MIPSfpga by Imagination

Lab 22-A

The Cache Controller – I$ Hit Management

These materials produced in association with Imagination.
Join our University community for more resources.
community.imgtec.com/university

Lab 22-A
The Cache Controller – I$ Hit Management

1. Overview

Given the high complexity of the Instruction and Data Cache Controllers, we have divided their explanation into three labs. In the first one, we analyze the management of an I$ hit (Lab 22-A) and a `lw` D$ miss (Lab 22-B). Then, in Lab 23, we study the different policies available in the core for managing the cache contents. Finally, in Lab 24 we analyze the Store Buffer, which holds temporarily the data to write by a store, and the Fill Buffer, which holds temporarily the block provided from main memory as a consequence of a miss.

As explained in Lab 21, microAptiv includes independent Data (D$) and Instruction Caches (I$). Each cache is managed via its Cache Controller. Thus, the Data Cache Controller (DCC), implemented in module **m14k_dcc**, manages the D$, and the Instruction Cache Controller (ICC), implemented in module **m14k_icc**, manages the I$. Figure 1 (obtained from [1]) illustrates a simplified block diagram of the DCC (the ICC is analogous but easier, as we will see throughout these labs). It includes the control logic for D$ lookup and replacement, dirty line eviction, cache tag comparators, way select multiplexers, dual line bypass-able fill buffer, and single word store buffer. The DCC supports load, store, *cacheops* and prefetch requests from the core. Loads and stores can be cached, i.e. the block containing the requested data is brought into the cache, or uncached, i.e. the requested data is read/written directly from/to main memory to/from the core, bypassing the cache. Moreover, loads and *cacheops* are blocking, whereas stores and prefetches are non-blocking.

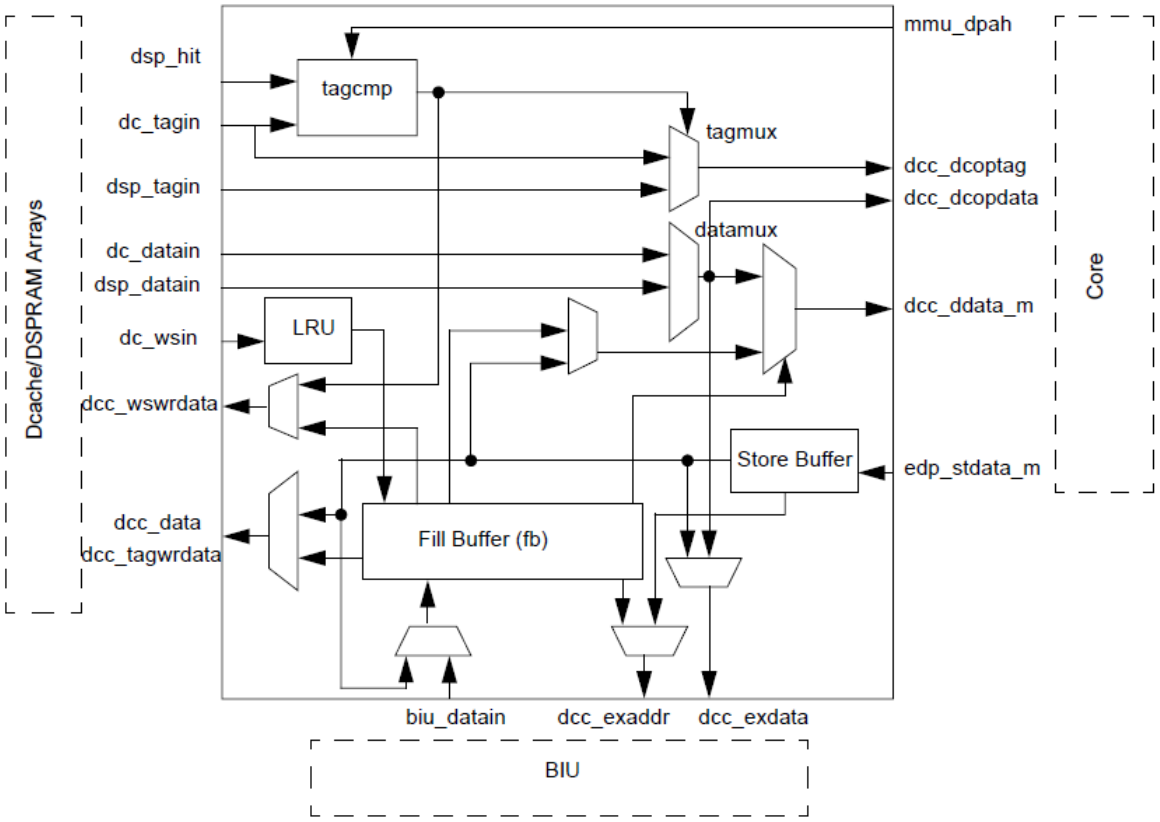

Figure 1. DCC block diagram [1].

When a hit takes place, the Way Select (WS) array is updated using a Least Recently Used (LRU) policy (we defer the explanation of cache replacement policies to Lab 23). Besides, if the D$ access comes from a load instruction, the requested word is sent to the core, specifically to the execution datapath, for being written to the register file. Instead, if the access comes from a store instruction, the data is written to the D$ through the Store Buffer (Lab 24 delves into the Store Buffer operation) and, in the case of a *write-back* policy (explained in Lab 23), the dirty bit of the line that contains the requested data is set in the WS array, whereas, for a *write-through* policy (also explained in Lab 23), the data is updated to main memory through the bus interface unit (dirty bits being not necessary in this case). In this lab (Lab 22-A) we analyze both an I$ hit and a D$ hit.

If a miss takes place (Lab 22-B examines a D$ miss), different tasks must be performed, depending on the scenario. If the request is for **uncached data**, the request is simply forwarded from the DCC to the bus interface unit (BIU) and to main memory. In MIPSfpga, access to the peripherals is a good example of uncached accesses: peripherals are mapped to virtual addresses 0xbf80XXXX, which belong to *kseg1*, which is an uncached segment. If the request is

for **cached data**, such as the access to the boot or program codes or the access to data in MIPSfpga, all of which are mapped to *kseg0*, different options are possible:

- If the D$ access comes from a store and the D$ uses a *write-through*, *no write allocate* (WT-NWA) policy, the data to write by the store instruction is buffered in the Store Buffer before being sent to main memory through the BIU, bypassing the D$. No information is provided from main memory in this case.
- If the D$ access comes from a store and the D$ uses a *write allocate* (WA) policy, or if the access comes from a load, several things must be done:
    - The missed line (made up of 4 words) must be brought to the D$. For that purpose, a request is sent to the BIU. A few cycles after the request, the missed line arrives at a rate of 1 word per cycle along 4 cycles. The missed line is stored temporarily in the Fill Buffer before being finally written into the D$, as we explain in Lab 24.
    - The WS array is read and the LRU algorithm determines the cache line that must be replaced if the destination set is full (obviously, if the set is not full, the inserted block is stored in an empty way). In a *write back* cache, 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 main memory is always up-to-date, thus the replaced line can always be discarded.
    - Loads just read data. Instead, stores must write new data to the D$. For that purpose, data to write by the store instruction is first saved in the Store Buffer, and then it is transferred to the Fill Buffer, where it will eventually be merged with the incoming line. Note that, if the D$ uses a *write-through* (WT) policy, the data to write by the store must also be updated in main memory through the BIU.
    - As we already mentioned above, stores are non-blocking. However, loads are blocking, thus the pipeline must be stopped for some cycles. Eventually, when the critical word arrives from main memory, execution will resume from the point where it stopped.

2. Introduction

In this first lab (Lab 22-A) concerning the cache controller, we focus on hit operations. Specifically, we analyze in detail the management of a hit in the I$, and ask you to analyze the management of a `lw` hit in the D$ as an exercise (we defer explanation of store instructions to Lab 24).

As briefly explained in Lab 14, at the I-Stage, the I$ is looked up for the next instruction to execute, using the Program Counter computed in the Pre-I-Stage. Also, as explained in Lab 16, memory instructions (`load` and `store`), access the Data Cache (D$) during the M-Stage, using the Effective Address computed in the E-Stage. In case of a hit in the I$/D$, the requested instruction/data is provided with a 1-cycle latency, thus the pipeline can continue working at its optimal throughput of 1 finalized instruction per cycle.

As we said above, in this lab we specifically study the management of an I$ hit. In addition to the I$ Arrays and the I$ Controller (ICC), other structures, such as the Memory Management Unit (implemented in module **m14k_tlb**), are involved in the I$ hit.

3. Pipeline Stages

In this section we explain the I$ hit management in microAptiv. Two stages are involved: the Pre-I-Stage, described in Subsection 3.a, and the I-Stage, which we analyze in detail in Subsection 3.b. Table 1 lists the main hardware modules and signals associated with each stage, and Figure 2 illustrates the main operations performed in these two stages.

Table 1. MIPSfpga pipeline: main modules and signals involved in an I$ hit operation

Pre-I-Stage	
Module/Signal Name	**Description**
m14k_edp	Execution datapath
edp_iva_p[31:0] / edp_iva_p_n[31:0]	Next PC
br_targ_e[31:0] / br_targ_e_n[31:0]	Next PC, computed at the E-Stage, for branch-like instructions
incsum_e[31:0]	Current PC (*edp_iva_i[31:0]*) + 4
preiva_p[31:0] / preiva_p_n[31:0]	Next PC, computed at the Pre-I-Stage as: Current PC + 4 (*incsum_e[31:0]*)
m14k_mpc_ctl	Main pipeline control – Control
mpc_eqcond_e	True if a branch instruction is at the E-Stage and the condition met. False otherwise
I-Stage	
Module/Signal Name	**Description**
m14k_edp	Execution datapath
edp_iva_i[31:0]	Current PC
ival_i[19:1]	Bits [19:1] of the Current PC
m14k_ic	Instruction Cache
data_addr[13:2]	Index into the I$
rd_data[63:0]	2 instructions read from the I$ Data Array, one from each way
tag_rd_data[47:0]	2 tags read from the I$ Tag Array, one from each way
m14k_tlb	Memory Management Unit
edp_cacheiva_i[31:0]	Virtual Address – Current PC

mmu_ipah	Physical Address
m14k_icc	Instruction cache controller
icc_dataaddr[13:2]	Index into the I$, obtained from the Current PC
m14k_cache_mux	Multiplexer selecting the way that hits
data_in[63:0]	2 instructions read from the I$ Data Array, one from each way
data_out[31:0]	Instruction selected by the multiplexer
waysel	Control signal for the multiplexer, provided from module m14k_cache_cmp
m14k_cache_cmp	Multiplexer selecting the way that hits
tag_cmp_pa[43:0]	Tags fetched from the 2-way Tag Array
tag_cmp_data[21:0]	Physical address from the MMU
line_sel[3:0]	True for the way that hits
cachehit	True in case of I$ hit
cache_hit	True in case of I$ hit
icc_imiss_i	True in case of I$ miss
icc_idata_i[31:0]	Fetched instruction, registered in the Instruction Register for the next stage

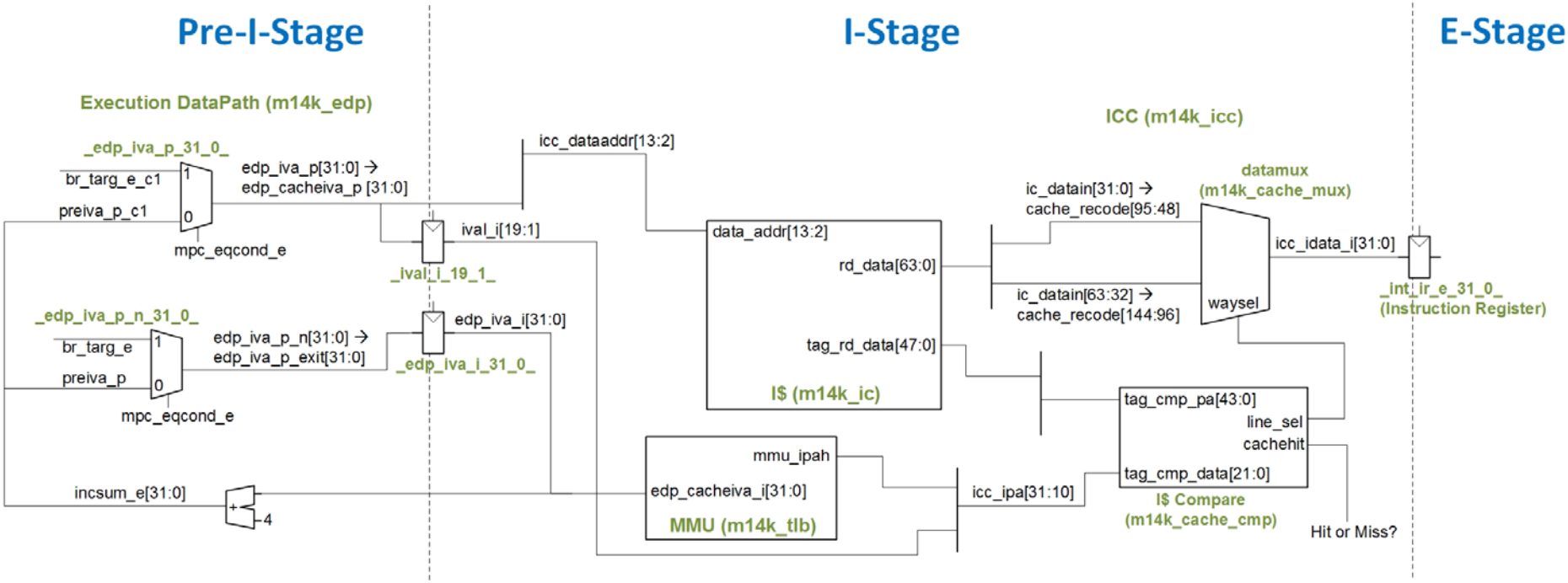

Figure 2. Simplified Pre-I-Stage and I-Stage in microAptiv for a 2-way I$.

a. Pre-I-Stage

The main purpose of this stage is to compute the Next PC, which, as shown in Figure 2, is provided in two signals (*edp_iva_p* and *edp_iva_p_n*). Each of these signals serves different purposes (we don´t go into details here about the reason for using two signals instead of one; you can confirm that, in the proposed examples and exercises, both signals always have the same value). The main uses of these signals are the following:

- Signal *edp_iva_p_n*: This signal is registered: *edp_iva_i* (Current PC) <= *edp_iva_p_n* (Next PC), and used in the next cycle for:
    - Virtual Address (VA) to Physical Address (PA) translation at the I-Stage.
    - Computation of the (Current PC + 4), at the Pre-I-Stage.
- Signal *edp_iva_p*:
    - A subset of bits is directly used (not through a register) for indexing the I$. As we explained in Lab 14, the inputs to the instruction memory must be ready at the clock edge, thus, during the Pre-I-Stage, the next PC is provided to the instruction memory so that it can be read along the I-Stage. Moreover, as we mentioned earlier, the I$ in microAptiv is virtually indexed (you can read more about this policy in Section B.3, *Sixth Optimization* of [3]).
    - A subset of bits are registered (*ival_i <= edp_iva_p*) and used in the next cycle for the computation of the Tag.

b. I-Stage

During the I-Stage, when the I$ access is enabled (*rd_strobe=1*), the I$ is read. As explained in Lab 21, the I$ is implemented in module **m14k_ic**, and it is made up of the Data Array (module **dataram_2k2way_xilinx**), the Tag Array (module **tagram_2k2way_xilinx**) and the Way Select Array (module **i_wsram_2k2way_xilinx**). The I$ is managed by the I$ Controller (module **m14k_icc**), and, in our examples, it is configured as a 4KB 2-Way Set-Associative cache (i.e. 2KB per Way).

The I$ read strobe (*rd_strobe*) and the index for reading/writing the I$ (*icc_dataaddr*[13:2]) must be available at the clock edge. Thus, as explained in the previous subsection, a subset of *edp_iva_p* (Next PC) is directly used (i.e. not through a register) from the Pre-I-Stage as the index for reading the I$ (*icc_dataaddr = edp_ival_p = edp_cacheiva_p = edp_iva_p*). That index (*icc_dataaddr*[13:2]) contains the number of bits required for indexing the maximum possible I$ size of 16KB per Way (14 bits). Note that the least 2 significant bits are not included in the index, as the access to the I$ is word-aligned. Also, when a smaller I$ is used, the unnecessary bits are just ignored (in our example, only bits [10:2] of *icc_dataaddr* are actually used for accessing the I$).

The 2-Way Set-Associative I$ outputs 2 32-bit instructions (one instruction per way), through signal *data_rd_data[63:0]*. If a hit takes place, the **datamux** multiplexer, included in the instruction cache controller (module **m14k_icc**), selects the correct instruction, which is assigned to signal *icc_idata_i* and then registered to the next stage (E-Stage) into signal *mpc_ir_e* (which is the Instruction Register, IR). The following code segments, extracted from module **m14k_mpc_dec** (lines 1698 and 1705), perform the updating of the Instruction Register.

```
mvp_cregister_wide #(32) _int_ir_e_31_0_(int_ir_e[31:0],gscanenable, (mpc_run_ie |
mpc_fixupi) & ~mpc_atomic_e & ~mpc_lsdc1_e, gclk, icc_idata_i);
…
assign mpc_ir_e[31:0] = sel_hw_e ? hw_ir_e : int_ir_e;
```

For determining if a hit takes place, at the I-Stage, in parallel with the I$ access, the Current PC (stored in signal *edp_cacheiva_i*) is translated by the Memory Management Unit (top-level module **m14k_tlb**). After translation, the Physical Page Number (provided at the output of the MMU through signal *mmu_ipah[`M14K_PAH]*) is concatenated with the Virtual Address Offset (*ival_i[`M14K_PALHI:1]*), obtaining the tag requested by the instruction in signal *icc_ipa[31:10]*. The following code segments, extracted from module **m14k_icc** (lines 1274 and 1157), perform this concatenation.

```
assign icc_tagcmpdata [31:9] = … {icc_ipa[31:10], ~mmu_icacabl};
…
assign icc_ipa [31:1] = {mmu_ipah[`M14K_PAH], ival_i[`M14K_PALHI:1]};
```

Given that the I$ in microAptiv is physically tagged, the tags read from the I$ Tag Array (*tag_cmp_pa[ASSOC*WIDTH-1:0]*) are compared (within module **m14k_cache_cmp**) with the tag requested by the instruction (*tag_cmp_data[WIDTH-1:0]*). This module generates two outputs: *cachehit*, which determines if a hit or a miss take place, and *line_sel*, which is used as the control signal in the **datamux** multiplexer for selecting the correct way in case of a hit. The following code segments, extracted from module **m14k_icc** (lines 1826 and 2088), correspond to the comparator and the multiplexor from Figure 2.

```
m14k_cache_cmp #(22, `M14K_MAX_IC_ASSOC) tagcmp (
        .tag_cmp_data( icc_tagcmpdata[31:10]),
        .tag_cmp_pa( tag_cmp_pa ),
        .tag_cmp_v( tag_cmp_v ),
        .valid( valid_raw ),
        .line_sel( line_sel_raw ),
        .hit_way( cache_hit_way_raw ),
        .cachehit( cache_hit_raw ),
         .spram_support( spram_support )
);
…
m14k_cache_mux #(48, `M14K_MAX_IC_ASSOC) datamux (
        .data_in( cache_recode[(48*(1+`M14K_MAX_IC_ASSOC))-1:48] ),
        .waysel( isachange1_i_reg ? isachange_waysel :  data_line_sel | spram_sel ),
        .data_out( datamuxout )
);
```

In the pipelined processor introduced in *DDCA* [2] and illustrated in Figure 7.58 of that book, the analogous stage to the I-Stage is called Fetch Stage. Along the Fetch Stage, the Instruction Memory, a black box which provides the requested instruction in one cycle (i.e. it is an ideal memory), is accessed. Note that, in case of a hit in the I$ of microAptiv, the high-level behavior of the I-Stage and the Fetch-Stage is the same, as the I$ is also capable of retrieving the requested instruction in 1 cycle.

4. Example Simulation

In this section we perform the simulation of the assembly program shown in Figure 3 (provided in folder *Lab22_CacheController_HitAndMissManagement\Simulations\SimulationSources_I$Hit*), which includes an infinite loop. You can focus on any iteration of the loop except on the first one, as our purpose is to analyze an I$ hit.

As in the previous sections, configure the I$ as a 4KB 2-Way Set-Associative cache (i.e. 2KB per Way). Moreover, as explained in Lab 14, make a copy the soft-core and create a new Vivado project or reuse an existing one, add the 8 memory files and the waveform configuration file (*testbench_boot_behav_I$-Hit.wcfg*), and configure the simulator properly. Figure 4 illustrates the results of the simulation of the program from Figure 3, focusing on the fetch of the add instruction.

```
80000204 <main>:
80000204:   240b0002    li    t3,2
80000208:   240c0003    li    t4,3
8000020c:   018b6820    add   t5,t4,t3
80000210:   1000fffc    b     80000204 <main>
80000214:   01ab7022    sub   t6,t5,t3
```

Figure 3. Example assembly program.

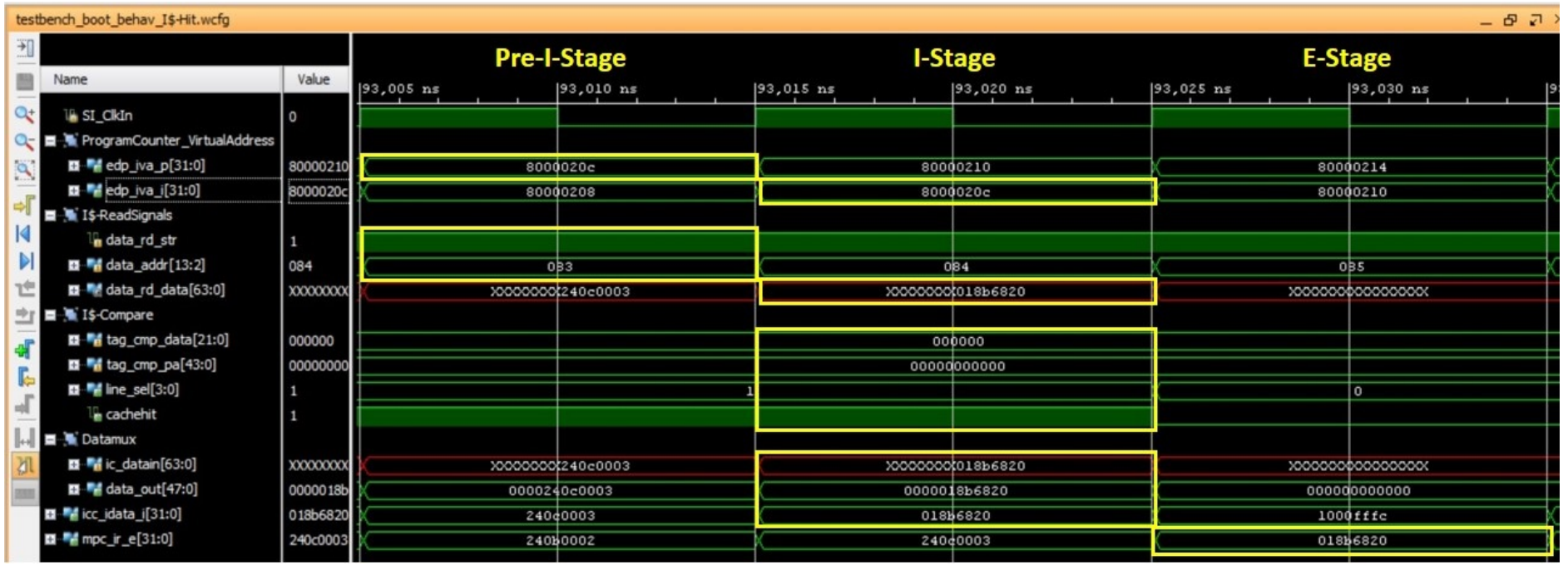

Figure 4. Timing diagram of the fetch of the second instantiation of an *add* instruction.

Next we detail the results obtained in the fetch of the second instantiation of the add instruction in the code from Figure 3.

- 1st cycle, Pre-I-Stage of the add instruction:
  - The Next PC is computed (*edp_iva_p*=0x8000020c) as *edp_iva_i + 4*. A subset of the Next PC bits is passed to the I$ as the index for reading it in the following cycle (*dataaddr*=0x83).
  - The Read Strobe is set to 1 (*data_rd_str*=1) for enabling the I$ reading in the next cycle.
  - At the end of this first cycle, the Next PC is registered (*edp_iva_i* <= *edp_iva_p_n*).
- 2nd cycle, I-Stage of the add instruction:
  - The I$ is read, using the signals received in the previous cycle. Signal *data_rd_data*=0xXXXXXXXX018b6820. Note that one way has not been initialized, thus it is undetermined (0xXXXXXXXX), whereas the other way contains the add instruction (0x018b6820).
  - In parallel with the I$ access, the Virtual Address (*edp_iva_i*) is translated to the Physical Address (*mmu_ipah*) in the MMU.
  - Tag comparison:
    - *tag_cmp_data[21:0]*=0x000000: This is the tag requested by the instruction, obtained by concatenating the PA number and the VA offset.
    - *tag_cmp_pa[43:0]*=0x00000000000: These are the two tags (2-Way I$) obtained from the I$ Tag Array.
    - Signals *cachehit*=1 and *line_sel[3:0]*=0001: Hit in the Way_0.
  - Datamux:
    - The way where a hit is found is selected and assigned to *icc_idata_i[31:0]*=0x018b6820.
  - The instruction read from the I$ is registered at the end of this cycle (*mpc_ir_e* <= *icc_idata_i*).
- 3rd cycle, E-Stage of the add instruction:
  - The add instruction is at the instruction register (*mpc_ir_e*=0x018b6820), and it goes through the E-Stage.

5. Exercises

Exercise 1: D$ Hit Management

Sketch the Verilog code related with the access to a 4KB 2-Way Set-Associative D$ (i.e. 2KB per Way) performed by a `lw` instruction. Figure 5 illustrates the main structures and signals involved in the M-Stage. Note that everything is handled analogously to the I$ access. Then, perform a simulation and analyze a D$ hit. You can use the memory files provided in folder *Lab22_CacheController_HitAndMissManagement\Simulations\SimulationSources_D$Hit* and the waveform configuration file *testbench_boot_behav_D$Hit*.

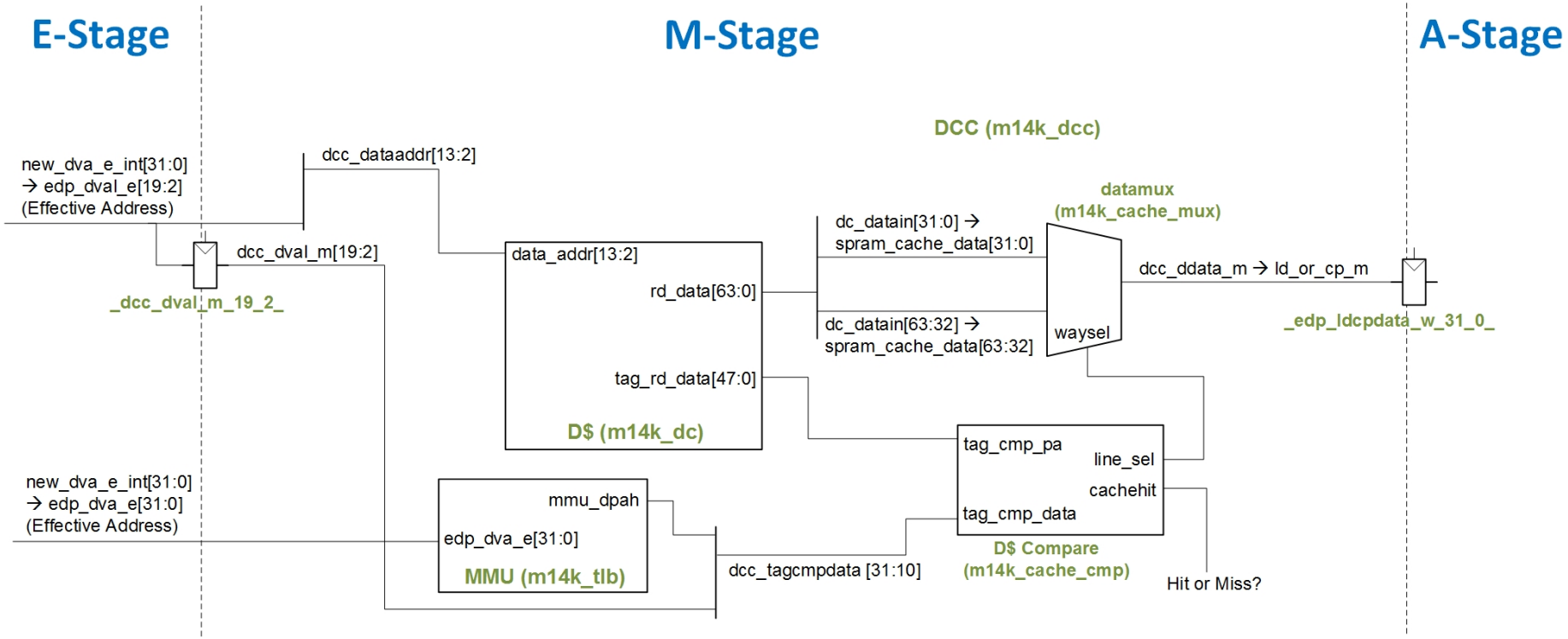

Figure 5. Structures and signals involved in the M-Stage of a `lw` instruction (2-way D$).

6. References

[1] “MIPS Technologies® - Data Cache Control Module Definition”, 2001.

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

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