CS3410 Spring 2015
Problem Set 2 (version 3)
Due Saturday, April 25, 11:59 PM
(Due date for Problem-5 is April 20, 11:59 PM)

NetID:
Name:

200 points total. Start early! This is a big problem set.

Problem 1: Data Hazards (25 points)

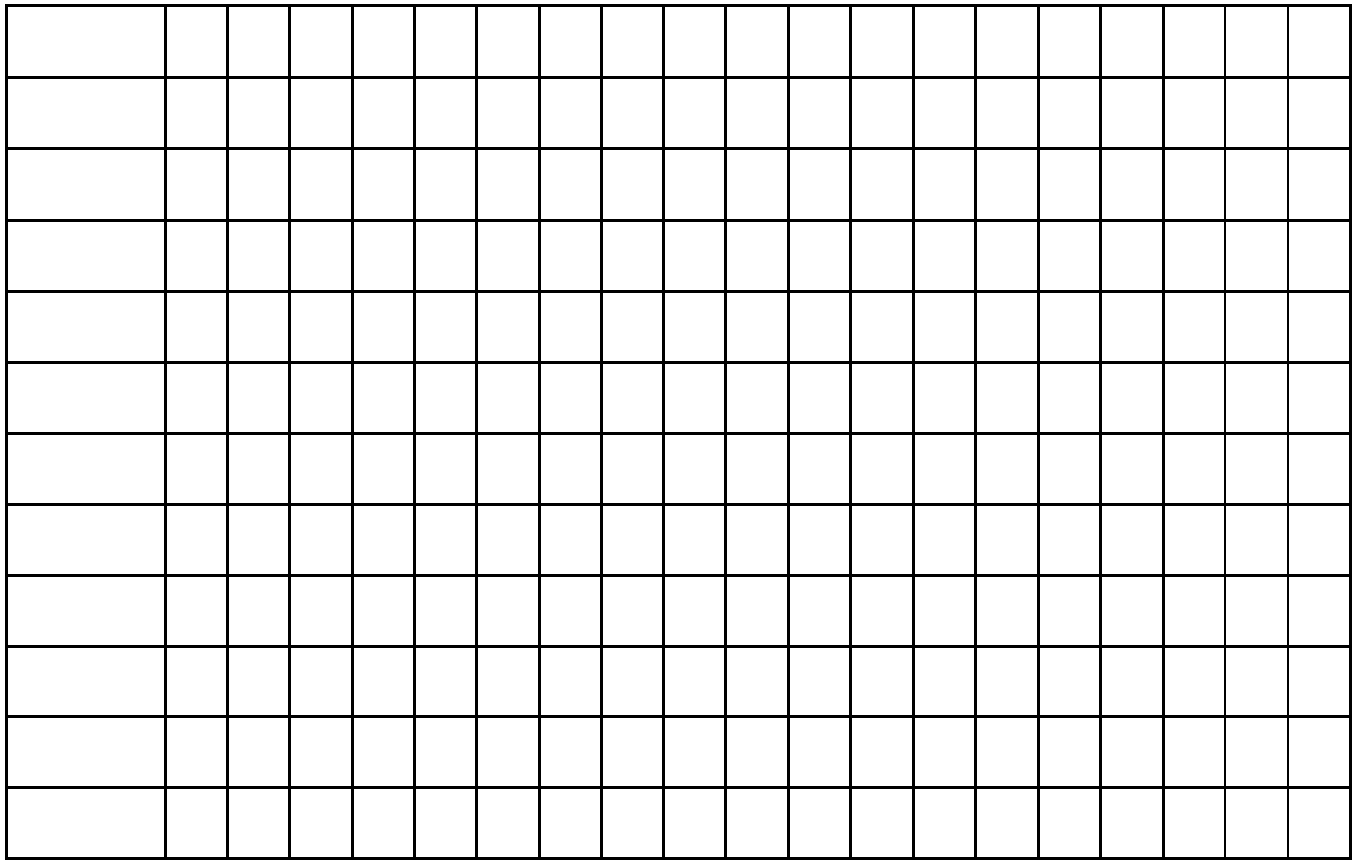
a) Do exercise 4.11.1 (5th Edition Patterson & Hennessy). Note: Branches are calculated in the execution stage.

b) Do exercise 4.11.2 (5th Edition Patterson & Hennessy). Additionally, on your cycle chart above, for each cycle mark any pipeline stage that isn't doing useful work in a different color.

Problem 2: Control Hazards (25 points)

Assume a fully bypassed 5 stage pipeline. It uses 1 branch delay slot, always predicts that branches are taken, and calculates branches in the execution stage. Instruction that follows the branch is the branch delay slot instruction for that branch. Draw a cycle chart of the instruction sequence given below:

```
        lw r2, 0(r4) (after doing this instruction the value in r2 is not equal to value in r1)
label 2: bne r1, r2, label 1
        lw r5, 0(r4)
        beq r5, r1, label 3
        lw r6, 0(r7)
label 1: beq r5, r2, label 2
        add r1, r2, r0
label 3: lw r7, 0(r8)
```



Problem 3: Calling Conventions (25 points)

a) Consider the following recursive C code to calculate the Fibonacci numbers –

```
int fib (int n) {
    if (n==0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return (fib(n-1) + fib(n-2));
}
```

Below is a partially filled MIPS code for the above C code. Fill in the blanks.
(1 Point for correctly filling each partially-filled instruction)

```
fib:  addiu                 # make room on stack
      sw                    # push $ra
      sw                    # push $fp
      sw                    # push $s0
      addiu                 # set new frame pointer
      sw                    # save $a0 on parents stack frame
      bgt                   # if n>0, jump to test to check if n>1
      nop                   # NOP-1
      add                   # else fib(0) = 0
      j rtn
      nop                   # NOP-2

test: addi $t0, $0, 1
      bne                   # if n>1, jump to gen
      nop                   # NOP-3
      add                   # else fib(1) = 1
      j rtn
      nop                   # NOP-4

gen:  addi                  # n-1
      jal fib               # call fib(n-1)
      nop                   # NOP-5
      add                   # copy fib(n-1)
      lw                    # need to get $a0 from the stack since it may have been clobbered
                            # by the JAL
      addi                  # n-2
```

```
        jal fib             # call fib(n-2)
        nop                 # NOP-6
        add                 # fib(n-1) + fib(n-2)

rtn:    lw                  # pop $s0
        lw                  # pop $fp
        lw                  # pop $ra
        addiu               # pop stack frame
        jr $ra
        nop                 # NOP-7
```

b) Assuming 5-stage pipelined architecture, with 1-delay slot and branch calculation done in Decode stage, what is the total number of MIPS instructions needed to execute the function when

1) n=0 (0.5 Point)
2) n=1 (0.5 Point)
3) n>=2 (write your answer as a function of n) (1 Point)

c) For the NOPs numbered 1-7 in the above code, state what instruction can replace it, if any. If no such instruction exists for a particular NOP, write NONE against it. (4 Points)

NOP-1 -
NOP-2 -
NOP-3 -
NOP-4 -
NOP-5 -
NOP-6 -
NOP-7 -

Problem 4: Caches (25 points)

Consider a two-way set associative cache with 2 sets and 8-byte lines. Virtual and physical addresses are 32 bits, and the cache is write-back and write allocate, with a Least Recently Used cache eviction policy.

Is each memory access in the program below a hit or a miss? If it is a miss, is it a cold miss, conflict miss, or capacity miss? Show the final state of the cache after the execution of the program. Assume the cache is initially empty. How many writes to main memory are actually made?

```
LW $t0 <- M[4]
LW $t1 <- M[16]
SW $t1 -> M[8]
LW $t2 <- M[4]
SW $t0 -> M[12]
SW $t1 -> M[20]
LW $t0 <- M[24]
LW $t2 <- M[36]
SW $t0 -> M[4]
SW $t0 -> M[8]
```

Main Memory	
Address	Data
0	0x00001234
4	0x0000DEAD
8	0x0000BEEF
12	0x0000CAFE
16	0x00000000
20	0x00008844
24	0x00003410
28	0x00005678
32	0x0000ABCD
36	0x0000FEED

Problem 5: Virtual Memory (15 points)

For Lab-4, you will be implementing the necessary components to support virtual memory in a real computer system. Virtual memory is an artificial mapping between the address space of an individual process to physical memory available on a system. This is necessary to give processes the illusion of exclusive access to memory for purposes of software compatibility as well as resource sharing and management.

Please complete the following problems **before** coming to the lab (though you should still submit it as part of this homework). This will help you understand the concept of virtual memory before implementing it in lab. The points will be applied to your Lab grade.

a) C Programming

Familiarize yourself with the following operators in C before coming to the lab:

- Left shift
- Right shift
- Logical operators: AND, OR, XOR
- Negation

Write C code that does the following:

1. Left shift variable x by 5 bits and store it in x.
2. Right shift variable x by 5 bits and store it in x.
3. AND two variables and store them into a third variable.
4. OR two variables and store them into a third variable.
5. XOR two variables and store them into a third variable.
6. Change a variable into its negation.

b) Structure

In this lab, we will be dealing with a two level page table. In this particular scheme, a CPU register contains the location of the top-level page directory. This is a special page of memory that contains listing of page tables. Page tables contain actual physical pages allocated to the current running process.

Draw out what the virtual memory mapping may possibly look like given a system running 3 processes.

c) Procedure

What happens exactly when the CPU requests memory access?

- Is it in the TLB? If so, fetch that value.
- Otherwise, we have to perform a page walk. Starting at the page directory, we find the corresponding page table, and then find the right page, followed by the data we actually want with the offset.
- But what if the address we want isn't to be found? We hit a page fault. This can be the result of either an invalid address (segfault) or a valid address that is not in memory. For the latter case, we either allocate a new page of memory if it has never been allocated, or we bring the page back to RAM if it was paged to disk.
- There is a third case: copy-on-write. Sometimes, processes may want to copy segments of memory around or share them. In this case, we want to delay actually copying the data as long as possible (why?). In such cases, multiple sets of virtual addresses point to the same physical page in memory. However, their permission bits in the page table are different (which ones and why?). When a change is made to a copy of the original data, we are forced to physically create a new copy and update the changes on the new copy as well as updating the page table to reflect this change. This is known as copy-on-write.

Draw out a flowchart for the above procedure.

d) Implementation

In Lab-4 we will give you the source code to the MIPS simulator that we used in Lab-3. You will then modify the simulator to simulate virtual memory, including a shared memory system call with copy-on-write semantics.

The original simulator code does not implement virtual memory: for each load and store, the simulator simply uses the address to index into a large array (allocated using `malloc`). We modified the simulator in a few simple ways to implement virtual memory. In particular, we implemented the simulation for the hardware part of the virtual memory system, but not the software part:

- The simulator already supports physical memory using the existing code.
- The simulator now calls `create_address_space()` before loading a program to create a new set of page tables for the program.
- The simulator stores a (physical) pointer to these page tables in Register 4 on Coprocessor 0, i.e. c0r4, which is the standard MIPS *Context* register, also known as the *Page Table Base Register (PTBR)* on x86.
- The simulator now calls `map_page()` to create virtual mappings for the (virtual) memory segments of the program being loaded.
- For loads and stores, the simulator now traverses the page tables stored at c0r4 to translate virtual addresses to physical addresses before it accesses the existing physical memory.
- Simple stub code for the shared memory system call has been added so that the simulator calls `map_shared_page()` as needed to create copy-on-write virtual mappings.

Before coming to Lab-4, we want you to familiarize yourselves with the source files for the simulator and implement a part of the virtual memory. All of the source files for the simulator are available in the course directory in the CSUG Lab. The top level directory contains the simulator source code and a Makefile for compiling it. You should copy the files to your own directory to work on them:

```
$ cp -r /courses/cs3410/lab4 ~/lab4
$ cd ~/lab4
```

The `mips` subdirectory contains an example MIPS program that uses shared memory (you can also write your own MIPS test programs if needed), and a Makefile for compiling it.

The most relevant files for you are:

- **mem.h** Contains detailed descriptions and prototypes for the functions you must implement, along with some prototypes for functions that you will likely need to call (e.g. to allocate physical pages).
- **vmem.c** You write this file. It should contain all of your implementation. The simulator will not compile without it.
- **mips/hello.c** This compiles into a MIPS executable. If your virtual memory code is working correctly, you should at least be able to run this program with no errors.

The `_mem_ref()` function is used by the simulator to traverse the page tables on each memory access to a virtual address; it is implemented in **mem.c**. You may find it helpful to at least look at this function to be sure you understand what it is doing. Otherwise, most of the simulator code is not terribly interesting. In rough order of interest, are:

- **pmem.c** Simulates physical memory as a big array of bytes.
- **syscalls.c** Simulates system calls.
- **mem.c** Helper routines for managing memory.
- **readelf.c** Parses and loads MIPS executable binaries.
- **run.c** Simulates the the CPU datapath.
- **main.c** Main entry point and command line processing.
- **sim.h** Prototypes and definitions.
- **debug.c** Interactive debugger.
- **disasm.c** Disassembler (used when debugging).
- **readline.c** Keyboard input (used when debugging).
- **readline.h** Keyboard input (used when debugging).

To compile the simulator (do this each time you edit vmem.c or anything in the lab4/ dir) from the lab4/ directory:

```
$ make
```

To compile the test program (only do this once) from the lab4/ directory:

```
$ cd mips/ $ make $ cd ..
```

To run the test program:

```
./vsimulate mips/hello
```

Before coming to the Lab-4, implement the `create_address_space()` function in vmem.c to create a virtual address space.

Problem 6: Traps (15 points)

a) What is the use of the privilege bit? (**1 point**)

b) State the difference between syscalls, exceptions and interrupts. (**1 point**)

c) Which traps will be caused by the following actions? (if no trap occurs, write N.A.) (**6 points**)

i	Call to malloc() leading to increase in heap size	
ii	Receiving a character from keyboard	
iii	Printing result to screen using printf()	
iv	addi instruction in MIPS resulting in an overflow	
v	System clock tick	
vi	Dereferencing a pointer which is NULL	

d) Why does the syscall save much less state than the exception handler? (**2 points**)

e) Write whether the following actions occur in syscalls, exceptions, both or neither: (**5 points**)

i	Saves old PC value	
ii	Saves caller-save registers	
iii	Saves callee-save registers	
iv	Allocate new registers	
v	Store result in $v0	

Problem 7: Multicore Performance (10 points)

a) Describe a possible scenario where multiple cores working on the same task might be slower than a single core of the same architecture and clock speed.

b) Suppose we have a program of which 30% can be parallelized upto any extent (call this section A), 30% can be parallelized up to 10 processors (call this section B), and 20% can be parallelized up to 2 processors (call this section C), and 20% cannot be parallelized at all (call this section D). Assume that each section must be executed sequentially, ie. A -> B -> C -> D. If the program takes 100 seconds to run on one processor, how long will it take to run on two processors working in parallel? On 10? On 20? What is the limit as the number of processors tends to infinity? What is the speedup when the number of processors tends to infinity?

Problem 8: Multicore Synchronization (15 points)

Suppose we have two threads with the following instructions in C. The threads operate in parallel and use a shared memory space. Array c is set such that c[0] = 4, c[1] = 2, and c[2] = 1.

Thread A	Thread B

c[0] = c[0] + 2;
c[1] = c[1] + 1;
	c[1] = c[1] – 2;
	c[2] = 3;
	c[0] = 0;

(a) Assuming the instructions can be executed in any order, what are all the possible values of c[0], c[1], and c[2] after both threads finish?

c[0] may be c[1] may be c[2] may be

(b) Below is the corresponding MIPS code. Rewrite it using the load linked and store conditional instructions. Assume the base address for c is in $s0. Also, be sure to include a delay slot for any branch instructions.

Thread A	Thread B

LW $t0, 0($s0)
ADDIU $t0, $t0, 2
SW $t0, 0($s0)
LW $t0, 4($s0)
ADDIU $t0, $t0, 1
SW $t0, 4($s0)
	LW $t0, 4($s0)
	ADDIU $t0, $t0, -2
	SW $t0, 4($s0)
	ADDIU $t1, $zero, 3
	SW $t1, 8($s0)
	SW $zero, 0($s0)

Problem 9: C Hashtable (45 points)

Note there are three parts to this problem, a, b, and c.

From *Wikipedia*:
"In computing, a **hash table** (**hash map**) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an *index* into an array of *buckets* or *slots*, from which the correct value can be found." - http://en.wikipedia.org/wiki/Hash_table

Let us define a hashtable with chaining and a load factor of 0.75.

Chaining: Let h = hash(key). The key/value pair is stored in a list stored at bucket (h mod N) where N is the table size. Hash collisions (when two different keys hash to the same bucket) are handled by adding the key/value pair to the list stored at the bucket.

Load factor 0.75: load factor is the number of elements in the hashtable divided by the total number of buckets. The performance of chaining drops significantly when the load factor exceeds some threshold. To avoid this, the table is doubled in size when the load factor > 0.75.

For this problem we will consider a hash table with integer keys and integer values.

a) After N insertions how big can the table be (worst case space)? How many insert operations are needed (worst case time)? Give your answer as an estimate of the constant factors (e.g., 2N, 3N log N, etc) for both questions. Remember to show your work and give an explanation for your answer.

Space:

Time:

b) Implement the hash table in C and submit the code and binary to CMS. The following functions are required:

```
void hashtable_create(struct hashtable *self);
void hashtable_put(struct hashtable *self, int key, int value);
int hashtable_get(struct hashtable *self, int key);
void hashtable_remove(struct hashtable *self, int key);
void hashtable_stats(struct hashtable *self);
```

The hashtable_stats function should print the number of elements in the hashtable, the table size (N), and the total number of insert operations performed so far in the format shown in the Example Usage section.

Example Usage:

```
struct hashtable a;
hashtable_create(&a);
hashtable_put(&a,0,99);
hashtable_stats(&a); /* prints: "length = 1, N = 2, puts = 1\n" */
hashtable_put(&a,1,42);
hashtable_stats(&a);
assert(hashtable_get(&a,0)==99);
hashtable_remove(&a,0);
hashtable_get(&a,0); /* causes program to exit with exit code 1 */
hashtable_remove(&a,0); /* would also cause program to exit with exit code 1 */
```

Output:

```
length = 1, N = 2, puts = 1
length = 2, N = 4, puts = 3
ERROR: key 0 not found.
```

Output explained: On the first insertion, the table is initialized with a size of 2. The second insertion will cause the load factor to exceed 0.75. So, prior to insertion the table is resized to N=4. Changing the table size requires that we re-insert all existing key/value pairs. (Why?) Therefore the total number of insertions is 3.

Error handling policy: your program must return 0 from main if there are no errors. Your program may choose to exit with exit code 1 if it detects a runtime error. Aborting the program with an assert is also acceptable.

Hash function: you may use any hash function or this provided hash function (from http://burtleburtle.net/bob/hash/integer.html):

```
unsigned int hash( unsigned int a)
    a = (a ^ 61) ^ (a >> 16);
    a = a + (a << 3);
    a = a ^ (a >> 4);
    a = a * 0x27d4eb2d;
    a = a ^ (a >> 15);
    return a;
}
```

c) Suppose we would like to handle complex keys and values (e.g. string keys and/or struct values). One idea is to use the same code but pass pointers as integers. For example,

```
char *key="hello";
struct mystuff value;
... /* initialize value */
hashtable_put(&a, (int)key,(int)(&value));
struct mystuff *p=(struct mystuff *)hashtable_get((int)"hello");
assert((*p)==value); /* is this true? */
```

Why might this not work? Explain in 1-2 sentences.