15-122 : Principles of Imperative Computation, Fall 2015

Written Homework 12

Due: Monday November 30, 2015 by 6:00 PM

Name: _____

Andrew ID: _____

Section: _____

This written homework will deal with integers in C and with the C0VM.

Print out this PDF double-sided, **staple** pages in order, and write your answers on these pages *neatly in ink or dark pencil*. You can hand in the assignment to your TA during lab or in the box outside of GHC 4117 (in the CS Undergraduate Program suite). **Warning: The box is removed promptly at 6PM.**

You must hand in your homework yourself;
do not give it to someone else to hand in.

Question	Points	Score
1	8	
2	7	
Total:	15	

1. **Integer Types**

5pts (a) Suppose that we are working with the usual 2's complement implementation of unsigned and signed `char` (8 bits, one byte), `short` (16 bits, two bytes) and `int` (32 bits, four bytes).

We begin with the following declarations:

```
signed char the_char = -7;
unsigned char un_char_1 = 248;
unsigned char un_char_2 = 5;
int the_int = -247;
```

Fill in the table below. In the third column, always use two hex digits to represent a `char`, four hex digits to represent a `short`, and eight hex digits to represent an `int`. You might find these numbers useful: $2^8 = 256$, $2^{16} = 65536$ and $2^{32} = 4294967296$.

Most, but not all, of these answers can be derived from the lecture notes. If you can't find an answer from the lecture notes, you can look at online C references or just compile some code.

C expression	Decimal value	Hexadecimal
`the_char`	-7	`0xF9`
`(unsigned char) the_char`	249	`0xF9`
`(int) the_char`	-7	`0xFFFFFFF9`
`un_char_1`	248	
`(int)(signed char)un_char_1`		
`(int)(unsigned int)un_char_1`		
`un_char_2`	5	`0x05`
`(int)(signed char)un_char_2`		
`(int)(unsigned int)un_char_2`		
`the_int`	-247	
`(unsigned int)the_int`		
`(char)the_int`		
`(short)the_int`		
`(unsigned short)the_int`		

3pts

(b) For this question, assume that `char` is a 1-byte signed integer type and that `unsigned int` is a 4-byte unsigned integer type.

Write the C function `pack_cui` which takes a `char` array of length 4 and packs it into a single `unsigned int`. We want the 0th character aligned at the most significant byte, and the last character aligned at the least significant byte. For example, given an array `C = {1, 2, -1, 4}`, `pack_cui(C)` should return 0x0102FF04.

For full credit,

- Do not cast (or otherwise convert types) directly between signed and unsigned types of different sizes.
- Do not rely on the *endianness*[1] of your machine. For example, the following code is incorrect:

  ```
  unsigned int pack_cui(char* C) { return *((unsigned int*) C); }
  ```
- Make sure your solution works for `char` arrays containing negative values.
- Write code that is clear and straightforward.

```
unsigned int pack_cui(char *C) {









}
```

[1]"Endianness" refers to the natural storage order of bytes for a particular hardware architecture; you can read about it on Wikipedia, and don't forget to read *Gulliver's Travels* in your no doubt copious spare time.

2. **C0VM**

Each of the following bytecode files was generated by the C0 compiler. Some comments may have been edited out, but all instructions are untouched. Write C0 programs that will generate these bytecode files.

2pts (a)

```
C0 C0 FF EE       # magic number
00 0D             # version 6, arch = 1 (64 bits)

00 00             # int pool count
# int pool

00 00             # string pool total size
# string pool

00 01             # function count
# function_pool

#<main>
00 00             # number of arguments = 0
00 02             # number of local variables = 2
00 26             # code length = 38 bytes
10 00    # bipush 0
36 00    # vstore 0
10 00    # bipush 0
36 01    # vstore 1
15 00    # vload 0
10 0A    # bipush 10
A1 00 06 # if_icmplt +6
A7 00 14 # goto +20
15 00    # vload 0
10 01    # bipush 1
60       # iadd
36 00    # vstore 0
15 01    # vload 1
15 00    # vload 0
60       # iadd
36 01    # vstore 1
A7 FF E8 # goto -24
15 01    # vload 1
B0       # return

00 00             # native count
# native pool
```

3pts (b) (Note that the bytecode continues on the following page.)

```
C0 C0 FF EE       # magic number
00 0D             # version 6, arch = 1 (64 bits)

00 00             # int pool count
# int pool

00 15             # string pool total size
# string pool
48 61 70 70 79 20 54 68 61 6E 6B 73 67 69 76 69 6E 67 21 0A 00

00 02             # function count
# function_pool

#<main>
00 00             # number of arguments = 0
00 03             # number of local variables = 3
00 0F             # code length = 15 bytes
14 00 00 # aldc 0
B7 00 00 # invokenative 0
57       # pop             # ignore result
10 00    # bipush 0
10 0A    # bipush 10
B8 00 01 # invokestatic 1
B0       # return

#<f>
00 02             # number of arguments = 2
00 03             # number of local variables = 3
00 23             # code length = 35 bytes
15 01    # vload 1
10 00    # bipush 0
9F 00 06 # if_cmpeq +6
A7 00 0A # goto +10
15 00    # vload 0
36 02    # vstore 2
A7 00 12 # goto +18
15 00    # vload 0
15 01    # vload 1
60       # iadd     # LINE 10
15 01    # vload 1  # LINE 11
10 01    # bipush 1 # LINE 12
64       # isub     # LINE 13
B8 00 01 # invokestatic 1
36 02    # vstore 2
15 02    # vload 2
B0       # return
```

```
00 01             # native count
# native pool
00 01 00 10       # print
```

2pts

(c) This question has to do with the function `f` in the bytecode given in part b above. When execution reaches the instruction marked `LINE 10` there are two values on the operand stack; assume they are `0x0000000A` and `0x00000009`. (It will be helpful to be aware of where these values came from.)

Draw the four operand stack states after each of lines 10–13 is executed. The elements in your stack should be 32-bit hexadecimal numbers. The top of your stack should be on the right-hand side.

Immediately after executing line 10: `iadd`

Immediately after executing line 11: `vload 1`

Immediately after executing line12: `bipush 1`

Immediately after executing line13: `isub`