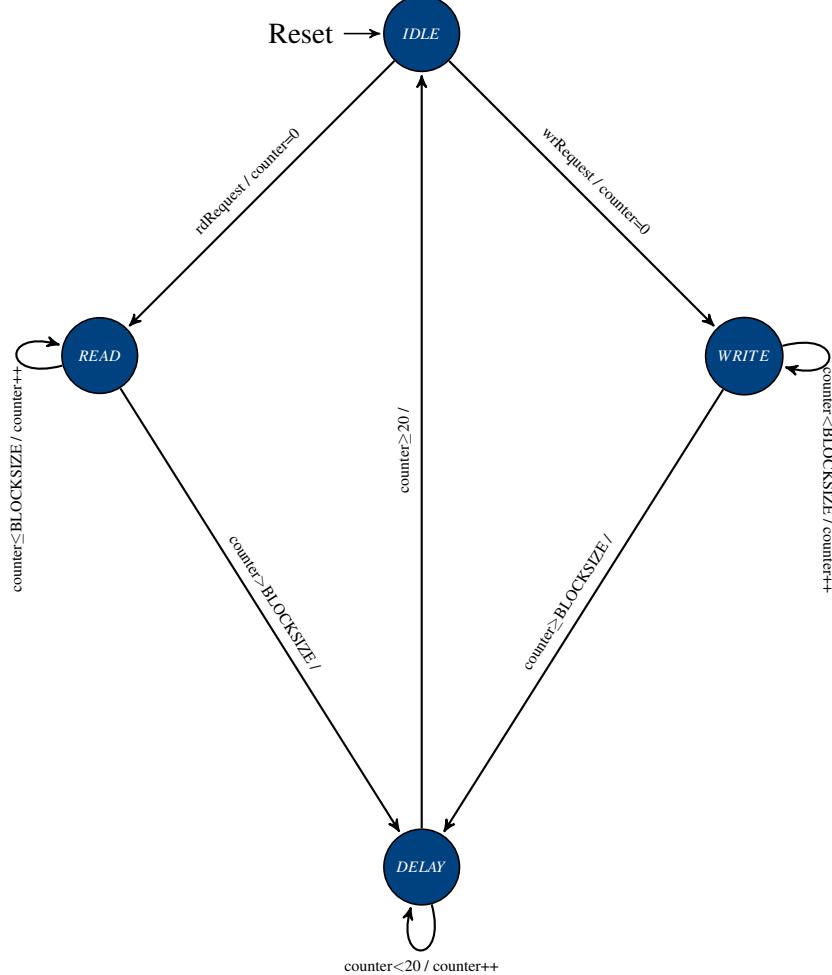
Abbreviation	Name	Description
stallCPU	Stall Processor	-
setDirty	Set Dirty Bit (Modified) Bit	-
wrMEM	Write To Memory	Write Replaced Block To Memory
dataCPU	Read Data Into CPU	-
rdMEM	Read Cache Block Into Cache From Memory	-
dataCPU2Cache	Write Data Into Cache	-



3.4.1 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 width 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 5.

Figure 6: Sketch of Mealy Automata - Main Memory Controller



3.5 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 table 5 we describe some test cases.

We have created the batch file *test_cacheTestbench.bat* to run the testbench with GHDL. Thus, we start the simulation by typing *test_cacheTestbench.bat* in the command-line user interface. Figure 6 shows that all test cases are successfully executed.

```

++++++ run the executable for 15us and save all waveforms ++++++
..\lisee\numeric_std-body.v93:2998:7:@0ms:(assertion warning): NUMERIC_STD_TO_INTEGER: metavalue detected, returning 0
..\lisee\numeric_std-body.v93:2998:7:@0ms:(assertion warning): NUMERIC_STD_TO_INTEGER: metavalue detected, returning 0
cache_tb.vhd:271:16:@10ns:(report note): Test 1 start validation...
cache_tb.vhd:276:16:@10ns:(report note): Test 1 successfully validated.
cache_tb.vhd:277:16:@10ns:(report note): -----
cache_tb.vhd:277:16:@10ns:(report note): Test 2 start validation...
cache_tb.vhd:276:16:@10ns:(report note): Test 2 successfully validated.
cache_tb.vhd:277:16:@5130ns:(report note): -----
cache_tb.vhd:271:16:@5134ns:(report note): Test 3 start validation...
cache_tb.vhd:276:16:@5134ns:(report note): Test 3 successfully validated.
cache_tb.vhd:277:16:@5134ns:(report note): -----
cache_tb.vhd:271:16:@21324ns:(report note): Test 4 start validation...
cache_tb.vhd:276:16:@21324ns:(report note): Test 4 successfully validated.
cache_tb.vhd:277:16:@21516ns:(report note): -----
cache_tb.vhd:271:16:@21516ns:(report note): Test 5 start validation...
cache_tb.vhd:522:16:@26640ns:(report note): change data in cache blocks...
cache_tb.vhd:544:16:@28688ns:(report note): read data from cache blocks with different tags...
cache_tb.vhd:276:16:@34836ns:(report note): Test 5 successfully validated.
cache_tb.vhd:277:16:@34836ns:(report note): -----
cache_tb.vhd:271:16:@34836ns:(report note): Test 6 start validation...
cache_tb.vhd:276:16:@39966ns:(report note): Test 6 successfully validated.
cache_tb.vhd:277:16:@39966ns:(report note): -----
cache_tb.vhd:271:16:@47144ns:(report note): Test 9 start validation...
cache_tb.vhd:276:16:@47144ns:(report note): Test 9 successfully validated.
cache_tb.vhd:277:16:@47144ns:(report note): -----
cache_tb.vhd:271:16:@54322ns:(report note): Test 10 start validation...
cache_tb.vhd:276:16:@54322ns:(report note): Test 10 successfully validated.
cache_tb.vhd:277:16:@54322ns:(report note): -----

```

Figure 7: Result Simulation - Cache

When illustrating the simulation results in Gtkwave, the number of clock cycles needed for a cache hit and cache miss can be determined. As you can see in figure 7, circa 6 clock cycles are needed for a cache hit.

Table 5: Test Cases for Simulation

Test Case 1 Reset Cache I

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

Test Case 2 Reset Cache II

If the cache is reset, then all cache blocks lines are invalid.

Test Case 3 Read Cache, Line Is Not Dirty

Assume, that there are already valid data in the cache block lines. Also, the data are not modified against the main memory. Now we read again with different tag values. Thus, the data are directly read from main memory into the cache. We expect that the miss counter is incremented and the stall signal is set to one.

Test Case 4 Read Cache, Different Offset

In the first step, we are going to read the first word from a cache block line. Following we read another word from the same cache block line. Thus, the miss counter will be incremented.

Test Case 5 Read Cache, Line is Dirty

There are already valid data words in the cache block line. Also, the data words are modified compared to the main memory. Now, we are going to read again from cache while the tag values are differen. Thus, we expect that the modified data words are written back to the main memory first. Afterwards the block is read from main memory into the cache. Hence, the miss counter will be incremented.

Test Case 6 Write Cache, Invalid Cacheblocks

Initially all cache blocks are invalid. If a cache block line is read, then the equivalent block is read from the main memory to the cache. Appropriate the miss counter will be incremented and the stall signal is set to '1'.

Test Case 7 Write Cache – Line is Dirty

Assume, that there are already valid data words in the cache block line. The data words are modified compared to the main memory. We are going to write again into the cache block lines whereupon the tag values are different. Hence, the data from the cache are written back to the main memory first. After that the correspondent main memory block is load into the cache with the new written data given from CPU. We expect that the miss counter is incremented.

Test Case 8 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.

Test Case 9 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.

Test Case 10 Write Cache - Check Values

In this test case we will check whether the new data word has been successfully written into the cache. Whatever the current status of the cache block/line is, we write new data into cache in the first step. After we have finished writing the cache, we can read the cache block line again. We expect, that we read the equal data from the cache, which we have written into the cache before.

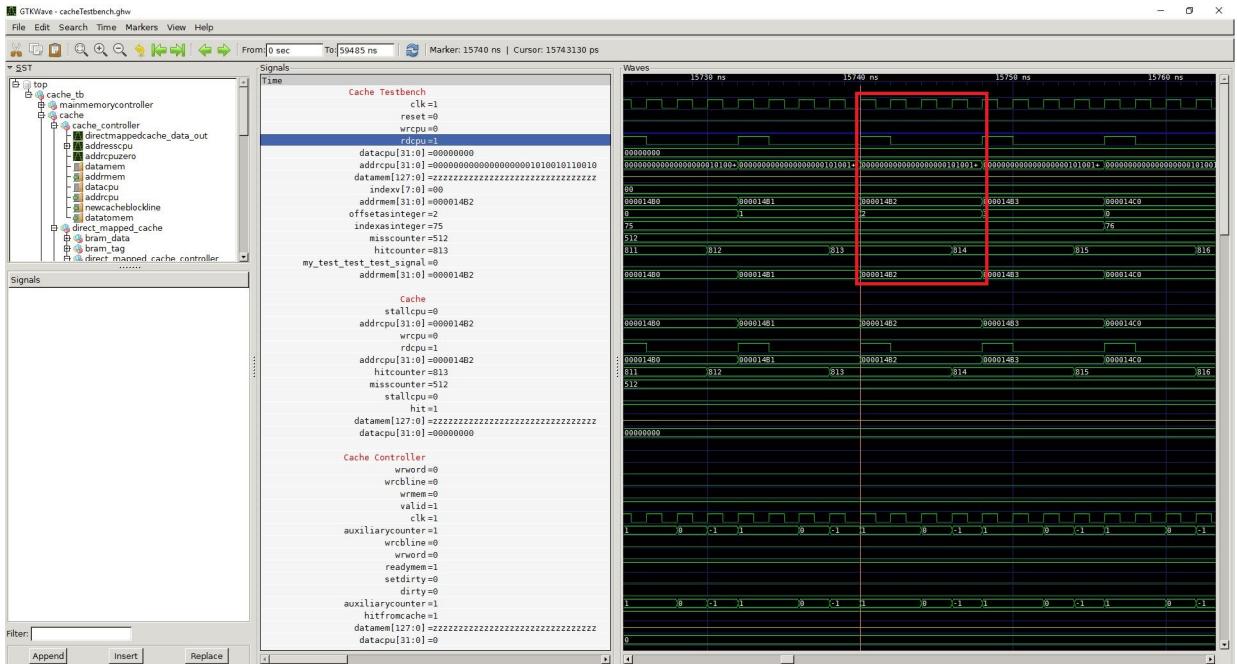


Figure 8: Result Simulation Gtkwave - Cache Hit

4 Task 5 - Branch Prediction

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 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.

6 Appendix

6.1 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 #
37 #           .text
38 #
39 #           li      $t0 , 16      # $t0 = number of rows
40 #           li      $t1 , 16      # $t1 = number of columns
41 #           move   $s0 , $zero    # $s0 = row counter
42 #           move   $s1 , $zero    # $s1 = column counter
43 #           move   $t2 , $zero    # $t2 = the value to be stored
44 #
45 #           Each loop iteration will store incremented $t1 value into next element of
46 #           matrix.
47 #
48 #           Offset is calculated at each iteration. offset = 4 * (row * #cols + col)
49 #
50 #           Note: no attempt is made to optimize runtime performance!
51 #
52 loop:   mult    $s0 , $t1      # $s2 = row * #cols (two-instruction
53 #           sequence)
54 #           mflo   $s2      # move multiply result from lo register to
55 #           $s2
56 #           add    $s2 , $s2 , $s1 # $s2 += col counter
57 #           sll    $s2 , $s2 , 2   # $s2 *= 4 (shift left 2 bits) for byte
58 #           offset
59 #           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 6: 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
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

```

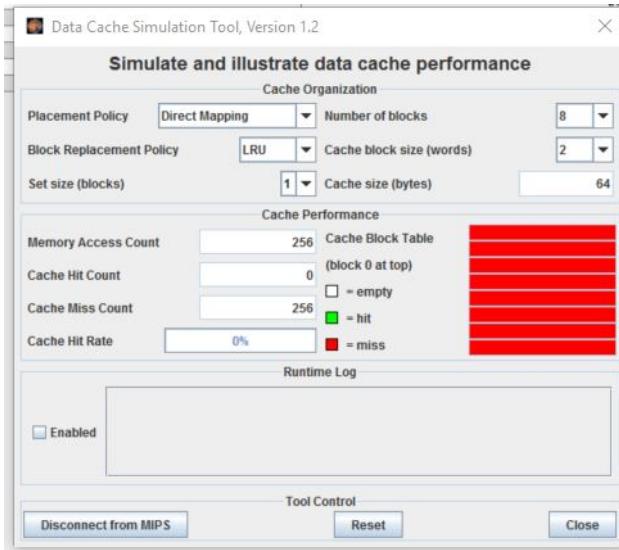


Figure 9: Column Major, Direct Mapping, Cache Block Size 2

```

48      increment row counter
#          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
#  We're finished traversing the matrix.
55      li     $v0, 10        # system service 10 is exit
56      syscall           # we are outta here.

```

Listing 7: row-major.asm

6.2 Cache Results - Snapshots

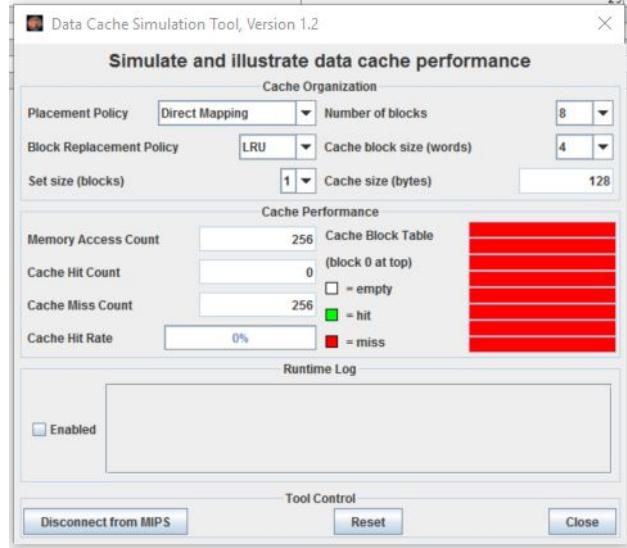


Figure 10: Column Major, Direct Mapping, Cache Block Size 4

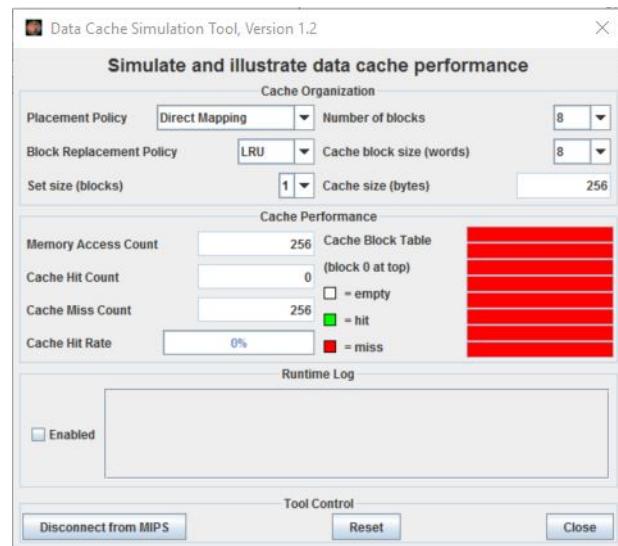


Figure 11: Column Major, Direct Mapping, Cache Block Size 8

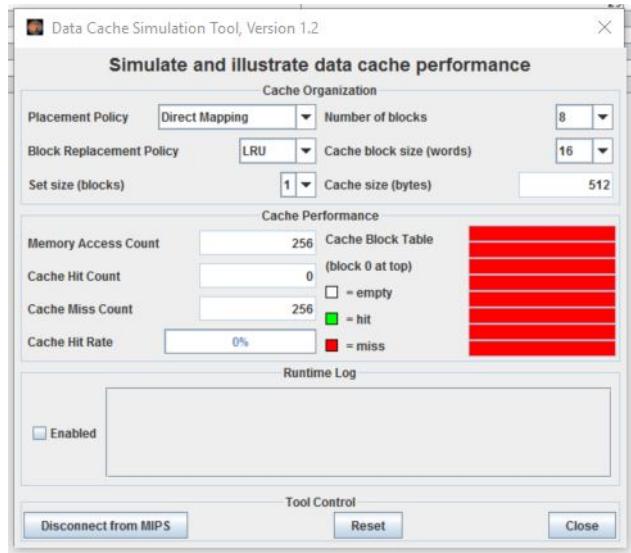


Figure 12: Column Major, Direct Mapping, Cache Block Size 16

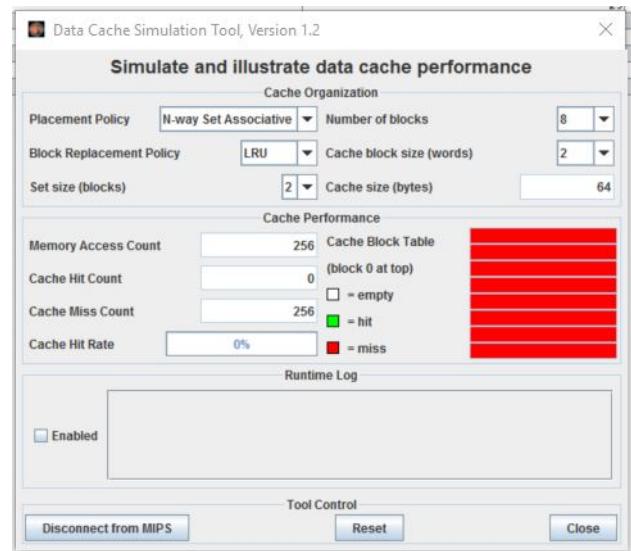


Figure 13: Column Major, 2-Way Associative, Cache Block Size 2

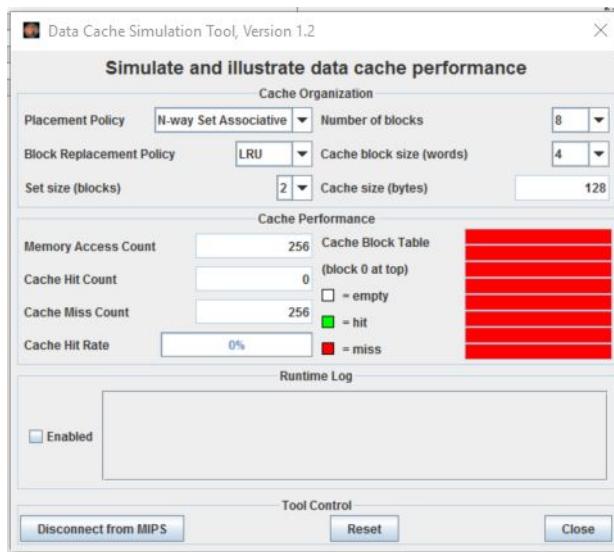


Figure 14: Column Major, 2-Way Associative, Cache Block Size 4

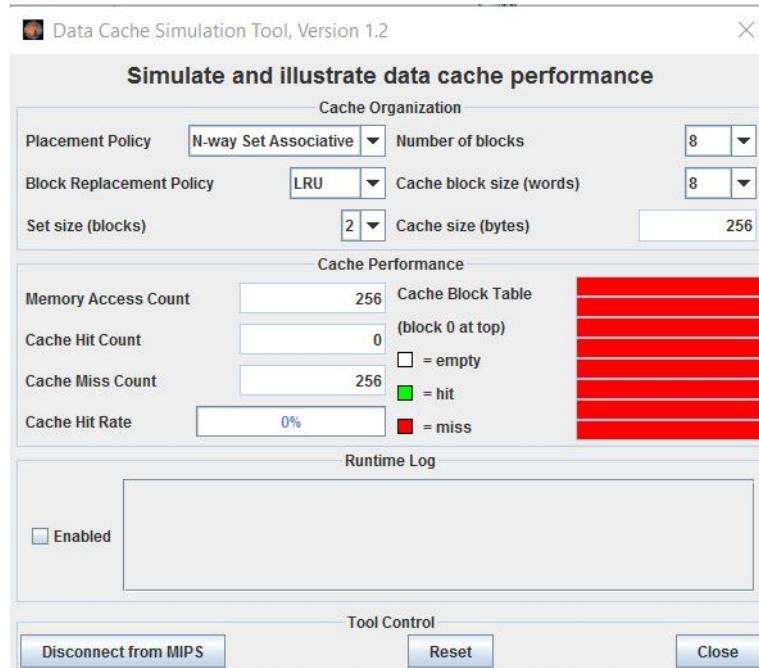


Figure 15: Column Major, 2-Way Associative, Cache Block Size 8

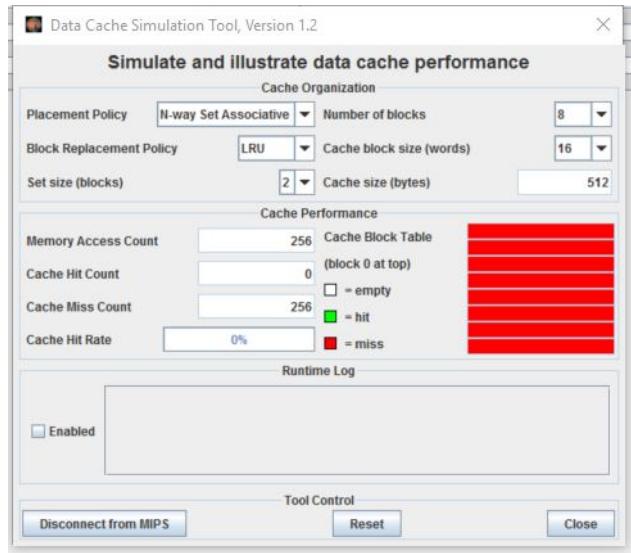


Figure 16: Column Major, 2-Way Associative, Cache Block Size 16

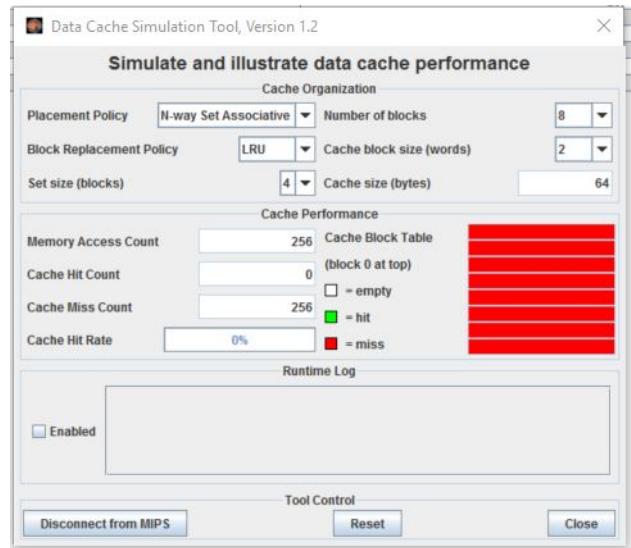


Figure 17: Column Major, 4-Way Associative, Cache Block Size 2

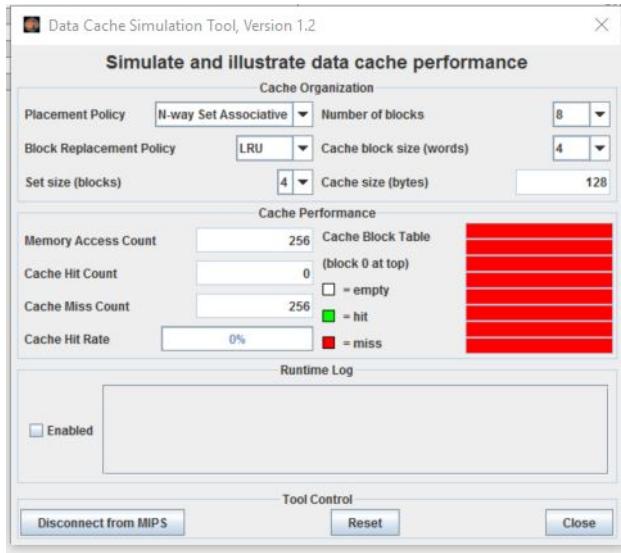


Figure 18: Column Major, 4-Way Associative, Cache Block Size 4

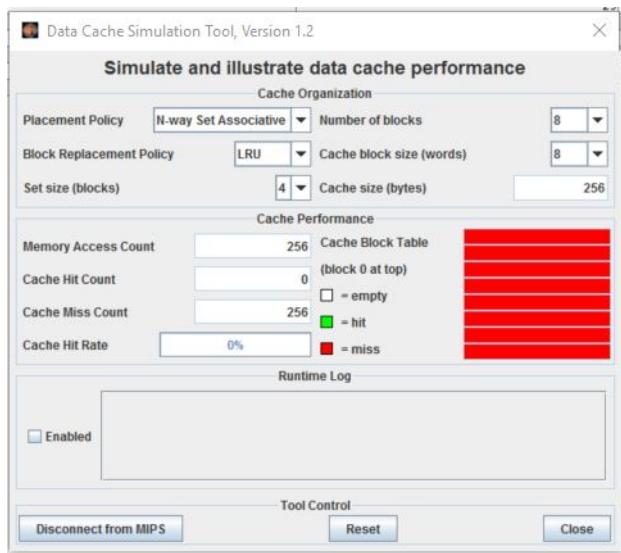


Figure 19: Column Major, 4-Way Associative, Cache Block Size 8

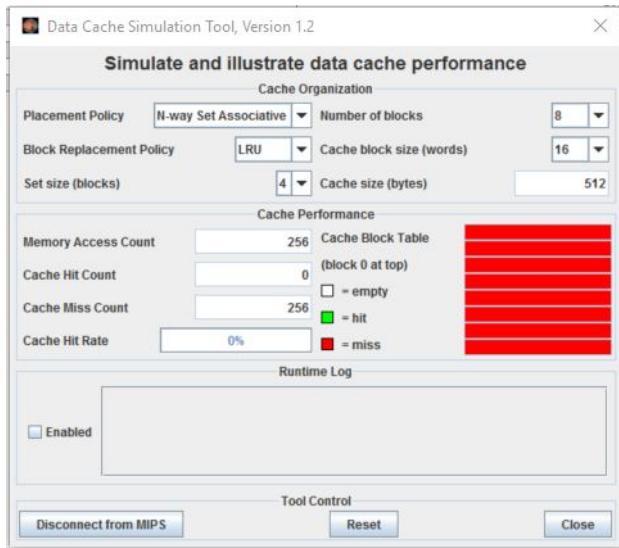


Figure 20: Column Major, 4-Way Associative, Cache Block Size 16

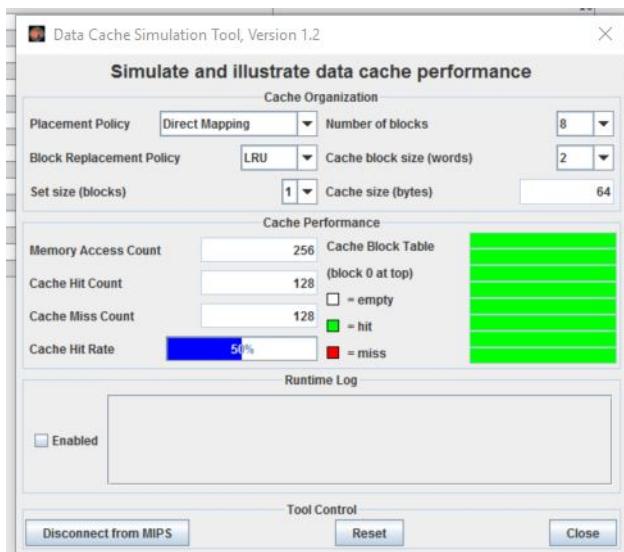


Figure 21: Row Major, Direct Mapping, Cache Block Size 2

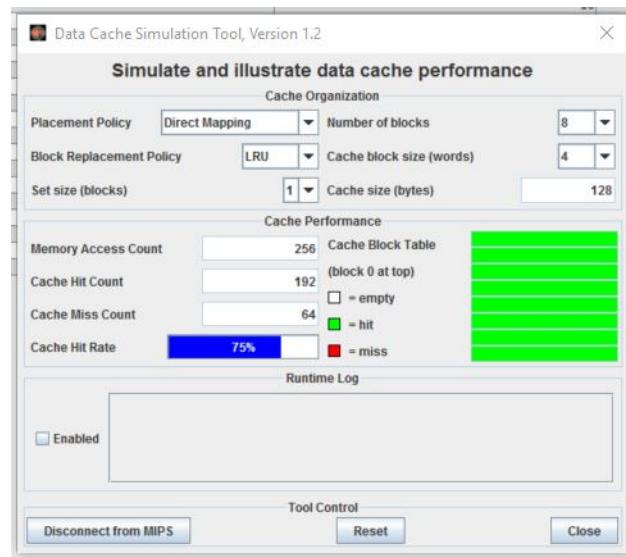


Figure 22: Row Major, Direct Mapping, Cache Block Size 4

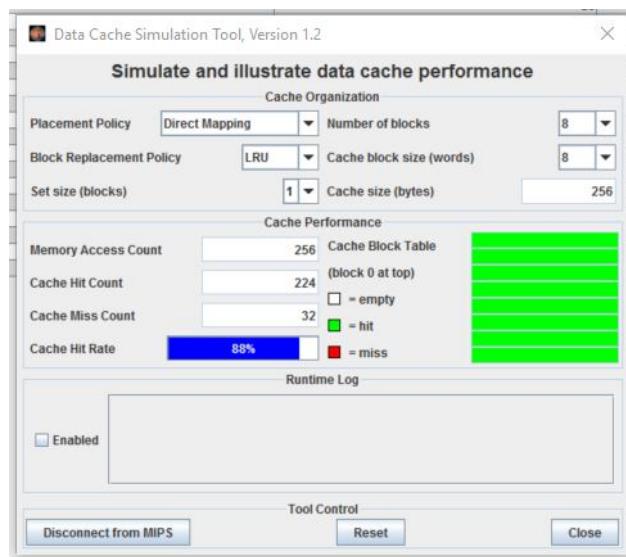


Figure 23: Row Major, Direct Mapping, Cache Block Size 8

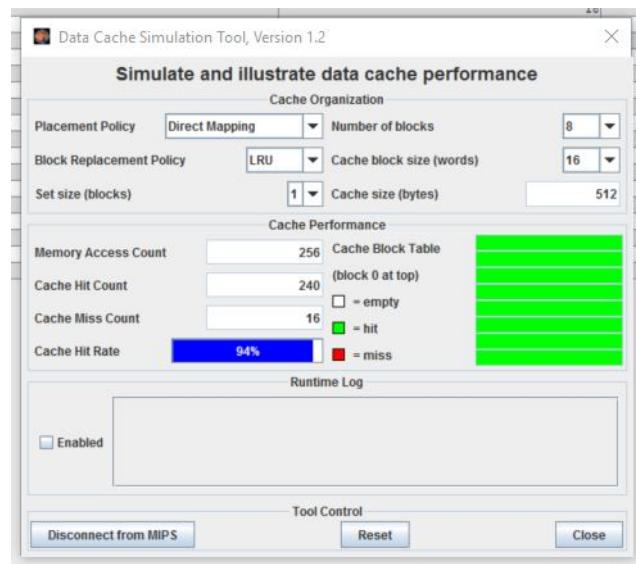


Figure 24: Row Major, Direct Mapping, Cache Block Size 16

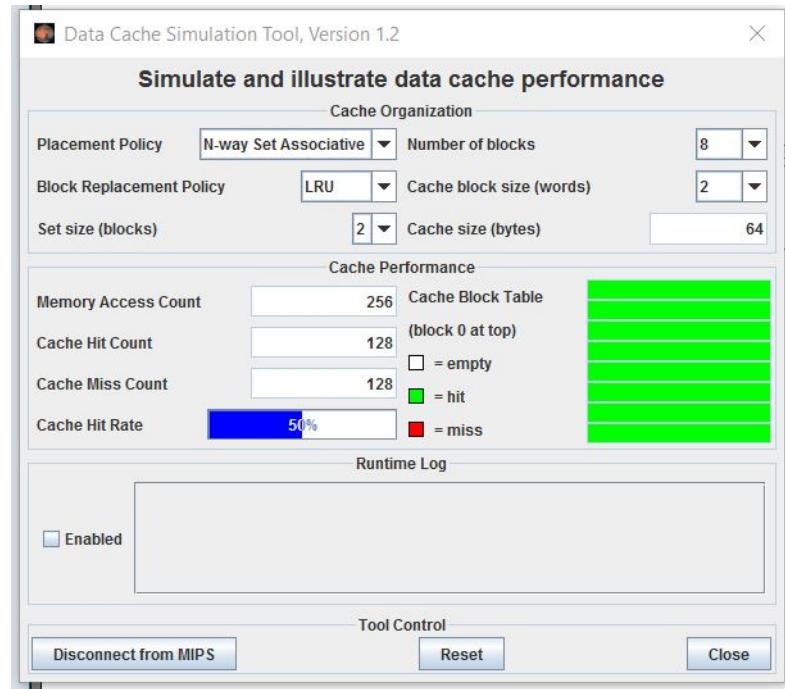


Figure 25: Row Major, 2-Way Associative, Cache Block Size 2

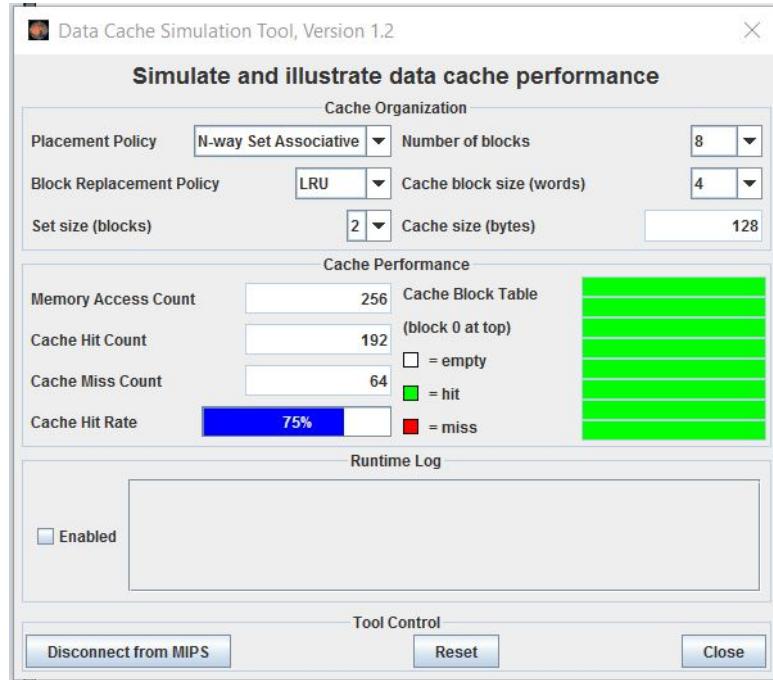


Figure 26: Row Major, 2-Way Associative, Cache Block Size 4

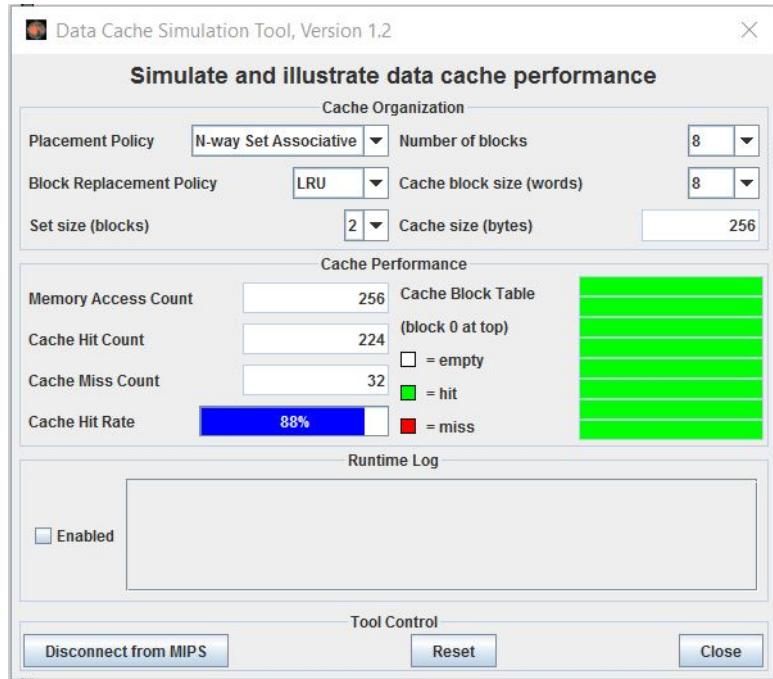


Figure 27: Row Major, 2-Way Associative, Cache Block Size 8

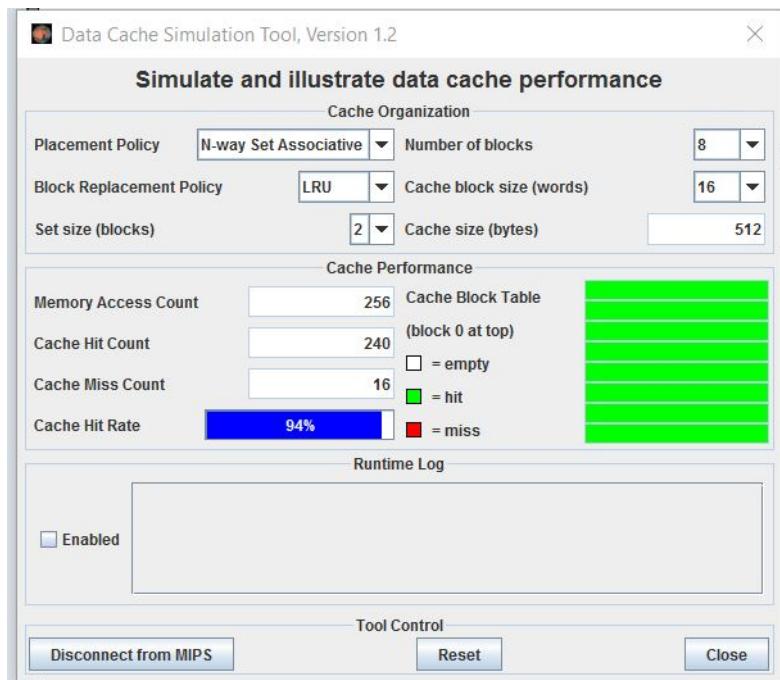


Figure 28: Row Major, 2-Way Associative, Cache Block Size 16

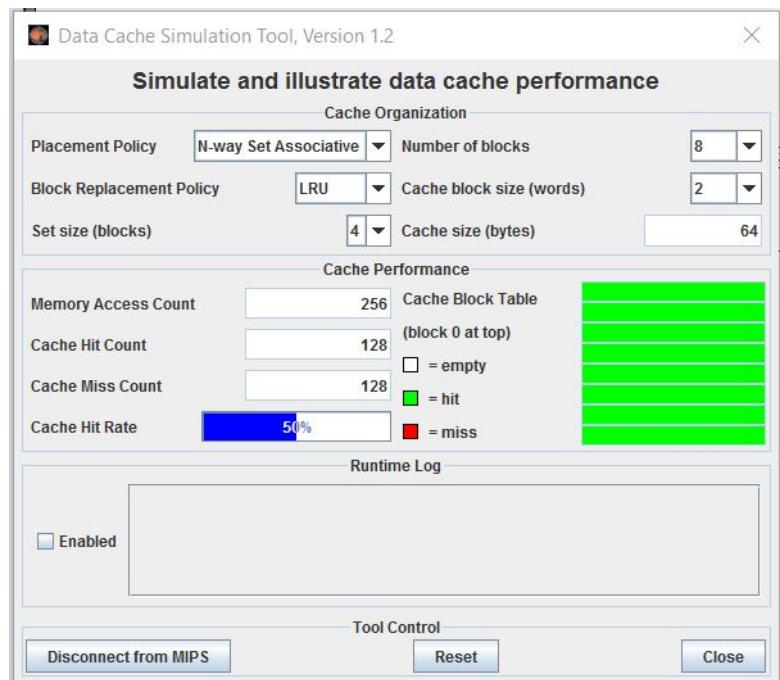


Figure 29: Row Major, 4-Way Associative, Cache Block Size 2

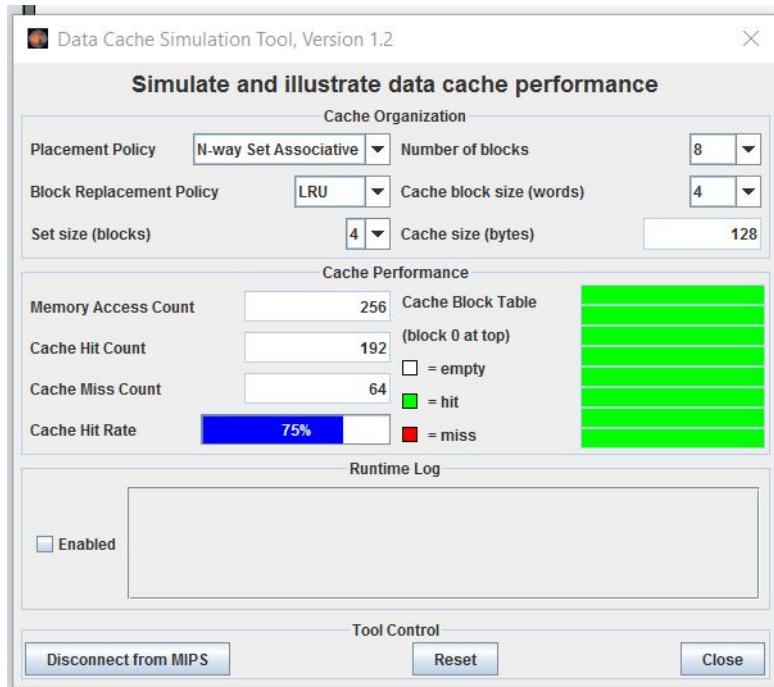


Figure 30: Row Major, 4-Way Associative, Cache Block Size 4

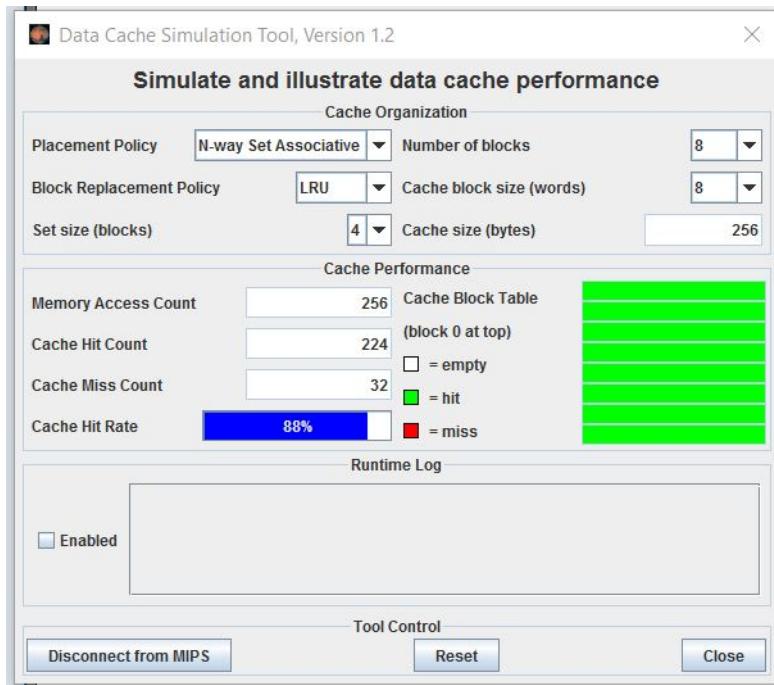


Figure 31: Row Major, 4-Way Associative, Cache Block Size 8

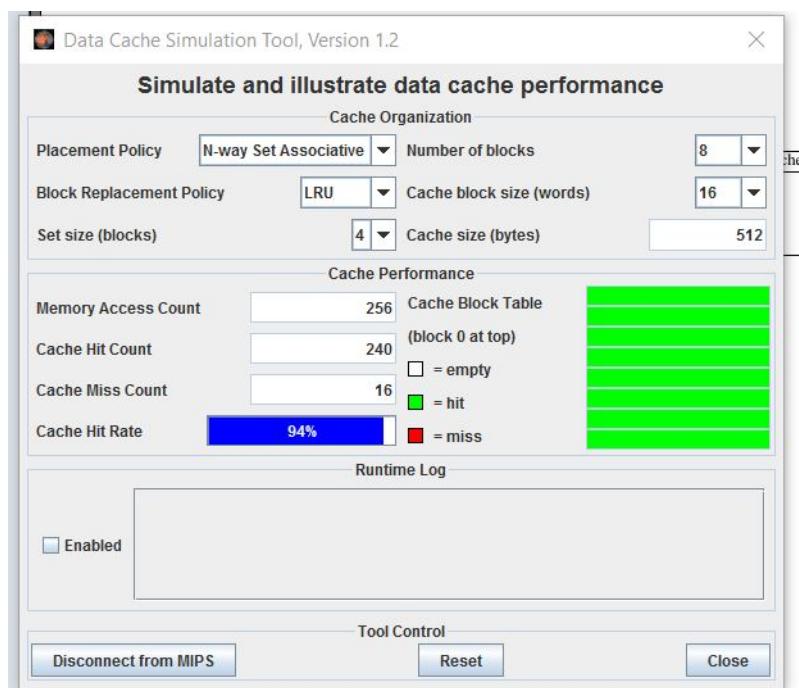


Figure 32: Row Major, 4-Way Associative, Cache Block Size 16