CS 2200 - Introduction to Systems
Fall 2016

Homework 1

Rules:

- Please print a copy of the assignment and handwrite your answers. No electronic submissions are allowed. Please staple the pages together.
- This is an **individual assignment**; No collaboration is permitted.
- **Due Date: 7th September 2016 – 6:05 PM (At the start of Recitation).** Bring your **Buzzcards**.

Name: Answer Key **GTLogin:** akey1

1. Unaligned memory accesses are faster than aligned memory accesses. Answer **True / False with justification**. **[10 Pts]**

False. Unaligned memory accesses usually take multiple instructions in order to get all parts of the value since these instructions can only load along the alignments.

ALL OR NOTHING. "False" with missing/incorrect justification receives no credit.

2. For the struct defined below, show how a smart compiler might pack the data to minimize wasted space and follow alignment restrictions. That is pack in such a way that you can **guarantee aligned** accesses to all the elements of the struct. Assume a char is 1 byte, int is 4 bytes, and a short is 2 bytes. Moreover, assume the architecture is **little endian** and supports load word, load byte, and load half word instructions. **[35 Pts]**

```
struct x {
      char c;  /* value 0x92*/
      int a;  /*value 0xBAADF00D */
      short s;  /* value 0xABCD */
      char d;  /* value 0x93*/
}
```

+3	+2	+1	+0	
			x92	0x1000
xBA	xAD	xF0	x0D	0x1004
	x93	xAB	xCD	0x1008
				0x100C

In the following memory picture each row represents a memory word comprising of 4 bytes, and each cell represents a byte. You do not necessarily need to use all rows. Write each byte in with the hexadecimal values.

+3	+2	+1	+0	
xBA	xAD	xF0	x0D	0x1000
x92	x93	xAB	xCD	0x1004
				0x1008
				0x100C

+10 correct endianness
+15 int and short are aligned properly (+7.5 each)
+10 space is minimized (via packing or by minimizing as much as possible while maintaining struct declaration order)

3. One common specification found in an ISA is the *addressing modes* of particular instructions. The addressing mode describes where a particular instruction gets its operands (the data the instruction operates on). For example, the operands of an *add* instruction would be the two Registers that are added together. **(Hint: Read the book and review class notes before you start)** **[30 Pts]**

Some common addressing modes are:

Register	The operand is located in one of the processor's general-purpose registers
Immediate	The operand is encoded in the instruction itself
Base + Offset	The operand is located at a memory address, computed as follows: Address = Base (stored in a register) + Offset (encoded in the instruction)
PC-relative	The operand is located at a memory address, at a specified offset (encoded in the instruction) from the current program counter. Note that at the time the instruction is executed, the PC has already been incremented to the address of the *next* instruction, thus: Address = PC (of instruction) + 1 + Offset.

Note, some instructions may use multiple addressing modes, for example one register-retrieved operand as well as an immediate value.

For each of the following **32-bit** LC-2200 instructions, provide the addressing mode used. If the addressing mode computes a memory address, write the address where the operand would be located:

```
! first instruction starts at memory address 0x0
! assume $sp = 0x1000

loop:
add $s0, $t0, $t1          ! answer: register
lw $s1,-4($sp)             ! answer: base+offset(0x0ffc)
addi $s1, $s1, 10          ! answer: immediate
beq $s0, $s1, done         ! answer: pc+offset (0x05)
beq $zero, $zero, loop     ! answer: pc+offset (0x0)

done:
halt                       ! answer: none
```

+6 per answer (for parts asking for memory address, +3 for mode and +3 for correct address)

Halt was not graded.

Reasonable variants such as "register, immediate" for addi were also accepted.

4. The following is a function call in assembly using the LC-2200 instructions but with some unnamed registers ($r1 - $r5, $sp, $at, $zero) and its own calling convention. Label each $rx register as caller-saved or callee-saved based on the code. Assume $sp is the stack pointer register and $at holds the starting address of the function (labeled func). **[25 pts]**

```
      ...
      addi $sp, $sp, -1     !start here
      sw   $r1, 0($sp)
      addi $sp, $sp, -1
      sw   $r3, 0($sp)
      addi $sp, $sp, -1
      sw   $r4, 0($sp)
      jalr $at, $zero
      …

func: addi $sp, $sp, -1
      sw   $r2, 0($sp)
      addi $sp, $sp, -1
      sw   $r5, 0($sp)
      …
```

$r1 caller

$r2 callee

$r3 caller

$r4 caller

$r5 callee

ALL OR NOTHING.