# Lab Assignment 1: Cache Memories

# Objective

The purpose of this assignment is to understand the functionality of cache memories, and to get an insight into various trade-offs related to the design of systems with cache memories.

# Preparation

- Lecture notes about cache memories
- Make sure you went through the Lab0 tutorial and the provided document for git basics

# Part 1: Cache Basics

Solve the following problems and include the solution in the submitted report.

## Problem 1

Consider a machine with a byte addressable main memory of 2^8 bytes and block size of 4 bytes. Assume that a direct mapped cache consisting of 8 lines is used with this machine.

1) How is an 8-bit memory address divided into tag, line number, and byte number?

2) Into what line would bytes with each of the following addresses be stored?

0001 1011
0011 0100
1101 0000
1010 1010

3) Suppose the byte with address 1010 0001 is stored in the cache. What are the addresses of the other bytes stored along with it?

4) How many total bytes of memory can be stored in the cache?

5) Why is the tag also stored in the cache?

## Problem 2

Consider the following code:

```
cout << ”Hello World”;
 cin >> a;
 for(i = 0; i < 50; i++)
cout<<i;
```

1) Give one example of the spatial locality in the code.
2) Give one example of the temporal locality in the code.

# Part 2: Writing the simulator

In this assignment, you will write a one-level cache simulator in c++.

- You need to develop your code using the cachesim.cpp empty file provided in the starter code. Feel free to add any auxiliary source/header files you need. However, your main function must be in this cachesim.c file
- You will take in the blocksize (in bytes), the total cache size (in bytes), the associativity, and write policy using the following flags (consult the appendix at the end of this document if you are not familiar with input arguments in c/c++):

  **-b [blocksize] -c [cachesize]  -w [writePolicy]-a [associativity]**

- Block size, cache size, and associativity are integers and writePolicy is either ‘t’ or ‘b’ for write-through and write-back, respectively.
- You will implement LRU replacement policy.
- To efficiently accommodate for different cache sizes, you have to use dynamically allocated memory.

## Reading Traces:

The simulator will read in a memory trace, which has the following form:

**`R/W:address`**

- Where R indicates a read access, and W indicates a write, and address indicates the memory address being referenced expressed as a hex number.
- Look into `ex_trace.trc` as an example of such trace file.

## Output:

You need to format your output as follows, for an example, looin into the provided `ex_output.out` file.

### Per Access Debugging:

For each access, you will print the address (in hex), and access type, the address as a binary number, the tag, the set number, the block offset, the way of the associativity that it was found in (Way), and the way that was updated (UWay). If the block was not found in the cache (cache miss), then Way will be -1, while UWay will indicate what way the memory location was stored in. If the block is found in the cache, then Way indicates what way it was found in, and a UWay of -1 indicates that we did not update the cache at all, a UWay of of -2 indicates that we only updated our LRU bits for this access, and any positive UWay indicates we actually had to store a value to that way location. You can format your output lines as:

**"%6x%c  %26s %8d %5d %3d %4d %4d %4d %4d\n"**

Notice that there are 26 bits of space for for the binary address: this is because you need to insert spaces between the block offset and index bits (see printouts later for details).

### Summary Statistics:

The summary statistics are printed after the entire trace has been processed. They will indicate the number of hits and misses in the cache, the hit and miss rate, the number of main memory reads, and the number of main memory writes (including dirty bit evictions).

### An example output file:

You are provided with an example output file called ex_output.out. You have to format your output to match the provided format.

# Part 3: Spatial Locality

For the following logic, where A and B are arrays of integer elements:

```
sum=0 // sum is int
for(i = 0; i < num_elements; i++)
    sum=sum+A[i]+B[i]
```

1. Assume that A[0] is placed in address 0x0 and B[0] is in address 0x100

2. (**Paper and Pen Analysis**): Experiment with num_elements=8, a four-way set associative cache of size 256-B and three different cache line sizes (16, 8, and 4B). For each configuration, obtain the hit rate and classify the misses, record your observations.

3. (**Simulation**): hand craft the traces for each of the configurations in Step 2 and run this through the simulator your created in Part 2.

4. Compare the results from step 3 and 4 and document all results.

# Part 4: Both Temporal and Spatial Locality

Let's modify the logic from part :

```
sum=0
for (j=0; j < num_iterations; j++)
    for(i = 0; i < num_elements; i++)
            sum=sum+A[i]+B[i]
```

1. Assume that A[0] is placed in address 0x0 and B[0] is in address 0x100

2. (**Paper and Pen Analysis**): Experiment with num_elements=8, a four-way set associative cache of size 256-B and line size of 16-B. Vary number of iterations as 1, 5, 10, and 100. For each configuration, obtain the hit rate and classify the misses, record your observations.

3. (**Simulation**): hand craft the traces for each of the configurations in Step 2 and run this through the simulator your created in Part 2.

4. Compare the results from step 3 and 4 and document all results.

5. What you expect to change if sum and both arrays are `uint_8` instead of `int`?

# Part 5: Both Temporal and Spatial Locality with Strides

For the following logic:

```
sum=0 // int
for (j=0; J < num_iterations; j++)
    for(i = 0; i < num_elements; i=i+stride)
            sum=sum+A[i]
```

1. Assume that A[0] is placed in address 0x0

2. Experiment (**both paper and pen and in simulation**) with num_elements=128, a four-way set associative cache of size 256-B and line size of 16-B and number of iterations of 100. Vary the stride size as 1,2,4,8,16,21,64.
    a. Obtain the hit rate and classify the misses

3. Repeat Step 2 for number of iterations=10 and 50

4. How does varying number of iterations and stride value affect the hit rate? How does this relate to spatial and temporal locality?

# Part 6: Cache Architecture and Replacement Policies

1. For the same logic in Part 5, experiment (**both paper and pen and in simulation**) with num_elements=128, a cache size of size 256-B and line size of 16-B, a stride of 1 and number of iterations of 100. Obtain the hit rate and identify the final cache content for the following cache architectures
    a. Direct Mapped
    b. 2-way Set Associative
    c. 4-way Set Associative
    d. Fully-Associative Cache

# Appendix: Helping Hints

## Command-Line Arguments

For those of you who are unfamiliar with getting command-line arguments in C/C++, here is a code fragment that would do so if our only flag was b. Please note that PrintUsage is a function that you need to implement if you want to print out the correct usage of this argument to the user.

```
int main(int nargs, char **args)
....

   for (i=1; i < nargs; i++)
   {
      if (args[i][0] != '-')
         PrintUsage(args[0], i);
      switch(args[i][1])
      {
      case 'b':
         if (++i == nargs)
            PrintUsage(args[0], i);
         *blksz = atoi(args[i]);
         break;
      default:
         PrintUsage(args[0], i);
      }
   }
```

# Bit-Level Operations

Students who are unfamiliar with bit-level operations will need to work on them. Here are two useful routines to get you started:

```
unsigned int GetMask(int nset)
/*
* RETURNS: int with nset least significant bits set to 1, others to 0
*/
{
        unsigned int mask, i;
        assert(nset <= 32); /* otherwise default int won't work */
        /*
        * Following two implementations are equivalent, but loop-based one may
        * be easier to understand at first
        */
        #ifdef FAST
                mask = (((long long)1))<<nset) - 1;
        #else
                for (mask=i=0; i < nset; i++)
                        mask |= (1<<i);
        #endif
        return(mask);
}

char *Int2Bits(unsigned int bits)
/*
* RETURNS: string indicating binary bit pattern in integer bits
*/
{
        static char cbits[NBITS+1]; /* NBITS macro defined elsewhere */
        int i;
        cbits[NBITS] = '\0';
        for (i=0; i < NBITS; i++)
                cbits[NBITS-1-i] = (bits == ((1<<i) | bits)) ? '1' : '0';
        return(cbits);
}
```