MIPSfpga by Imagination

Lab 23

The Cache Controller – Content Management Policies

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

Lab 23
The Cache Controller – Content Management Policies

1. Introduction

In this lab we analyze the different policies available in microAptiv for managing the data cache (D$) content, although most explanations are also valid for the instruction cache (I$). Specifically, we study and experiment with three policies: the **allocation policy**, the **replacement policy**, and the **write policy**. Recall that, as explained in previous labs, microAptiv contains one level of cache, with separate data and instruction caches. It uses 16B blocks (i.e. 4 words per block), and supports 1-way (usually called direct-mapped) and 2/3/4-way set-associative caches, where each way can have a variable size in the range from 1KB to 16KB.

When a cache miss takes place, the **allocation policy** decides if the missed block is copied into the cache (*allocate*) or not (*no-allocate*). The missed block is usually brought to the cache under a read miss, i.e. a read-allocate policy is usually employed. As for write accesses, under some scenarios it may be more convenient to use a *no write allocate* policy, in which the block is not copied to the cache under a write miss but directly updated in main memory.

In the case of set-associative caches, in addition to the allocation policy, a **replacement policy** must be used, that, when a block is going to be inserted in a set that is already full (i.e. all ways of the set hold valid blocks), decides which block to evict (obviously, if the set is not full, the inserted block is stored in an empty way). The most typical replacement policy, and also the one used in microAptiv, is *Least Recently Used* (LRU). Based on the principle of temporal locality, this policy replaces from the cache the block which was referenced the furthest in the past.

Finally, another policy must be used for write operations. Store instructions modify the contents of a block in the cache (except under a write miss in a cache using a *no write allocate* policy). The modified block must be updated in main memory. A **write policy** decides *when* that updating takes place. As such, if a *write-back* policy is employed, main memory updating is postponed to the time when the modified block is evicted from the cache. A dirty bit, associated with each block in the cache, is necessary in this case for the cache controller to be able to determine if the block must be written to main memory or not when an eviction takes place. Conversely, in a *write-through* policy, the write is performed simultaneously to the cache and to main memory, and thus no dirty bits are required in the cache.

2. Replacement Policy

In this section we first explain the replacement policy used in microAptiv and how it is implemented, and then we provide an example supporting these explanations. MicroAptiv implements a *true* LRU replacement policy, meaning that the LRU algorithm is exact, as opposed to pseudo-LRU approximate algorithms typically used in many processors. As explained in Lab 21, the WS Array in the cache stores the LRU bits, which can go from 0 (in a direct-mapped cache configuration) to 6 bits per set (in a 4-way set-associative cache configuration). Specifically, for the cache configuration provided by default in MIPSfpga (2-way set-associative), only 1 bit is required for encoding the LRU state of each set. In this Section we analyze the LRU algorithm for the highest supported associativity in microAptiv, 4-ways, as the remaining configurations can be easily understood once this one is explained.

Figure 1 illustrates the interpretation of the 6-bit LRU state for two scenarios. The ways in each set are fictitiously linked by pairs, where each link has an arrow direction determined by the LRU bit associated with that pair (for example, LRU[2] is associated with the pair *Way0* - *Way1*). On the left-hand side of Figure 1, all the LRU bits are assumed to be 0 (LRU[5:0]=000000). On the right-hand side of Figure 1, the opposite scenario is shown, where LRU[5:0]=111111 and all the arrows go in the opposite direction.

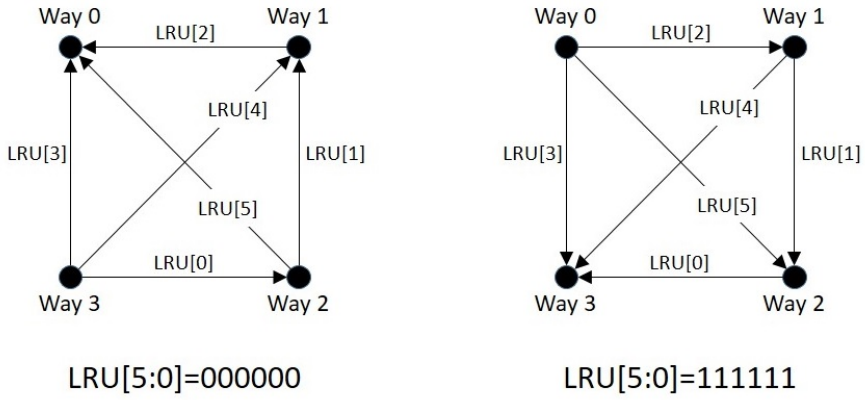

Figure 1. Interpretation of the LRU bits.

a. LRU decoding algorithm

In this subsection we explain the decoding algorithm, used for determining which block to replace. We first explain the algorithm theoretically and then describe the implementation used in microAptiv.

i. Theoretical explanation

When a new block is inserted in the cache, if its destination set is full, a block within that set must be replaced. For that purpose, the LRU state for that set is read from the WS Array, and the *decoding* algorithm of the LRU replacement policy, implemented in module **m14k_dcc**, is

used for determining which block to replace. Based on the interpretation of the LRU state explained in Figure 1, the way to replace is determined by going in the direction of the arrows until we cannot go any further. Put it another way, the way to replace is the one with 3 arrows pointing into it. As such, in the examples from Figure 1, the algorithm would select Way 0 under the scenario on the left-hand side of the figure, whereas it would select Way 3 under the scenario on the right-hand side.

ii. Implementation

The *decoding* algorithm is implemented within the **m14k_dcc** module (you can find it by searching for tag "*LRU Replacement algorithm - (decode)*" in that module). The way to replace is determined by signal *drepl[3:0]*. This is a one-hot signal (meaning that only one of its bits can be 1 at the same time), in which the way selected to replace is 1. Signal *drepl[3:0]* is computed as follows:

```
// decode
assign drepl[0] = ~unavail[0] & ((~lru[5] | unavail[2]) & (~lru[3] | unavail[3]) & (~lru[2] | unavail[1]));
assign drepl[1] = ~unavail[1] & ((~lru[4] | unavail[3]) & (lru[2] | unavail[0]) & (~lru[1] | unavail[2]));
assign drepl[2] = ~unavail[2] & ((lru[5] | unavail[0]) & (lru[1] | unavail[1]) & (~lru[0] | unavail[3]));
assign drepl[3] = ~unavail[3] & ((lru[4] | unavail[1]) & (lru[3] | unavail[0]) & (lru[0] | unavail[2]));
```

Signal *lru[5:0]* is computed as follows:

```
// FB way encoded (see lru encoding diagram in lru encoding section)
assign lru[5] = (dc_wsin_full[5] & ~fb_repl[2]) | (fb_repl[0]);
assign lru[4] = (dc_wsin_full[4] & ~fb_repl[3]) | (fb_repl[1]);
assign lru[3] = (dc_wsin_full[3] & ~fb_repl[3]) | (fb_repl[0]);
assign lru[2] = (dc_wsin_full[2] & ~fb_repl[1]) | (fb_repl[0]);
assign lru[1] = (dc_wsin_full[1] & ~fb_repl[2]) | (fb_repl[1]);
assign lru[0] = (dc_wsin_full[0] & ~fb_repl[3]) | (fb_repl[2]);
```

For the sake of simplicity, assume that *fb_repl[3:0]*=0000, thus *lru[5:0]* = *dc_wsin_full[5:0]*, which contains the LRU state read from the WS Array for the involved set.

Signal *unavail[3:0]* determines, for every way within a set, if the way is selectable for replacement or not. A way may not be selectable for replacement under different situations. For example, in a 2-way set-associative cache, *unavail[3:0]*=1100, thus Ways 2 and 3 are not available. Or, as another example, a way can be locked, meaning that it cannot be replaced. In that case, *unavail[i]*=1, being *i* the locked way.

b. LRU encoding algorithm

In this subsection we explain the encoding algorithm, used for updating the LRU state of a block. We first explain the algorithm theoretically and then describe the implementation used in microAptiv.

i. Theoretical explanation

When some data are requested, either for reading or writing it, the LRU bits of the set in which the data is placed must be updated according to Table 1. This table applies both for the case when the block is found in the cache (i.e. cache hit) and for the case when it is not found (i.e. cache miss, in which case the block is inserted from main memory into the cache, unless a *no write allocate* policy is used).

The purpose of the algorithm from Table 1 is to avoid a recently requested block to be selected in the forthcoming replacements in that set. This is achieved by moving away all the arrows from the way where the block is placed: the LRU bits corresponding to a write enable signal equal to 1 are modified, turning the arrows away from the way where the block is stored, whereas the LRU bits corresponding to a write enable signal equal to 0 are left unchanged. For example, suppose that the LRU state of a set is the one shown in the left-hand side of Figure 1 (LRU[5:0]=000000), and the block stored in Way 0 is read. According to Table 1, the LRU bits must be updated to LRU[5:0]=101100, turning all the arrows away from Way 0.

Table 1. LRU bits updating for a cache hit or a block insertion

Way	LRU bits to be written	Write enable
0	1X11XX	101100
1	X1X01X	010110
2	0XXX01	100011
3	X00XX0	011001

The LRU bits must also be updated when a block is invalidated, in which case the way where the block is stored becomes available for allocating a new block. Thus, in this case, the algorithm turns the arrows into the way where the block is stored, so that, in the next replacement in that set, the invalidated way is selected. Table 2 illustrates the LRU state updating under a block invalidation.

Table 2. LRU bits updating for a block invalidation

Way	LRU bits to be written	Write enable
0	0X00XX	101100
1	X0X10X	010110
2	1XXX10	100011
3	X11XX1	011001

One of the advantages of the LRU algorithm used in microAptiv is that the *encoding* algorithm needs not know the previous LRU state of the set, i.e. the LRU bits need not be read before updating them. This comes at the expense of using some more bits than the minimum that would be strictly necessary for storing the LRU state.

ii. Implementation

Similarly to the *decoding* algorithm, the *encoding* algorithm is implemented within the **m14k_dcc** module (you can find it by searching for the tag "*LRU update data - (encode)*" in that module). When some data are requested, the new LRU state for the set in which the data is placed is computed in two signals: *lru_data[5:0]*, which encodes the value of the second column of Table 1 and Table 2, and *lru_mask[5:0]*, which encodes the value of the third column of Table 1 and Table 2. Once computed, these two signals are communicated from the DCC to the Way-Select Array through bits [5:0] of signals *dcc_wswrdata[13:0]* and *dcc_wswren[13:0]* respectively. Two multiplexers are used for computing *lru_data[5:0]* and *lru_mask[5:0]*, as follows:

```
assign lru_mask[5:0] = (dcop_lru_write  | dcop_hit_nolru_reg) ?
       ({6{~dcop_hit_nolru_reg}}) : (cacheread_m & cache_hit) ? lru_hitmask[5:0] :
       lru_fillmask[5:0];
…
assign lru_data[5:0] = dcop_lru_write ? cpz_taglo[9:4] :
       (cacheread_m & cache_hit) ? lru_hitdata[5:0] : ev_wrtag_inv_ws & ~dcop_ws_valid ?
       ~(lru_filldata[5:0]) : lru_filldata[5:0];
```

Next we analyze these two multiplexers.

- When the access hits in the D$ (in which case Table 1 is used), `lru_mask[5:0]=lru_hitmask[5:0]` and `lru_data[5:0]=lru_hitdata[5:0]`. These two signals are computed as follows (note that signal *block_hit_way[3:0]* encodes the way where the hit takes place):

```
assign lru_hitmask[5] = (block_hit_way[2] | block_hit_way[0]);
assign lru_hitmask[4] = (block_hit_way[3] | block_hit_way[1]);
assign lru_hitmask[3] = (block_hit_way[3] | block_hit_way[0]);
assign lru_hitmask[2] = (block_hit_way[1] | block_hit_way[0]);
assign lru_hitmask[1] = (block_hit_way[2] | block_hit_way[1]);
assign lru_hitmask[0] = (block_hit_way[3] | block_hit_way[2]);
…
assign lru_hitdata[5] = (~block_hit_way[2] | block_hit_way[0]);
assign lru_hitdata[4] = (~block_hit_way[3] | block_hit_way[1]);
assign lru_hitdata[3] = (~block_hit_way[3] | block_hit_way[0]);
assign lru_hitdata[2] = (~block_hit_way[1] | block_hit_way[0]);
assign lru_hitdata[1] = (~block_hit_way[2] | block_hit_way[1]);
assign lru_hitdata[0] = (~block_hit_way[3] | block_hit_way[2]);
```

- When the access misses in the D$ (in which case Table 1 is used), and thus the block is inserted from main memory, `lru_mask[5:0]=lru_fillmask[5:0]` and `lru_data[5:0]=lru_filldata[5:0]`. These two signals are computed as follows (note that signal *ws_en[3:0]* encodes the way where the new block is allocated):

```
assign lru_fillmask[5] = (ws_en[2] | ws_en[0]);
assign lru_fillmask[4] = (ws_en[3] | ws_en[1]);
assign lru_fillmask[3] = (ws_en[3] | ws_en[0]);
assign lru_fillmask[2] = (ws_en[1] | ws_en[0]);
```

```
assign lru_fillmask[1] = (ws_en[2] | ws_en[1]);
assign lru_fillmask[0] = (ws_en[3] | ws_en[2]);
…
assign lru_filldata[5] = (~ws_en[2] | ws_en[0]);
assign lru_filldata[4] = (~ws_en[3] | ws_en[1]);
assign lru_filldata[3] = (~ws_en[3] | ws_en[0]);
assign lru_filldata[2] = (~ws_en[1] | ws_en[0]);
assign lru_filldata[1] = (~ws_en[2] | ws_en[1]);
assign lru_filldata[0] = (~ws_en[3] | ws_en[2]);
```

- Finally, when a block is invalidated (in which case Table 2 is used), `lru_mask[5:0]=lru_fillmask[5:0]` and `lru_data[5:0]=~lru_filldata[5:0]`. In this case, *ws_en[3:0]* encodes the way that must be invalidated.

c. Example

For completing the explanation of the LRU algorithm, let´s analyze the evolution of the LRU state of a given set for an example access pattern, assuming a 4-way set associative D$. The students can analyze on their own computation of signals *lru_data*[5:0] and *lru_mask*[5:0] for the *encoding* algorithm. The example is explained in the following items and in Figure 2.

1. Initially, LRU[5:0]=000000.
2. Assume that there is a **hit on Way 2**, thus, applying the *encoding* algorithm (Table 1), bits 0, 1 and 5 of the LRU state must change: 0XXX01. The new value for the LRU state is LRU[5:0]=000001. Figure 2 shows in red the links that change their direction.
3. Now, there is a **hit on Way 0**, thus, applying again the *encoding* algorithm (Table 1), bits 2, 3 and 5 of the LRU state must change: 1X11XX. The new value for the LRU state is LRU[5:0]=101101.
4. After the second hit, assume that there is **a miss**. Thus, a block must be replaced using the *decoding* algorithm. According to this algorithm, the block to replace is the one stored in the way with 3 arrows pointing into it, which is, in this case, Way 1. Thus, the new block is stored in Way 1 and the new LRU state, applying again the *encoding* algorithm (Table 1), is: LRU[5:0]=111011.
5. Now, assume that the block stored in **Way 0 is invalidated**. Thus, applying again the *encoding* algorithm (in this case according to Table 2), the LRU state changes to: LRU[5:0]=010011.
6. Finally, assume that another **miss** takes place. Using the decoding algorithm, Way 0 is selected for inserting the new block. However, in this case, no block is replaced, as Way 0 is empty. The LRU stage changes, as determined by the *encoding* algorithm (Table 1), to: LRU[5:0]=111111.

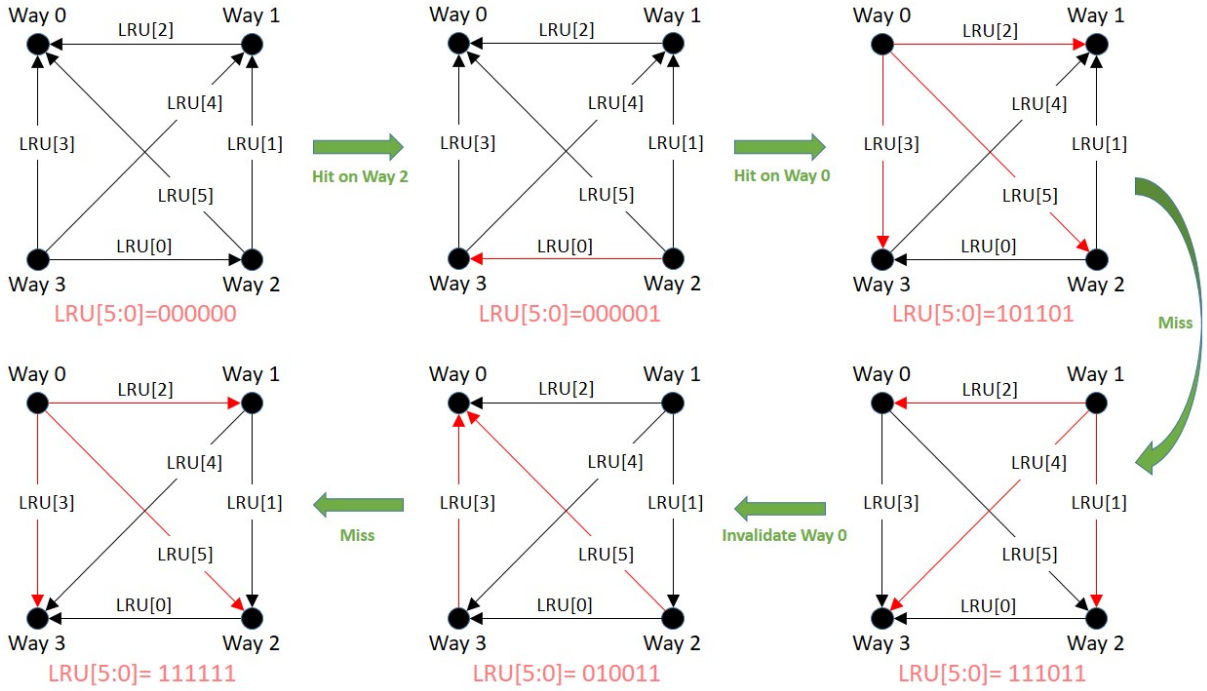

Figure 2. Evolution of the LRU state for an example pattern.

3. Write Policy and Allocation Policy

The write policy and allocation policy on MIPSfpga must be configured by software during the reset exception process. It is configurable per address segment (see Section 4.2.1 of [1]). Similarly to the performance counters, the write policy configuration is performed through the System Control Coprocessor (CP0). In this case, *CP0 Register* 16, with a *Select number* of 0, is used (see Section 6.2.30 of [1]). Figure 3 illustrates the different fields of this register. We are mainly interested on the right-most field, as it defines the *kseg0* cache coherency attributes (which includes the definition of the write policy and the allocation policy) and the programs in MIPSfpga are always mapped to that segment. Table 3 illustrates the encoding of the cache coherency attributes for the 3 possible combinations of cacheable policies.

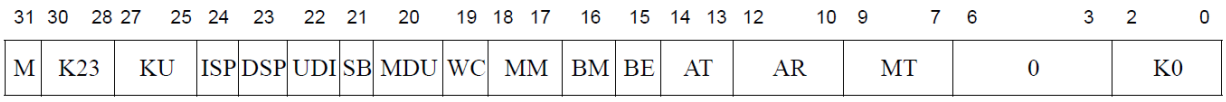

Figure 3. CP0 Register 16, Select 0.

Table 3. Cache coherency attributes encoding

Bits [2:0] of CP0 Register 16, Select 0	Cache Coherency Atrribute

0	Cacheable, non-coherent, write-through, no write allocate
1	Cacheable, non-coherent, write-through, write allocate
3*, 4, 5, 6	Cacheable, non-coherent, write-back, write allocate
2, 7	Uncached

* These two values are required by the MIPS32 architecture. In the microAptiv UP processor core, only values 0, 1, 2 and 3 are used. For example, values 4, 5 and 6 are not used and are mapped to 3. The value 7 is not used and is mapped to 2. Note that these values do have meaning in other MIPS Technologies processor implementations. Refer to the MIPS32 specification for more information.

The program **boot.S**, provided as part of MIPSfpga boot software, initializes the caches. At line 108 (tag *enable_k0_cache*), the cache coherency attributes are configured as follows:

```
mfc0    v0, C0_CONFIG           # read Config
li      v1, 3                   # CCA for single-core processors
ins     v0, v1, 0, 3            # insert K0
mtc0    v0, C0_CONFIG           # write Config
```

Let´s analyze this code:

- `mfc0 v0, C0_CONFIG`: The value of *CP0 Register* 16, and *Select number* 0 (recall that *C0_CONFIG* is defined in file *m32c0.h*, included in Imagination Codescape) is copied into register *v0*.
- `li v1, 3`: Register *v1* is initialized with a value of 3 (note that, according to Table 3, a value of 3 corresponds to a *write-back, write-allocate* policy).
- `ins v0, v1, 0, 3`: The right-most 3 bits of *v1* are merged with *v0* starting at position 0.
- `mtc0 v0, C0_CONFIG`: The value contained in *v0* is copied to *CP0 Register* 16, *Select number* 0.

Thus, in order to modify the write and allocation policies of segment *kseg0*, we just have to change the value assigned to *v1* (`li v1, 3`), according to Table 3, i.e.:

- *Write-through* and *no write allocate* (K0 field = 0)
- *Write-through* and *write allocate* (K0 field = 1)
- *Write-back* and *write allocate* (K0 field = 3)

4. Exercises

Exercise 1: Write and allocation policies

In this exercise you compare the different write and allocation policies available in microAptiv, using the following program:

```
volatile int A[32];
volatile int B[32];
```

```
volatile int C[32];
volatile int R[16];
volatile int D[512];
volatile int result, i;

result=0;
for (i = 0; i < 32; i = i + 1){
        A[i] = i;
        B[i] = i;
}
for (i = 0; i < 512; i = i + 1){
        D[i] = i;
}

// CLEAN DCACHE

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

for (i = 0; i < 16; i = i + 1)
        R[i] = C[31-i]-C[i];

for (i = 0; i < 512; i = i + 1)
        result = result + D[i];
```

Perform the following steps:

1. As explained in Lab 14, make a copy of the soft-core and create a new Vivado project, or reuse an existing one. Don´t forget to expand MIPSfpga as explained in Lab 5 for supporting the 7-segment displays. For this exercise, you must use a **direct-mapped 2KB** D$ configuration (one of the new D$ configurations that you created in Lab 21).
2. Folder *Lab23_CacheController_CacheContentManagementPolicies\Simulations\SimulationSources_WriteAllocationPolicies_BlueFragment* provides the source files that you must use in this first test. It contains the C program shown above translated into MIPS assembly language. Examine the source files (note that the *makefile* uses the –O1 optimization option in the compilation) and then execute the program on MIPSfpga for the following policies:
    a. A *write-back*, *write allocate* policy
    b. A *write-through*, *no write allocate* policy

    For each of these policies, measure the amount of *D$ accesses* and *D$ misses* of the blue fragment and explain the results.
3. Folder *Lab23_CacheController_CacheContentManagementPolicies\Simulations\SimulationSources_WriteAllocationPolicies_GreenFragment* provides the source files that you must use in this second test. Examine the source files and then execute the program on MIPSfpga for the following policies:
    a. Try first a *write-back*, *write allocate* policy
    b. Try then a *write-through*, *write allocate* policy

For each of these policies, measure the amount of *D$ accesses* and number of *writebacks* (*event* 10 of *counter* 1) of the green fragment and explain the results.

Exercise 2: Implement a FIFO replacement policy and test it

In this exercise, you will implement a new D$ replacement policy and test it executing a simple benchmark and measuring the results using performance counters. Specifically, you must implement a first-in first-out (FIFO) policy. In this policy, when a new block is to be inserted in a full set, the block to replace is the one that was inserted in that set the furthest in the past.

1. Analyze how a FIFO replacement policy works.
2. Implement a FIFO replacement policy in MIPSfpga, trying to modify the original LRU policy as little as possible. Observe that a FIFO policy is analogous to an LRU policy where an access that hits just skips the update of the LRU state.
3. Analyze the synthetic benchmark provided in folder *Lab23_CacheController_CacheContentManagementPolicies\Simulations\SimulationSources_ReplacementPolicy*. This benchmark has the following access pattern: $(a_1\ b_1\ a_2\ b_2\ a_1\ b_1\ a_3\ b_3\ a_2\ b_2\ a_3\ b_3)^N$, where x_i represents a cache block (i) that maps to set x, and N is the number of times that the sequence repeats.
4. Justify analytically how many accesses and misses would obtain this benchmark in a processor with a 2-Way Set-Associative D$, for each of the two replacement policies studied (LRU and FIFO).
5. Finally, execute the synthetic benchmark in MIPSfpga for a 4KB 2-Way Set-Associative D$ (i.e. 2KB per way), for each of the two replacement policies studied (LRU and FIFO), measuring the number of D$ accesses and misses. Compare the analytical and the experimental results.
6. As an extension of this exercise, you could implement other policies (such as LIFO, DRRIP, etc.) and also look for other access patterns in real algorithms and compare them under the different policies. For example, you can compare the matrix multiplication used in Lab 22-B for the two replacement policy studied in this exercise (LRU and FIFO).

5. References

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