Problem 1: Cache simulation and associativity.

For Problems 1 through 8, You can learn about the "go" SPEC95 benchmark by looking at the web page http://www.spec.org/osg/cpu95/news/099go.html.

This problem introduces the sim-cheetah cache simulator. In order to simulate several cache configurations at one time, the simulator utilizes an algorithm devised by Rabin Sugumar and Santosh Abraham while they were at the University of Michigan. You can look at the sim-cheetah summary by entering the command sim-cheetah by itself.

Use a single run of sim-cheetah to simulate the performance of the following cache configurations for the SPEC95 benchmark "go".

Configurations:

* least-recently-used (LRU) replacement policy
* 128 to 2048 sets
* 1-way to 4-way associativity
* 16-byte cache lines

The command line will be:

```
sim-cheetah <arguments> go.ss 2 8 go.in
```

where

* <arguments> is the list of sim-cheetah parameters needed to produce results for the specified cache configurations.
* go.ss is the SimpleScalar binary for the "go" benchmark.
* "2 8" are the play quality and the board size for the go simulation.
* "go.in" is a file that specifies the starting board position (in this case an empty file.)

You can watch the two computer players as they present their moves. The game (and therefore the simulation) will end at move number 40.

Using the output from sim-cheetah, for caches of equivalent size, verify using simple calculations whether increasing associativity or the number of sets in the cache gives the most benefit. Explain how you did your computations, give your results, and discuss the relative benefits of cache associativity verses the number of sets in the cache. You should include at least four comparisons. Is the trend always the same? Suggest reasons for the trends or the of lack of trends.

Note: An example comparison would be going from 128/1 (sets/associativity) to 128/4 (an increase in cache size of 4) and from 128/1 to 512/1 (also an increase in cache size of 4). Another example would be 124/4 to 512/4 and 512/1 to 2048/1.

Problem 2: Cache simulation and unified/separate instruction and data caches.

This problem introduces the sim-cache simulator. It allows more flexibility than sim-cheetah (see Problem 1), but it may require more simulation time. You can look at the sim-cache summary by entering the command sim-cache by itself.

(a) Use sim-cache to verify the results for the following sets/associativity combinations from the sim-cheetah simulation in problem 1:

* 128/1
* 128/2
* 128/4
* 2048/1
* 2048/2
* 2048/4

Note: You will have to set the cache parameters in sim-cache to make them the same as those used by the sim-cheetah simulation. If you are not getting the correct results, be sure that you are setting all of the parameters correctly (i.e., line size, replacement policy).

Note: You can make the simulation run faster by including parameters to tell sim-cache not simulate parts of the memory hierarchy that it normally would. These parameters are:

```
-cache:il2 none -cache:dl2 none -tlb:dtlb none -tlb:itlb none
```

Discuss your results, characterizing any differences. Using the simulation time numbers output by each of the simulators, discuss the relative speed of the simulators for this particular case.

(b) Using the additional flexibility of sim-cache, compare the performance of separate instruction and data caches to the unified cache that is simulated by sim-cheetah. (Note: You can simulate data- or instruction-only caches with sim-cheetah, but sim-cache is much more flexible.) The total cache size should be the same as the size of the unified cache. The simulator will give you the miss rates for the instruction and data caches. Use these to compute an overall miss
rate for the two combined. Do this for each of the configurations in part (a).

Discuss the relationship between the miss rates from part (a) and the separate and combined miss rates. What do your results suggest about using combined/separate instruction and data caches? What other factors might affect this choice?

Note: Each simulation will now contain a single L1 instruction cache and a single L1 data cache. The size of each of these will be half that of the unified cache to which you are comparing results. The sizes should be adjusted by halving the number of sets in each cache.

Problem 3: Cache replacement policies.

The sim-cheetah simulator implements the MIN set-associative cache replacement policy that was first suggested by Laszlo Belady. This unimplementable--but simulatable--policy uses oracle information to determine the block in a set that will be used the farthest in the future. This is different than the LRU replacement policy which "guesses" the block in a set that will be used farthest in the future by using information about when the blocks were used in the past. The MIN algorithm knows which block will be used furthest in the future. Therefore, using the MIN replacement policy should result in better performance than using the LRU replacement policy.

For a direct-mapped cache (set size 1), would you expect the results for LRU and MIN replacement policies to be different? Why or why not?

Redo the simulation in Problem 1 using the MIN replacement policy (sim-cheetah calls the policy opt). If you have not done the simulation in Problem 1, do it first. Discuss the results for associativity 1.

Define the reduction in miss-rate as

(original miss-rate - new miss-rate) / original miss-rate.

Compute the reduction in miss-rate when changing from the LRU to the MIN replacement policy for all cache sizes and associativities 2 and 4.

What do your results suggest about cache replacement policies? If MIN works better than LRU and MIN cannot be implemented (it uses information about the future), then what can be done to improve replacement policies?

Problem 4: Cache efficiency.

In his Ph.D. thesis written at the University of Wisconsin, Doug Burger defined cache efficiency to be the portion of time that data in the cache is considered to be live, divided by the total time that data could be live.

A cache line is live if it contains valid data that will be read before the program halts, the data is overwritten, or it is evicted from the cache. The total time that a cache line could be live is the entire execution time of the program. Thus, the efficiency of a cache is computed as the sum of the live times of all of the lines in the cache divided by the execution time multiplied by the number of lines in the cache.

Burger found that cache efficiencies tend to be lower than we might expect. He suggested that because much of the data in the cache will never be used again (this is what efficiency measures), the miss-rate of the cache might be tied to this measurement. If this is the case, then finding a better way to decide what will be kept in the cache will raise the efficiency and also lower the miss-rate of the cache.

Modify the source code of the sim-cache simulator to include a calculation of cache efficiency. The way to do it is to keep track of the total time (define the time of event X as the number of cache accesses that have occurred before X in the simulation) that data is live in the cache. When a cache line becomes valid, mark it with a time-stamp. If it is used later, compute the difference between the current time and the saved time-stamp, then update the time-stamp so that a future reference to the line will add to the live time of this line from this point. Keep a sum of all of the live times for all cache lines. When the simulation ends, use this sum to calculate the cache efficiency as defined above.

Note: This modification will include adding a place to keep the sum and adding timestamps to each cache line.

The sum will be a variable of type SS_COUNTER_TYPE. Like the cache-hits counter, it will be defined in cache.h. It will also be initialized, updated, and output in the same places in cache.c that the cache-hits counter is.

The timestamps will be of type SS_TIME_TYPE. They will be defined in the same place in cache.h as the cache-line status variable. They will also be initialized, updated, and used to update the sum in the same places in cache.c that the cache-line status variable is modified.

Run the experiments in Problem 2(a) using your modified version of the sim-cache simulator. For each cache size, plot the hit rate (defined as 1 - miss rate) and the efficiency of the cache on the same plot. Does the efficiency of the cache predict the miss rate?

For each of the cache sizes 128 cache lines and 2048 cache lines, run the same experiment as in Problem 2(a), but make the cache fully associative (a single set with 128 and 2048 lines, respectively). For each cache size, do two runs: one using the LRU replacement policy and one using the "random" replacement policy. Discuss the following:

* Compare the results for each of the two fully-associative cache sizes using the LRU replacement policy with each of the results for a cache with the same total capacity from Problem 1. Do the results make sense?

* Compare the results for each fully-associative cache using the LRU replacement policy with the results for the same cache using the random replacement policy. What could have caused the results?

Problem 5: Generating profile information.

The sim-profile simulator allows you to find out how often certain events happen during the execution of a program. Enter sim-profile by itself to see a list of all of the parameters that allow you to see profile information.

Run each of the following (either one at a time or all at once using the -all parameter) for the SPEC95 benchmark "go". In each case, describe the information that is given and what you learned about the go program from it (e.g., it does no floating-point computations.) Be creative and be specific. The command line you will use is:

```
sim-profile <arguments> go.ss 2 8 go.in
```

where

* <arguments> is the list of sim-profile parameters.
* go.ss is the SimpleScalar binary for the "go" benchmark.
* "2 8" are the play quality and the board size for the go simulation.
* "go.in" is a file that specifies the starting board position (in this case an empty file.)

You should generate and discuss the output for the following profile parameters:

* -iclass   Counts of each instruction type.
* -iprof   Counts of each instruction.

* -brprof   Counts of each branch type.
* -amprof   Counts of each addressing mode.
* -segprof Counts of accesses to each data segment.

Generate the output for these profile parameters and describe the information that they provide.

* -tsymprof   Count of accesses to text segment symbols.
* -tsymprof -internal Same as above, but include compiler symbols.
* -taddrprof   Count of accesses to address in the text segment.
* -dsymprof   Count of accesses to data segment symbols.
* -dsymprof -internal Same as above, but include compiler symbols.

Problem 6: Introduction to timing (e.g., execution-driven) simulation.

The sim-cache and sim-cheetah simulators simulate only the state of the memory system--they do not keep track of the timings of events. The sim-outorder simulator does. In fact, it simulates everything that happens in a superscalar processor pipeline, including out-of-order instruction issue, the latency of the different execution units, the effects of using a branch predictor, etc. Because of this, sim-outorder runs more slowly, but it also generates much more information about what happens in a processor. Enter sim-outorder by itself to see a list of all of the parameters that you can set. Look through this list carefully, and try to identify what each one is for.

Because sim-outorder keeps track of timing, it can report the number of clock cycles that are needed to execute the given program for the simulated processor with the given configuration. Run the SPEC95 benchmark "go" with the default parameters for sim-outorder using the command line:

```
sim-outorder <arguments> go.ss 2 8 go.in
```

where

* <arguments> is the list of sim-outorder parameters.
* go.ss is the SimpleScalar binary for the "go" benchmark.
* "2 8" are the play quality and the board size for the go simulation.
* "go.in" is a file that specifies the starting board position (in this case an empty file.)

with <arguments> empty, so that the defaults are used. Record the instructions simulated per second, instructions-per-cycle (IPC), and the cache miss rates for the L1 and L2 caches.

Now execute the same simulation with a "perfect" memory system. (Note: This is not a perfect memory system, in that the memory access delay will be from 1 to 4 cycles, but it is as close to perfect as you can get without modifying the simulator source code.) By setting the delay for every memory unit (L1 cache, L2 cache, memory, and TLB) to 1 cycle, you can simulate what would happen if the processor did not have to wait for the memory system to deliver or accept data. Record the instructions simulated per second, the IPC, and the cache miss rates for the L1 and L2 caches.

Verify that your second simulation set the latency for all memory devices to 1 cycle by looking at the output generated before the simulation started. Also verify that the same memory activity took place in both simulations by comparing the cache miss rates.

Compute the change in IPC from the first run to the second. Define the percent change in IPC as

% change = 100 x new IPC / old IPC.

What does this tell you about the importance of the memory system in creating fast processors (at least for the go benchmark)? Could you have come to this conclusion using the non-timed simulator, sim-cache? Explain you answer.

Speed is almost always a concern when building a simulator. Your go simulations simulated about 3 million instructions. Researchers commonly simulate billions of instructions using the same simulators. Using the instructions simulated per cycle, calculate the percent change between sim-outorder and sim-cache. Your answer should be larger than 1. What is your opinion as to the difference in simulation speed? How many times slower would sim-outorder have to be before it would not be easy to do this problem using it?

Problem 7: The need for branch prediction.

Run sim-profile to find out the distribution of instruction types for the SPEC95 benchmark "go". You will use the command line:

```
sim-profile -iclass go.ss 2 8 go.in
```

where

* go.ss is the SimpleScalar binary for the "go" benchmark.
* "2 8" are the play quality and the board size for the go simulation.
* "go.in" is a file that specifies the starting board position (in this case an empty file.)

What percentage of the instructions executed are conditional branches? Given this percentage, how many instructions on average does the processor execute between each pair of conditional branch instructions (do not include the conditional branch instructions).

Using your textbook, class notes, other references, and your own opinion, list and explain several reasons why the performance of a processor like the one simulated by sim-outorder (e.g., out-of-order-issue superscalar) will suffer because of conditional branches. For each reason also explain how--if at all--a branch predictor could help the situaltion.

Problem 8 asks you to use the sim-outorder simulator to investigate the effects of branch predictors on the execution of the go benchmark.

Problem 8: Branch predictors.

If you do not have the results from Problem 7, run sim-profile to find out the distribution of instruction types for the SPEC95 benchmark "go". You will use the command line:

```
sim-profile -iclass go.ss 2 8 go.in
```

where

* go.ss is the SimpleScalar binary for the "go" benchmark.
* "2 8" are the play quality and the board size for the go simulation.
* "go.in" is a file that specifies the starting board position (in this case an empty file.)

What percentage of the instructions executed are conditional branches? Given this percentage, how many instructions on average does the processor execute between each pair of conditional branch instructions (do not include the conditional branch instructions).

The sim-outorder simulator allows you to simulate 6 different types of branch predictors. You can see the list of them by looking at the -bpred parameter listed in the output generated when you enter sim-outorder by itself. For each of the 6 default predictors, run the same go simulation as you did above and note the branch-direction prediction rate and the IPC for each. Each command line will be similar to:

```
sim-outorder -bpred nottaken go.ss 2 8 go.in
```

For each of the 6 branch predictors, describe the predictor. Your description should include what information the predictor stores (if any), the amount of storage (in bits) that is required, how the prediction is made, and what the relative accuracy of the predictor is compared to the others (use the branch-direction measurements for this). Finally, describe how the prediction rate effects the processor IPC for the go benchmark. Do you believe that the results for go will generalize to all programs? If not, describe a program for which the prediction rates or the IPCs would be different.

Extra Credit:

The sim-outorder simulator does not include a way to disable branch prediction. This is because out-of-order processors always benefit from predicting branches as taken or not taken, even if a more resource intensive predictor is not implemented.

Modify the sim-outorder simulator to include the parameter -bpred none. This parameter will cause the processor to simulate a processor with no branch prediction be treating every conditional branch as a mis-predicted branch. To do this, modify the file sim-outorder.c. Look at the code that handles the perfect branch prediction command-line parameter, and create new code for the parameter none.

Run the simulator with no branch prediction. How do the IPC results compare to those for the other 6 branch prediction schemes?

Problem 9: Simple C Loop

Using sim-safe, write and test a program in assembly language that implements the piece of C code in Figure 1.

```
for(i=0; i ≤ 100; i = i + 1) {
      A[i] = B[i] + C;
}
```

Figure 1

Assume A and B are arrays of 32-bit integers, and C and i are 32-bit integers. Assume that all data is stored in data memory and A, B, and i are stored at addresses 0x40000000, 0x10000000, and 0x20000000, respectively. In addition, assume that the values in the registers are lost after every iteration. Calculate how many cycles are executed?

Once the program is tested, use sim-profile to assess the relative frequency of operations and sim-outorder to determine how many instructions are executed, the total simulation time in cycles, and the CPI or cycles per instruction.

Problem 10: 32-bit Division Algorithm

Using sim-safe, write and test a program in assembly language that implements the division algorithm in Figure 8-16 for 32-bit integers. This algorithm typically uses one 64-bit register to hold the remainder and the quotient. However, because the register file for the simplescalar architecture only has 32-bit registers, the divide algorithm for the simplescalar architecture makes use of an internal 64-bit register. At the end of n iterations, the remainder is stored in the most significant 32-bits. On the other hand, the quotient is stored in the least significant 32-bits. To allow the user to easily access this 64-bit register and not waste valuable registers, two registers are created called HI and LO which can be accessed to obtain the remainder and quotient, respectively. However, since the built-in opcode for divide is not being used in this problem, use two registers and make certain a shift transfers the appropriate bits between the two 32-bit registers. Check your results against the built-in divide opcode div.

Problem 11: Euclid's GCD Algorithm

One of the most-often-studied algorithms in computing sciences is called Euclid's Algorithms for finding the greatest common divisor of two numbers. The greatest common divisor or GCD is also called the highest common factor and is very important in fields such as cryptography and error-correcting codes. Test and implement Euclid's GCD algorithm for two 32-bit integers using assembly language. The pseudo-code for performing the GCD is listed in Figure 2.

```
int GCD(int num, int den) {
      int remainder;
      while(remainder != 0) {
            num = den;
            den = remainder;
            remainder = mod(num, den);
}
return den;
}
```

Figure 2

Problem 12: Unrolling Loops

This exercise considers the benefits of performing loop unrolling on pipelined machines such as the pipelined RISC architectures found in Chapter 12. Loop unrolling is used to exploit more parallelism inside a computer program. Most modern-day compilers come equipped with options to allow loop unrolling. This technique allows a typical loop, such as in Figure 3, to be more efficient.

```
for(i=0; i ≤ 1000; i = i = i + 1) {
      A[i] = A[i] + C;
}
```

Figure 3

In loop unrolling, multiple copies of the loop body are made and instructions from different iterations are linked together. For example, the code fragment in Figure 4 can suffer from data hazards because of the dependency between the load and the addition.

```
Loop: l.d   f0, 0($1)
      Add.d f4, f0, f2
      Subi  $1, $1, 8
      s.d   f4, 0($1)
      bne   $1, Loop
```

Figure 4

In Figure 5, the stall lengthens each iteration to 6 cycles/iteration. However, if the loop is replicated once and each iteration performs two computations of the loop in Figure 3. Then, the number of cycle/iteration is decreased to 8 / 2 or 4 cycles/iteration. Figure 6, shows the optimized scheduled code which is unrolled twice. Note the counter is decremented by 16 instead of 8, because each iteration performs two copies of the loop body in Figure 3.

```
Loop: l.d   f0, 0($1)
      stall
      add.d f4, f0, f2
      subi  $1, $1, 8
      s.d   f4, 0($1)
      bne   $1, Loop
```

Figure 5

```
Loop: l.d   f0, 0($1)
      l.d   f6, 8($1)
      add.d f4, f0, f2
      add.d f8, f6, f2
s.d   f4, 0($1)
s.d   f8, 8($1)
      subi  $1, $1, 16
      bne   $1, Loop
```

Figure 6

This optimization by a compiler does have the disadvantage of wasting registers as highlighted in Figure 6. However, the careful use of such a technique can have a big effect on a computer without having to change the hardware.

In this problem, write and test code assembly code for Figure 3 and compare it to a version that has its loops unrolled four times. Determine speedup of the scheduled code compared to the unscheduled code using sim-outorder. Assume the clock speed is 100 MHz and that all variables are 32-bit integers or arrays.