Project 2

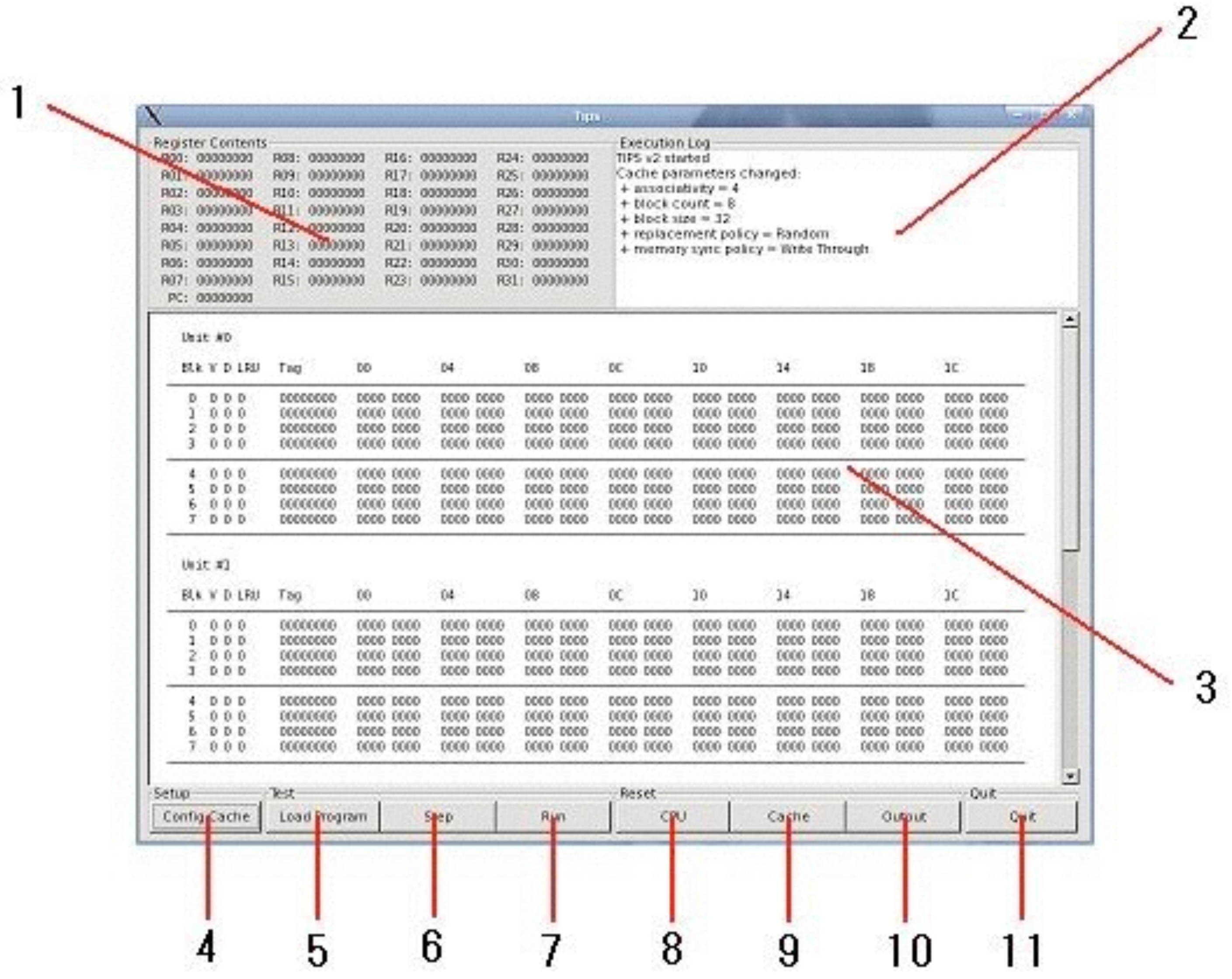
This project will give you intimate knowledge of cache logic through implementation of the actual caching logic. We will be providing you a MIPS simulator called **TIPS** (Thousands of Instructions Per Second), that will be able to run MIPS instructions. However, the cache logic is broken (read: conveniently non-existent)!

Your job is to implement the cache logic behind the cache so that **TIPS** can make use of the many benefits caching entails. You may choose to complete this project either by yourself or with a partner.

The initial **TIPS** code has a default cache size of 0 since there is no cache logic present. You can configure the cache by clicking on the "Config Cache" button at the lower left of the interface.

GUI Walkthrough

The GUI was designed to be straightforward. There are four main components to the GUI interface: register display, execution log, cache display, and control panel.



```
R00: 00000000   R08: 00000000   R16: 00000000   R24: 00000000
R01: 00000000   R09: 00000000   R17: 00000000   R25: 00000000
R02: 00000000   R10: 00000000   R18: 00000000   R26: 00000000
R03: 00000000   R11: 00000000   R19: 00000000   R27: 00000000
R04: 00000000   R12: 00000000   R20: 00000000   R28: 00000000
R05: 00000000   R13: 00000000   R21: 00000000   R29: 00000000
R06: 00000000   R14: 00000000   R22: 00000000   R30: 00000000
R07: 00000000   R15: 00000000   R23: 00000000   R31: 00000000
 PC: 00000000
```

```
TIPS v2 started
Cache parameters changed:
 + associativity = 4
 + block count = 8
 + block size = 32
 + replacement policy = Random
 + memory sync policy = Write Through
```

```
Unit #0

Blk V D LRU  Tag       00         04         08         0C         10         14         18         1C
---------------------------------------------------------------------------------------------------------
 0  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
 1  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
 2  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
 3  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
---------------------------------------------------------------------------------------------------------
 4  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
 5  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
 6  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
 7  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
---------------------------------------------------------------------------------------------------------

Unit #1

Blk V D LRU  Tag       00         04         08         0C         10         14         18         1C
---------------------------------------------------------------------------------------------------------
 0  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
 1  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
 2  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
 3  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
---------------------------------------------------------------------------------------------------------
 4  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
 5  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
 6  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
 7  0 0 0    00000000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000  0000 0000
---------------------------------------------------------------------------------------------------------
```

A description of each of the GUI widgets are described as follows:

1. Register display -- detailed view of the current state of the registers
2. Execution log -- log of actions by **TIPS**. Messages can be displayed in this box using the append_log()function.
3. Cache display -- current snapshot of the state of the cache. The meaning of the column headings on each unit are:
   - Blk - block number
   - V - valid bit
   - D - dirty bit
   - LRU - LRU data
   - Tag - Tag for the block
   - Numbers (00, 04, etc.) - offset in the cache block data
4. Config Cache -- configure the cache parameters
5. Load Program -- loads a dump file for execution
6. Step -- execute one instruction
7. Run -- automate execution
8. CPU -- reset the PC and reinitialize registers
9. Cache -- flush the cache
10. Output -- clear the execution log
11. Quit -- exit **TIPS**

There is also a text-based version of the GUI for those who prefer it. You can run it with the following call:

```
  $ ./tips -nogui
```

Type help at the TIPS prompt to get a list of commands usable in this mode.

Your Task

To complete this project, you must complete the accessMemory() function in cachelogic.c. This function will handle accessing actual memory, using theaccessDRAM() function. Thus, the code in the accessMemory() function will behave as a cache that will call the accessDRAM() function as needed (for cache misses).

To ensure a variety of caches can be simulated, you will be required to dynamically support 5 properties:

- Associativity (ranges from 1 to 5)
- Number of unique indexes (2^n where n ranges from 0 to 4)
- Block size (2^n bytes where n ranges from 2 to 5)
- Replacement Policy (LRU and Random)
- Memory Synchronization Policy (Write Back and Write Through)

More information about the variables you will be working with and the functions at your disposal can be ascertained by looking over tips.h.

You should keep the following things in mind when formulating the code:

- accessDRAM() requires a byte pointer when it is called.
- There are 4 bytes in 1 word.
- The tag information must be right aligned. For example, if the tag is only 25 bits for a given cache configuration, the top 7 bits must always be 0.

- When you are moving things between cache and physical memory, a **BLOCK** is transferred, **NOT** just a word nor a byte. Thus, if the block size is 16 bytes, when you want to move data from cache to memory (or vice versa) you must make sure 16 bytes travel between the cache and physical memory on youraccessDRAM() function call.
- Write Through policy requires the **ENTIRE** block be transferred to physical memory on a write operation.
- To move data to and from a cache block, the memcpy() function should be used. The function prototype of memcpy() is defined as follows:

  void* memcpy(void* dest, void* src, size_t amount);
  where dest is the destination pointer, src is the memory to be copied, and amount is the number of bytes to copy. A more detailed description of this function can be found in K&R.

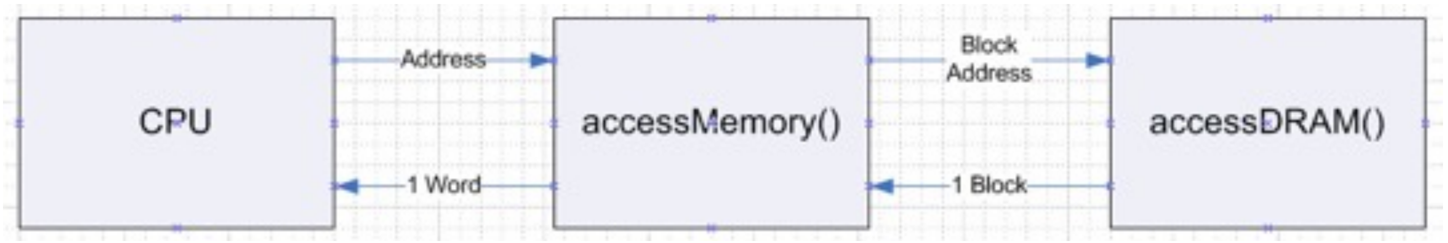
Getting Started

Look over tips.h and cachelogic.c. tips.h gives you an overview of how the cache simulator is put together. A section of that file has been marked as important, so read it to get an idea of what functions and variables you will be using. cachelogic.c contains a slightly more detailed explanation of what you will be writing in the accessMemory() function.

The cache data structure is divided into three levels:

- The first level of entry is selecting which set you want. For example, cache[2] states that you are going to be accessing the 3rd set of the cache. This level is regulated by set_count.
- The next level is selecting the block you want in the set. That is specified by the block field. For example, cache[2].block[4] accesses the 5th block of the 3rd set. In the block is the tag, valid bit, dirty bit, and lru information. This level is regulated by the associativity of the cache.
- The final level of entry is selecting which bytes of the block do you want retrieve or modify. The data contained in a block is represented by the data field of that block. Using the offset, a particular byte can be referenced.

There are two methods to access the LRU information of a block. The first method is via lru.value, a field that will hold LRU information in integer format. The other method is via lru.data, a field that will hold LRU information represented in another format (its type is void*, which means it can be a pointer to anything). You are free to use either method to represent the LRU information, so long as the LRU behaves in a deterministic fashion (i.e. no two valid blocks will ever be candidates for replacement at the same time).



Organization of the memory functions in TIPS

In a nutshell, the accessMemory() function acts as a communication layer between the CPU and DRAM. The code within accessMemory() manipulates the cache data structure defined in tips.h.

All your code <u>MUST</u> be contained in cachelogic.c - specifically in accessMemory() and possibly the LRU functions (depending on how you implement the LRU algorithm). You may add helper functions as long as you do not modify any code outside of cachelogic.c. Do not change any file other than cachelogic.c. Do not change the prototypes of existing functions. To summarize, you can modify only the *body* of the given functions, adding helper functions if needed.

Creating Dump Files for Testing

If you prefer to use the Mars GUI, go ahead and start Mars and open your assembly file. Ensure that the "Delayed Branching" setting is **enabled** in the Settings menu. Now, assemble the source file. Then, select the "Dump Memory" option from the File menu. Ensure that the ".text" memory segment is selected and that the dump format is Binary, and you should be good to go. If you prefer to use the command line, you can do something like this:

```
java -jar Mars.jar a db dump .text Binary output.dump input.s
```

Questions to Answer

Answer the following questions in the Project2.txt/doc file. You can safely assume that all associative caches in these questions use an LRU replacement policy.

Question 1:

For this question, assume 1KB of addressable memory, 64B of L1 cache with 4B blocks. You will probably find it useful to draw the cache as you work through this problem.

The code in question:

```
     #define ARRAY_SIZE 64
     int total = 0;
     char * array = malloc(ARRAY_SIZE * sizeof(char));      /* Line 1 */
// Note: chars are 1-byte wide
     for(int x = 0; x < ARRAY_SIZE; x++) array[x] = x;      /* Line 2 */
     for(int x = 0; x < ARRAY_SIZE; x++) total += array[x]; /* Line 3 */
     for(int x = 0; x < ARRAY_SIZE; x++) total += array[x]; /* Line 4 */
```

a. Assume the cache is direct-mapped and array is a pointer to the memory at offset 0b1001111000, what is the ratio of cache hits to cache misses while Line 4 is being executed?

b. Assume the cache is fully associative and array is a pointer to the memory at offset 0b1001111000, what is the ratio of cache hits to cache misses while Line 4 is being executed?

c. Assume the cache is direct-mapped and array is a pointer to the memory at offset 0b1001111101, what is the ratio of cache hits to cache misses while Line 4 is being executed?

d. Assume the cache is fully associative and array is a pointer to the memory at offset 0b1001111101, what is the ratio of cache hits to cache misses while Line 4 is being executed?

e. Do these results surprise you? Why or why not? Explain what's going on. What would happen if the cache were 2-way set associative? 4-way?

Question 2:

Explain the differences between write-through and write-back caches, and when/why one might be preferred over the other.

Question 3:

Describe a situation in which a 2-way set associative cache would out-perform a fully associative cache. Vice-versa? You should describe these situations in relation to the attributes of the cache. (i.e. accesses are made to fill every block of the cache, hitting memory block-size apart each time...) Otherwise, the examples can be as contrived as you wish.

Question 4:

Briefly describe the changes you would have to make to your implementations in this project to implement an L2 cache. Assume you're given a second cache struct. How would your L1 cache implementation change? How, if at all, would your L2 cache implementation need to be different from your original cache implementation?**YOU ARE NOT EXPECTED TO IMPLEMENT THIS.**

Question 5:

Suppose for a moment that virtual memory was included in this project. How would you expect the input to accessMemory() to change? (in other words, what type of information should the cache now receive in order to do its thing?) **YOU ARE NOT EXPECTED TO IMPLEMENT THIS.**

Submission

* Only submit cachelogic.c (where all your code will go)

* Project2.txt/doc with answers to Questions 1-5