Midterm Exam

601.229 Computer Systems Fundamentals

Spring 2020 Johns Hopkins University

Instructors: Xin Jin and David Hovemeyer

9 March 2020

Complete all questions.

Use additional paper if needed.

Time: 50 minutes.

Name of student: Solution

Q1. Integer representation

20 points

Powers of 2 ($2^{y} = x$):

y	0	1	2	3	4	5	6	7	8	9	10	11	12
x	1	2	4	8	16	32	64	128	256	512	1,024	2,048	4,096

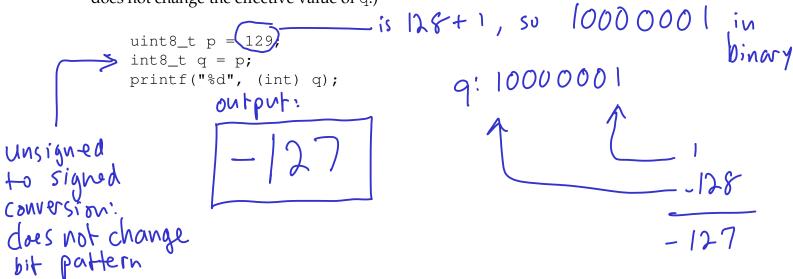
y	13	14	15
\boldsymbol{x}	8,192	16,384	32,768

Assume that int8_t and int16_t are 8 and 16 bit signed integer types represented using two's complement, and uint8_t and uint16_t are 8 and 16 bit unsigned integer types.

(a) [6 points] Write the binary representations of the values a, b, and c.

(b) [6 points] Write the binary representations of the values d, e, and f.

(c) [4 points] What output is printed by the following code? (Note that the cast to int does not change the effective value of q.)

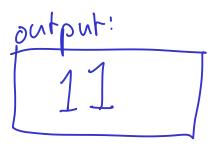


(d) [4 points] What output is printed by the following code? (Note that the cast to unsigned does not change the effective value of s.)

```
uint16_t r = 267;

uint8_t s = r;

printf("%u", (unsigned) s);
```



Q2. Integer arithmetic

Assume that int8_t, int16_t, and int32_t are 8, 16, and 32 bit signed integer types represented using two's complement, and uint8_t, uint16_t, and uint32_t are 8, 16, and 32 bit unsigned integer types. Assume that signed overflow follows two's complement semantics.

(a) [6 points] Given the following incomplete code:

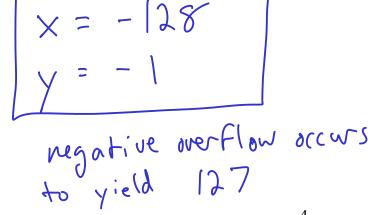
uint8_t
$$x = ___, y = ____$$

assert(x + y < x);

State values for x and y that will make the assertion true.

(b) [6 points] Given the following incomplete code:

State values for x and y that will make the assertions true.



I note: we are assuming that x and y are not implicitly promoted which they would be in an actual C program). Instead we're assuming 8 bit unsigned arithmetic.

Again, assume 10 implicit promotion occurs, and assume 8-Lit signed arithmetic.

(c) [4 points] Consider the following C function:

```
int16_t negate16(int16_t x) {
  return -x;
}
```

Also consider the following incomplete code:

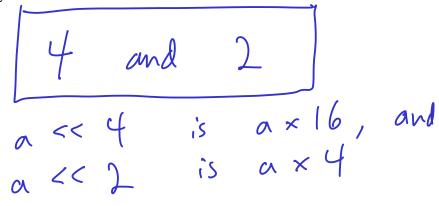
```
int16_t y = \underline{\hspace{1cm}}; assert( (int32_t) negate16(y) != -((int32_t) y) );
```

State a value for y that will make the assertion true. Explain briefly.

(d) [4 points] Consider the following incomplete function:

```
uint32_t times20(uint32_t a) {
  return (a << ____) + (a << ____);
}</pre>
```

State values that can be substituted for the two blanks so that the times20 function returns a value that is 20 times greater than its parameter a (ignoring the possibility of overflow). Explain briefly.



Q3. x86-64 50 points

Things to know about x86-64 code:

- Arguments are passed in %rdi, %rsi, %rdx, %rcx, %r8, %r9
- The return value is returned in %rax
- %r10 and %r11 are caller-saved registers, which may change as a result of a function call (the argument registers are also effectively caller-saved)
- %rbx, %rbp, and %r12-%r15 are callee-saved: functions modifying them must save and restore their values using pushq and popq, and they may be assumed *not* to change as a result of a function call
- Code should go in the .text section
- Global variables should go in the .bss or .data sections (.data allows data to be initialized)
- Read-only data such as string constants should go in the .rodata section
- Operand size suffixes are b (8 bit byte), w (16 bit word), 1 (32 bit long word), and q (64 bit quad word)
- Instructions may have at most one memory operand
- Indexed/scaled addressing is (RegA,RegB,Scale), and accesses address RegA + RegB \times Scale, where Scale is 1, 2, 4, or 8

Selected registers and 8 bit sub-registers: Selected conditional jump instructions:

Register	8 bit sub-register	Instruction	Meaning
%rax	%al (same pattern for	jl	jump if less
	%rbx,%rcx,%rdx)	jg	jump if greater
%rdi	%dil	jle	jump if less than or equal
%rsi	%sil	jge	jump if greater than or equal
%r8	%r8b (same pattern for		
	%r9-%r15)		

Note that assigning to the 8 bit sub-register does *not* clear the other bits of the larger register.

[Actual problem is on the next page.]

Write an x86-64 assembly language function called <code>countLessThan</code> which takes three parameters <code>arr</code>, <code>n</code>, and <code>val</code>. The parameter <code>arr</code> is a pointer to an array of 64 bit signed integers. The parameter <code>n</code> is a single 64 bit integer which indicates the number of elements in the array that <code>arr</code> points to. The parameter <code>val</code> is a single 64 bit integer. The C function prototype for the function is the following:

```
long countLessThan(long *arr, long n, long val);
```

The countLessThan function should return a count of the number of elements in the array which are less than val.

Here are C test code showing the expected behavior of countLessThan:

```
long testArr[] = \{-5, 6, -1, 8, 3, 8, 4, -5\};
ASSERT(\{4L == countLessThan(testArr, 8, 4)\};
```

Requirements and hints:

- Your function must follow correct register-use and calling conventions
- Remember that the first parameter is a pointer, and the array elements are in memory
- Index/scaled addressing may be useful

[Write your code in the next page(s).]

.section .text

.globl countLessThan

countLessThan:

subg \$8, Porsp

move \$0, 90 rax

Ltop: cmpq \$0, Porsi jle Ldone

cmpq %rdx, (%rdi)

jge . Ladvance

incq Porax

.Ladrance:

adda \$8, Tordi dece % rsi jup. Ltop

Laone:

adda \$8, %, 50

RF

/* align stack*/

/* counter */

/* n ≤ 0 ? */

/ if so, done */

1/ is or elt < val 7/k/

/* if not, don't inc. count */

1/ incr. count */

/ advance to next elt. */

/* decr. n */

/k restore stack */ * return result in 2 rax*/ [Continue your answer to Q3 on this page if necessary.]

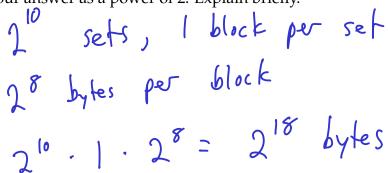
Q4. Performance and caching

10 points

(a) [5 points] Assume that on a system with 32 bit addresses, the addresses have the following format:

14 bits	10 bits	8 bits
tag	index	offset

If the cache is direct-mapped, how many bytes of data can be stored in the cache? You may choose to express your answer as a power of 2. Explain briefly.



(b) [5 points] Consider the following x86-64 instructions:

Is it possible for these instructions to execute in parallel on a CPU with multiple non-pipelined functional units capable of integer multiplication? Explain briefly.



[Use this page for scratch work if necessary.]