PART I

Objective: Build a data cache performance simulator and run a set of experiments using your simulator.

Authorship: You may discuss ideas with one another, but you may not share any code with other people. Code that is clearly copied from another team will be considered academic misconduct.

Specifications:
1) You are to write your program in C++, and run it on the dept Linux machines.

2) Your Level 1 Data Cache Simulator should accept the following command-line options:
* capacity <kbytes> - 4, 8, 16, 32, 64
-cap <capacity>
* block size <in bytes>: 4, 8, 16, 32, 64, 128, 256, 512
-block <blocksize>
* set associativity <integer size of set>: 1, 2, 4, 8, 16
-assoc <associativity>
The BASE testcase (i.e. the first one that you should test) is:
capacity = 8K, blocksize = 16 bytes (4words), set associativity = 4

3) Additionally, your cache should have the following functionality:
* write back (for write hits),
* write allocate (for write misses),
* and LRU replacement

4) The input to the cache simulator will be a sequence of memory access traces, one per line (with 2-3 integers per line), terminated by end of file.
For loads the format will be:
0 <address>
For stores the format will be:
1 <address> <data value>
* each address will be the location of a 32bit data word, whose address is word aligned (i.e. divisible by The addresses and data are expressed in hexadecimal format.

5) Other details:
* a word is 4 bytes
* the configuration parameters will be specified in the command line,
as 3 integers (capacity, block size, associativity)

6) To support your testing of the data cache simulator, you should also
implement a simple model of main memory. The capacity of the memory
should be 4 Megabytes (1 Megawords). Initialize each word to its address.

7) The Output of your program will consist of both statistics and contents of the cache and memory. See below for the suggested format.
To print the formatted values in hex, you will need to include <iostream.h> and <iomanip.h> and use setfill, setw, and hex as follows:
```
cout << setfill('0') << setw(8) << hex << (int)_byteArray[j];
```

given setfill prepends '0's to pad the result to be the specified width, setw [i.e. set width] stream manipulator specifies the # of digits to print, hex is a stream manipulator that will cause the following integer to be written in hex.

8) Your program will be graded for correctness and for design.

For Design: I expect to see well chosen classes that reflect the application. Be sure that your method names and variable names reflect memory hierarchy terms. When done, anyone reading your code should have learned how a cache works.

So, towards that end, it may help to define constants such as “dirty”. And, you should definitely make sure that you define (e.g. using typedef) new types for common concepts such as “word” (rather than sprinkling “int” all around) --- use terms from the application-domain.

You should create your own trace files to help you test and debug your cache configurations.

9) The next part will specify the experiment that you are to perform once your simulator is complete, will provide more info on the test cases that will be provided, and will provide the turn in instructions.

OUTPUT FORMATTING:

Key to the following description:
base of output: (10 = decimal and 16 refers to hex)
rate should be a real number, with 4 digits of accuracy
T/F: print either T for True or F for False

STATISTICS:

Misses: <Total (10)> <DataReads (10)> <DataWrites (10) >
Miss Rate: <for total> <for dataread> <for datawrites>
Number of Dirty Blocks Evicted From the Cache: <Total (10)>
CACHE CONTENTS:
Set Number Valid Tag Dirty Word0 ....
<base 10> T/F <16> T/F <16> ....
.....

MAIN MEMORY: {we will ask you to print out some # of words of memory starting at address X}

Print 8 words of memory per line, all values should be base 16 --- here's an example, assuming X is 0}
<address of word0> <word0> <word1> <word2>.... <word7>
<address of word8> <word8> <word9> <word10>.... <word15> ...
<address of word1016> <word1016> <word1017> <word1018>....
<word1023>
<No need to print out the rest of memory.>

PART II

1. Objective

Incorporate the data cache simulator from the project part I, to the SimpleScalar processor simulator. Run a set of experiments and analyze the performance.

Specification:

- Configurable cache parameters: capacity, block size, and associativity.
- Cache hit time is determined by the three cache parameters. See below for details.
- Cache should support write-back, write-allocate and two replacement policies: LRU and random.
- Incorporate the cache model to the SimpleScalar simulator.
- Find CPIs for a set of benchmarks.
- At the end of execution, the simulator should show CPI, the number of hits and misses, and the cache contents without data values.

2. Installing SimpleScalar and running benchmarks.

To install SimpleScalar in your linux account, unzip the file and do make. The executable binary is **sim-fast.**

```
tar xvfz sim-fast.tar.gz
cd simplesim-3.0
make
```

To run a program binary with the compiled SimpleScalar simulator (sim-fast), just add the name of the binary after sim-fast. Suppose test-printf is a target program you are simulating on sim-fast. The following command will start the simulation.

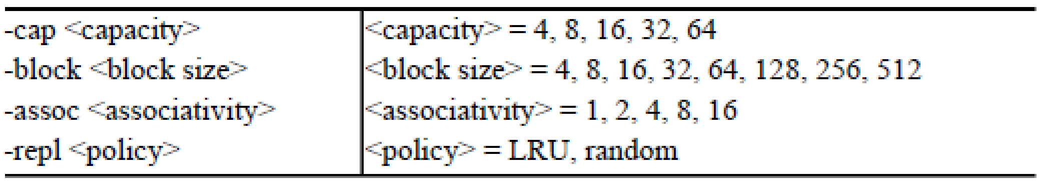
```
sim-fast test-printf
```

3. Getting cache parameters

You should get cache parameters from commandline options. sim-fast has a set of utility functions at options.c for commandline options. In the provided sim-fast.c, there is a sample usage of the utility (line # 135, sim-fast.c). The following command should start sim-fast with three cache parameters.

```
sim-fast –cap 32 –block 16 –assoc 4 –repl LRU test-printf
```

The following four cache parameters must be accepted by your simulator:
<capacity>, <block size>, and <associativity> are integer numbers, and <policy> is a character string.



-cap <capacity>	<capacity> = 4, 8, 16, 32, 64
-block <block size>	<block size> = 4, 8, 16, 32, 64, 128, 256, 512
-assoc <associativity>	<associativity> = 1, 2, 4, 8, 16
-repl <policy>	<policy> = LRU, random

One more option should be supported to control the output. If –verbose is set to 1, your cache simulator should show the final contents of the cache without data portion, as specified later in the output specification.

4. Cache organization and interfaces

For this project, you should simulate only a data cache, assuming you have a perfect instruction cache. Instruction fetching will not increase CPI. Only load and store instructions will access the data cache. sim-fast.c is the main loop executing one instruction at each iteration. For this project, you need to implement two interfaces:

```
void cache_init(int cap, int block, int assoc);
```

cache_init is called to create a cache. You need to implement this function to create your cache and initialize the content of the new cache. Cache hit time should be calculated from the three parameters. You should use the following expression:

Cache hit time = (0.5*capacity (in Kbytes) / 4 + 0.1*block_size/16) *(1+0.01*assoc*assoc)

The actual cache hit time must be the smallest integer number equal to or larger than the above expression. You should set external memory access time for cache misses to **60** cycles. Your cache should support **write-back, write-allocate,** and **LRU replacement**.

```
int cache_access(md_addr_t addr,
enum mem_cmd cmd,
int size);
```

cache_access function should be called for every load and store. The inputs are address, command (Read or Write) and size (1, 2, 4, 8 bytes). md_addr_t type is the data type for address in SimpleScalar, and enum mem_cmd is either "Read" or "Write". This function should return access latency. Load and store instructions use READ_* and STORE_* macros (READ_BYTE, READ_HALF,...) defined in sim-fast.c. You should call **cache_access** from the macros and adjust the current cycle from the returned access latencies.

For this project, you do not need to keep the actual data in the cache. Your cache simulator just needs to return correct access times by tracking hits and misses of cache accesses. MEM_READ_* and MEM_WRITE_* in READ/ WRITE macros will actually copy the data to registers. In the source codes, cache.c and cache.h currently have dummy functions to implement the interfaces, which always return 5 cycles as access latency. You can replace the dummy functions with a real cache implementation. Sim-fast is written in C. You can use the cache.c and cache.h as a C interface for your C++ cache model.

5. Calculating CPI

The unmodified sim-fast simulator models a processor with a perfect memory which has one CPI. (i.e. it always execute one instruction per cycle.) You should incorporate your cache model into the sim-fast, and find the new CPI with a realistic memory hierarchy. For this project, you need to add only a data cache. There are two global variables necessary to calculate the CPI.

```
counter_t sim_num_insn; /* the number of instructions executed. */
counter_t sim_cycle; /* the number of total cycles */
```

sim_cycle should be increased by one for non-memory instructions and by access latency for loads and stores.

You are responsible for adjusting sim_cycle appropriately from returned cache latencies. Your simulator should show the final CPI after finishing the execution.

6. Output and performance analysis

By default, your simulator must show the following statistics: CPI :

# of Misses:	**<total>**	**<reads>**	**<writes>**
# of Hits :	**<total>**	**<reads>**	**<writes>**
Miss rates:	**<total>**	**<reads>**	**<writes>**

If **–verbose** options is set to 1, the simulator should show the cache contents without data portion too.

Cache Contents:
Set Number Valid_bit Dirty_bit Tag_bits

The SPEC Benchmarks have been provided and you should find the best cache configuration by varying three parameters: capacity, block size, and associativity. Which parameters have the best CPI for each of the three benchmarks? Which is the best on average across the three benchmarks? Upload your projects to moddle before **November 17th 2013 (Midnight).** A demo will be conducted after the deadline. Each group have to give detailed presentation of the code including the role of every group member. The marking may vary among the group members.