Lab 24

The Cache Controller – The Store Buffer and the Fill Buffer

Lab 24
The Cache Controller – The Store Buffer and the Fill Buffer

1. Introduction

In this final lab concerning the cache controller, we analyze the operation of two important structures included in the D$ controller: the Store Buffer, which temporarily holds the data to write in the D$ by a store, and the Fill Buffer, which temporarily holds the block to fill into the D$ after a miss. Note that the I$ controller also has a (smaller) fill buffer but does not include a store buffer, as the I$ is never written from the core.

The Store Buffer extracts the store writing of the critical path, allowing store instructions to be *non-blocking* (i.e. a D$ miss caused by a store instruction does not stall the pipeline). The Fill Buffer extracts the block filling after a miss of the critical path, allowing the pipeline to resume execution before the missed line has been copied into the D$.

2. The Store Buffer – Analysis of a D$ hit caused by a `sw` instruction

In this section we analyze the Store Buffer by means of an example of a `sw` instruction that hits in the D$. We first explain some theoretical concepts and then perform a guided simulation.

a. Explanation

Many actions performed by the `sw` instruction are handled identically to a `lw` instruction: for example, during the E-Stage of a `sw` the Effective Address is computed as explained in Lab 16 for a `lw`; or, as another example, during the M-Stage of a `sw`, a D$ hit or a miss is determined exactly as described in Lab 22 for a `lw`. However, whereas the aim of a `lw` instruction is to bring a word from the D$ to the register file, a `sw` instruction writes a word from the register file to the D$. Thus, as we explain below, several actions are handled differently in the `sw` instruction.

Figure 1 highlights the main new actions carried out by a `sw` instruction. Most things shown in Labs 14 (`add` instruction), 16 (`lw` instruction) and 22 (hit and miss management) are omitted here for the sake of simplicity. As shown in the figure, at the E-Stage, the word to write by the `sw` is read from the register file (signal *rf_bdt_e[31:0]*) or forwarded from a subsequent stage (signal *prebbus_e[31:0]*), then it is prepared at the Shifter/Rotator/Insert/Extract (SRIE) unit, available in the *m14k_edp* module, and finally it is registered to the M-Stage (register *_shf_rot_m_31_0_*). Besides, at the E-Stage, the Effective Address is computed as explained in Lab 16 for the `lw` instruction and registered to the M-Stage (register *_dcc_dval_m_19_2_*).

At the M-Stage, differently to load instructions, store instructions do not read the D$ Data Array, but only read the D$ Tag Array to determine if a hit or a miss takes place as explained in Lab 22-A. In order to decouple the Core and the D$, at the end of the M-Stage, the word to write to the D$ (signal *storebuff_data_mx[31:0]*) and the index (signal *storebuff_idx_mx [19:2]*) are stored in an intermediate one-word buffer, called the Store Buffer (Figure 1), which allows the pipeline to continue executing instructions both for a hit (as we show in this subsection) and for a miss (as we show in the next subsection). Eventually, the word held in the Store Buffer is written in the D$ Data Array and the Store Buffer is released.

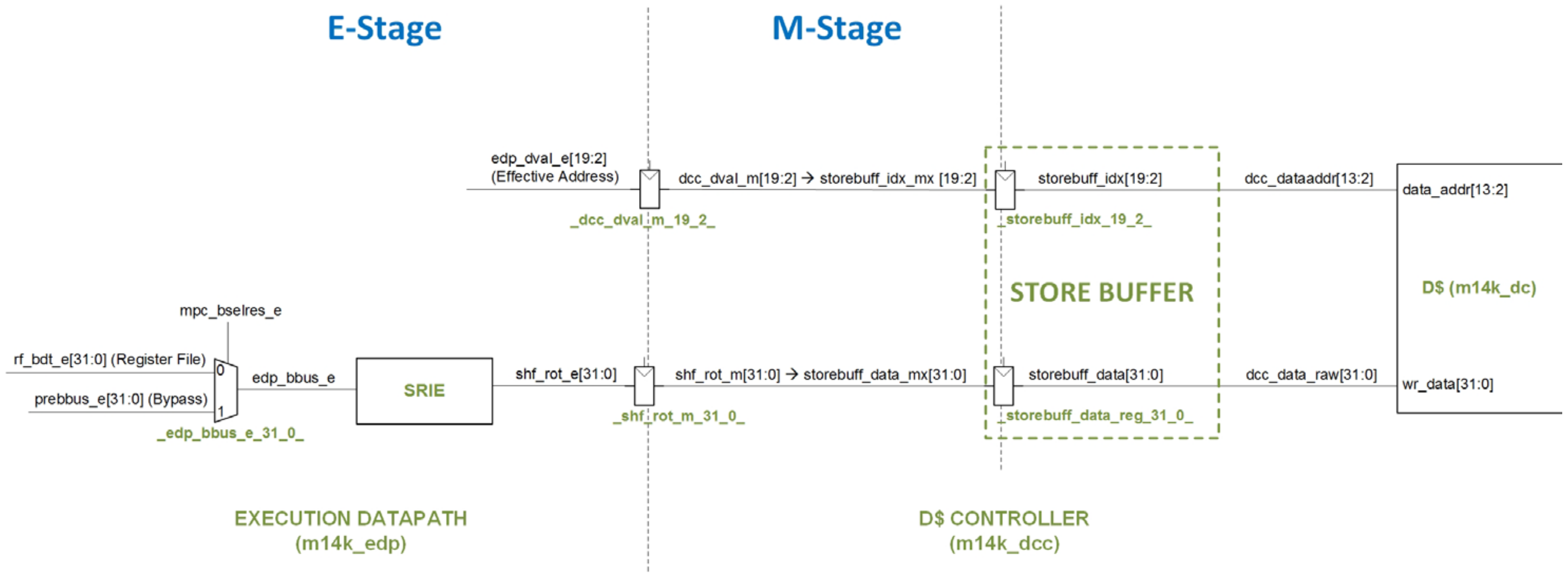

Figure 1. The Store Buffer.

The D$ Tag Array needs not be modified when a `sw` hits in the D$. Instead, the Way-Select Array must be modified in order to update the LRU state and, in the case of a *write-back* D$, to mark the written way as dirty. Note that, if the D$ uses a *write-through* policy, there is no dirty bit, as the new word is immediately updated in memory. For that purpose, the word to write in the D$ is also buffered in the Bus Interface Unit.

b. Example Simulation

In this section, we illustrate an example simulation of the program shown in Figure 2, for a processor with a 4KB 2-Way Set-Associative D$ (i.e. 2KB per Way). We avoid the first iteration of the loop, as we want to force both I$ and D$ hits (note that the `sw` instruction hits in the D$ as we are accessing the same array component in every iteration). The simulation source files are provided in folder *Lab24_CacheController_StoreBufferAndFillBuffer\Simulations\SimulationSources_StoreD$Hit* and the waveform configuration file is provided in *Lab24_CacheController_StoreBufferAndFillBuffer\Simulations\testbench_boot_behav_StoreD$Hit.wcfg*. As explained in Lab 14, make a copy the soft-core and create a new Vivado project, or reuse an existing one. Moreover, add the 8 memory files and the waveform configuration file, and configure the simulator accordingly.

```
80000204:   240a000a    li    t2,10
80000208:   3c0e8000    lui   t6,0x8000
8000020c:   25ce0250    addiu t6,t6,592
80000210 <loop>:
      80000210:   00000000    nop
      80000214:   adca0004    sw    t2,4(t6)
      80000218:   00000000    nop
      8000021c:   254a0001    addiu t2,t2,1
      80000220:   1000fffb    b     80000210 <loop>
      80000224:   00000000    nop
```

Figure 2. Example assembly program, with the sw instruction highlighted in blue.

In file **main.c** you can also view definition and initialization of the *test_data[10]* array.

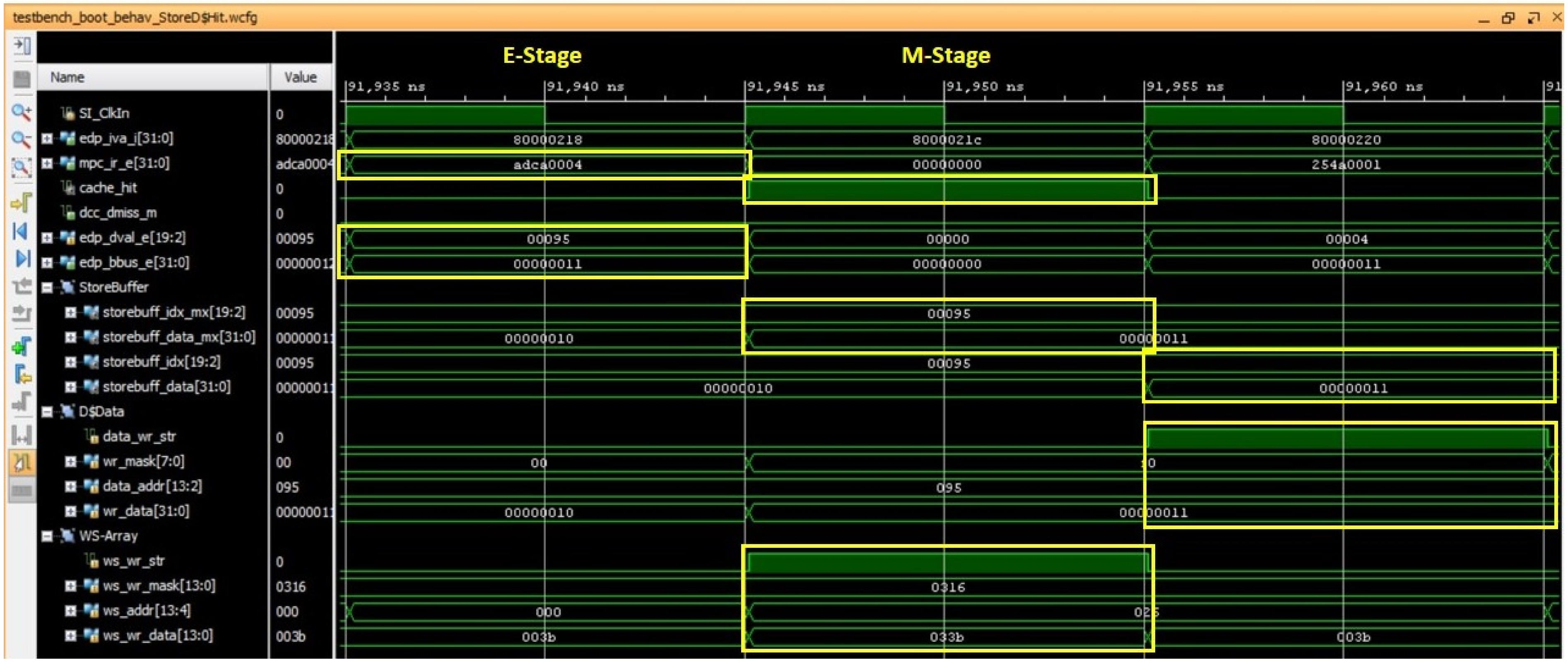
testbench_boot_behav_StoreD$Hit.wcfg
E-Stage
M-Stage
Name
Value
91,935 ns
91,940 ns
91,945 ns
91,950 ns
91,955 ns
91,960 ns
SI_ClkIn 0
edp_jva_i[31:0] 80000218 80000218 8000021c 80000220
mpc_ir_e[31:0] adca0004 adca0004 00000000 254a0001
cache_hit 0
dcc_dmiss_m 0
edp_dval_e[19:2] 00095 00095 00000 00004
edp_bbus_e[31:0] 00000012 00000011 00000000 00000011
StoreBuffer
storebuff_idx_mx[19:2] 00095 00095
storebuff_data_mx[31:0] 00000011 00000010 00000011
storebuff_idx[19:2] 00095 00095
storebuff_data[31:0] 00000011 00000010 00000011
D$Data
data_wr_str 0
wr_mask[7:0] 00 00 f0
data_addr[13:2] 095 095
wr_data[31:0] 00000011 00000010 00000011
WS-Array
ws_wr_str 0
ws_wr_mask[13:0] 0316 0316
ws_addr[13:4] 000 000 025
ws_wr_data[13:0] 003b 003b 033b 003b

Figure 3. Timing diagram for the program shown in Figure 2.

We next explain the results of the simulation shown in Figure 3.

- 1st cycle, `sw` E-Stage:
  - The `sw` instruction (*mpc_ir_e*=0xadca0004) computes the Effective Address (*edp_dval_e[19:2]*=0x00095) as explained in Lab 16. The Effective Address is used to access the D$ in the next cycle.
  - The data to write by the `sw` is read from the Register File (*edp_bbus_e*[31:0]=0x00000011) and properly prepared by the SRIE unit. Note that this value depends on the iteration that you chose to analyze.
- 2nd cycle, `sw` M-Stage:
  - D$ hit or miss is determined as explained in Lab 22-B. In this case, a hit is detected (*cache_hit*=1 and *dcc_dmiss_m*=0).
  - The index and the data to write are inserted in the Store Buffer at the end of this cycle (*storebuff_idx* <= *storebuff_idx_mx* = 0x00095 and *storebuff_data* <= *storebuff_data_mx* = 0x00000011).
  - The LRU state is updated in the WS Array, setting as MRU the way where the new word is stored.
  - The pipeline continues executing instructions even though the `sw` instruction has not written the new data to the D$ yet.
- 3rd cycle:
  - The store buffer is updated with the new value to write (*storebuff_data* = 0x00000011).
  - The inputs to write the D$ in the next cycle are established along this cycle: *data_wr_str*=1, *data_addr*=0x095, *wr_data*=0x00000011.

3. The Fill Buffer – Analysis of a D$ miss caused by a `sw` instruction

In this section we analyze the Fill Buffer by means of an example of a `sw` instruction that misses in a *write-allocate* D$. We first explain some theoretical concepts and then perform a guided simulation.

a. Explanation

Lab 22-B explains many actions performed when a D$ miss is detected for a `lw` instruction. In that lab we focus on the main actions carried out along the miss handling of the `lw`: stop the execution (load instructions are *blocking* in microAptiv), request the missed block through the bus, and restart the execution as soon as possible with the use of the *critical word bypass* policy. In this section, our aim is to look ahead a bit further after the miss, mainly explaining how the new line is filled into the D$ when a miss is detected for a `sw` instruction.

As explained in Lab 22-B, at the M-Stage, a `lw`/`sw` instruction determines if a D$ hit or miss takes place. When a D$ miss is detected, the block containing the word that the `lw`/`sw` instruction reads/writes must be filled into the D$ (except in the case of a `sw` instruction executing in a processor with a *write-no-allocate* D$). As we advanced in Lab 22-B, this block is not filled directly into the D$ but through an intermediate structure named Fill Buffer. Thanks to the cooperation between the Store Buffer and the Fill Buffer, there is no need to stop/restart the pipeline when a miss is detected for a store instruction (i.e. stores are *non-blocking*).

The Fill Buffer, included in the data cache controller, holds temporarily the missed line brought from main memory. Figure 4 extends Figure 1, including the Fill Buffer (module *m14k_dcc_fb*). This structure contains 2 entries, named *front entry* and *back entry*, each one with the same size as a D$ block (i.e. 4*32B). By default, the *front entry* (corresponding to registers *_fb_data*_31_0_*) receives the missed block directly from the BIU through signal *biu_datain[31:0]*. If the front entry is occupied, the *back entry* (corresponding to registers *_fb_data_back*_**) is used, so that, when the *front entry* empties, the block stored in the *back entry* is transferred to the *front entry* (through signals *trans_fb_data**). In addition to these two entries, the Fill Buffer also provides to the D$ the index where the new line must be stored. This index is computed by concatenating two signals: *fb_index[19:4]*, which is obtained from the missed address (*dcc_dval_m[19:2]*) and is held in register *_fb_index_19_4_*, and *raw_fill_dword[3:0]*, which provides the least significant bits of the indexes of the four words conforming the missed cache line (computation of this signal is not detailed in Figure 4, so it can be sketched by the students as an exercise). The Fill Buffer fills the new block into the D$ opportunistically (i.e. when the D$ is free), and does not need to wait for the whole line to be in the buffer before the fill into the D$ begins.

In the case of a D$ miss caused by a `sw` instruction, one of the words within the missed block must be modified. As explained in the previous section for a D$ hit, the new word is held temporarily in the Store Buffer, from where it is eventually updated in the D$. However, in the case of a miss, the word stored in the Store Buffer (signal *storebuff_data[31:0]*) is not written directly to the D$, but transferred to the Fill Buffer (observe that Figure 4 includes a path from the Store Buffer into the Fill Buffer), where it is merged with the new line before being filled into the D$.

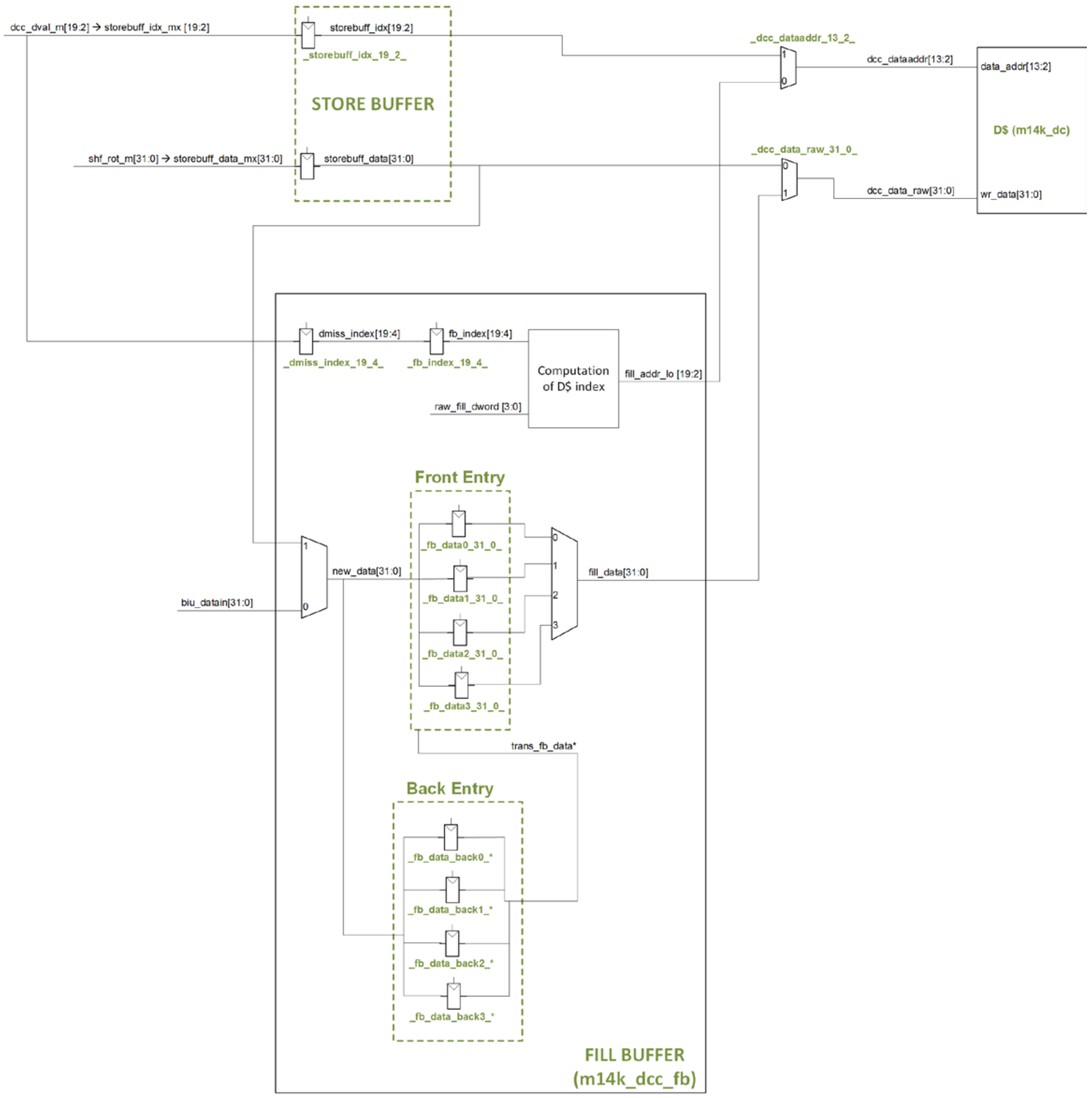

Figure 4. The Fill Buffer.

b. Example Simulation

In this section, we illustrate an example simulation of the program shown in Figure 5, for a processor with a 4KB 2-Way Set-Associative *write-allocate* D$ (i.e. 2KB per Way). The program is similar to the one used in the previous section (Figure 2) except that the `sw` instruction is forced to access a different cache block in the second iteration so that it misses in the D$. We focus on the second iteration of the loop, as we want to force I$ hits but a D$ miss. The simulation source files are provided in folder *Lab24_CacheController_StoreBufferAndFillBuffer\Simulations\SimulationSources_StoreD$Miss* and the waveform configuration file is provided in *Lab24_CacheController_StoreBufferAndFillBuffer\Simulations\testbench_boot_behav_StoreD$Miss.wcfg*. Recall from Lab 23 that we can configure the D$ allocation policy in file **boot.S**.

```
80000204:   24090002    li    t1,2
80000208:   240a0000    li    t2,0
8000020c:   3c0e8000    lui   t6,0x8000
80000210:   25ce0260    addiu t6,t6,608
80000214 <loop>:
      80000214:   00000000    nop
      80000218:   adca0004    sw    t2,4(t6)
      8000021c:   00000000    nop
      80000220:   254a0001    addiu t2,t2,1
      80000224:   25ce0010    addiu t6,t6,16
      80000228:   0149082a    slt   at,t2,t1
      8000022c:   1420fff9    bnez  at,80000214 <loop>
      80000230:   00000000    nop
      80000234:   1000ffff    b     80000234 <loop+0x20>
      80000238:   00000000    nop
```

Figure 5. Example assembly program.

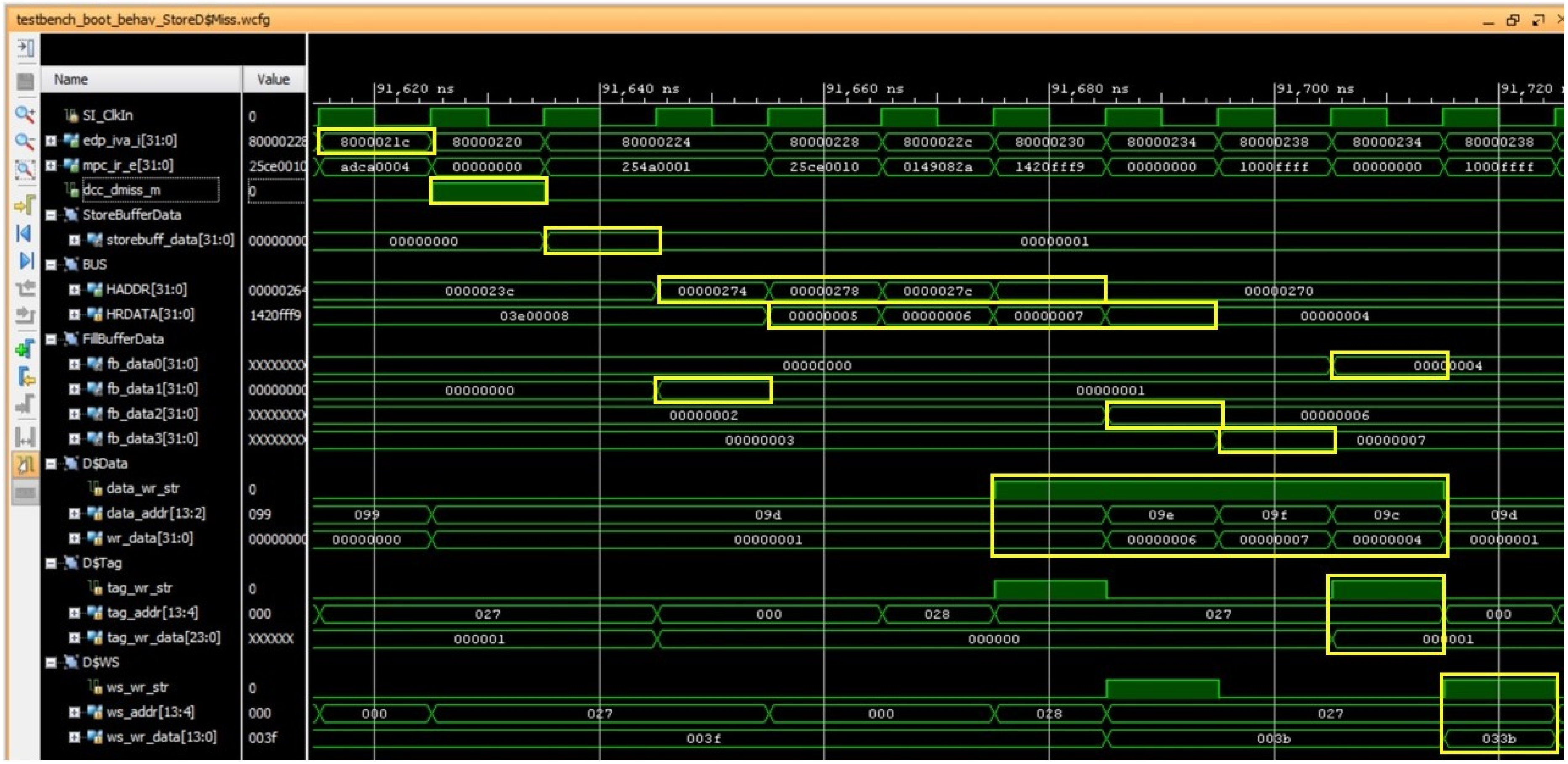

Figure 6. Store D$ miss.

Figure 6 shows the simulation of the program from Figure 5. Below, we discuss the results.

- 1st cycle, `sw` E-Stage:
  - The store instruction (*mpc_ir_e*=0xadca0004) calculates the Effective Address and passes it as the index to access the D$ in the next cycle.
- 2nd cycle, `sw` M-Stage:
  - The `sw` instruction determines if there is a D$ hit or a D$ miss as explained in Lab 22. In this case, a miss is detected (*dcc_dmiss_m*=1).
  - Note that the execution continues as store instructions are non-blocking.
- 3rd cycle:
  - The data to write by the `sw` instruction is saved in the store buffer (*storebuff_data*=0x00000001).
- 4th cycle:
  - Data from the Store Buffer is transferred to the second word of the *front entry* (*fb_data1*=0x00000001), where it will be combined with the missed line once it is provided from main memory.
- 4th to 8th cycles:
  - The missed line is requested to (*HADDR[31:0]*) and returned from (*HRDATA[31:0]*) main memory.
- 8th to 10th cycles:
  - The missed line is filled into the Fill Buffer (*fb_data2*=0x6, *fb_data3*=0x7 and *fb_data0*=0x4). Observe that the second word, which was already in the Fill Buffer (*fb_data1*=0x1), is not modified with the word brought from main memory.
- 7th to 10th cycles:
  - The missed line is written from the Fill Buffer to the D$ Data Array (*data_wr_str*=1). Note that the second word was provided from the `sw` instruction (*wr_data*=0x1), whereas the first, third and fourth words were provided from main memory.
- 10th cycle:
  - The tag of the missed line is stored into the D$ Tag Array (*tag_wr_str*=1).
- 11th cycle:
  - The dirtiness and LRU states of the missed line are stored into the D$ WS Array (*ws_wr_str*=1).

4. Exercises

Simulate the following sequences, analyze them, and explain the results (we provide the simulation source files and the waveform configuration files for this exercise in folder *Lab24_CacheController_StoreBufferAndFillBuffer\Simulations\Sequences*). You should also sketch the logic involved in each sequence.

1. **Sequence 1.** Use a *write-allocate* D$ (Lab 23 explains how to configure the write and allocation policies of the D$). A `sw` instruction misses, and it is followed by a `lw` instruction to the same word. You should analyze the second iteration of the loop, where all instructions fetches hit in the I$.
2. **Sequence 2.** Use a *write-allocate* D$. A `sw` instruction misses, and it is followed by a `lw` instruction to a different word within the same line.
3. **Sequence 3.** A `lw` instruction misses, and it is followed by another `lw` instruction to a different word within the same line.
4. **Sequence 4.** A `lw` instruction misses, and it is followed by a `sw` instruction to a different word within the same line.