TUHH
Hamburg University of Technology

HAMBURG UNIVERSITY OF TECHNOLOGY

PROBLEM-BASED LEARNING

Advanced System-on-Chip Design

Carsten Hoppe
carsten.hoppe@tuhh.de

Hendrik Meyer zum Felde
hendrik.meyer@tuhh.de

Leonard Püttjer
leonard.puettjer@tuhh.de

Documentation
Advanced System on Chip

Tutor
Dipl.-Ing. Wolfgang BRANDT

April 13, 2017

Contents

List of Figures

List of Tables

Listings

1 Introduction

This document describes a student project in the winter term 2016/2017 at the Hamburg University of Technology (TUHH). The task of the project was to extend and improve the functionality of a comparatively simple CPU, a 16-bit MIPS processing unit. For the development of our code we used the Sigasi-plugin for the Eclipse-editor. The VHDL code was compiled using GHDL and MARS and then simulated, visualized and debugged using GTKWave. We developed our source code on the three different operating systems:

- Windows 10 - 64 bit
- Ubuntu 14 - 64 bit
- macOS 10.12.4 - 64 bit

To compile our code on macOS, we used a VirutalBox with Ubuntu 15.10 - 64 bit as operating system to execute the GHDL - script. Thus, we developed two different scripts to execute ASM - files with our MIPS - CPU, one for Windows and one for Linux.

All pictures for the FSMs, simulation results and code snippets were, if not denoted different, created on our own using Word or Latex for the FSMs, screenshots of Eclipse for the code snippets and screenshots of GTKWave for simulation results. To accomplish the project, at first we had to familiarize ourselves with the assembler-commands of a MIPS - CPU. We used MARS to develope and test simple assembler programs, which we could later use as benchmarks to verify, that our CPU behaves as we would expect. In addition, we received further, more complex ASM - programs, which we could use to verify the functionality of our CPU.

The goal of the course was to add further functionality to improve the CPU. We added the following features in the respective order:

- Implementation of pipeline stages using stalling and forwarding to resolve data hazards and structural hazards.
- Exchange of Distributed RAM by Block RAM
- Design direct mapped cache and 2-way-associative cache
- Implementation of a direct mapped instruction cache
- Static branch prediction
- Dynamic branch prediction using register files as a 2-bit branch history table (BHT)
- Branch target buffer using 2-way-associative cache to store jump/branch targets

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 maid. 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 occured and all steps were taken according to the modification plan precise testing had to take place. Small simple programs were executed and

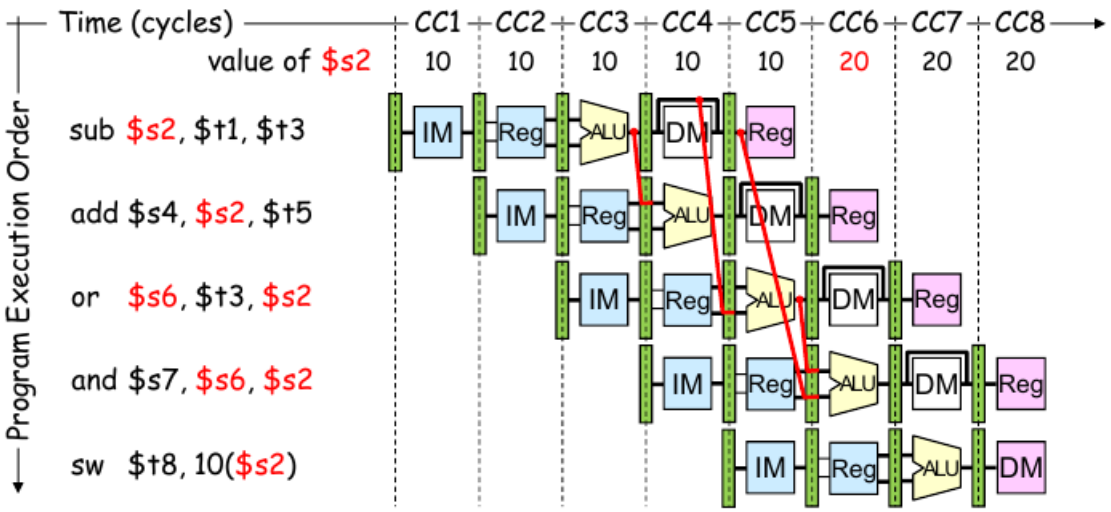


Figure 1: FEC Slides, Diagram of Forwading Logic

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 cock 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 to behaviour that needed to be checked and varied the numbers of auxiliary nops inbetween ciritical 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 avticated, 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) comnmands 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 2. The code that was executed was consisting of three nop operations followed bei a JAL command. The PC of the 4th command, JAL, ist 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 occured 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 occured 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 - One Nop at the Beginning Needed

After our logic was implemented and the test runs were successfull it was interesting to see that some odd behaviour occured at the beginning of the program. It might be important to mention this behaviour in order to clearify the situation and make bug hunting easier. The first command that is processed by the modified mips is a nop command due to the initialized state. But it turned out that the second command will be executed twice in a row. In order to bypass this strange behaviour we simply inserted a nop at the beginning of each asm program. The consequence was a doubled nop command, which has no impact on the program at all.

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 dissvoled 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. In a final step we replace the instruction memory in the MIPS CPU by our designed direct mapped cache in section 3.7.

3.1 Introduction to Memories

3.1.1 The Need for Different Storage Types

In the world of computer science not all memory storage types have the same properties and abilities. Major distinctions can simply be made by looking at price and performance information. For instance cache memory is extremely fast compared to hard disk storage or even tape libraries but on the other hand its storage space is highly limited compared to hdd or tape storage due to its small physical space. Another big difference is of course the price. Buying tape which is able to store TBs of data might cost less than 50 euros. Extremely fast storage types like caches or RAM would cost thousands of euros if one wanted to use a TB of RAM.

It often occurs that a computer doesn't simply need one type of storage but several different ones for different tasks. That is why storage that has few space but high access times (Caches) were developed as well as moderate access times with more space (SSD, HDD). The same applies for the higher levels of the memory hierarchy with Registers, Level 1 Caches, Level 2 Caches and Main Memory. Storing all information in Registers would be too expensive. Additional levels inbetween magnetic or flash discs and registers are needed as well.

The problem of fitting all needs can be managed by letting the different storage types interact with each other whenenver required. If data is accessed extremely often a cache can be used or even registers, whereas rare use can easily be stored in memory that is less expensive and less fast. The more types of storage are available, the easier it is so handle problems needing multiple demands at once.

3.1.2 Classifying Caches - Advantages and Disadvantages

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.

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 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

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

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 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 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.

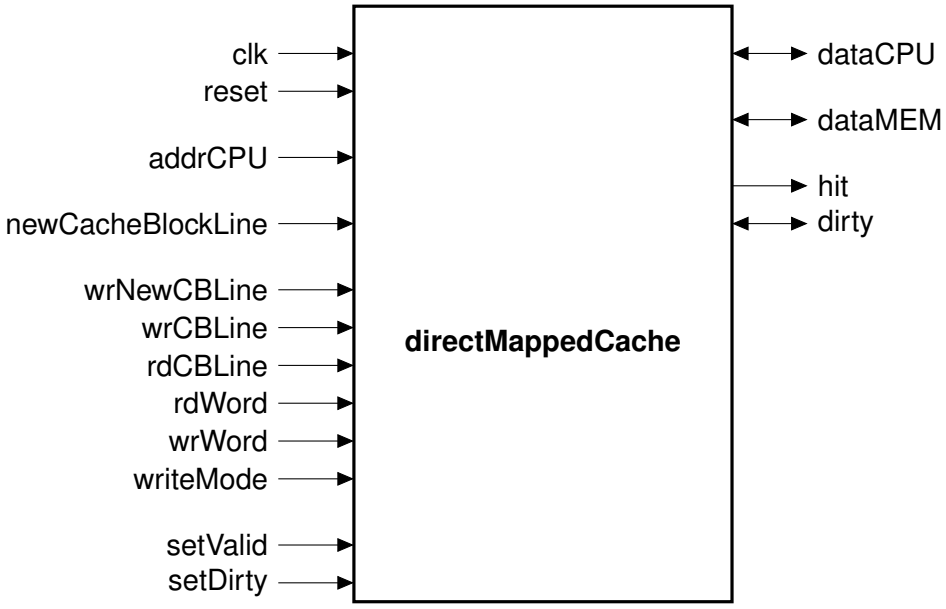


Figure 3: Entity regarding direct mapped cache.

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 number of occurences of 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 4 the state diagram of the cache controller is illustrated. The state diagram represents a Mealy automaton.

3.4.1 Description of the state space

The initial state of the cache controller is *IDLE*. In this state the cache is waiting for a request from the CPU. Either the CPU wants to read a data word from the cache or prefers to write new data into the cache. If the CPU is writing into cache, the state is switched to state *CHECK1*. Now, the cache controller checks whether the correspondent memory block is already in the cache. We get a cache hit, if the memory block is already in the cache. Therefore, we return to state *IDLE*. Otherwise, the cache controller must check whether the current cache block line is dirty and valid. If the cache block line is valid and modified, then the controller reaches the state *WRITEBACK1*. Here, the cache writes the current cache block line back to the main memory. The cache controller is waiting for the main memory since the operation of writing back takes a couple of cycles. Once the main memory is ready the cache controller traverse the delay state *WRITE_BACK1_DELAY* and the state *WRITE*. The delay state is needed as a delay because the correspondent signals have to be updated. If the cache controller does not recognizes a cache hit and the current cache block line is either invalid or unmodified, then the controller will go from

state *CHECK1* to state *WRITE*. Here, the cache reads the correspondent cache block line from the main memory. Also, the cache controller is waiting for the main memory until the main memory finished the read request. Finally, the read memory block and the new data word from the CPU are placed in the correspondent cache block line. This procedure is done in state *TOCACHE1*. Surely, the cache controller finishs the write request of the CPU and idles in the initial state. Similar to the write operation the cache controller handles a read request from the CPU. In state *CHECK2* the cache controller tests the cache block line for cache hit or cache miss. In case of a cache hit, the controller directly returns to state *IDLE*. Otherwise, the cache block line could be dirty. Then the cache controller have to write back the cache block line to the main memory. Thus, the controller goes to state *WRITEBACK2*. Inside this state the controller is waiting for the main memory. When the main memory finished reading, the the controller changes its state to state *READ*. Previously the controller traverses the state *WRITE_BACK2_DELAY* because the signals are been updated. Equally, the cache block line could be unmodified or invalid. Then the cache controller transits from state *CHECK2* to state *READ*. In state *READ* the controller reads the correspondent memory block from the main memory. Afterwards, this memory block is written into the cache line. The successor state of state *TOCACHE2* is afresh state *IDLE*.

3.4.2 Description of the outputs

Since the cache counts the number of occurrences of cache hits and cache misses, two registers are used. The register *rHitCt* counts the cache hits and *rMissCt* counts the cache misses. The output *cacheLine2MEM* means that the current cache block line must be written back to main memory. Thus, the correspondent control signals are set. The control signal *rdMEM* indicates whether the main memory is just read by the cache controller. Furthermore, the control signal *wrNewCBLine* signalizes when a memory block, which has been read from the main memory, and the new data word given by the CPU are placed into a cache block line at the same time. The output *data2CPU* declares that the correspondent data word is transferred from the cache to the CPU. This represents the data word, which is enquired by the CPU.

3.4.3 Description of the inputs

The input *wrCPURequest* and *rdCPURequest* represents whether the CPU wants to read or write the cache. Therefore, either the signal *wrCPU* or the signal *rdCPU* is set to '1' to indicate the CPU request. If both signals are '1' then the controller will ignores the request because it cannot read and write the cache at the same time. The input signal *hit* indicates whether a cache hit occurs. The cache hit only appears if the correspondent cache block line is valid and the tags in this cache block line and the tag from the current CPU address are equal. Otherwise, a cache miss is incurred. In this case, we distinguish between three cases. On the one hand, the cache block line could be invalid. Then, the input signal *lineIsDurty* is set to '1'. On the other hand, the cache block line could be invalid and the input signal *lineIsInvalid* is fulfilled. The last case is that the cache block line is valid and unmodifed w.r.t. the main memory. This is represented by the control signal *lineIsNotDirty*. The input signal *readyMEM* identifies whether the main memory finished a read or write operation. Therefore, this signal is set to '0' if the main memory is still operating. Otherwise, it is set to '1' if the main memory has already finished the operation.

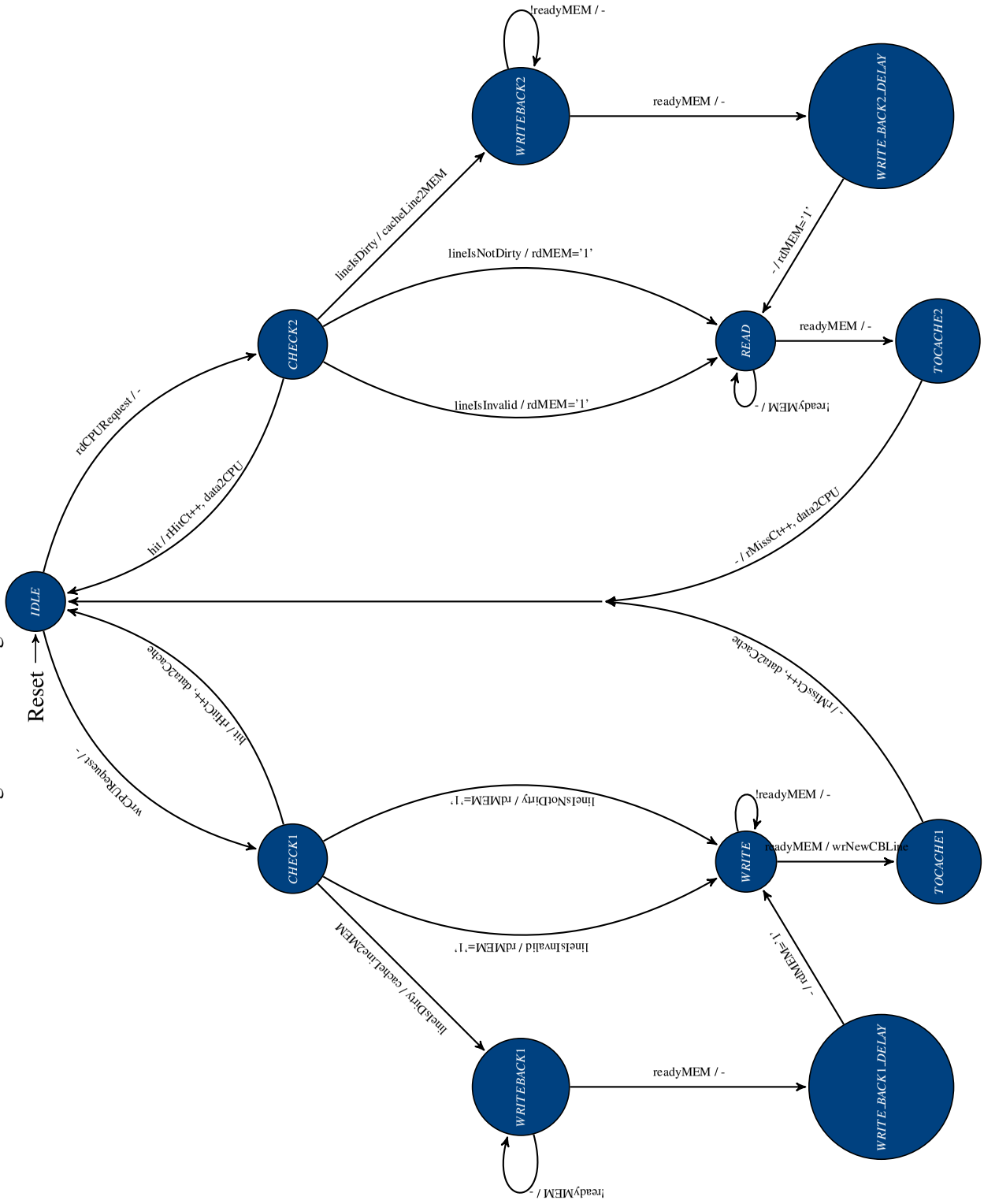


Figure 4: State diagram of the cache controller.

3.4.4 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 5. The controller could be arranged in one of the following states. The state *IDLE* represents that the controller is waiting for a read or write request from the cache. In state *READ* the controller is reading the main memory. Assume, a cache block line contains four data words. Then the main memory controller must read four data words from main memory in state *READ*. Respectively, the controller writes the cache block lines to main memory in state *WRITE*. The additional state *DELAY* is needed to simulate that the *readyMEM* signal is set after 20 cycles.

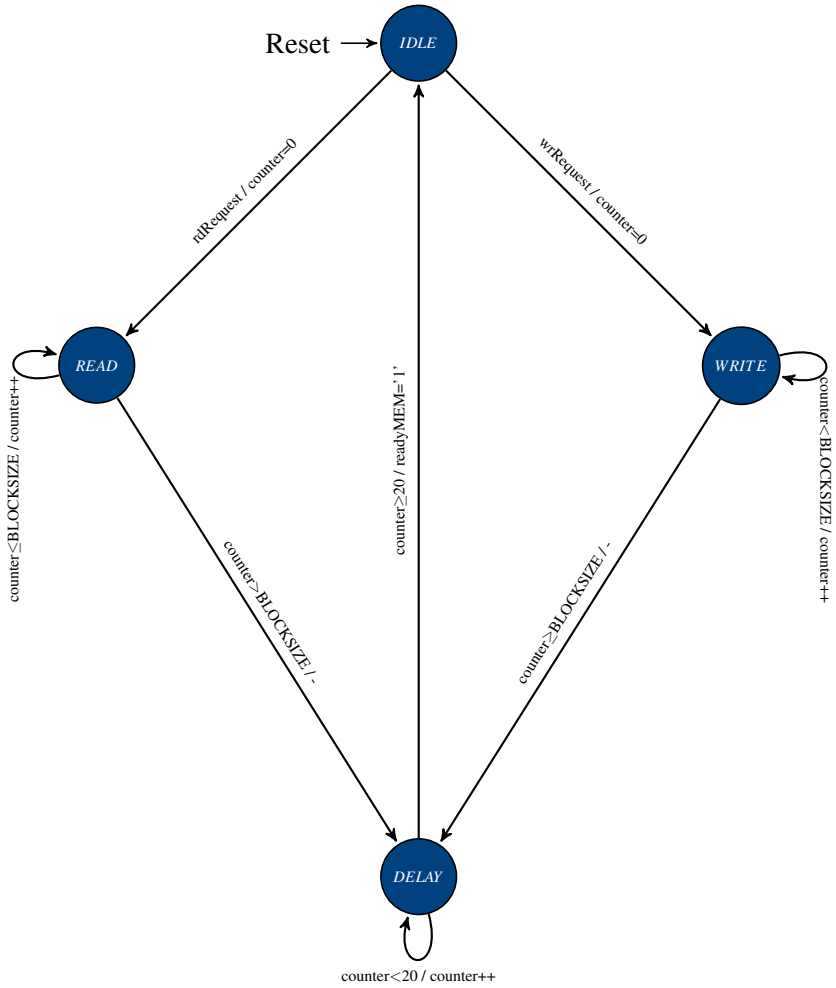


Figure 5: Sketch of Mealy Automata - Main Memory Controller

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 3 we describe some test cases. All these test cases are implemented in VHDL in the file *cache_tb.vhd*.
We have created the batch

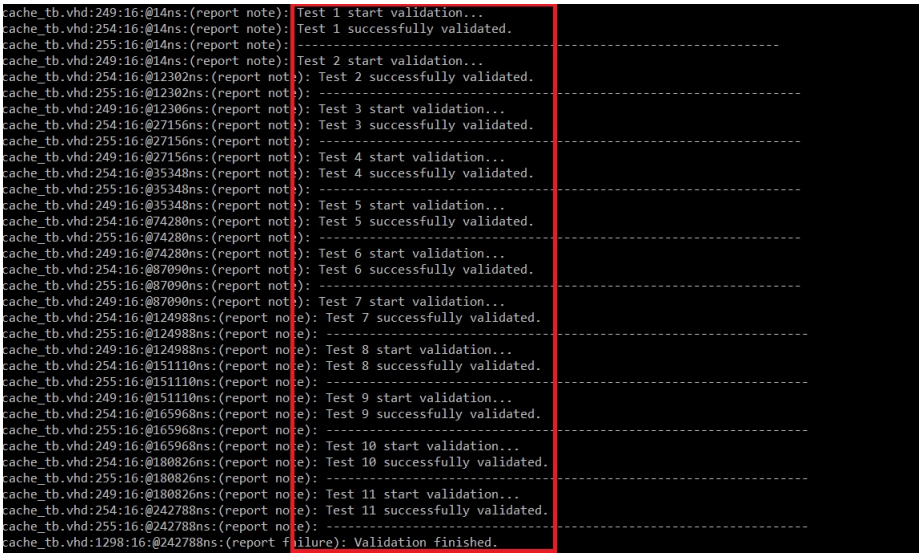


Figure 6: Result Simulation - Cache

file *run_ghdl_task4_cacheTestbench_windows.bat* to run the testbench with GHDL under Windows. Thus, we start the simulation by typing *run_ghdl_task4_cacheTestbench_windows.bat* in the command-line user interface with a name of an assembler file. During execution of the testbench some notes are shown indicating which test case succeeds. Figure 6 shows that all test cases are successfully executed. Respectively, we create the
script *run_ghdl_task4_cacheTestbench_Linux.sh* to run the simulation under Linux. Consider, that the variables have to be updated as necessary in the scripts.
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 3 clock cycles are needed for a cache hit. In contrast the cache controller needs circa 23 clock cycles in case of a cache miss. A read operation with cache miss is represented in figure 8.

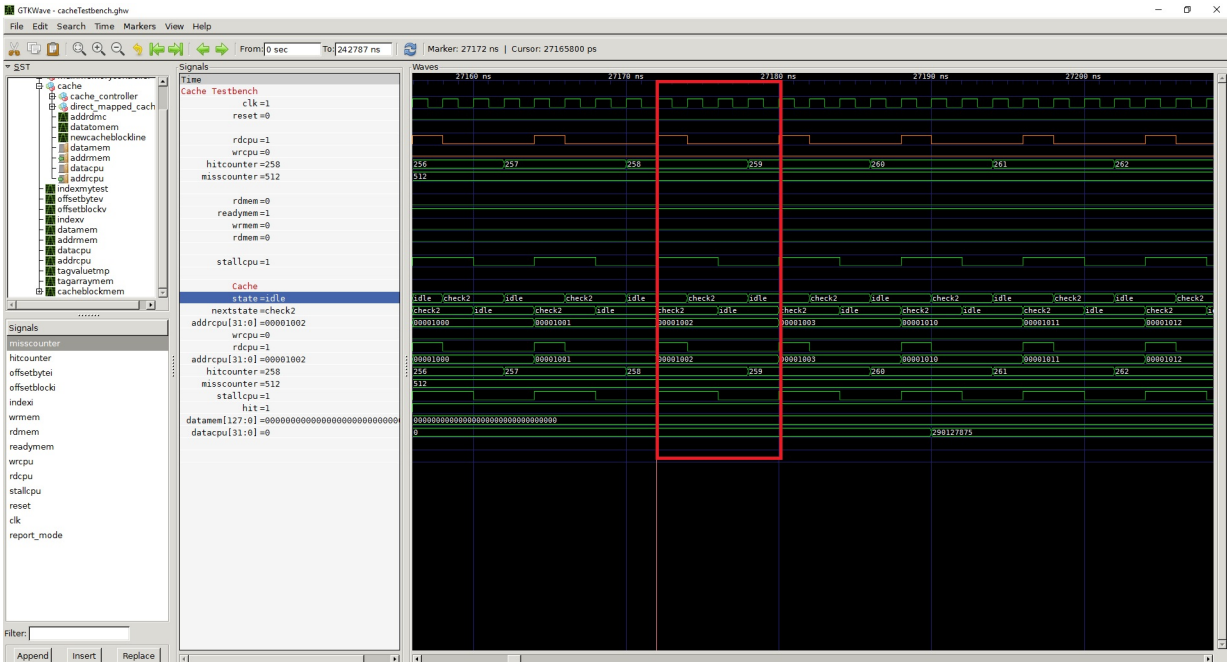


Figure 7: Result Simulation Gtkwave - Cache Hit

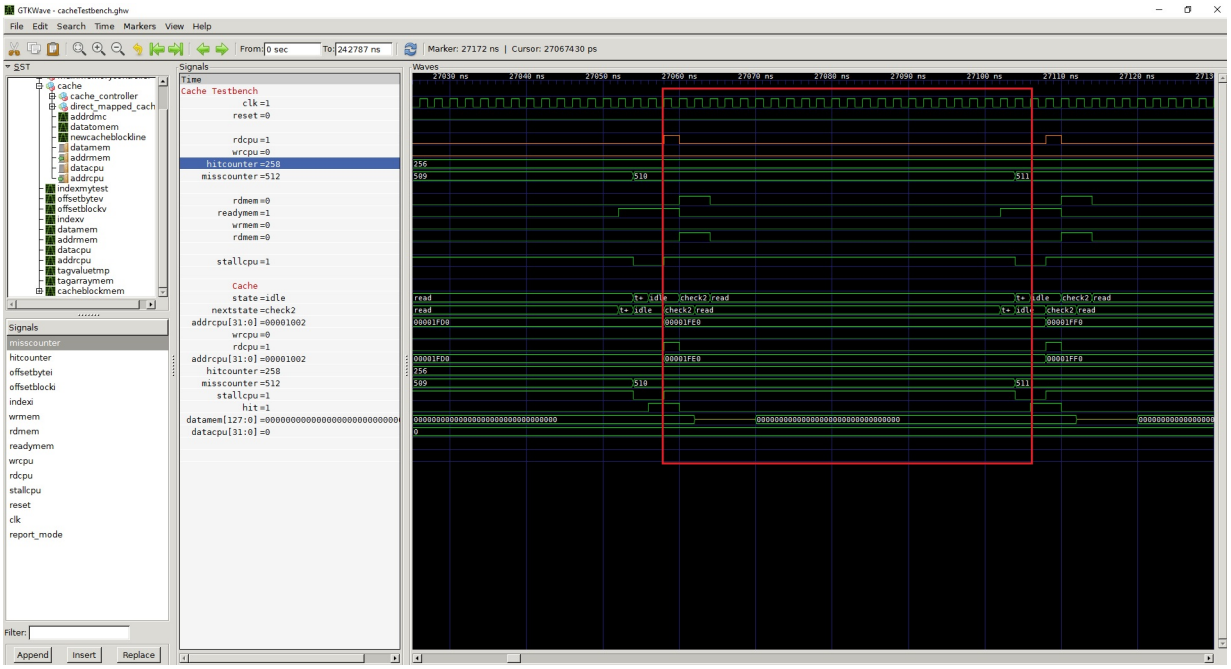


Figure 8: Result Simulation Gtkwave - Cache Miss

Table 3: 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.	

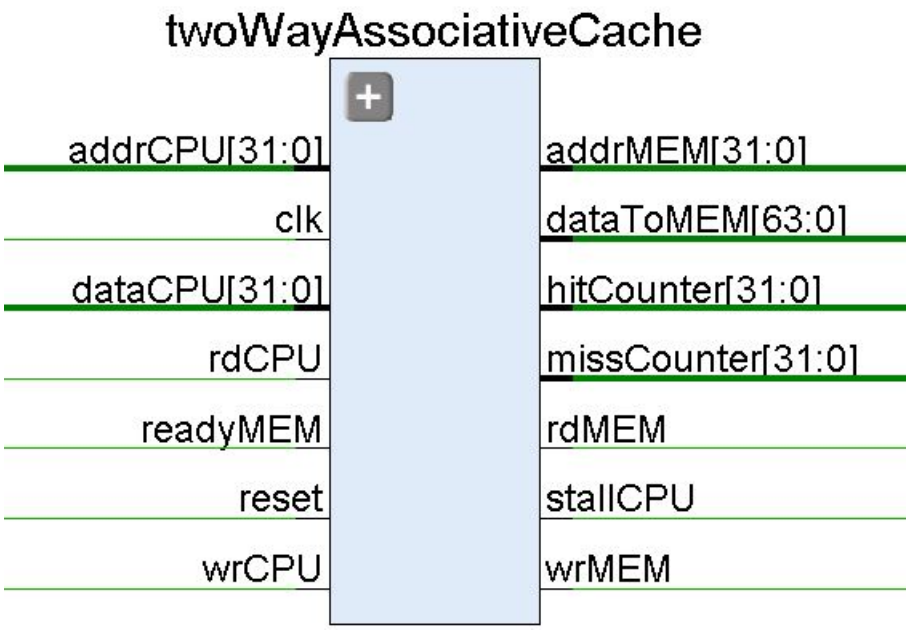
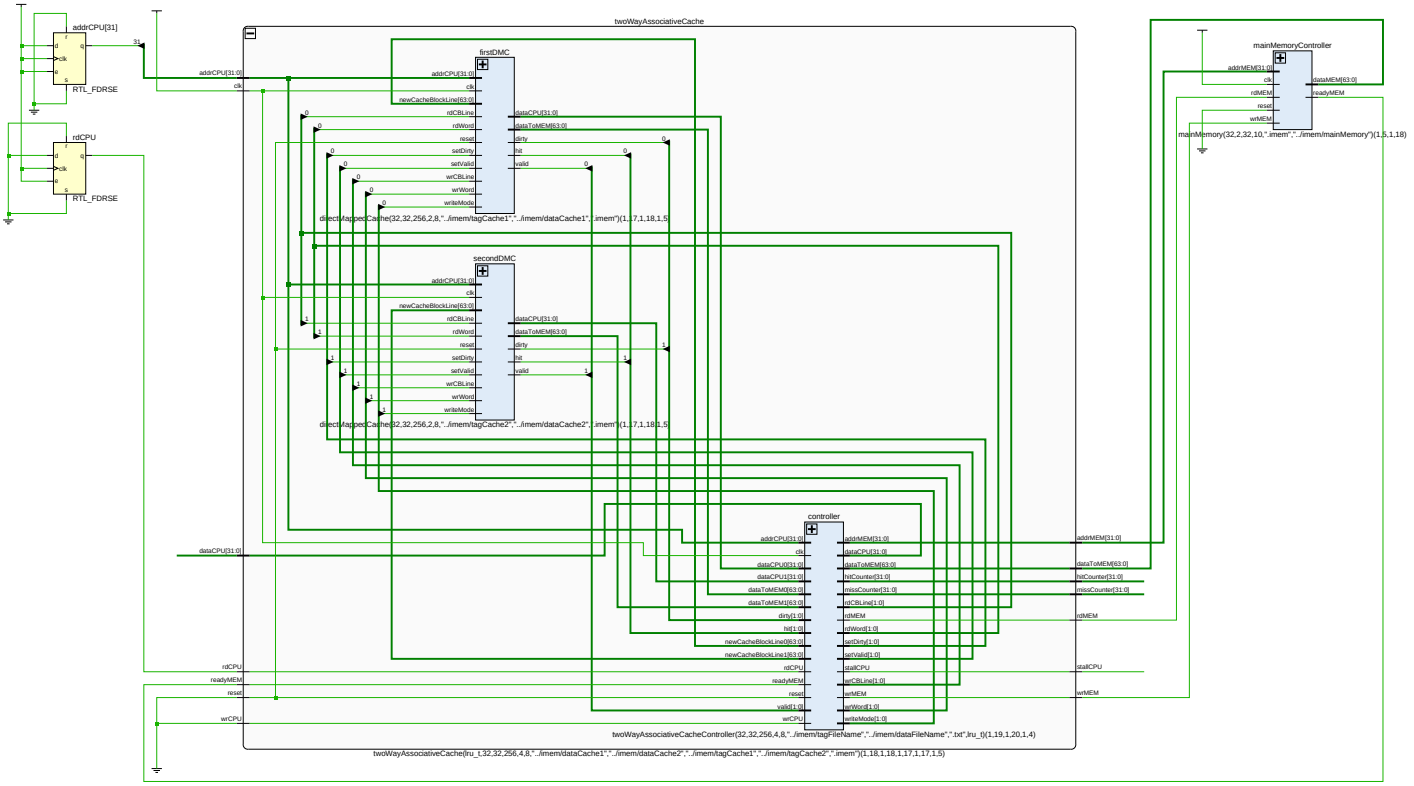


Figure 9: 2 way associative cache

3.6 Design a 2-way associative Data Cache incl. Controller

In the following step a two way set associative cache is implemented in file *twoWayAssociativeCache*. The corresponding interface of the entity is shown in figure 9. Similar to the directed mapped cache the two way set associative cache is connected with the CPU and with the main memory. Therefore, the input ports *rdCPU* and *wrCPU* controls that the cache is read or written by the CPU. the correspondent *addrCPU* is the address of the word which should be read or written. The signal *dataCPU* contains the data word which should be written into cache or read from cache. While the cache is working it indicates via the signal *stallCPU* whether it is ready or not. Respectively, the CPU must be stall when the cache is just working. Also, the two way cache is connected to the main memory. The cache could either read or write the main memory. Thus, the two control signals *rdMEM* and *wrMEM* are used. The two signals *addrMEM* and *dataToMEM* represents the correspondent address and data word which is read or written to main memory. The in port *readyMEM* indicates when read or write operation to the main memory has been finished. In figure 10 the RTL schematic of the 2-way associative cache is illustrated. As you can see the cache consists of one controller and two direct mapped caches. Here, the two direct mapped caches stores the correspondent entries of the whole cache. The provided cache controller determines which of the two direct mapped caches should be used for the next operation. It is possible to choose between the LRU or a random strategy. The implementation of the controller is given in file *twoWayAssociativeCacheController.vhd*.
In command line you can execute the batch script run *_ghdl_task4_2wayCacheTestbench_windows.bat* with a name of an assembler program. This script will simulate the correspondent testbench.

Figure 10: RTL Schematic of 2-way associative cache



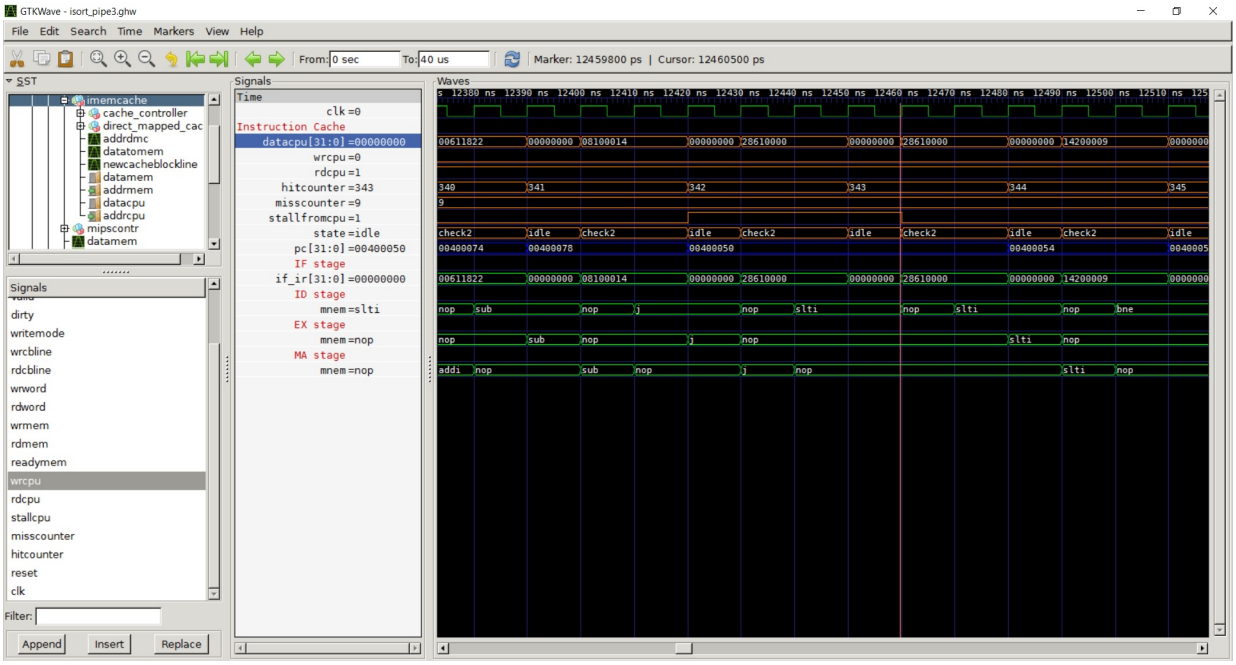


Figure 11: Result Simulation Gtkwave - MIPS with instruction cache

3.7 Replace the Instruction Memory of the MIPS CPU with the Cache

Finally, we integrate the direct mapped cache into the MIPS CPU. Thus, we replace the instruction memory with the cache. We rewrite the MIPS code to generate the one entity of the cache as the instruction cache. Also, we remove the code lines with the instruction memory. These changes are shown in 6. Anymore, we connect the correspondent input and output ports if the cache entity. The input port *rdCPU* and *wrCPU* are constantly set to '0' and '1' since the MIPS always reads the instruction cache and does not change the instructions. The output port *dataCPU* represents the data word which should be read from cache or to be written into cache. Thus, it is connected to the signal *IF_ir*. The additional signal *stallFromCache* indicates whether the CPU must be stalled. Therefore, this signal is connected to the output port *stallCPU*.
The result of the simulation run is shown in figure 11. The CPU reads the next instruction word from the instruction cache by specifiying the current program counter. Therefore, the control signal *rdCPU* is always set to '1'. This signal orders the instruction cache to return the corresponding instruction word. Then the instruction cache switch its state from *IDLE* to state *CHECK2*, since it have to check whether the inquired instruction word is already stored in the cache or not. This figure represents the case that the asked instruction word is already located in the cache. Therefore, the hit counter will be incremented by value one. Furthermore, the pipeline is stalled from the instruction cache while it is in checking state. Thus, the *nop* instruction is inserted into the pipeline while the instruction cache is in state *CHECK2*.

```
1 -- ------------------------------------------------------------
2 -- Instruction cache.
3 -- ------------------------------------------------------------
```

```
4  imemCache: entity work.cache
5      generic map(
6          MEMORY_ADDRESS_WIDTH => MEMORY_ADDRESS_WIDTH,
7          DATA_WIDTH           => DATA_WIDTH,
8          BLOCKSIZE            => BLOCKSIZE,
9          ADDRESSWIDTH         => ADDRESSWIDTH,
10         OFFSET               => OFFSET,
11         TAG_FILENAME         => TAG_FILENAME,
12         DATA_FILENAME        => DATA_FILENAME,
13         FILE_EXTENSION       => FILE_EXTENSION
14     )
15     port map(
16         clk          => clk,
17         reset        => reset,
18         hitCounter   => hitCounter,
19         missCounter  => missCounter,
20         stallCPU     => stallFromCache,
21         rdCPU        => '1',
22         wrCPU        => '0',
23         addrCPU      => pc,
24         dataCPU      => IF_ir,
25         readyMEM     => readyMEM,
26         rdMEM        => rdMEM,
27         wrMEM        => wrMEM,
28         addrMEM      => addrMEM,
29         dataMEM      => dataMEM
30     );
31
32 --  imem:    entity work.bram  generic map ( INIT =>  (IFileName & ".imem"))
33 --           port map (clk, '0', pc(11 downto 2), (others=>'0'), IF_ir);
```

Listing 6: Replacing the instruction memory by direct mapped cache.

4 Task 5 - Branch Prediction

4.1 Branch Prediciton

The task was to implement a branch prediction unit in three ways:

- "Static branch prediction"
- "Dynamic branch prediction using a branch history table (BHT)"
- "dynamic branch prediction using a branch target buffer (BTB)"

4.2 Static Branch Prediciton

For our static branch prediction we needed perform the check if a branch will be taken in the ID phase. The comfortable restriction that was given was the fact that we only need to handle the commands BNE and BEQ since they were the ones used for testing programs like isort_pipe. In fact no other assembler file given was using other branch commands. Therefore our branch predicition only needed to handle a comparison check of two given registers. This was implemented using the following technique.

We inserted a new signal called branchIdPhase. This signal was set to 1 whenever a BEQ or BNE command was detected in the ID phase and simultaneously not in MA or EX phase. We wanted it to be set to '1' in order to mark the arrival of a new branch command. Simultenously our signal predictionError receives the input to perform static jumping if our StaticBranchAlwaysTaken signal is set to '1' and the branch condition is not met. In other words if we assume a branch is always taken and the condition to take a branched step is not given our predictionError signal is set to '1'. Whenever we have a predictionError we decide to take another path than the one that was predicted.

```
1 branchIdPhase        <=  '1'  when
2                 ((i.Opc = I_BEQ.Opc) and (EX.i.Opc /= I_BEQ.Opc) and
3                 (MA.i.Opc /= I_BEQ.Opc)) or
4                 ((i.Opc = I_BNE.Opc) and (EX.i.Opc /= I_BNE.Opc) and
5                 (MA.i.Opc /= I_BNE.Opc)) else '0';
6
7 predictionError      <=
8     StaticBranchAlwaysTaken when ((a /= b)  and i.Opc = I_BEQ.OPC)  else
9     StaticBranchAlwaysTaken when ((a = b)   and i.Opc = I_BNE.OPC)  else '0';
```

Listing 7: BranchCheckMovedToIDPhase

The impact of our code was to let the system assume a certain strategy like always "always assume to take a branch" and performing stalling whenever the prediction was not right. We needed to test whether this strategy was running as well as the possibility to switch the strategy at any time since the logic would be used in the dynamic branching also.

4.3 Dynamic Branch Prediction Using a BHT

In the beginning understanding of the functionality of a BHT needed to be gained. This was done by using the preinstalled feature from the MARS simulator. Which can be seen in figure 13 on page 31.

The BHT basically uses a slot for each branch command and remembers the last one or two jumping behaviours. According to the history the next jump should be taken in order to guess as many tries as possible correctly. Our problem is that whenenver we guess our branch jump behaviour wrong, we lose clock cycles. Therefore we want a prediction rate as high as possible.

The interesting thing is if we compare values for changed input like Intitial state either "TAKE" or "NOT TAKE" and History size is either 2 or only 1 entry we can see following behaviour in table 6 on page 36. Additional screenshots of results can be seen in the appendix.

BHT Settings	1BNE Precision	2BNE Precision	BEQ Precision
BHT with History 1 Initial TAKE	80%	88.57%	72.73%
BHT with History 2 Initial TAKE	70%	88.57%	72.73%
BHT with History 1 Initial NOT TAKE	90%	91.43%	69.70%
BHT with History 2 Initial NOT TAKE	90%	94.29%	75.76%

Table 4: of successful prediction using BHT in isort_pipe

After creating a running Static branch prediction in the previous task it was afterwards possible to give this branch prediction dynamic inputs. Since static always taken as well as static never taken were modes that could be selected. Figure 13 on page 31 shows how the prediction of a branch command is compared to the real result and influences the program flow. The vertical red line shows the state of a falling edge at which the variable a and b are compared whether they are equal or not equal in the ID phase. Hald a clock cycle later the result is at hand. Since the Prediction from the BHT says the branch is not taken and the values of a and b are equal to each other and we have a BNE (branch when not equal) command we see that the system will not perform a branch jump. The area of a distance of half a clock cycle to the right of the vertical red marker is not of importance for our logic since it reads out the prediction error of the next PC, instructed by the yellow signal at the top of the screenshot because it will be read at a later state. The read signals show that its BHT status has switched from weakly not taken into strongly not taken.

Figure 14 on page 31 shows a situation where the BHT predicts that the Branch command BEQ will not be taken but actually it will be taken. Its BHT status switches from weakly not taken to weakly taken.

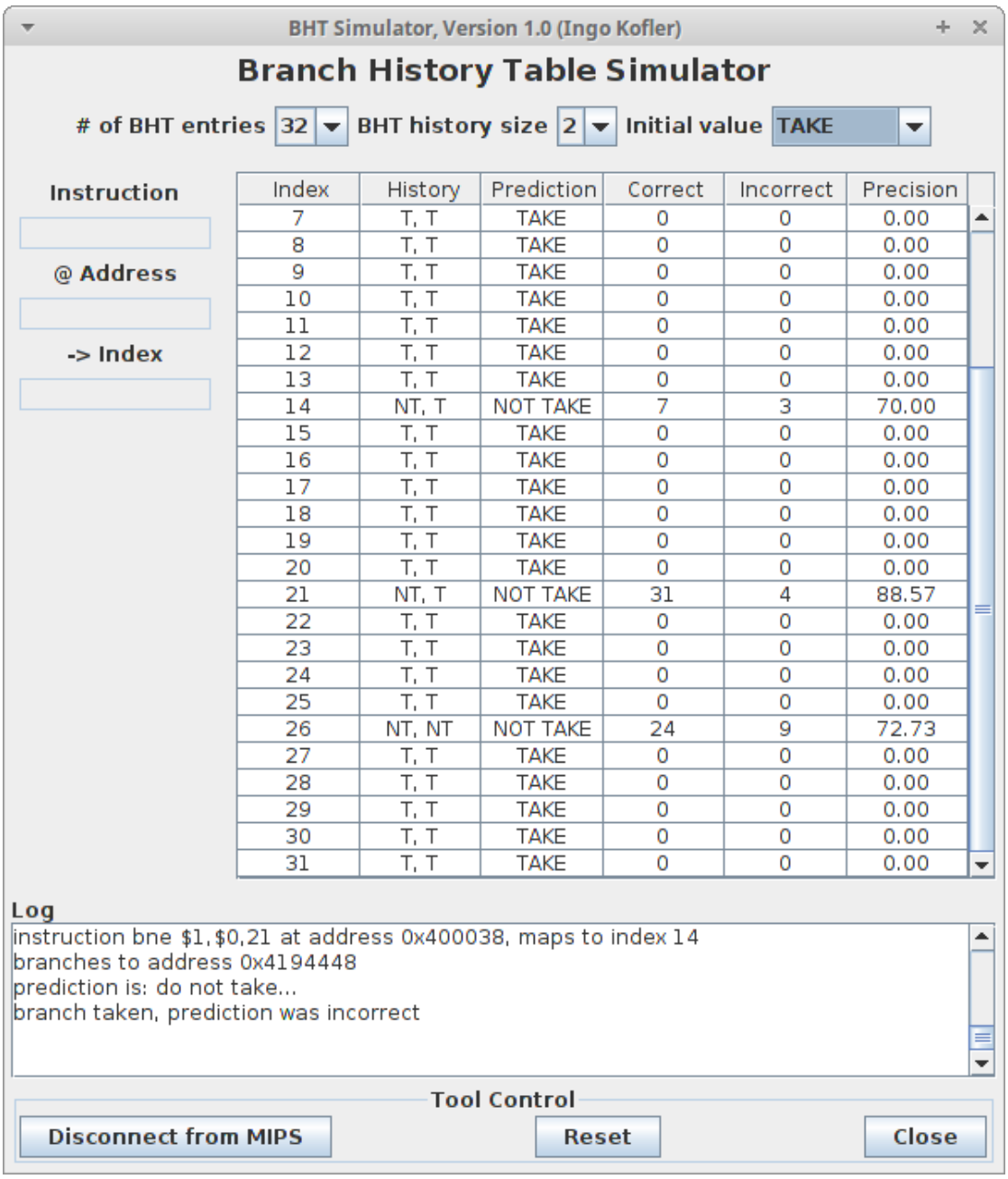


Figure 12: Screenshot of BHT Simulation in MARS using isort_pipe from lecture

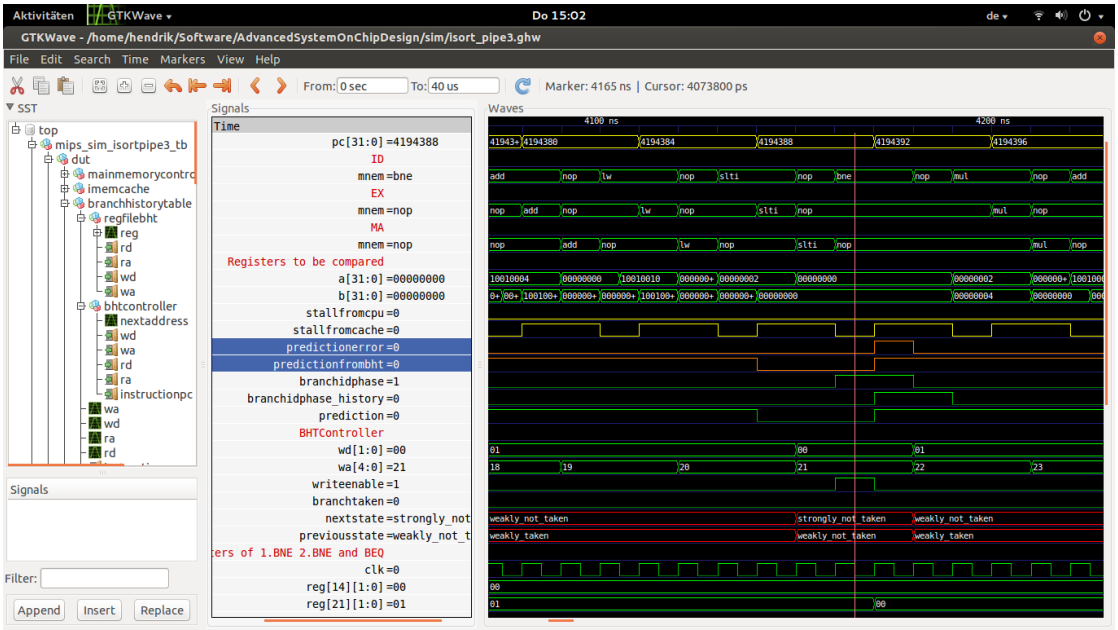


Figure 13: Behaviour of BHT without prediction error

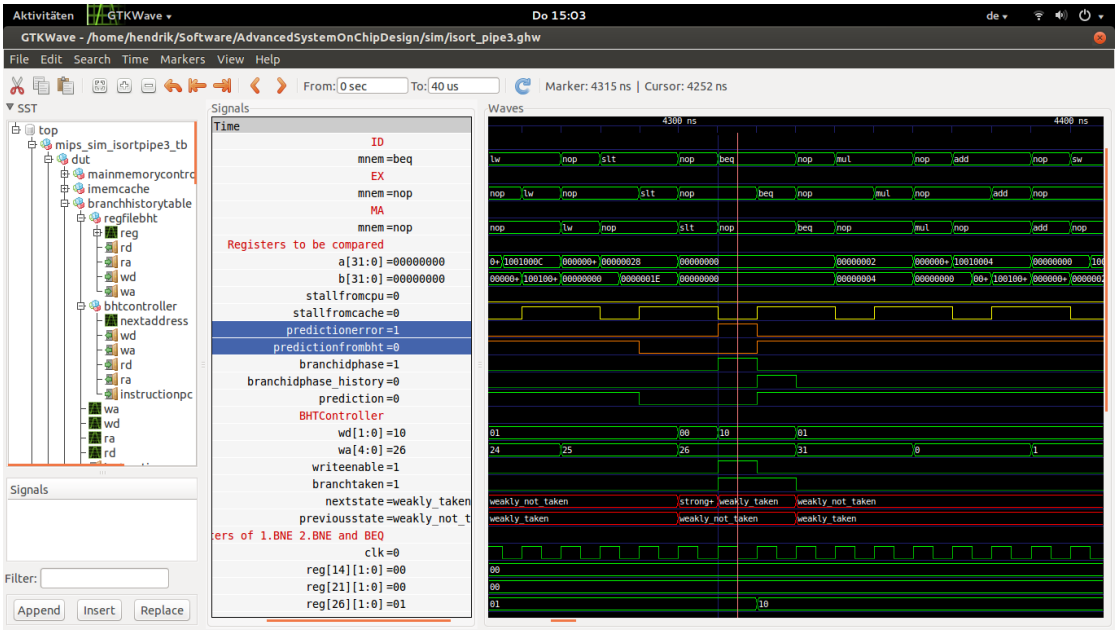
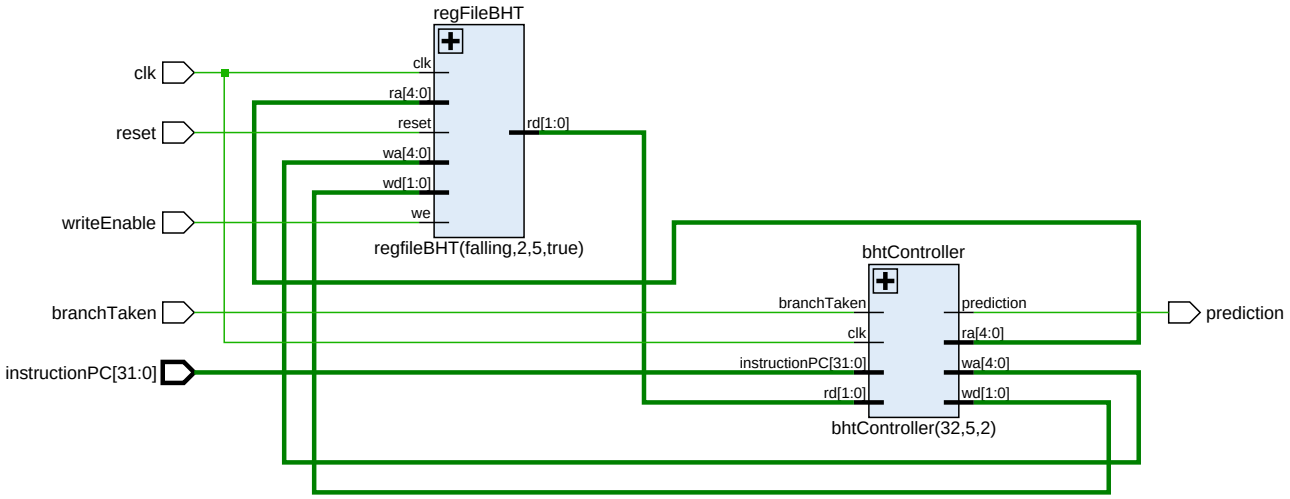


Figure 14: Behaviour of BHT with prediction error

The RTL schematic is shown in figure 15. It is essential that the BHT contains one register file *regFileBHT* and a appropriate controller *bhtController*. The register file contains for each branch instruction the saturation bit which determines whether the branch instruction should be taken or should be not taken. The controller instance manages the accesses between the register file and the CPU. Thus, the BHT returns a prediction bit which indicates whether the branch is taken or not taken. Therfor, the CPU has to pass the current pc value to the BTB controller. If a branch prediction was incorrect, then the CPU has to inform the BTB by setting the signal *writeEnable* and *branchTaken*. These two signals identicate that the correspondent register should be rewritten with the state of the saturation counter. The next state of the saturation counter depends on the current state and of the state of the input signal *branchTaken* which shows whether the last branch has been taken or not. The associated implementation of the BHT is given in file *bht.vhd* and *bhtController.vhd*.

Figure 15: RTL Schematic of BHT



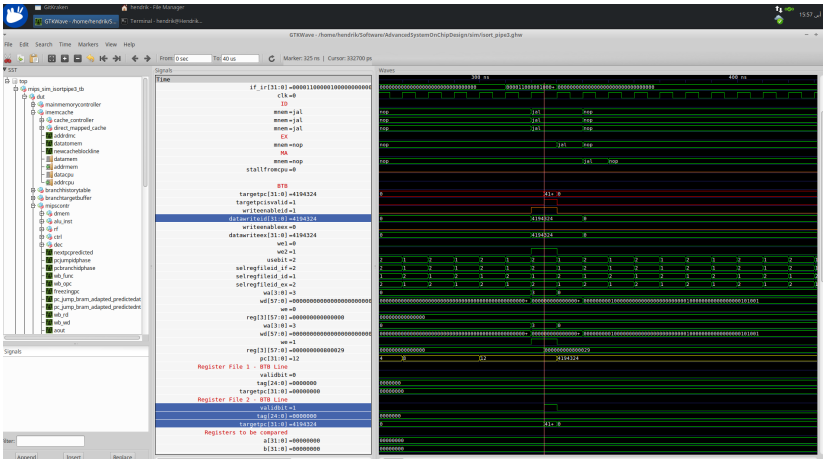


Figure 16: GTKWAVE Screenshot of BTB saving a jump's target at a falling edge

4.4 Dynamic Branch Prediction using BTB

We wanted to let our mips use a Branch Target Buffer which keeps track of all taken branches and their target Program Counters. In figure 16 on page 33 we can see how a jump command is executed for the first time and its target is saved into the BTB. In figure 17 on page 33 we can see how the process of another jump command which is processed later on is increased in speed because the command has been taken a previous time and the BTB still remembers which target adress it was using. The BTB can be read combinatorically whereas the writing process of writing into the BTB is clock cycle dependant.

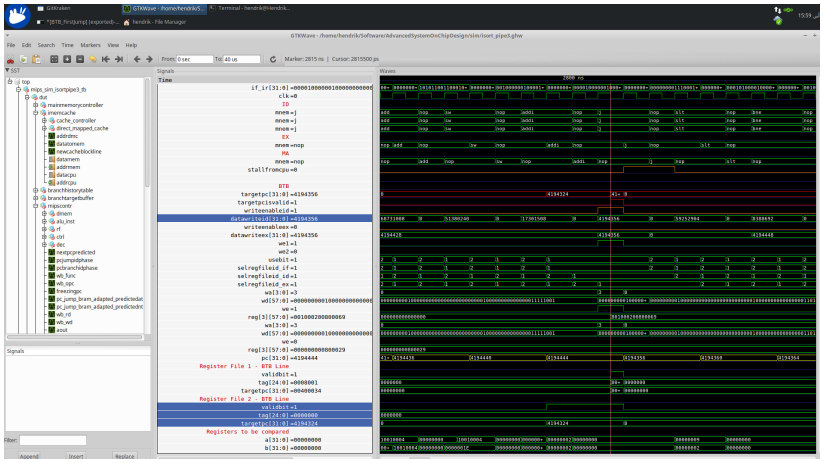
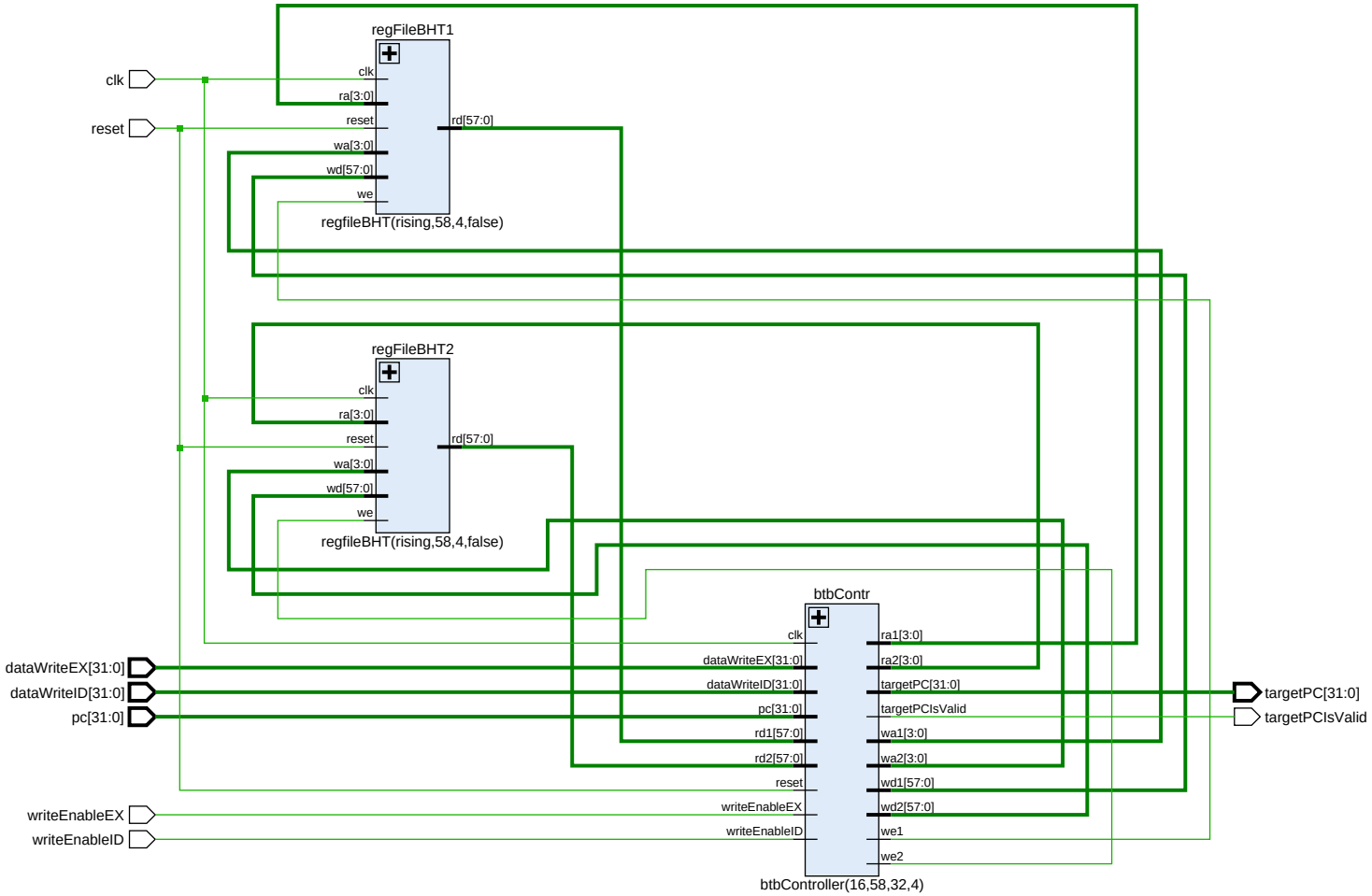


Figure 17: GTKWAVE Screenshot of a jump predicted from BTB

The implementation of the BTB is given in file *btb.vhd*. The correspondent RTL schematic is illustrated in figure 18. As you can see the BTB contains of one BTB controller and two register files since the BTB should be implemented by register files as a two way associative cache. The BTB controller realizes the behavior of the BTB. So this controller has to read both register files if a target pc should be determined. Also, the BTB controller must rewrite the correspondent register file in ID stage (jump) or in EX stage (branch). The implementation of the BTB and its BTB controller is given in file *btb.vhd* and *btbController.vhd*.

Figure 18: RTL Schematic of BTB



4.5 Lessons Learned

While doing the work of this section another problem occured. Since we were three people using three different OS one of our computers was not able to accept a correct Eclipse Sigasi Certificate on an emulated Linux machine running on a physical Macbook. The workaround at hand was to implement the code on the Mac OS, upload the code using github and pulling the modified state on the virtual Linux machine. Tests were run in the virtual box but the coding had to be done on the physical OS. This was a bit of a slow down factor since each change of code caused a push and pull action for this team member. We recommend again to use the same OS on all systems and the same Development Software on all machines. For our case we needed some workarounds.

5 Summary

5.1 How to Use the Scripts and Testbenches

After having answered different questions at different stages of the development process of our mips it turned out that a system to easily run and test new as well as old configurations of the mips was necessary. Simply switching between states within the repository was not fully satisfying our needs. Therefore a specified testbench for each program was created with corresponding configurations to chose from.

- Configuration1 is mips with pipelining, stalling and forwarding
- Configuration2 is Configuration1 with instruction cache
- Configuration3 is Configuration2 with static branch prediction
- Configuration4 is Configuration3 with Branch History Table implemented
- Configuration5 is Configuration4 with Branch Target Buffer implemented

Test program	Config c1	Config c2	Config c3	Config c4	Config c5
FAC	-	OKAY	OKAY	OKAY	
FIB	-	OKAY	OKAY	OKAY	
GCD	-	OKAY	OKAY	OKAY	
ISORT_PIPE3	OKAY	OKAY	OKAY	OKAY	

Table 5: Table showing successful testing events

Test program	Config c1	Config c2	Config c3	Config c4	Config c5
FAC	cfac1	cfac2	cfac3	cfac4	cfac5
FIB	cfib1	cfib2	cfib3	cfib4	cfib5
GCD	cgcd1	cgcd2	cgcd3	cgcd4	cgcd5
ISORT_PIPE3	cisort1	cisort2	cisort3	cisort4	cisort5

Table 6: Table showing command parameters to call run_ghdl_mips_xxx_script

The corresponding windows and linux scripts in the hdl folder to call with parameter from above are presented here.

- run_ghdl_mips_fac_linux.sh
- run_ghdl_mips_fac_windows.bat
- run_ghdl_mips_fib_linux.sh
- run_ghdl_mips_fib_windows.bat

- run_ghdl_mips_gcd_linux.sh
- run_ghdl_mips_gcd_windows.bat
- run_ghdl_mips_isortPipe3_linux.sh
- run_ghdl_mips_isortPipe3_windows.bat

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
       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 8: column-major.asm

```
#################################################################################
#  Row-major order traversal of 16 x 16 array of words.
#  Pete Sanderson
#  31 March 2007
#
#  To easily observe the row-oriented order, run the Memory Reference
#  Visualization tool with its default settings over this program.
#  You may, at the same time or separately, run the Data Cache Simulator
#  over this program to observe caching performance.  Compare the results
#  with those of the column-major order traversal algorithm.
#
#  The C/C++/Java-like equivalent of this MIPS program is:
#     int size = 16;
#     int[size][size] data;
#     int value = 0;
#     for (int row = 0; col < size; row++) {
#        for (int col = 0; col < size; col++) }
#           data[row][col] = value;
#           value++;
#        }
#     }
#
#  Note: Program is hard-wired for 16 x 16 matrix.  If you want to change this,
#        three statements need to be changed.
#        1. The array storage size declaration at "data:" needs to be changed from
#           256 (which is 16 * 16) to #columns * #rows.
#        2. The "li" to initialize $t0 needs to be changed to new #rows.
#        3. The "li" to initialize $t1 needs to be changed to new #columns.
#
        .data
data:   .word     0 : 256       # storage for 16x16 matrix of words
        .text
        li        $t0, 16       # $t0 = number of rows
        li        $t1, 16       # $t1 = number of columns
        move      $s0, $zero    # $s0 = row counter
        move      $s1, $zero    # $s1 = column counter
        move      $t2, $zero    # $t2 = the value to be stored
#  Each loop iteration will store incremented $t1 value into next element of matrix.
#  Offset is calculated at each iteration. offset = 4 * (row*#cols+col)
#  Note: no attempt is made to optimize runtime performance!
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 += column counter
        sll       $s2, $s2, 2   # $s2 *= 4 (shift left 2 bits) for byte offset
        sw        $t2, data($s2) # store the value in matrix element
        addi      $t2, $t2, 1   # increment value to be stored
#  Loop control: If we increment past last column, reset column counter and
```

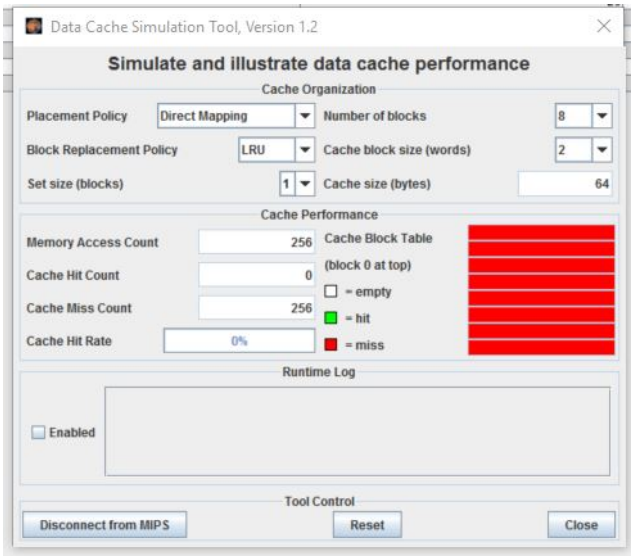


Figure 19: Column Major, Direct Mapping, Cache Block Size 2

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

Listing 9: row-major.asm

6.2 Cache Results - Snapshots

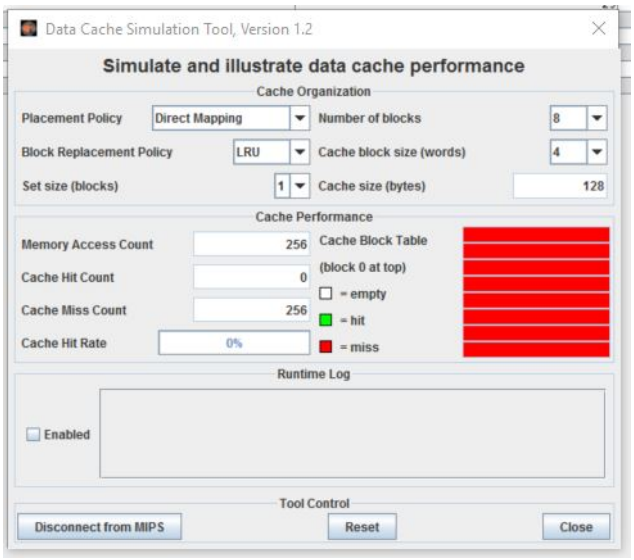


Figure 20: Column Major, Direct Mapping, Cache Block Size 4

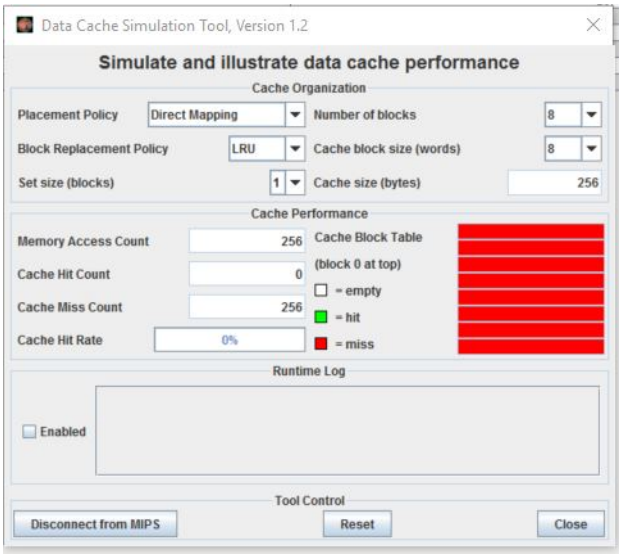


Figure 21: Column Major, Direct Mapping, Cache Block Size 8

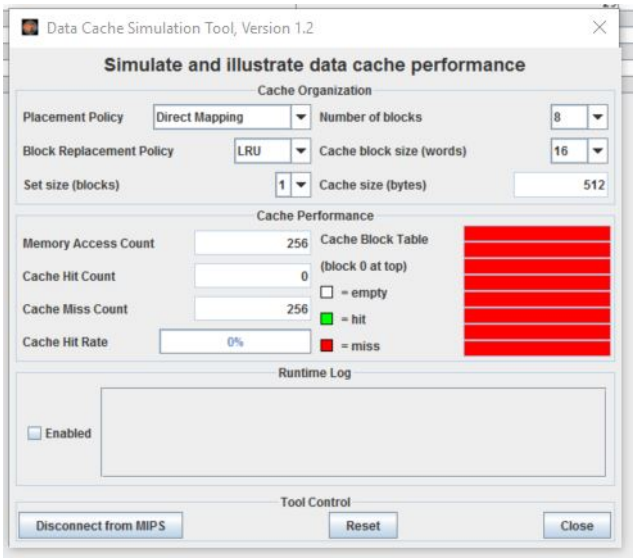


Figure 22: Column Major, Direct Mapping, Cache Block Size 16

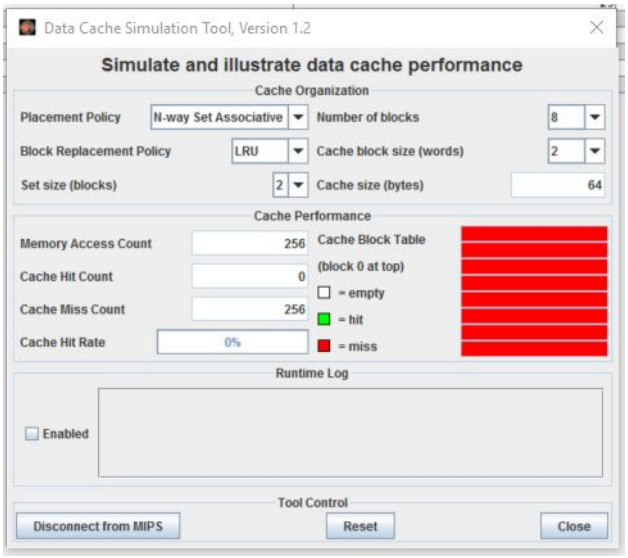


Figure 23: Column Major, 2-Way Associative, Cache Block Size 2

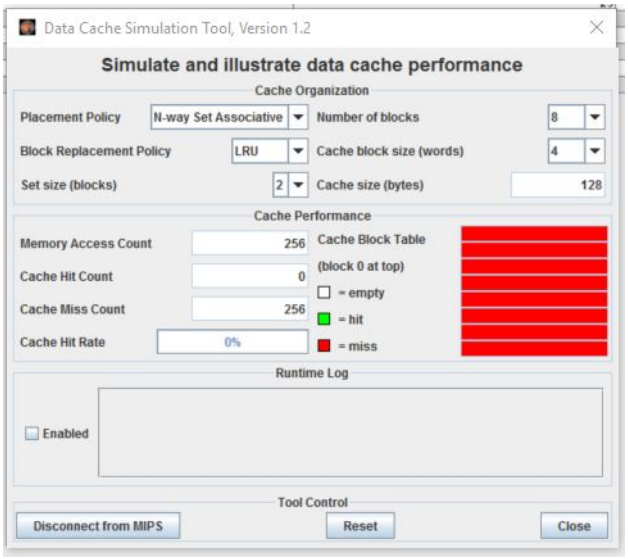


Figure 24: Column Major, 2-Way Associative, Cache Block Size 4

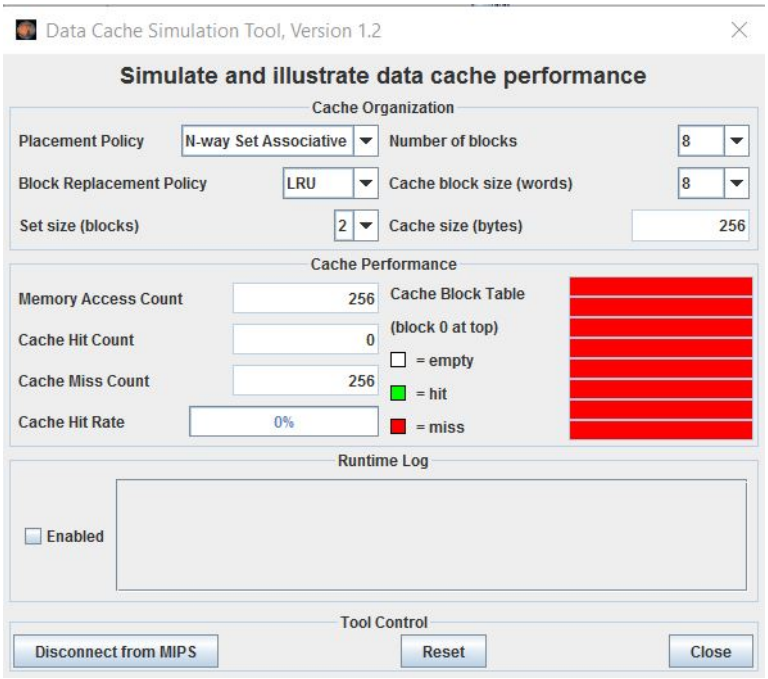


Figure 25: Column Major, 2-Way Associative, Cache Block Size 8

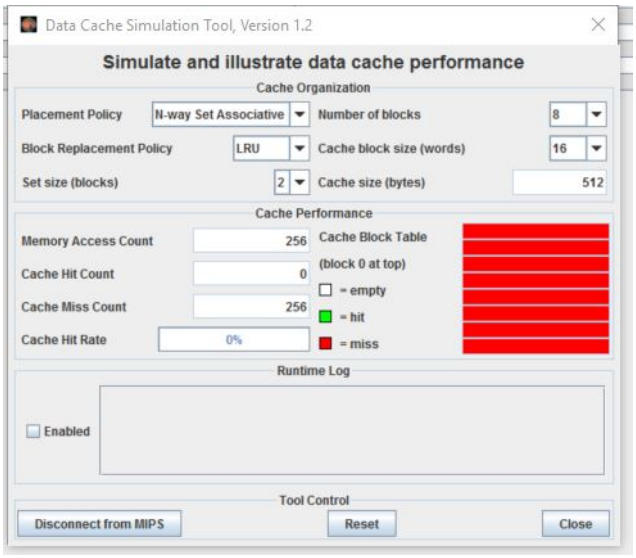


Figure 26: Column Major, 2-Way Associative, Cache Block Size 16

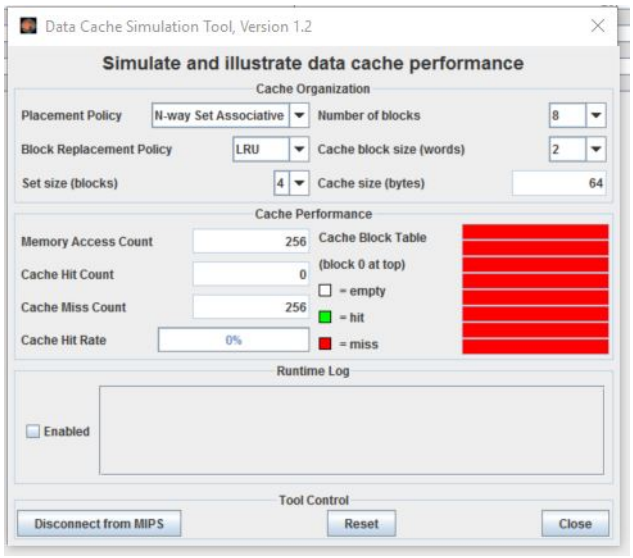


Figure 27: Column Major, 4-Way Associative, Cache Block Size 2

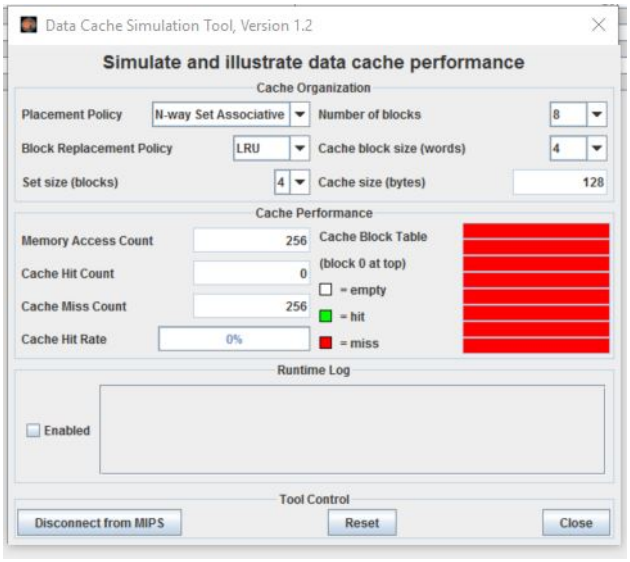


Figure 28: Column Major, 4-Way Associative, Cache Block Size 4

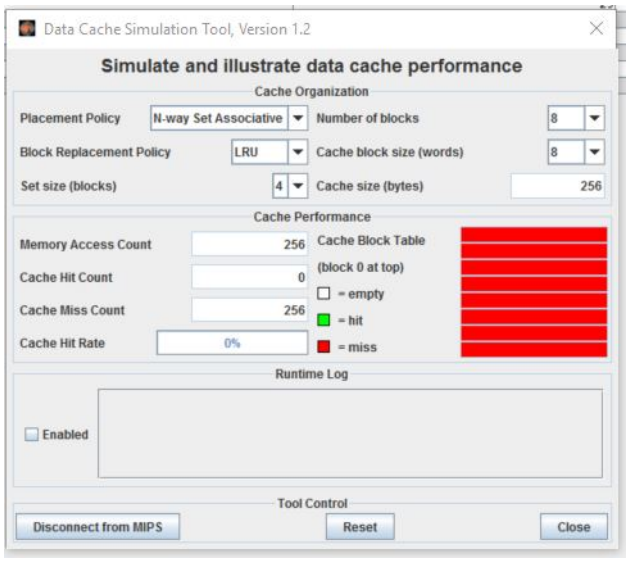


Figure 29: Column Major, 4-Way Associative, Cache Block Size 8

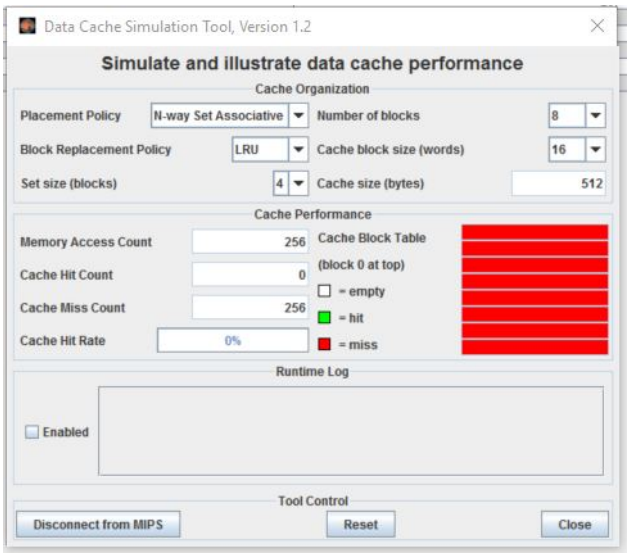


Figure 30: Column Major, 4-Way Associative, Cache Block Size 16

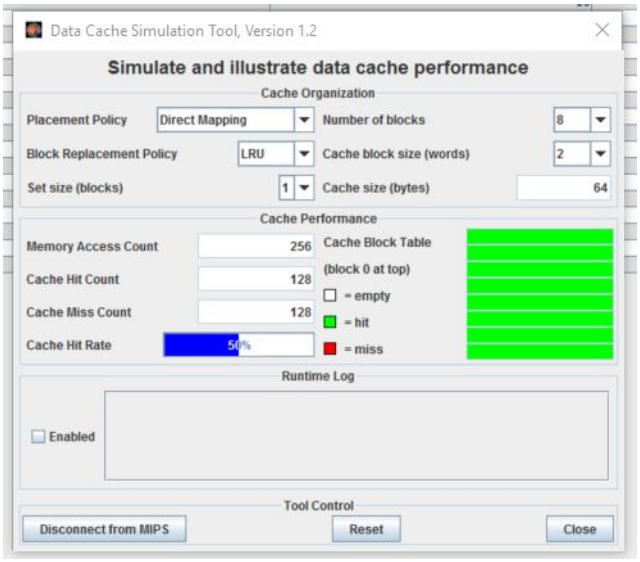


Figure 31: Row Major, Direct Mapping, Cache Block Size 2

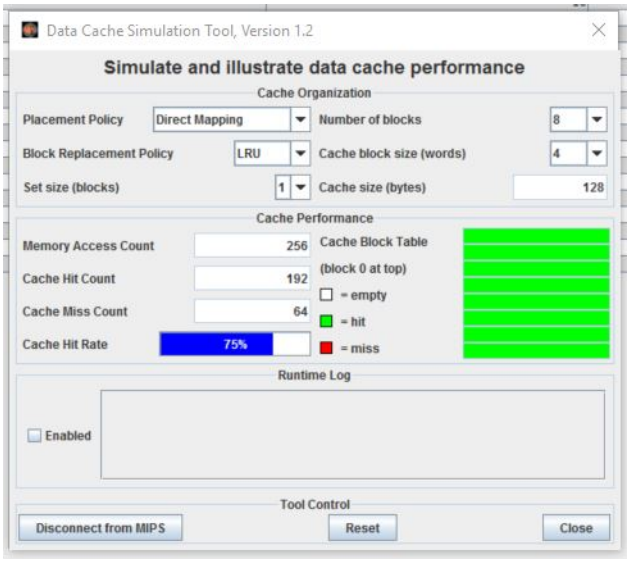


Figure 32: Row Major, Direct Mapping, Cache Block Size 4

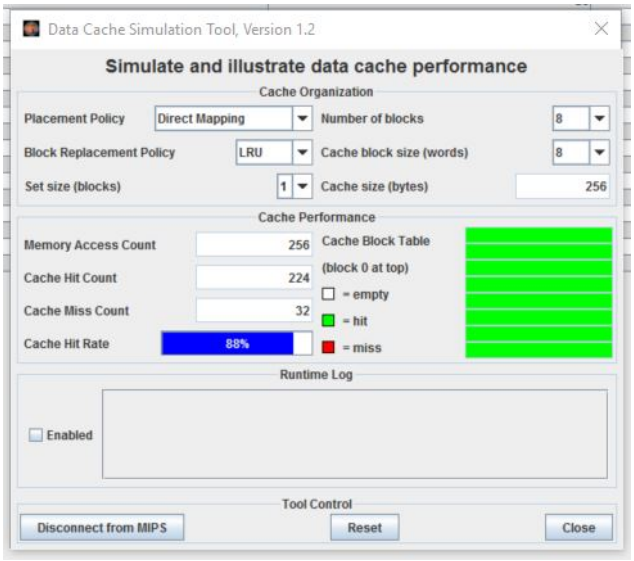


Figure 33: Row Major, Direct Mapping, Cache Block Size 8

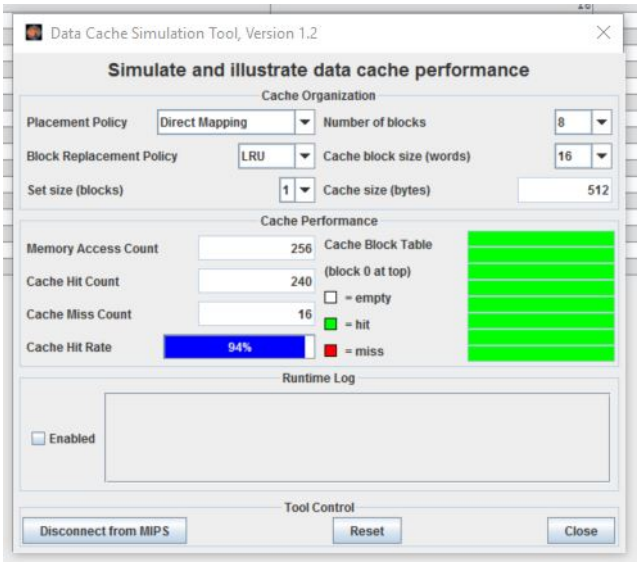


Figure 34: Row Major, Direct Mapping, Cache Block Size 16

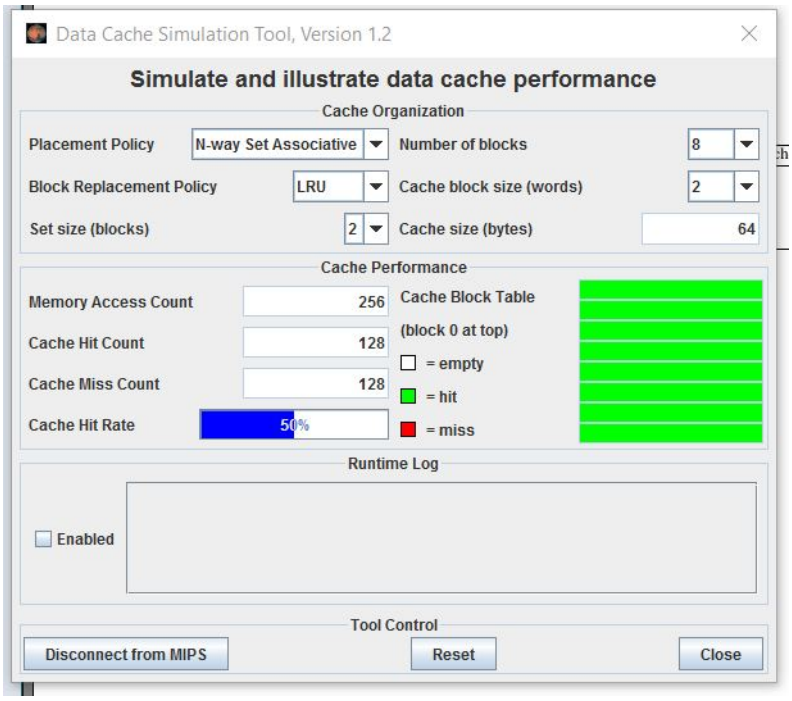


Figure 35: Row Major, 2-Way Associative, Cache Block Size 2

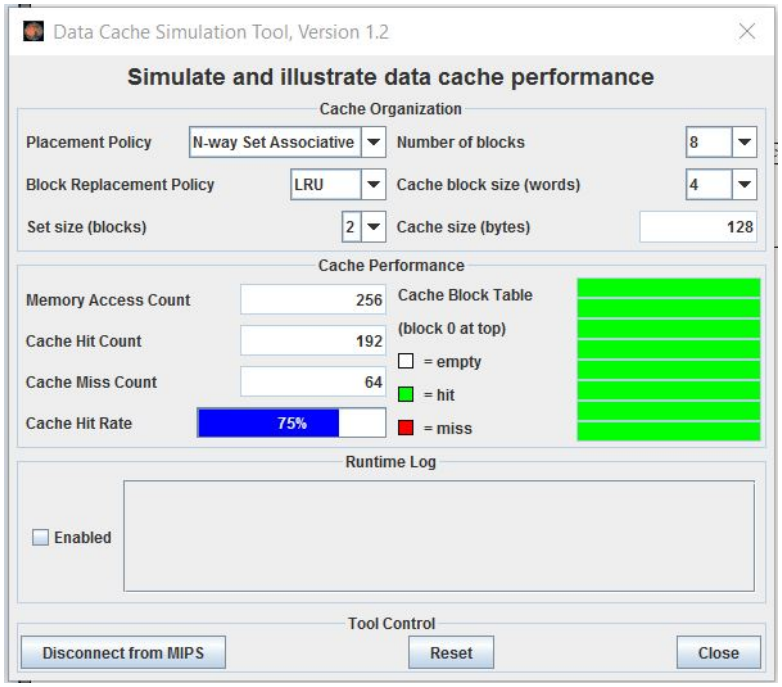


Figure 36: Row Major, 2-Way Associative, Cache Block Size 4

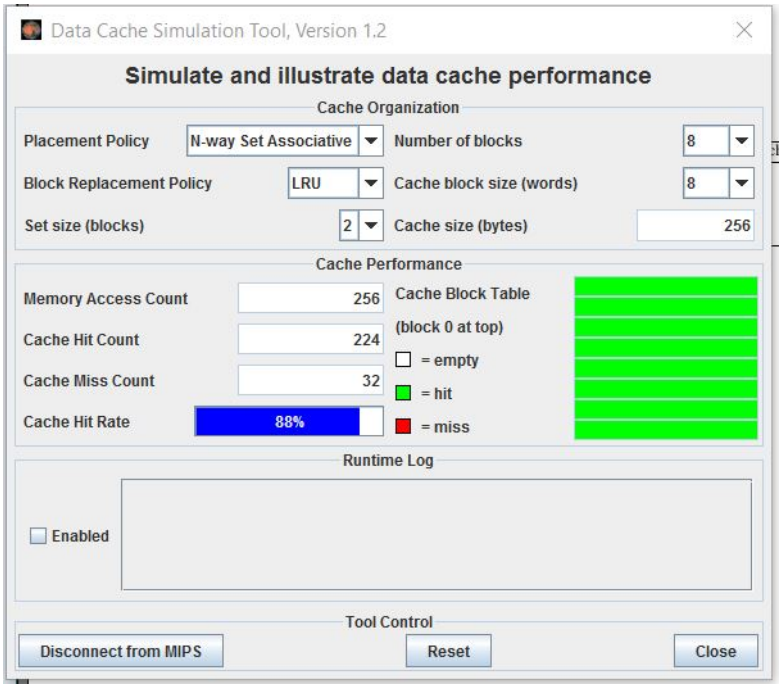


Figure 37: Row Major, 2-Way Associative, Cache Block Size 8

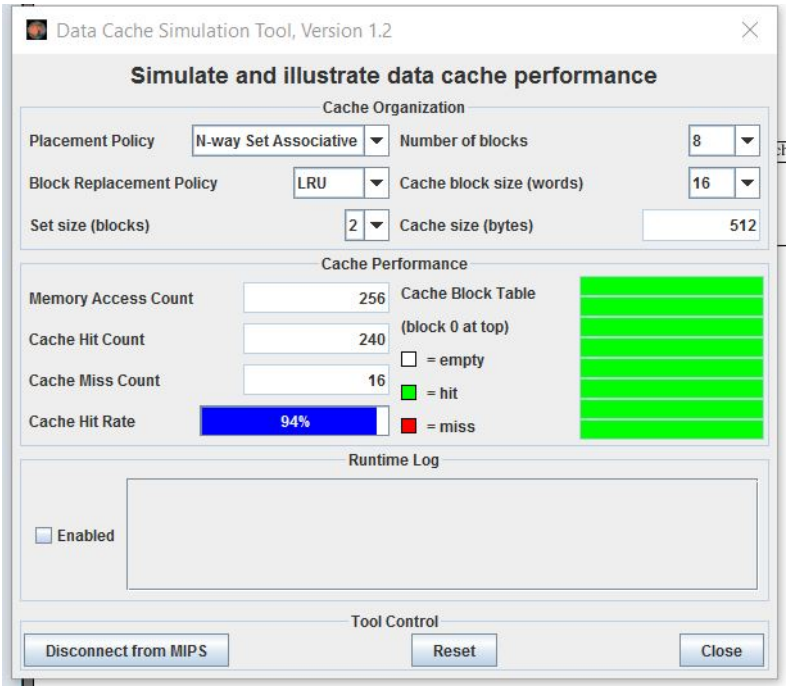


Figure 38: Row Major, 2-Way Associative, Cache Block Size 16

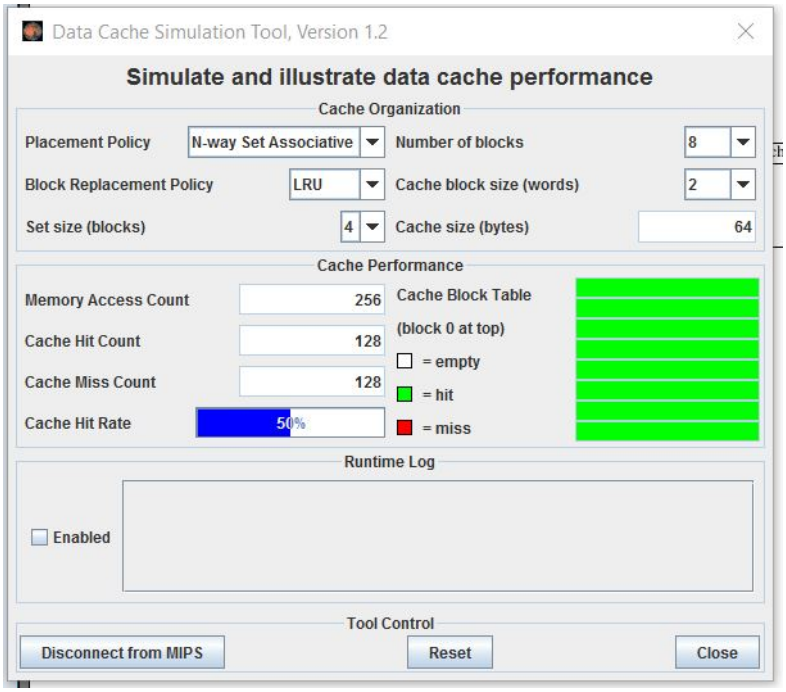


Figure 39: Row Major, 4-Way Associative, Cache Block Size 2

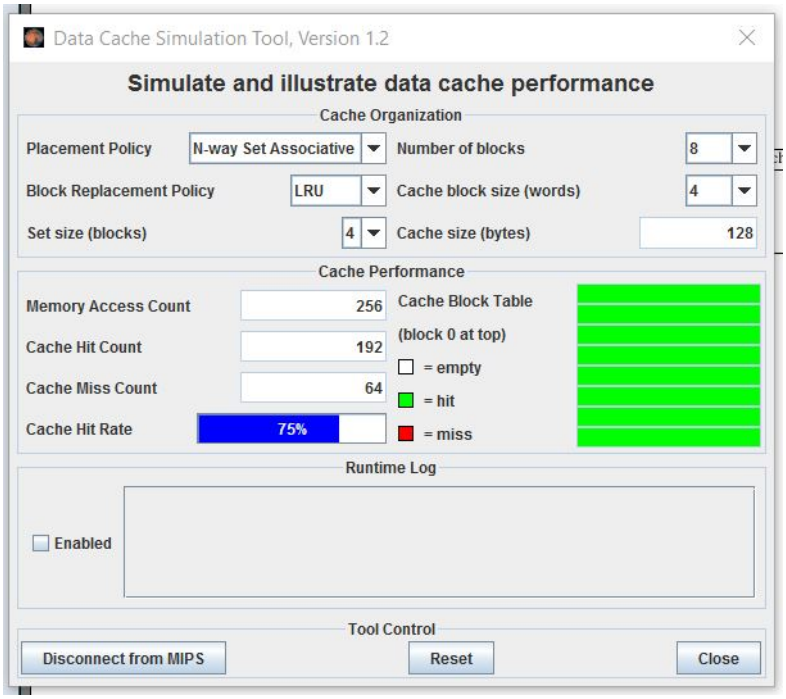


Figure 40: Row Major, 4-Way Associative, Cache Block Size 4

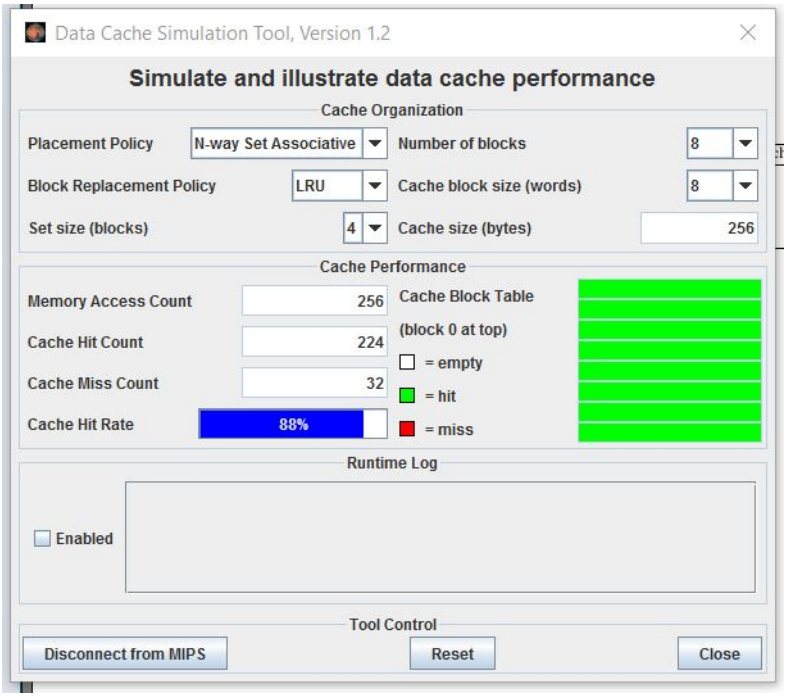


Figure 41: Row Major, 4-Way Associative, Cache Block Size 8

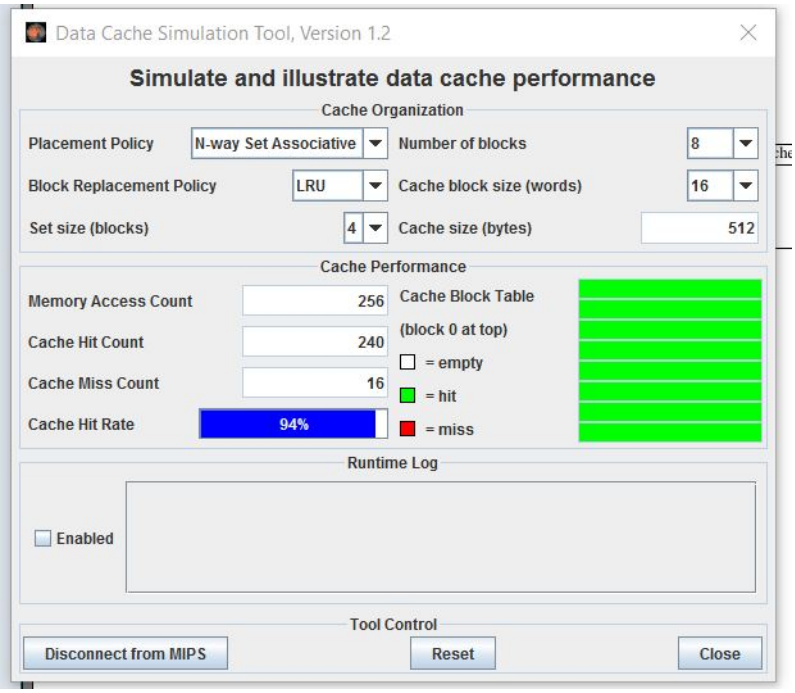


Figure 42: Row Major, 4-Way Associative, Cache Block Size 16

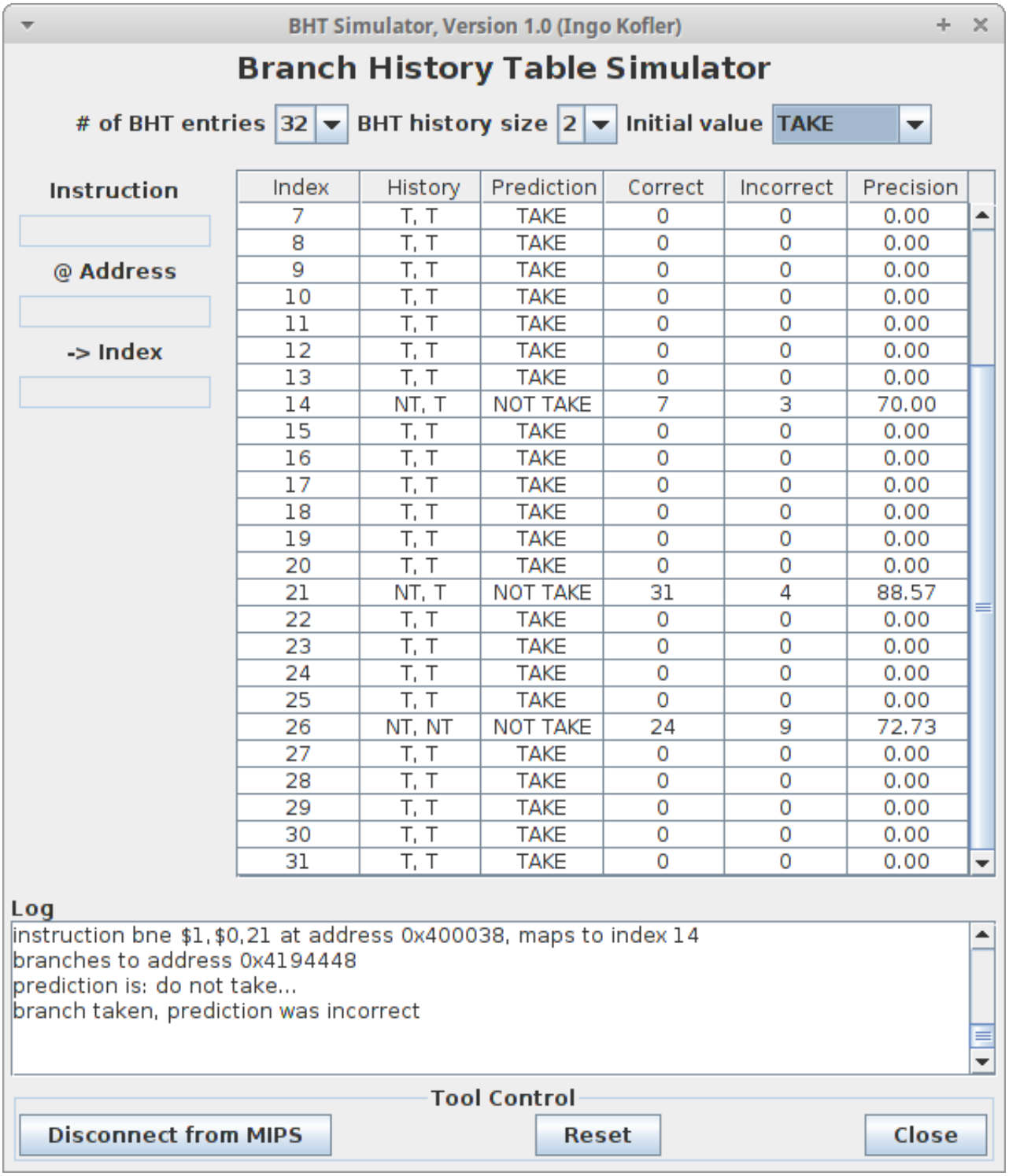


Figure 43: Screenshot of BHT Simulation in MARS using isort Size 2 Init Take

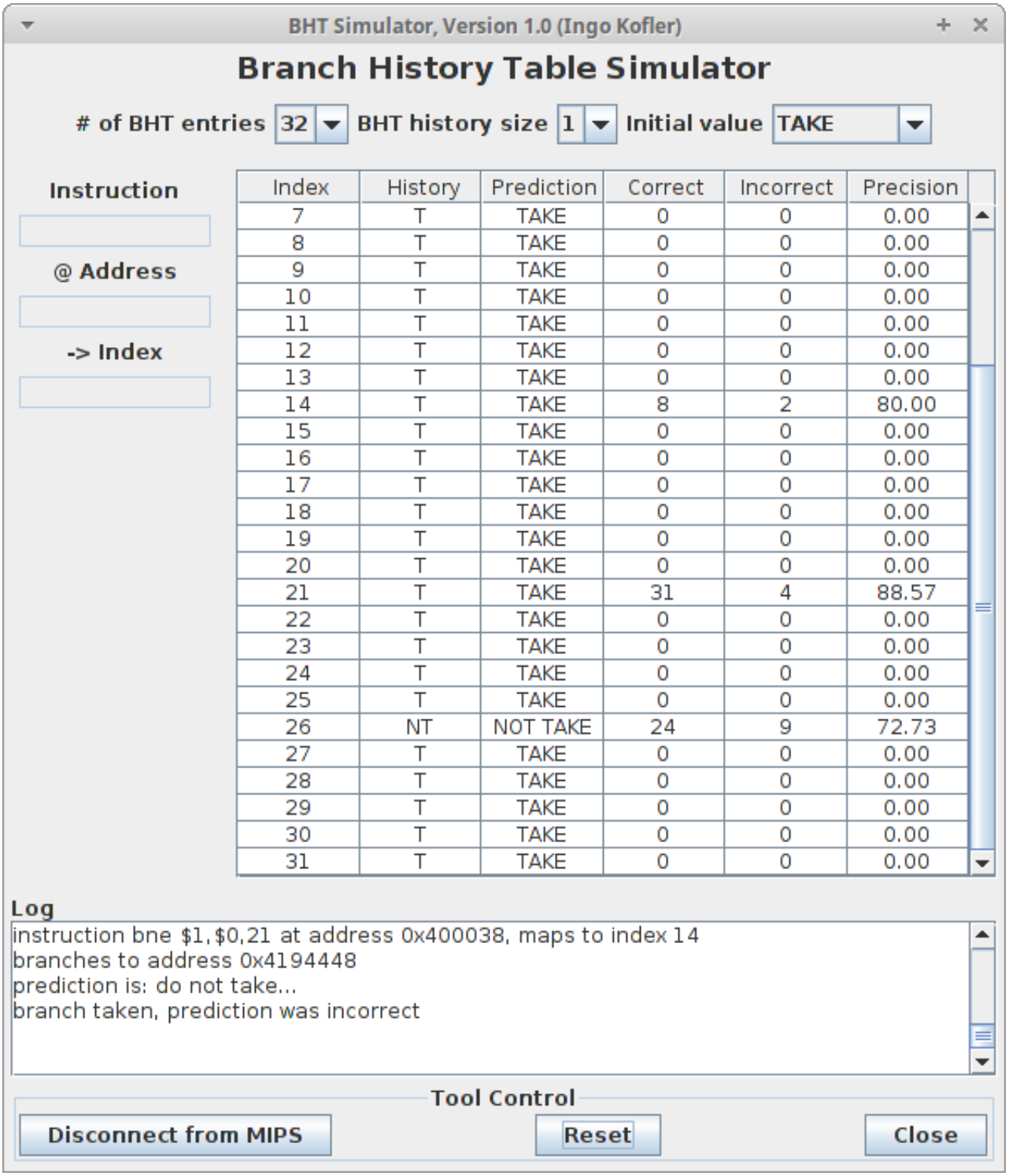


Figure 44: Screenshot of BHT Simulation in MARS using isort Size 1 Init Take

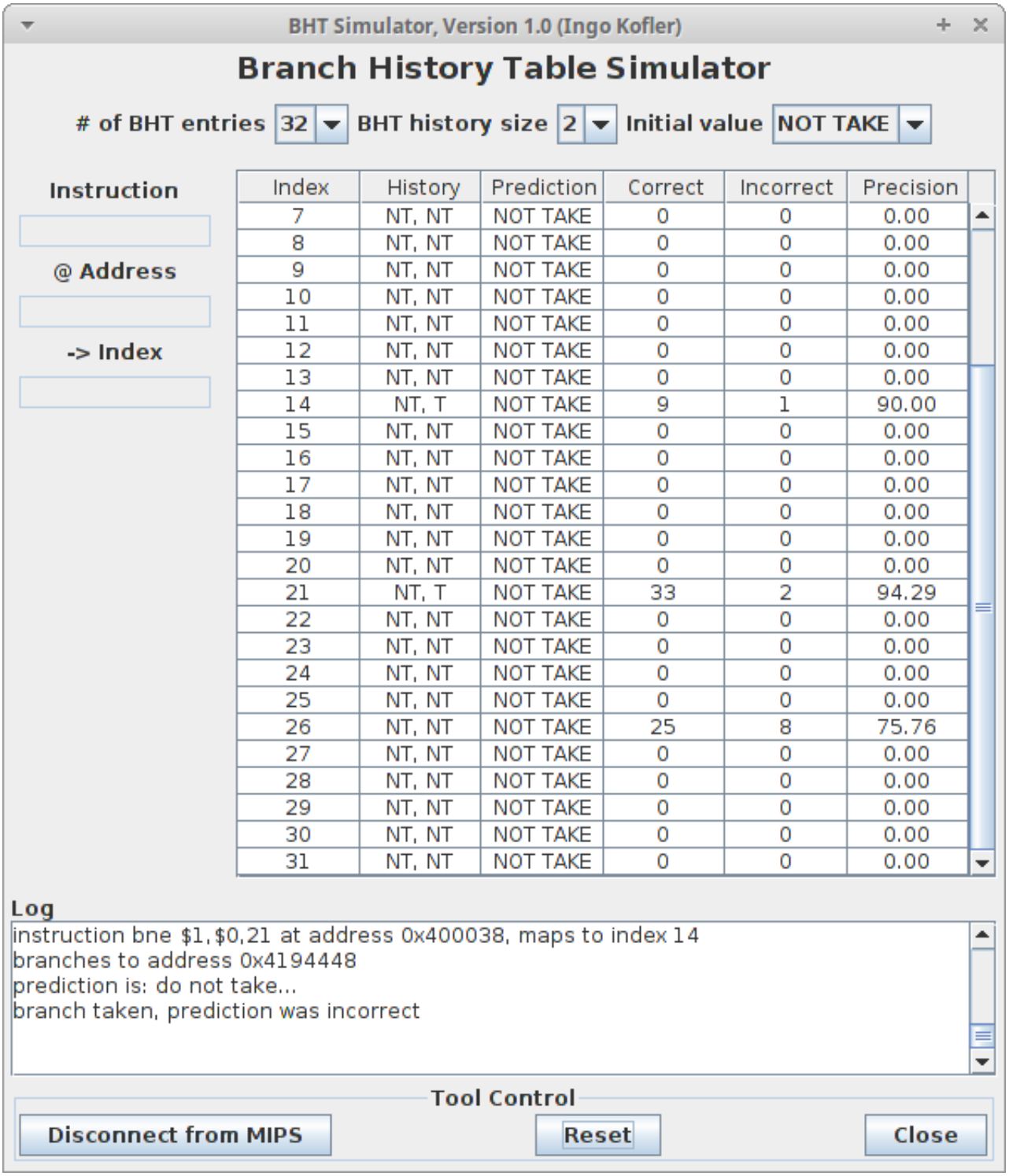


Figure 45: Screenshot of BHT Simulation in MARS using isort Size 2 Init Not Take

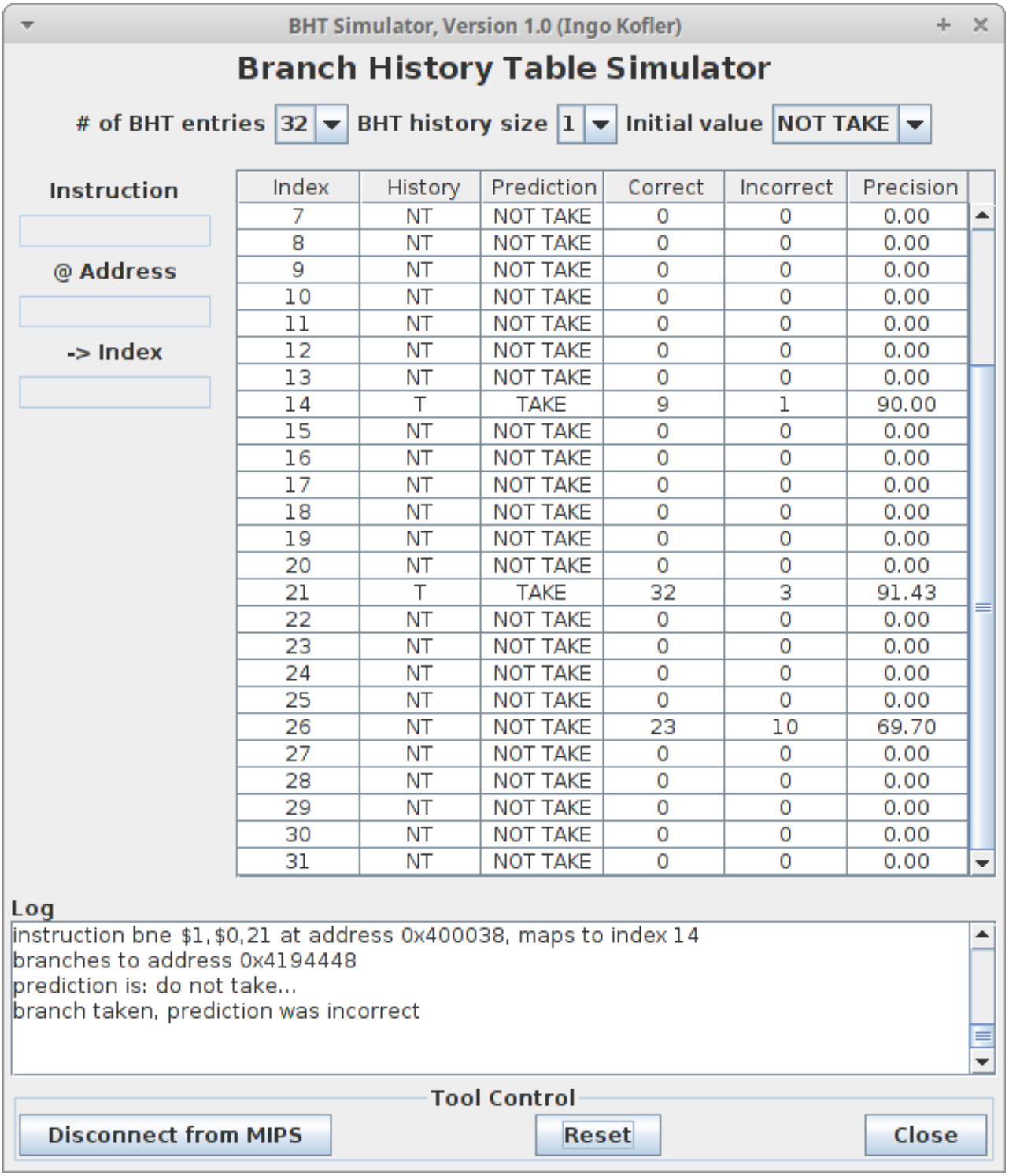


Figure 46: Screenshot of BHT Simulation in MARS using isort Size 1 Init Not Take