MIPSfpga

by Imagination

Lab 22-B

The Cache Controller – D$ Miss Management

Lab 22-B
The Cache Controller – D$ Miss Management

1. Introduction

In this lab we focus on cache miss operations. Specifically, we analyze the management of a `lw` instruction miss in the Data Cache (D$). The management of an I$ miss is completely analogous (and in fact quite simpler) and will be analyzed as an exercise at the end of this lab. As for store instructions, they are explained in Lab 24.

As explained in Lab 22-A, hit/miss detection is performed at the M-Stage of the `lw` instruction, when the requested tag is compared with the tags read from the D$ at the **m14k_cache_cmp** module available within the data cache controller (see Figure 4 of the previous lab, where we analyzed the M-Stage of a `lw` instruction). When a hit is detected at the data cache controller (DCC), *cache_hit*=1 and *dcc_dmiss_m*=0, whereas when a miss is detected, *cache_hit*=0 and *dcc_dmiss_m*=1. Miss detection triggers several actions, which we analyze in the following section by means of theoretical explanations and simulations. Then, in the final section, we propose several exercises.

2. D$ Miss management

This section analyzes how the core manages a D$ read miss generated by a `lw` instruction. The DCC is responsible for detecting and notifying the miss to other modules, but some of the main actions triggered by the miss are delegated to other core structures outside the DCC.

When a block is requested to the D$, if the block is not found in the cache (nor in the Fill Buffer or the Data ScratchPad RAM, two structures that we will examine in Labs 24 and 25), signal *dcc_dmiss_m* is set to 1. The main actions triggered by the miss detection of a `lw` instruction (stores introduce some differences explained in Lab 24), which we analyze in detail in this section, are the following:

1. **Stop execution**: Given that *load* instructions are blocking the pipeline must be stopped for some cycles. Eventually, when the requested word arrives from main memory, execution will restart from the point where it stopped.
2. **Request block to main memory through the bus, select a block to replace and evict it**: A request is sent to the bus interface unit (BIU). A few cycles after the request, the missed line arrives at a rate of 1 word per cycle, along 4 cycles. Moreover, the WS array is read and the LRU algorithm (which Lab 23 explains in detail) determines the cache

block that must be replaced if the destination set is full (obviously, if the set is not full, the new block is inserted in an empty way). Depending on the write policy employed, the replaced block is copied to main memory or just discarded.

3. **Resume execution as soon as possible**: As soon as the requested word (*critical* word) arrives from main memory, execution resumes from the point where it stopped.

Along this section we provide simulations to back up the theoretical explanations. For that purpose, we use the program shown in Figure 1, available in folder *Lab22_CacheController_HitAndMissManagement*\Simulations\SimulationSources_Miss. In this case, we want the instructions to be found in the I$, so we focus on the second iteration of the loop, but instead we want the `lw` instruction to miss in the D$, so we access a different 16B block in each iteration (by means of instruction `addiu t6,t6,16`).

```
80000204 <main>:
80000204:       24090002        li      t1,2
80000208:       240a0000        li      t2,0
8000020c:       3c0e8000        lui     t6,0x8000
80000210:       25ce0270        addiu   t6,t6,624

80000214 <loop>:
        80000214:       012a6022        sub     t4,t1,t2
        80000218:       012a6024        and     t4,t1,t2
        8000021c:       8dcd0004        lw      t5,4(t6)
        80000220:       012a6020        add     t4,t1,t2
        80000224:       012a6027        nor     t4,t1,t2
        80000228:       012a6025        or      t4,t1,t2
        8000022c:       00000000        nop
        80000230:       25ce0010        addiu   t6,t6,16
        80000234:       254a0001        addiu   t2,t2,1
        80000238:       0149082a        slt     at,t2,t1
        8000023c:       1420fff5        bnez    at,80000214 <loop>
        80000240:       00000000        nop
        80000244:       1000ffff        b       80000244 <loop+0x30>
        80000248:       00000000        nop
```

Figure 1. Simulated program used for examining a lw D$ miss.

a. Stop Execution

i. Explanation

The pipeline must be stopped when a D$ miss for a load is detected, given that the MIPSfpga pipeline is in-order and caches are blocking. Eventually, when the data arrives from Main Memory (MM), the execution resumes from the point where it was stopped.

For stopping execution, the Pipeline Registers (PRs), used for communicating information between consecutive stages are not updated during the miss management. As we explained in lab 14, microAptiv contains 3 Control PRs in the core (*_ie_pipe_out_50_0_*, *_m_pipe_out_46_0_*, *_w_pipe_out_5_0_*), updated in module **m14k_mpc_ctl**, and several individual PRs, such as the Instruction Register (*_int_ir_e_31_0_*), the Program Counter (*_edp_iva_i_31_0_*), etc. PR updating is only enabled when signals *mpc_run_** are 1.

Suppose that, in cycle *i*, the `lw` instruction from Figure 1 is at the M-Stage, where it accesses the D$. The data requested by the load is not found in the D$, so a miss is signaled (*dcc_dmiss_m*=1). The pipeline is allowed to advance one more cycle though, as *mpc_run_** are 1 in cycle *i*. Thus, in cycle *i+1*, the pipeline state is the one shown in Figure 2. As the simulation performed below illustrates, in the cycle after the miss is detected, signals *mpc_run_** change to 0, thus the pipeline remains in this state until the requested data arrives from main memory. The only exception is that the instruction at the W-Stage (`and` instruction) is allowed to finish execution in cycle *i+1*.

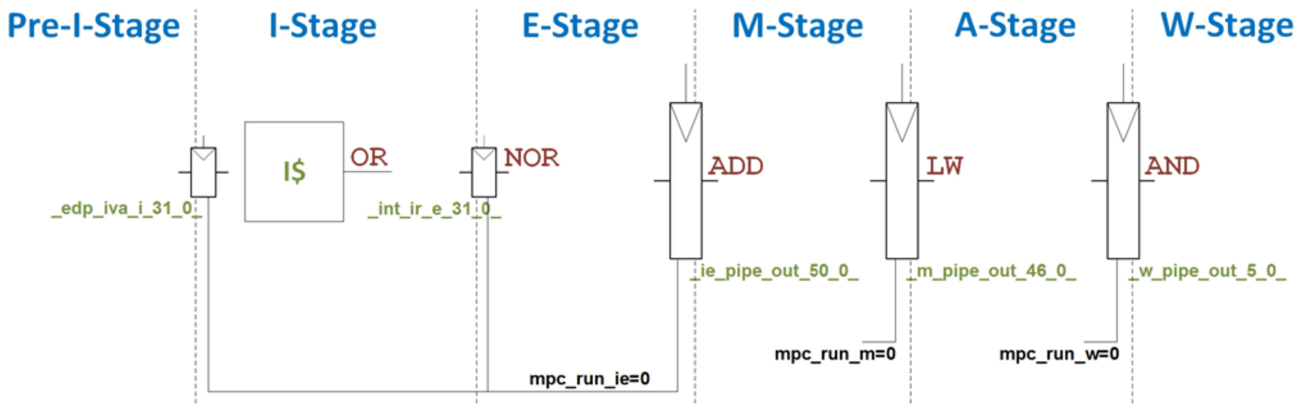

Figure 2. Pipeline state in cycle *i+1*.

Eventually, at cycle *i+n* (being *n* the miss penalty), the processor receives the requested data from main memory and resumes execution from the state shown in Figure 2, by setting *mpc_run_** to 1. Thus, at cycle *i+n*, the I$ reads the `or` instruction, which is stored in the Instruction Register (*_int_ir_e_31_0_*) at the end of the cycle, the `nor` instruction executes and registers its result in register *_prealu_m_31_0_* (see Lab 15) at the end of the cycle, the `add` instruction bypasses the M-Stage, and the data received from main memory is aligned and passed to the Register File, where it will be written along the next cycle.

We should highlight that some of the tasks performed at the different pipeline stages are gated during the miss handling, mainly for saving energy. For example, I$ access is gated during the miss handling process (by assigning *data_rd_str*=0). Instead, some other tasks are combinational and thus are not gated. For example, the Register File keeps being read at the E-Stage along the D$ miss handling, or the ALU at the E-Stage keeps working. Note however that, in general, when the inputs to some combinational logic remain unchanged, transistors do not switch and hence dynamic energy consumption is zero.

ii. Example Simulation

In this section we perform the simulation of the assembly program shown in Figure 1, highlighting the processor stall. As explained in Lab 14, start by creating or opening a Vivado project, add the 8 memory files available in folder *Lab22_CacheController_HitAndMissManagement**Simulations**SimulationSources_Miss*, and the waveform configuration file (*testbench_boot_behav_Miss-StopPipeline.wcfg*), and configure the simulator properly.

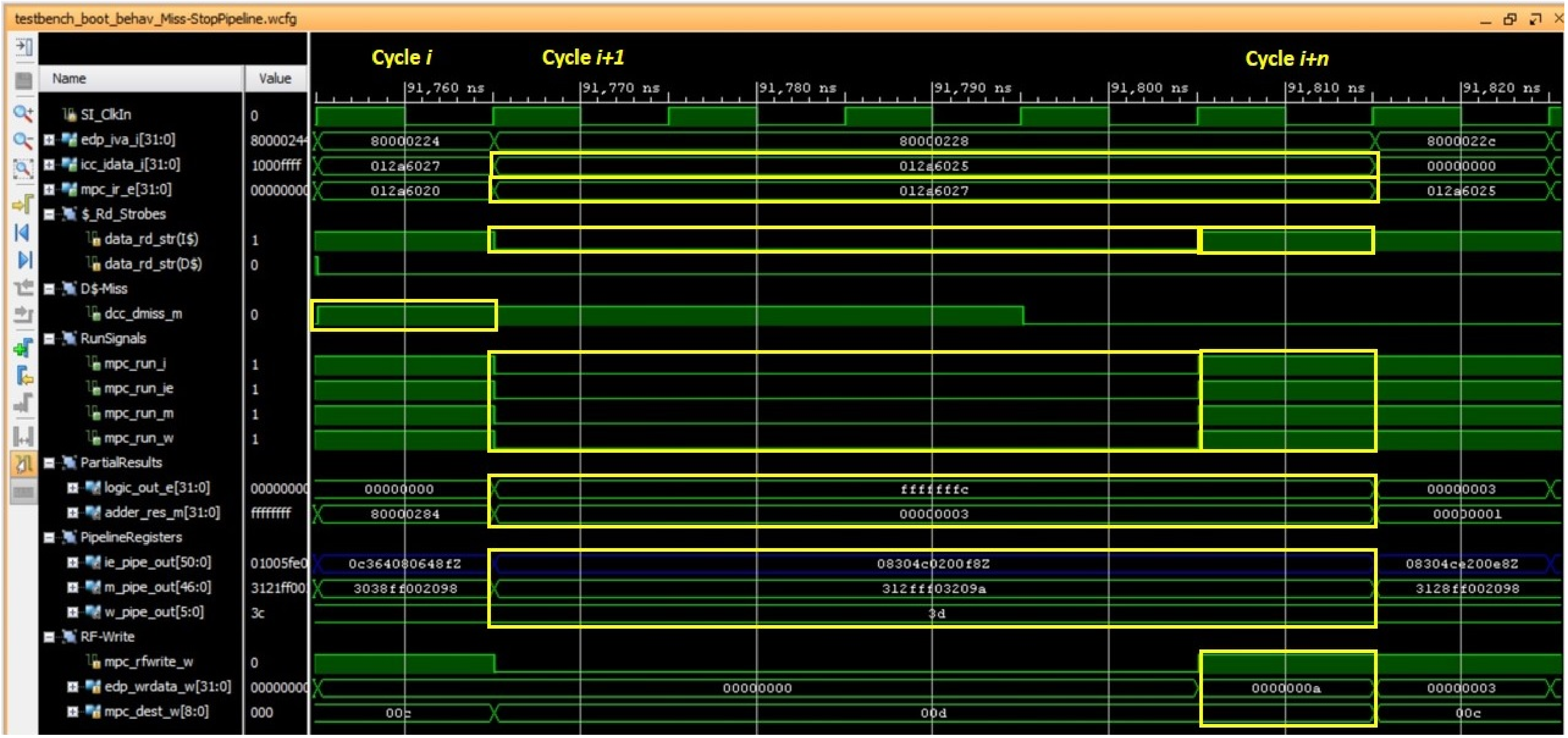

Figure 3. Timing diagram illustrating the first action triggered after a D$ miss: stop the pipeline.

Next, we analyze the results of this simulation shown in Figure 3.

- 1st cycle (cycle *i*):
  - A miss is detected for the `lw` at the M-Stage (*dcc_dmiss_m*=1), which triggers the miss handling process.
  - *mpc_run_**=1, so the pipeline is not yet frozen, but instructions can advance one more cycle.
- 2nd cycle (cycle *i+1*) to 5th cycle (cycle *i+n-1*):
  - The Pipeline Registers are not updated (*mpc_run_**=0) during the miss handling. The pipeline is frozen with the state shown in Figure 2 (for example, we can easily see from the simulation results that the instruction fetched from the I$ at the I-Stage is the `or` instruction, as *icc_idata_i*=0x012a6025, or that the instruction at the E-Stage is the `nor` instruction, as *mpc_ir_e*=0x012a6027).
  - Observe that some structures are gated with the aim of saving energy during the miss handling, such as the I$ access (by setting *data_rd_str*=0).
  - Note also that the combinational logic keeps working along the miss handling, but its results are not registered to the PRs. For example, the Logic Unit at the E-Stage keeps doing the operation for the `nor` instruction (*logic_out_e*=0xfffffffc), but the result is not registered.
- 6th cycle (cycle *i+n*):
  - The core receives the missed word during this cycle. Signals *mpc_run_** change to 1, thus execution resumes.
  - The register file receives the data to write in the next cycle for the `lw` instruction (*edp_wrdata_w*=0x0000000a and *mpc_dest_w*=0x00d).

b. Request block to main memory through the bus, select a block to replace and evict it

i. Explanation

In order to exploit spatial locality, not only the missed word is brought from main memory to the D$ but also the words that surround it (i.e. the whole block). In this section we analyze this process in detail. Note that the destination set for the new block may be full, in which case the replacement policy (which is analyzed in Lab 23) must select a block to replace (obviously, if the set is not full, the inserted block is simply stored in an empty way). In a *write back* cache (also analyzed in Lab 23), the replaced line must be written back to main memory only if it is dirty; otherwise, it can simply be discarded. In a *write through* cache (also analyzed in Lab 23), main memory is always up-to-date (store instructions always update the cache and main memory), thus the replaced line can always be discarded. For the sake of simplicity, in the program analyzed in this lab (Figure 1) the destination set for the `lw` instruction is not full, thus no line must be replaced; however, after completing Labs 23 and 24, you can simulate a scenario where a block must be replaced and written back to main memory.

As explained in Section 5.2 of the Getting Started Guide, an AHB-Lite read operation consists of two consecutive phases: the address phase (first cycle) and the data phase (second cycle). During the address phase, the master (the core) sends the address on *HADDR* and deasserts *HWRITE*. During the data phase, the slave (MM in this case) sends the data on *HRDATA*. Note that the two phases can be overlapped, meaning that the address phase of a transfer can occur during the data phase of the previous transfer.

The D$ line in microAptiv is made up of 4 words. Given that the width of *HRDATA* is 32 bits, at most one word can be sent from MM to the D$ per cycle. Thus, for transferring a whole cache line from MM to the D$, 5 cycles are needed in MIPSfpga: in each cycle a new word is requested by supplying its address through *HADDR* and the word requested in the previous cycle is sent from MM through *HRDATA* (exceptionally, in the first cycle no word is sent from MM and in the fifth cycle no new address is requested). We should highlight here that the first word requested to MM from the core is the word requested by the `lw` instruction. That way, execution can continue as soon as that word arrives to the core (this is called *critical word first* policy, and is explained as the sixth optimization in Section 2.2 of [3]).

The logic involved in this process is explained below and illustrated in Figure 4.

1. <u>Generation of control signals for managing the bus request:</u>
    - In the cycle after the miss is detected (*dcc_dmiss_m*=1), *dreq_val*=1 which unleashes 4 consecutive pulses:
        - First pulse (first cycle): Signal *dreq_st0*=1.
        - Second pulse (second cycle): Signal *dreq_st1*=1.
        - Third pulse (third cycle): Signal *dreq_st2*=1.
        - Fourth pulse (fourth cycle): Signal *dreq_st3*=1.
    - In addition, signal *dreq* is computed as follows:

      ```
      dreq = dreq_st0 || dreq_st1 || dreq_st2 || dreq_st3.
      ```

      Thus, signal *dreq* is true along the duration of the transfer.

2. <u>Request the cache line (4 words) through the Bus:</u>
    - The address requested to MM is provided through signal *HADDR*, which is computed in module **m14k_biu** by concatenating signals *burst_addr_nxt*[31:2] and *be_nxt_address*[1:0] as follows:

      ```
      mvp_cregister_wide #(32) _HADDR_31_0_(HADDR[31:0],gscanenable, new_addr,
      gclk, {burst_addr_nxt[31:2], be_nxt_address[1:0]});
      ```

Along our explanations and examples, signal *be_nxt_address*[1:0]=00, thus a word-aligned access is always performed.

- When *dreq*=1, a new *burst_addr_nxt*[31:2] is computed every cycle in module **m14k_biu** by concatenating two signals, as follows:

```
assign burst_addr_nxt [31:2] = wreq_a ?{wtaddr[31:4], wrb_addr_wd[1:0]} :
        dreq ?  {daddr[31:4], dword_nxt} :
        ireq ?  {iaddr[31:4], iword_nxt} :
        {30'b0};
```

  where:

  - Signal *daddr*[31:4] contains the address of the missed line, provided from the DCC (module **m14k_dcc**), and remains unchanged along the whole transfer.
  - Signal *dword_nxt*[1:0] selects, in each pulse, a different word within the requested line, and is computed in module **m14k_biu** as follows:

```
assign dword_nxt [1:0] = (beat_cnt == 0) ? daddr[3:2]
        : (change_transfer_reg) ? (HADDR[3:2] + 2'h1)
                : (dword_nxt_reg [1:0]);
…
assign beat_cnt [1:0] = (ireq_st3 || dreq_st3 || wreq_sta3) ? 2'h3:
        (ireq_st2 || dreq_st2 || wreq_sta2) ? 2'h2:
        (ireq_st1 || dreq_st1 || wreq_sta1) ? 2'h1: 2'h0;
```

    Initially, *beat_cnt*=00, thus *dword_nxt*[1:0]=*daddr*[3:2], thus the first word requested to MM is the word requested by the `lw` instruction (i.e. the *critical word*). In the 3 following cycles (where *beat_cnt* is 01, 10 and 11), the remaining words within the cache line are requested by means of a new modulo 4 adder: `HADDR[3:2] + 2’h1`.

3. <u>Main memory provides the cache line (4 words) through the Bus:</u>
   - Simultaneously to the previous action, in the cycle after each address request, a word belonging to the missed cache line, starting with the *critical word*, arrives through *HRDATA.* The words provided from main memory are buffered in a structure called Fill Buffer (module **m14k_dcc_fb**) before being finally written to the D$.

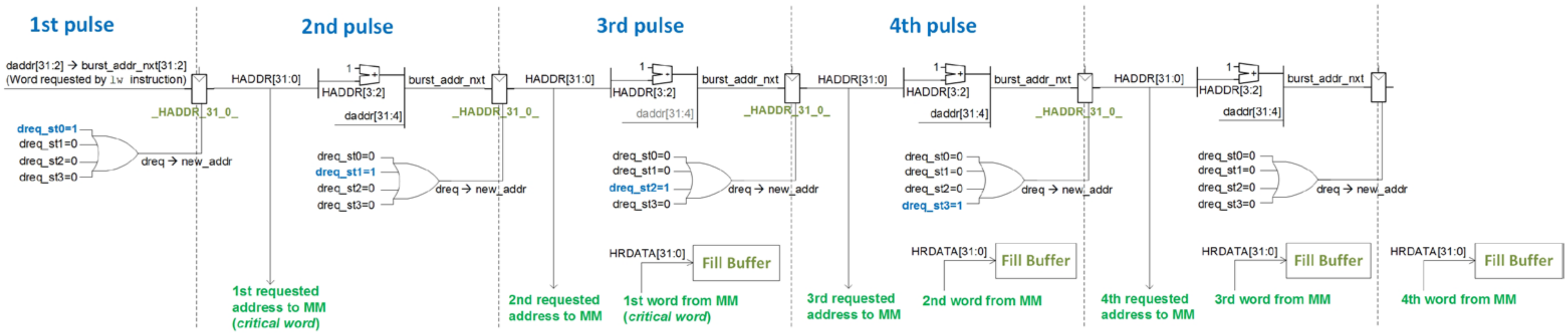

Figure 4. Data line request to Main Memory.

ii. Example Simulation

In this section we perform the simulation of the assembly program shown in Figure 1, highlighting the request of the missed D$ block through the AHB-Lite bus used in MIPSfpga. You can use the same Vivado project used in the previous section, adding the waveform configuration file called *testbench_boot_behav_Miss-RequestAddress.wcfg*. Figure 5 illustrates the results.

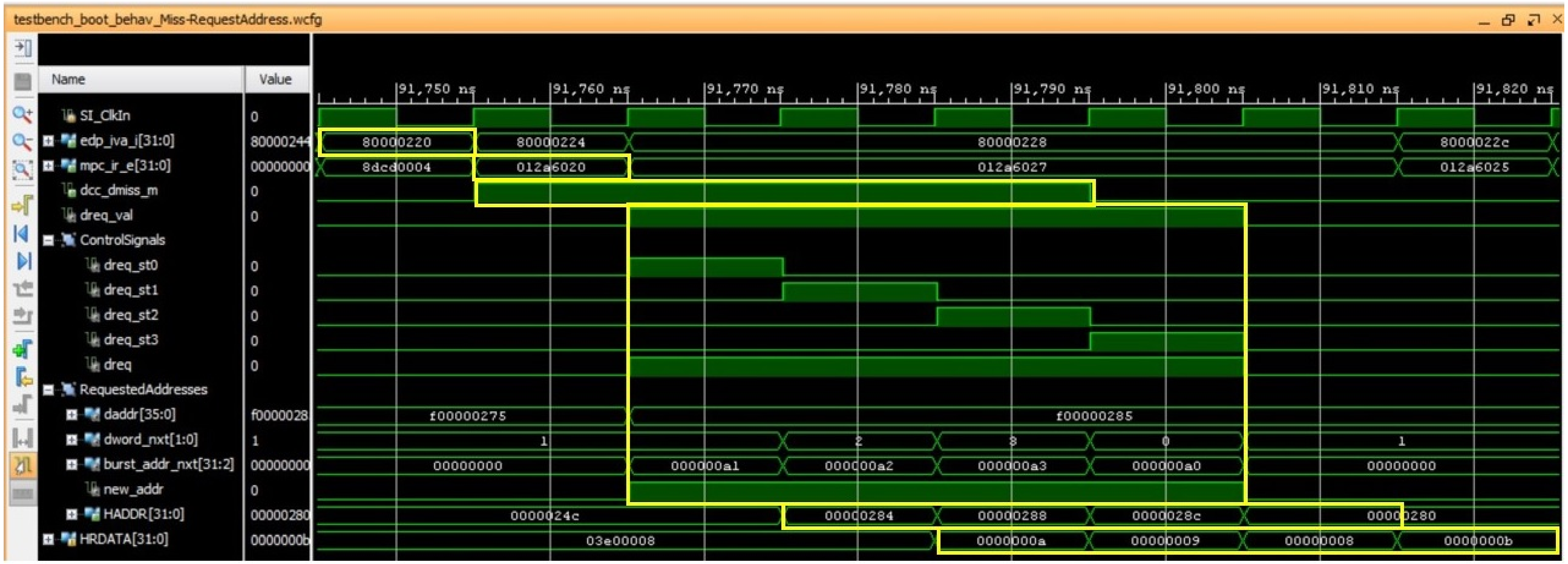

Figure 5. Timing diagram illustrating a request to MM.

Next we detail the results obtained in Figure 5.

- 1st cycle:
    - The `lw` instruction is at the E-Stage.
- 2nd cycle:
    - The `lw` instruction is at the M-Stage. A D$ miss is detected (*dcc_dmiss_m*=1).
- 3rd cycle:
    - First pulse (Figure 4).
    - The first address to request through the bus, corresponding to the address of the word requested by the `lw` instruction (*critical word*), is assigned into signal *burst_addr_nxt[31:2]*=0x000000a1 from signal *daddr[31:2]* (which remains constant along the whole transfer).
    - Signal *burst_addr_nxt[31:2]* is registered, at the end of this cycle, into signal *HADDR*.
- 4th cycle:
    - Second pulse (Figure 4).
    - The *critical word* is requested to main memory (*HADDR[31:0]*=0x00000284), which corresponds to the second word within the D$ block.
- 5th cycle:
    - Third pulse (Figure 4).
    - The *critical word* is provided from main memory (*HRDATA[31:0]*=0x0000000a).
    - The second word of the missed D$ block is requested (*HADDR[31:0]*=0x00000288).
- 6th cycle:
    - Fourth pulse (Figure 4).
    - The second word of the missed D$ block is provided from main memory (*HRDATA[31:0]*=0x00000009).
    - The third word of the missed D$ block is requested (*HADDR[31:0]*= 0x0000028c).
- 7th cycle:
    - The third word of the missed D$ block is provided from main memory (*HRDATA[31:0]*=0x00000008).
    - The fourth word of the missed D$ block is requested (*HADDR[31:0]*=0x00000280).
- 8th cycle:
    - The fourth word of the missed D$ block is provided from main memory (*HRDATA[31:0]*=0x0000000b).

c. Resume execution as soon as possible

i. Explanation

In this subsection we explain the finalization of the D$ miss and how the execution is resumed. When the *critical word* arrives from main memory through signal *HRDATA[31:0]*, it is stored into register *_hrdata_reg_31_0_*. Then, in the next cycle, after traversing a hierarchy of multiplexers (Figure 6), the *critical word* is bypassed to the execution datapath through signal *ld_or_cp_m[31:0]* (this signal is also included in Lab 16 and Lab 21-A), so that execution can resume immediately, even though part of the block is still being transferred from main memory as explained in the previous section.

If we zoom into the hierarchy of multiplexers (Figure 6), you can observe that the *critical word* must first traverse a 4:1 multiplexer, given that the requested word can be in any position within the cache block. The control signal for this multiplexer is a subset of the missed address (*dval_m_sel*[3:2]) that determines the specific word within the cache line requested by the `lw` (i.e. the *critical word*). This multiplexer is defined in module **m14k_dcc_fb** (line 702) as follows:

```
assign fb_data [31:0] = (dval_m_sel[3:2] == 2'h0) ? fb_data_next0[31:0] :
        (dval_m_sel[3:2] == 2'h1) ? fb_data_next1[31:0] :
        (dval_m_sel[3:2] == 2'h2) ? fb_data_next2[31:0] :
        fb_data_next3[31:0];
```

Once the *critical word* is available in signal *fb_data[31:0]*, some more multiplexers are traversed as shown in Figure 6. These multiplexers are defined in module **m14k_dcc** (lines 2194-2207). Finally, the *critical word* is stored in register *_edp_ldcpdata_w_31_0_* (*edp_ldcpdata_w[31:0]* <= *ld_or_cp_m[31:0]* = *dcc_ddata_m[31:0]*) and execution resumes with the `lw` instruction traversing the A-Stage.

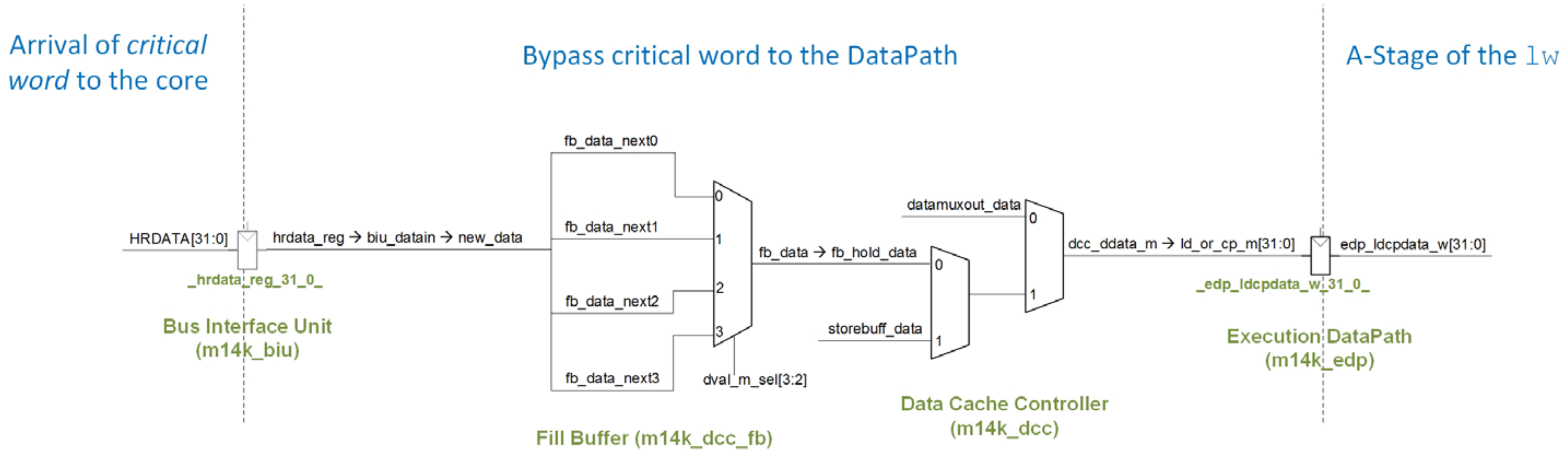
Arrival of *critical word* to the core
Bypass critical word to the DataPath
A-Stage of the lw
HRDATA[31:0]
hrdata_reg → biu_datain → new_data
_hrdata_reg_31_0_
Bus Interface Unit (m14k_biu)
fb_data_next0
fb_data_next1
fb_data_next2
fb_data_next3
0
1
2
3
dval_m_sel[3:2]
Fill Buffer (m14k_dcc_fb)
fb_data → fb_hold_data
storebuff_data
datamuxout_data
dcc_ddata_m → ld_or_cp_m[31:0]
Data Cache Controller (m14k_dcc)
edp_ldcpdata_w[31:0]
_edp_ldcpdata_w_31_0_
Execution DataPath (m14k_edp)

Figure 6. Critical word bypass.

ii. Example Simulation

In this section we perform the simulation of the assembly program shown in Figure 1, highlighting the execution resuming after the D$ miss and the *critical word* bypass. You can use the same Vivado project used in the previous sections, adding the waveform configuration file called *testbench_boot_behav_Miss-CriticalWordBypass.wcfg*. Figure 7 illustrates the results.

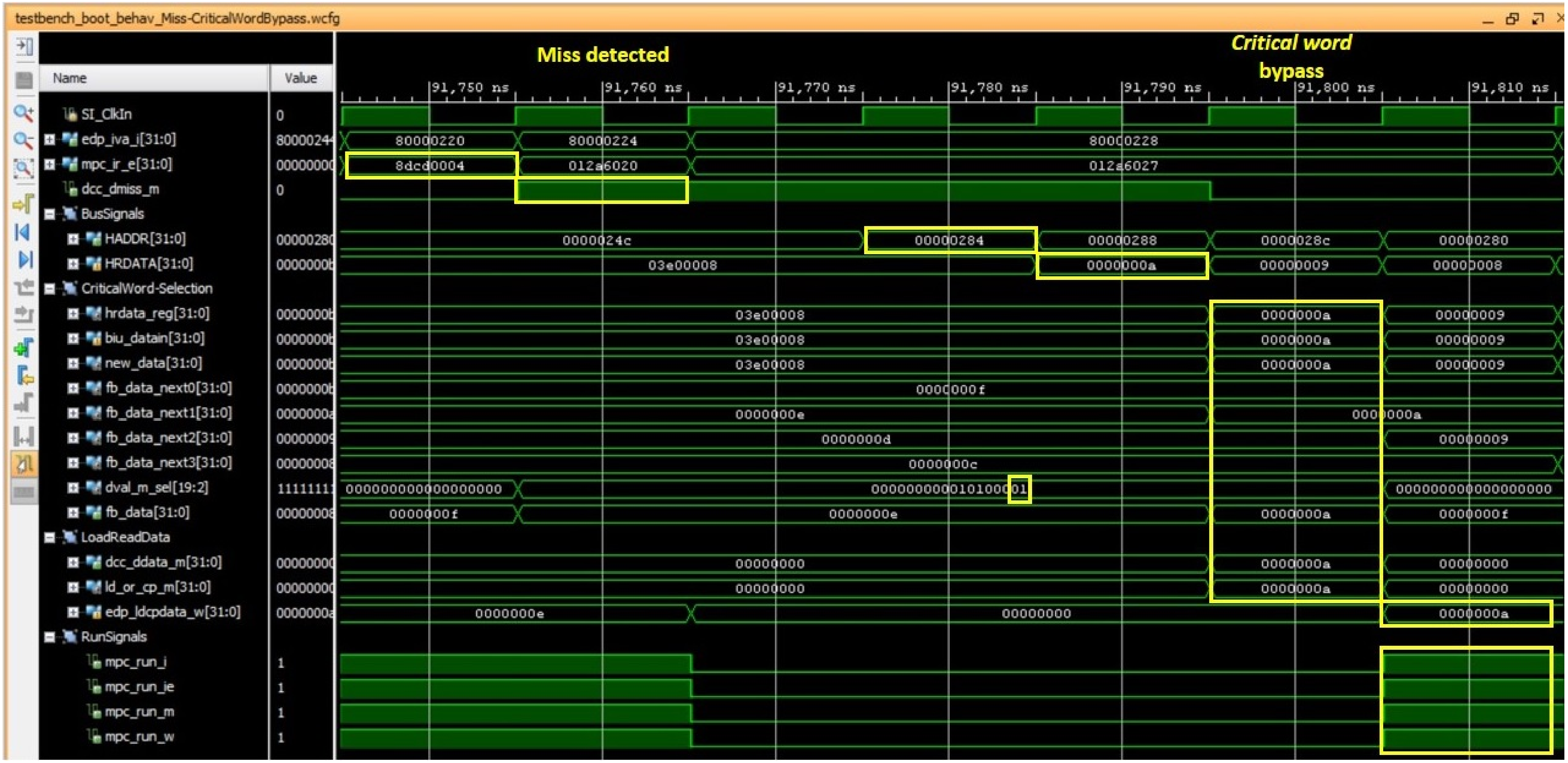

Figure 7. Timing diagram illustrating the execution resuming.

Next we detail the results obtained in Figure 7.

- 1st cycle:
  - The `lw` instruction is at the E-Stage.
- 2nd cycle:
  - A D$ miss is detected at the M-Stage.
- 4th cycle:
  - The *critical word* is requested through the bus (*HADDR*=0x00000284).
- 5th cycle:
  - The *critical word* arrives from main memory (*HRDATA*=0x0000000a).
- 6th cycle:
  - The *critical word* is bypassed to the datapath as explained in Figure 6 (*ld_or_cp_m[31:0]*=0x0000000a).
- 7th cycle:
  - The word requested by the `lw` is at the A-Stage (*edp_ldcpdata_w[31:0]*=0x0000000a) and execution can resume (*mpc_run_**=1).

3. Exercises

Exercise 1: Analyze an I$ Miss

Sketch the Verilog code related with the management of an I$ miss. An I$ miss is managed analogously to the D$ miss described in the previous sections. Once you understand the logic involved, perform a simulation and analyze an I$ miss.

Exercise 2: Determine the *miss penalty*

Examining Figure 7, determine the *miss penalty* and explain each factor contributing to it.

Exercise 3: Code optimization techniques

In this exercise, you execute several programs on the FPGA board and evaluate the following code optimization techniques, using the performance counters as explained in Lab 13:

- Array enlargement and array merging
- Loop interchange
- Blocking

Before developing this exercise, we recommend you to read Section 2.2 – Eighth Optimization of [3].

i. Array enlargement and array merging

Use the following C program for evaluating the first technique:

```
volatile int A[512];
volatile int B[512];
…
for (i = 0; i < 512; i = i + 1)
       C = C + (A[i] + B[i]);
```

Perform the following steps:

1. Go into folder *Lab22_CacheController_HitAndMissManagement\Simulations\CodeOptimizations\ArrayEnlargement_ArrayMerging\SimulationSources_Baseline* and open file **main.c**. Note that we are using the skeleton provided in Lab 13. The evaluated program is the C program mentioned above, translated into MIPS assembly language. Analyze the program on your own.
2. Execute this program on a processor (don´t forget to expand MIPSfpga as explained in Lab 5 for supporting the 7-segment displays) with a **Direct-Mapped 2KB D$** configuration (one of the configurations that you created in Lab 21) and on a processor with a **2-way D$ with 2KB per Way**, and measure the *D$ accesses* and *D$ misses*. Then, configure the performance counters for accounting for *instructions completed* and *cycles*. Explain the results.
3. **Array Enlargement**:
    a. Given the access pattern in this code, we can use a software optimization technique called *array enlargement*. In this case, the first array is slightly extended, which permits to avoid all conflict D$ misses.

    ```
    volatile int A[516];
    volatile int B[512];
    ```

    b. Go into folder *Lab22_CacheController_HitAndMissManagement\Simulations\CodeOptimizations\ArrayEnlargement_ArrayMerging\SimulationSources_ArrayEnlargement*, open file **main.c**, and analyze the program on your own.
    c. Execute this program on MIPSfpga for a **Direct-Mapped 2KB D$** configuration and measure the amount of *D$ accesses* and *D$ misses*. Then, configure the performance counters for accounting for *instructions completed* and *cycles*. Explain the results. Can you infer the average miss penalty from the results achieved?
4. **Array Merging**:
    a. In this program we can also use a software optimization technique called *array merging*. In this case, the two arrays are merged in a single *struct*, avoiding again all conflict D$ misses. Note that, in this case, the code also has to change a bit.

    ```
    struct fusion{
        int A;
    ```

```
        int B;
    } array[512];
    …
    for (i = 0; i < 512; i = i + 1)
           C = C + (array[i].A + array[i].B);
```

   b. Go into folder *Lab22_CacheController_HitAndMissManagement\Simulations\CodeOptimizations\ArrayEnlargement_ArrayMerging\SimulationSources_ArrayMerging*, open file **main.c**, and analyze the program on your own.
   c. Execute this program on MIPSfpga for a **Direct-Mapped 2KB D$** configuration and measure the amount of *D$ accesses* and *D$ misses*. Then, configure the performance counters for accounting for *instructions completed* and *cycles*. Explain the results. Can you infer the average miss penalty from the results achieved?

ii. Loop interchange

For evaluating the second optimization technique, we use the following C program:

```
for (i=0; i<32; i=i+1)
       for (j=0; j<32; j=j+1)
              C = C + A[j][i];
```

Perform the following steps:

1. Go into folder *Lab22_CacheController_HitAndMissManagement\Simulations\CodeOptimizations\LoopInterchange\SimulationSources_Baseline* and open file **main.c**. Note that we are using the skeleton provided in Lab 13. The evaluated program is the C program mentioned above. Analyze the program on your own.
2. Execute this program on a processor with a **Direct-Mapped 2KB D$** configuration and on a processor with a **Direct-Mapped 4KB D$** (two configurations that you created in Lab 21), and measure the *D$ accesses* and *D$ misses*. Then, configure the performance counters for accounting for *instructions completed* and *cycles*. Explain the results.
3. **Loop interchange**:
   a. We can use a software optimization technique called *loop interchange*. In this case, the program is modified as follows:

```
for (i=0; i<32; i=i+1)
       for (j=0; j<32; j=j+1)
              C = C + A[i][j];
```

   b. Go into folder *Lab22_CacheController_HitAndMissManagement\Simulations\CodeOptimizations\LoopInterchange\SimulationSources_LoopInterchange*, open file **main.c**, and analyze the program on your own.

c. Execute this program on MIPSfpga for a **Direct-Mapped 2KB D$** configuration and measure the amount of *D$ accesses* and *D$ misses*. Then, configure the performance counters for accounting for *instructions completed* and *cycles*. Explain the results. Can you infer the average miss penalty from the results achieved?

iii. Blocking

A very typical operation used in many computing algorithms is matrix multiplication. The following program constitutes a possible implementation of this algorithm:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        r = 0;
        for (k = 0; k < N; k++) {
            r = r + y[i][k]*z[k][j];
        }
        x[i][j] = r;
    }
}
```

A useful optimization of this code is to use **blocking**, as explained in page 89 of [3]. The resulting code would look as follows:

```
for (jj = 0; jj < N; jj=jj+B) {
    for (kk = 0; kk < N; kk=kk+B) {
        for (i = 0; i < N; i++) {
            for (j = jj; j < MIN((jj+B),N); j++) {
                r = 0;
                for (k = kk; k < MIN((kk+B),N); k++) {
                    r = r + y[i][k]*z[k][j];
                }
                x[i][j] = x[i][j] + r;
            }
        }
    }
}
```

Test both C programs for different matrix sizes, blocking values and D$ configurations, and discuss the results. The source files for this exercise are provided in folder *Lab22_CacheController_HitAndMissManagement\Simulations\CodeOptimizations\MatrixMultiplication*.

4. References

[1] “MIPS32® microAptiv™ UP Processor Core Family Software User’s Manual -- MD00942”.

[2] “Digital Design and Computer Architecture”, 2nd Edition. David Money Harris and Sarah L. Harris. Morgan Kaufmann, 2012.

[3] “Computer Architecture: A Quantitative Approach”, 4th Edition. John L. Hennesy and David A. Patterson. Morgan Kaufmann.

[4] “Computer Organization and Design”, 5th Edition. David A. Patterson and John L. Hennesy. Morgan Kaufmann, 2013.