Lab YP5

The First Glance into Caches

MIPSfpga 2.0. Lab YP5 – The first glance into caches

1. Introduction

This lab demonstrates how the work of CPU cache can be directly observed in the most obvious and straigtforward fashion when running MIPSfpga-based system on FPGA board. During this lab a student first makes the design clock running really slow (a dozen of beats per second), and then the student's program fills a two-dimensional memory array in one of two different ways: either with moving row index first or with moving column index first. The resulting cache hit and miss patterns are clearly visible on blinking LED, which is connected to a signal that indicates cache fill line request. The student can see that these patterns are different for raw-first and column-first runs.

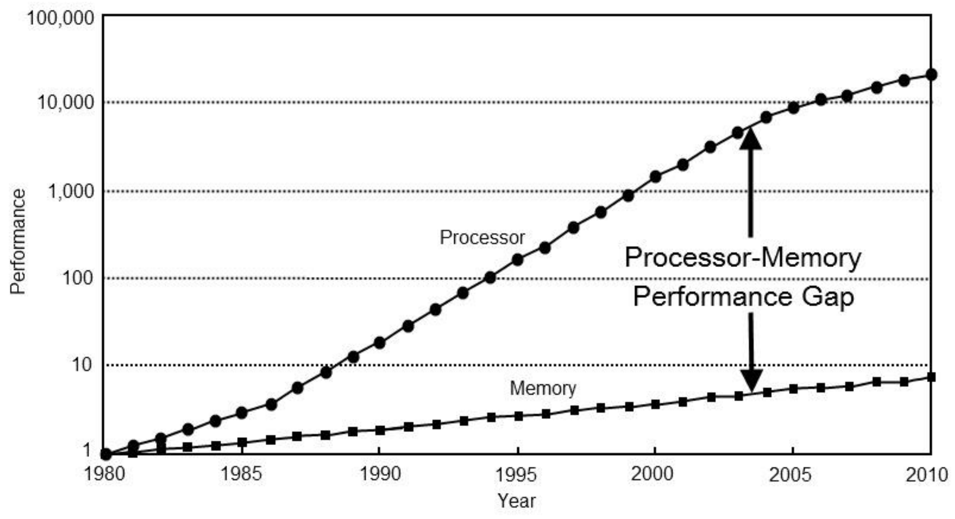
Each cache line in MIPSfpga CPU cache holds four 32-bit words. As a result, when the program fills the memory array by moving column index first, the pattern is going to be "cache miss – hit – hit – hit – miss – hit – hit – hit – miss ...". However when the program fills the array by moving raw index first, the pattern is going to be a series of cache misses followed by a long series of cache hits, since a large portion of the array is already in the cache.

This lab does not attempt to measure, quantify or analyze cache behavior in details. The only goal of this lab is to create "A-ha" moment for a student who wants to visualize for himself the role a CPU cache plays in the computer system. The precise analysis requires using different methods, for example performance counters, as described in other cache-related labs in MIPSfpga 2.0 package.

2. The theory of operation

Why do we need CPU cache? It solves the problem of so-called "processor-memory performance gap". The issue is: during the past decades the speed of CPUs grew faster than the speed of dynamic memory, used for storing a large amount of data. This difference in growth is illustrated on Figure 1. As a result, without cache, a typical transaction (fetch, load or store) from CPU to the memory could take the same amount of time (the same number of clock cycles) as tens, sometimes a hundred or more arithmetic operations inside the CPU. It means that with a typical memory-intensive application a CPU without caches would waste most program execution time idling, waiting to complete the memory operations.

Figure 1. Processor-memory performance gap, from Computer Architecture: A Quantitative Approach by David A. Patterson and John L. Hennessy



One of the best descriptions of cache used in MIPS CPUs is in book [See MIPS Run, Second Edition by Dominic Sweetman](). A fragment of this description is below:

The cache's job is to keep a copy of memory data that has been recently read or written, so it can be returned to the CPU quickly. For L1 caches, the read must complete in a fixed period of time to keep the pipeline running.

MIPS CPUs always have separate L1 caches for instructions and data (I-cache and D-cache, respectively) so that an instruction can be read and a load or store done simultaneously.

Conceptually, a cache is an associative memory, a chunk of storage where data is deposited and can be found again using an arbitrary data pattern as a key. In a cache, the key is the full memory address. Produce the same key back to an associative memory and you'll get the same data back again. A real associative memory will accept a new item using an arbitrary key, at least until it's full; however, since a presented key has to be compared with every stored key simultaneously, a genuine associative memory of any size is either hopelessly resource hungry, slow, or both.

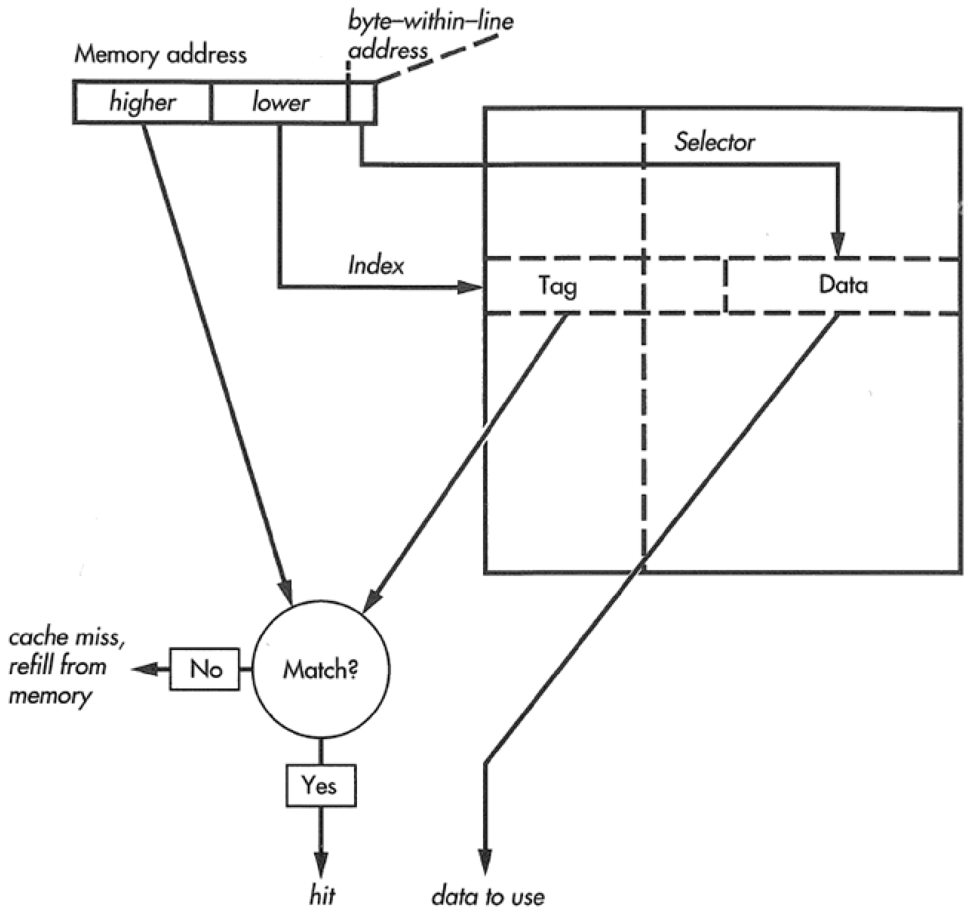
So how can we make a useful cache that is fast and efficient? **Figure 2** shows the basic layout of the simplest kind of cache, the direct-mapped cache used in most MIPS CPUs up to the 1992 generation.

The direct-mapped arrangement uses a simple chunk of high-speed memory (the cache store) indexed by enough low address bits to span its size. Each line inside the cache store contains one or more words of data and a cache tag field, which records the memory address where this data belongs.

On a read, the cache line is accessed, and the tag field is compared with the higher addresses of the memory address; if the tag matches, we know we've got the right data and have "hit" in the cache. Where there's more than one word in the line, the appropriate word will be selected based on the very lowest address bits.

If the tag doesn't match, we've missed and the data will be read from memory and copied into the cache. The data that was previously held in the cache is simply discarded and will need to be fetched from memory again if the CPU references it.

Figure 2. Direct-mapped cache.

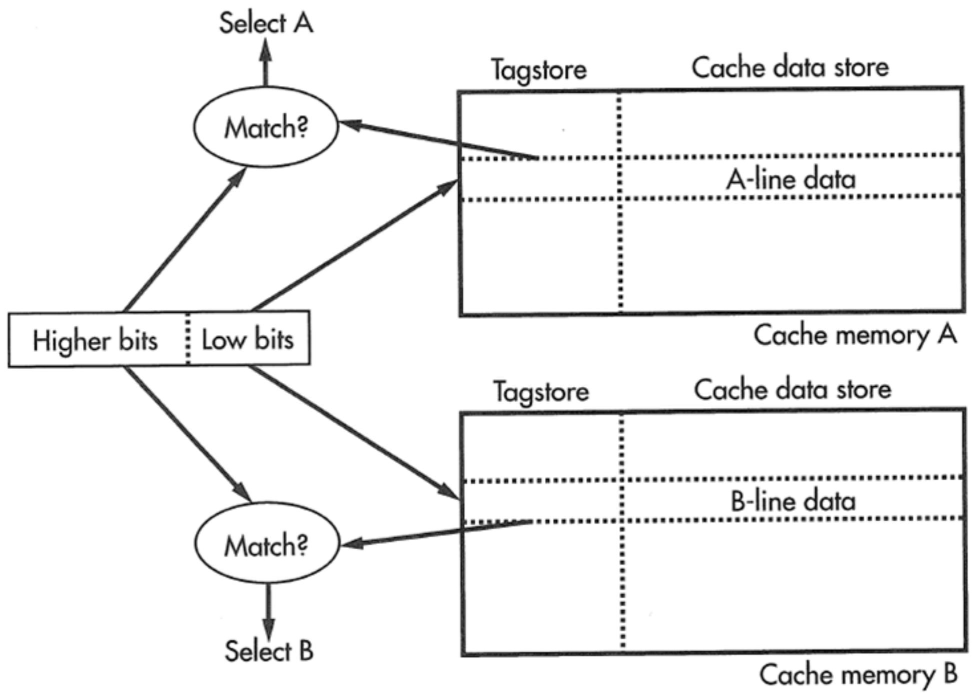


A direct-mapped cache like this one has the property that, for any given memory address, there is only one line in the cache store where that data can be kept.[1] That might be good or bad; it's good because such a simple structure will be fast and will allow us to run the whole CPU faster. But simplicity has its bad side too: If your program makes repeated reference to two data items that happen to share the same cache location (presumably because the low bits of their addresses happen to be close together), then the two data items will keep pushing each other out of the cache and efficiency will fall drastically. A real associative memory wouldn't suffer from this kind of thrashing, but it is too slow and expensive.

A common compromise is to use a two-way set-associative cache—which is really just a matter of running two direct-mapped caches in parallel and looking up memory locations in both of them, as shown in Figure 3.

[1]. In a fully associative memory, data associated with any given memory address (key) can be stored anywhere; a direct-mapped cache is as far from being content addressable as a cache store can be.

Figure 3. Two-way set-associative cache.



Now we've got two chances of getting a hit on any address. Four-way setassociative caches (where there are effectively four direct-mapped subcaches) are also common in on-chip caches.

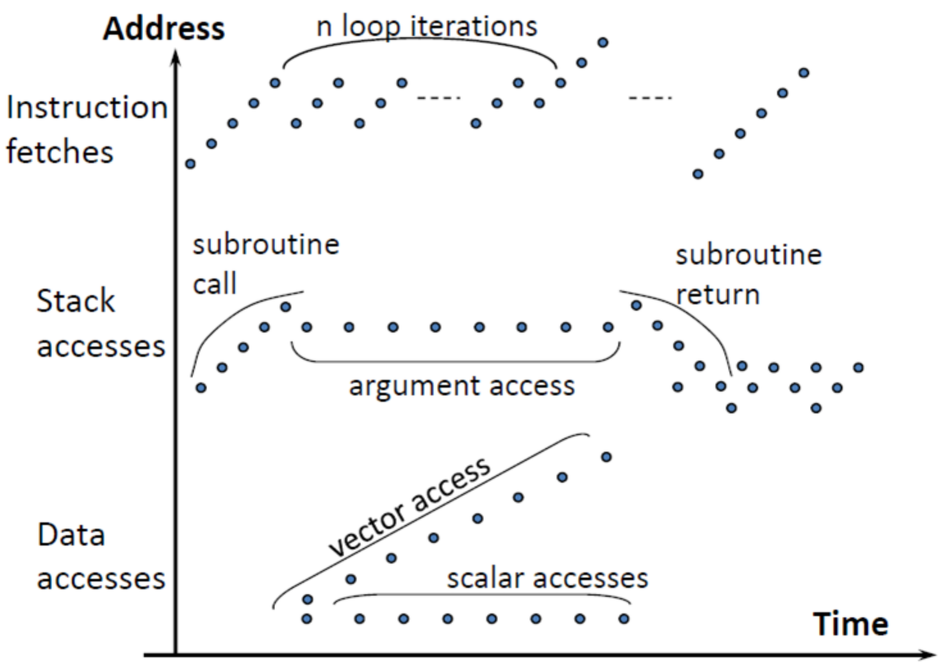
In a multiway cache there's more than one choice of the cache location to be used in fixing up a cache miss, and thus more than one acceptable choice of the cache line to be discarded. The ideal solution is probably to keep track of accesses to cache lines and pick the "least recently used" ("LRU") line to be replaced, but maintaining strict LRU order means updating LRU bits in every cache line every time the cache is read. Moreover, keeping strict LRU information in a more-than-four-way set-associative cache becomes impractical. Real caches often use compromise algorithms like "least recently filled" to pick the line to discard.

CPU caches exploit so-called principle of locality of memory access patterns found in almost all programs. Below is the explanation for two major types of locality from Wikipedia and Figure 4 that illustrates the patterns with more specifics. The understanding of this principle is relevant to this lab because it explains, for example, why MIPSfpga cache loads four words on cache miss (the cache line) rather than just one word.

From https://en.wikipedia.org/wiki/Locality_of_reference:

- Temporal locality: If at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future.
- Spatial locality: If a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future.

Figure 4. Memory access patterns from Computer Architecture course by David Wentzlaff from Princeton University. 2011.

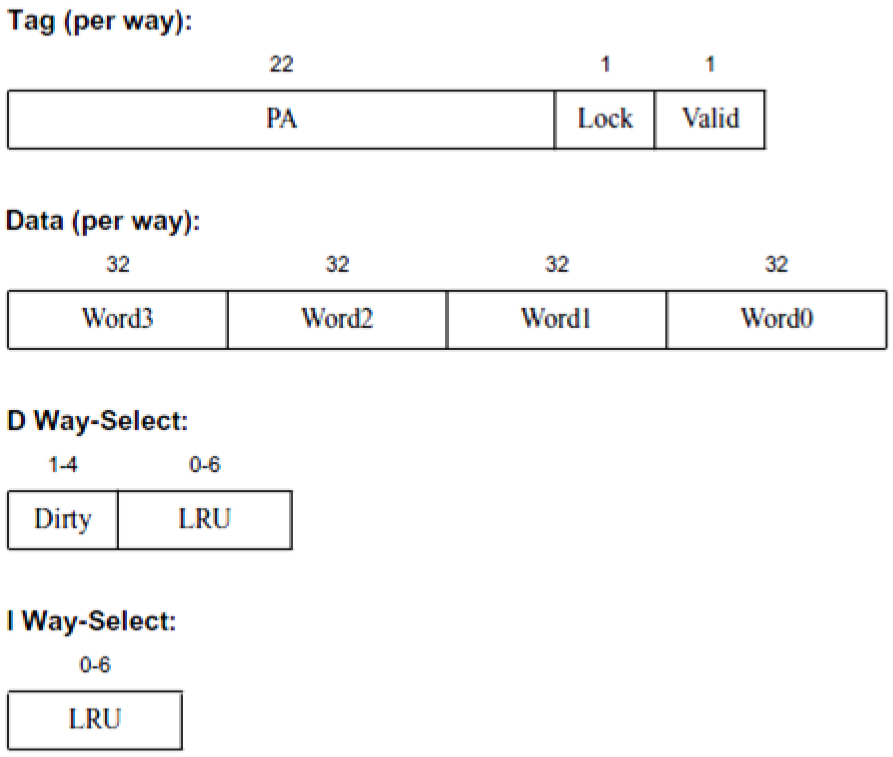


Finally, Figure 5 shows the structures associated with each cache line in MIPS microAptiv UP processor core. The particular configuration of this core used in MIPSfpga has 2-way set-associativity for both instruction and data caches; each cache way has size of 2 KB in both I-cache and D-cache. The parameters of caches used during the synthesis can be found in Verilog header file *m14k_config.vh* located in the directory *core_rtl*:

File *core_rtl/m14k_config.vh*

```
// ***********************************************************************
// Cache module parameters
// Not used with scache or cache stub modules
// ***********************************************************************
// Cache Associativity
//      1-4 way set associative

`define M14K_ICACHE_ASSOC 2
`define M14K_DCACHE_ASSOC 2

// Cache Way Size
//      Size/Way in KB
//      1,2,4,8,16 KB

`define M14K_ICACHE_WAYSIZE 2
`define M14K_DCACHE_WAYSIZE 2


// ***********************************************************************
// Cache Controller parameters
// ***********************************************************************
// ! If changing manually, remember to change WS Ram Width below to match !
// Maximum Cache Associativity
//      1-4 way set associative (including Spram)

`define M14K_MAX_IC_ASSOC 2
`define M14K_MAX_DC_ASSOC 2
```

Figure 5. The structures associated with each cache line in MIPS microAptiv UP processor core processor core.



3. Lab steps

This section outlines the sequence of steps, necessary to complete the lab. Almost all generic steps in this lab are the same as in *MIPSfpga 2.0 Lab YP1. Using MIPSfpga with Serial Loader Flow that does not require BusBlaster board and OpenOCD software*. Such generic steps are not described in this section. Only the steps different from *Lab YP1* are explained in details.

3.1. Connect the board to the computer

For *Digilent* boards, such as *Nexys4*, *Nexys4 DDR* or *Basys3*, this step is obvious.
For *Altera/Terasic* boards some additional steps required:

1. Connect USB-to-UART connector to FPGA board. Either *FT232RL* or *PL2303TA* that you can by from AliExpress or RadioShack will do the job. *TX* output from the connector (green wire on *PL2303TA*) should go to pin 3 from right bottom on Terasic DE0, DE0-CV, DE1, DE2-115 (right top on DE0-Nano) and *GND* output (black wire on *PL2303TA*) should be connected to pin 6 from

right bottom on Terasic DE0, DE0-CV, DE1, DE2-115 (right top on DE0-Nano). Please consult photo picture in *Lab YP1* to avoid short-circuit or other connection problems.
2. For *FT232RL* connector: make sure to set 3.3V/5V jumper on *FT232RL* part to 3.3V.
3. For the boards that require external power in addition to the power that comes from USB, connect the power supply. The boards that require the extra power supply include *Terasic DE2-115*.
4. Connect FPGA board to the computer using main connection cable provided by the board manufacturers. Make sure to put USB cable to the right jack when ambiguity exists (such as in *Terasic DE2-115* board).
5. Make sure to power the FPGA board (turn on the power switch) before connecting the UART cable from USB-to-UART connector to the computer. Failing to do so may result in electric damage to the board.
6. Connect USB-to-UART connector to FPGA board.

3.2. Select the appropriate hardware system configuration for the lab

Before running synthesis it is necessary to review and possibly modify the file *system_rtl/mfp_ahb_lite_matrix_config.vh* that includes a set of Verilog *`define* statements that determine the functionality of the synthesized MIPSfpga system. The configuration should be the following:

File *system_rtl/mfp_ahb_lite_matrix_config.vh*

```
//
//  Configuration parameters
//

// `define MFP_USE_WORD_MEMORY
// `define MFP_INITIALIZE_MEMORY_FROM_TXT_FILE
   `define MFP_USE_SLOW_CLOCK_AND_CLOCK_MUX
   `define MFP_USE_UART_PROGRAM_LOADER
// `define MFP_DEMO_LIGHT_SENSOR
   `define MFP_DEMO_CACHE_MISSES
// `define MFP_DEMO_PIPE_BYPASS
```

3.3. Run the synthesis and configure the FPGA with the synthesized MIPSfpga system

This step is identical to the synthesis step in *Lab YP1*

3.4. Go to the lab directory and clean it up

Under Windows:

```
cd programs\05_cache_misses
00_clean_all.bat
```

Under Linux:

```
cd programs/05_cache_misses
./00_clean_all.sh
```

3.5. Review the lab program code

The main program is located in file *programs/05_cache_misses/main.c*.

The loop at the beginning (*"while ((MFP_SWITCHES & 4) == 0) ;"*) is needed so that the program can start with high clock speed (25 MHz) and the switch to very low clock speed (12.5 Hz) happens only after you first switch the speed by turning on switch one and then let the program to continue by turning on switch two. For more details about the switchable clock see *MIPSfpga 2.0 Lab YP2. Using switchable clock to observe the CPU cycle-by-cycle*.

The program has two variants – with uncommented *"a [i][j] = i + j;"* and with uncommented *"a [j][i] = i + j;"*. For the first run make sure *"a [i][j] = i + j;"* is uncommented and *"a [j][i] = i + j;"* is commented:

File *programs/05_cache_misses/main.c*

```
#include "mfp_memory_mapped_registers.h"

int a [8][8];

int main ()
{
    int n = 0;
    int i, j;

    // Wait for switch 2

    while ((MFP_SWITCHES & 4) == 0)
        ;


    for (i = 0; i < 8; i ++)
        for (j = 0; j < 8; j ++)
              a [i][j] = i + j;
            // a [j][i] = i + j;

    return 0;
}
```

3.6. Prepare the first software run

Following the procedure described in *Lab YP1*, compile and link the program, generate Motorola S-Record file and upload this file into the memory of the synthesized MIPSfpga-based system on the board.

Under Windows:

1. cd programs\05_cache_misses
2. run 02_compile_and_link.bat
3. run 08_generate_motorola_s_record_file.bat
4. run 11_check_which_com_port_is_used.bat
5. edit 12_upload_to_the_board_using_uart.bat based on the result from the previous step – set the working port in "set a=" assignment.

6. Make sure the switches 0, 1, 2 on FPGA board are turned off. Switches 0 and 1 control the speed of the clock, while switch 2 gates the program execution – see *3.5. Review the lab program code*. If the switches 0 and 1 are not off, the loading through UART is not going to work; if the switch 2 is not off, the program will execute before you have a chance to change the clock speed.
7. run 12_upload_to_the_board_using_uart.bat

Under Linux:

If uploading program to the board first time during the current Linux session, add the current user to *dialout* Linux group. Enter the *root* password when prompted:

```
sudo adduser $USER dialout
su - $USER
```

After that:

1. cd programs/05_cache_misses
2. run ./02_compile_and_link.sh
3. run ./08_generate_motorola_s_record_file.sh
4. run ./11_check_which_com_port_is_used.sh
5. edit ./12_upload_to_the_board_using_uart.sh based on the result from the previous step – set the working port in "set a=" assignment
6. Make sure the switches 0, 1, 2 on FPGA board are turned off. Switches 0 and 1 control the speed of the clock, while switch 2 gates the program execution – see *3.5. Review the lab program code*. If the switches 0 and 1 are not off, the loading through UART is not going to work; if the switch 2 is not off, the program will execute before you have a chance to change the clock speed.
7. ./run 12_upload_to_the_board_using_uart.sh

3.7. The first run

1. Make sure the switches 0, 1, 2 on FPGA board are turned off. Switches 0 and 1 control the speed of the clock, while switch 2 gates the program execution – see *3.5. Review the lab program code*. If the switches 0 and 1 are not off, the boot sequence (a sequence of processor instructions before *main* function) will take too long. If the switch 2 is not off, the program will execute before you have a chance to change the clock speed.
2. Reset the processor. The reset buttons for each board are listed in the table below:

Board	**Reset button**
Digilent Basys3	Up
Digilent Nexys4	Dedicated CPU Reset

Digilent Nexys4 DDR	Dedicated CPU Reset
Terasic DE0	Button/Key 0
Terasic DE0-CV	Dedicated reset button
Terasic DE0-Nano	Button/Key 0
Terasic DE1	Button/Key 0
Terasic DE2-115	Button/Key 0
Terasic DE10-Lite	Button/Key 0

3. Turn the switch 1 on. This will switch the system clock from 25 MHz to 12.5 Hz. You should see LED 7 start blinking, it is connected straight to the system clock.
4. Now turn your attention to LED 6, next right from blinking LED 7. LED 6 is connected to the signal that signifies cache fill burst.

   File *system_rtl/mfp_ahb_lite_matrix.v*

```
`ifdef MFP_DEMO_CACHE_MISSES

wire burst = HTRANS == `HTRANS_NONSEQ && HBURST == `HBURST_WRAP4;

assign IO_GreenLEDs = { { `MFP_N_GREEN_LEDS - (1 + 1 + 6) { 1'b0 } },
                        HCLK, burst, HADDR [7:2] };

`endif
```

5. Turn the switch 2 on. It will let the program to proceed to fill the array. Continue watching LED 6.
6. LED 6 starts blinking. The first couple of requests are coming from instruction fetches filling L1 instruction cache, not L1 data cache, but after this you can see blinks with the frequency of cache fills.

3.8. The second run

Turn all switches to off position and go back to the program code. Comment out *"a [i][j] = i + j;"* and uncomment *"a [j][i] = i + j;"*. Now we are going to fill the array by rows, not by columns.

File *programs/05_cache_misses/main.c*

```
#include "mfp_memory_mapped_registers.h"

int a [8][8];

int main ()
{
    int n = 0;
    int i, j;

    // Wait for switch 2

    while ((MFP_SWITCHES & 4) == 0)
        ;


    for (i = 0; i < 8; i ++)
        for (j = 0; j < 8; j ++)
            // a [i][j] = i + j;
               a [j][i] = i + j;

    return 0;
}
```

Repeat the compile, link and other steps as described in sections 3.6 and 3.7. You should see a different memory access pattern as manifested on LED 6. Instead of "cache miss – hit – hit – hit – miss – hit – hit – hit – miss ..." manifested in the first run, the pattern is going to be a series of eight cache misses followed by long period of hits (no LED 6 light) with eight back-by-back cache misses followed by silence again.

4. Additional question for self-study

Read the article in Wikipedia Row-major order. Is C programming language a row-major order language or a column-major order language? How the lab results would change if C uses the opposite method for arranging multidimensional arrays in memory?