SOLUTION

Simon Fraser University
Computing Science 295
Fall 2021
Friday Oct. 15 2021
Midterm Examination 1
Time: 45 minutes

First name (please write as legibly as possible within the boxes)

Last name

Student ID

This examination has 11 pages.

Read each question carefully before answering it.

- No textbooks, cheat sheets, calculators, computers, cell phones or other materials may be used.
- All assembly code must be x86-64 assembly code.
- Hand in your scrap sheet(s) along with your examination paper. The scrap sheet(s) will not be marked.
- The marks for each question are given in [ ]. Use this to manage your time:
  - One (1) mark corresponds to one (1) minute of work.
  - Do not spend more time on a question than the number of marks assigned to it.

Good luck!

Part 1 - Each question is 2 marks – There are no part marks given!

Answer the following multiple choice questions on the **bubble sheet** at the back of this examination paper.

1. Consider the following syntactically correct C code fragment:

```
float aFloat = 3.1415;
int sum = (int) aFloat + 0xFFFFFFFE;
```

Which value does the variable `sum` contain when the above C code fragment has executed on our *target machine*?

a. `1.1415`

b. `-1`

c. `0x00000001`

```
aFloat = 3.1415
0xFFFFFFFE = -2
(int) aFloat = 3
sum = 3 + (-2) = 1 or 0x00000001 (int in hex)
```

d. `5`

e. None of the above

2. Which step in the compilation process transforms our C code into assembly instructions?

a. The step called the preprocessor

b. The step called the compiler – See Lecture 8 Slide 7

c. The step called the assembler

d. The step called the linker

e. None of the above

3. Consider the following syntactically correct C function:

```
char mystery( char someParam ) {
    char result = 0;
    if ( someParam > 0 ) result = someParam;
    else result = -someParam;
    return result;
}
```

What will it return once it has executed on our *target machine* with the parameter `someParam` set to the value `-128`?

a. `127`

b. `128`

c. `-127`

d. `-128`

```
char mystery( -128 ) {
    char result = 0;
    if ( -128 > 0 ) ~~result = someParam;~~
    else result = -(-128);
    So, it seems that result = 128
    What is the bit pattern of 128?
    Interpreting 128 as an unsigned char we get:
    B2U(10000000) -> 2^7 -> 128
    but we cannot interpret 128 as a signed char
    because 128 is outside the range of signed char
    -> [-128 .. 127], so the bit pattern 10000000
    interpreted as a signed char is -128
    Therefore even though it seems that
    result = 128 (10000000)
```

```
It is actually the case that
result = -128
return result i.e., -128
EXTRA:
What is -128 as a bit pattern?
-128 -> T2B(X) -> (~(U2B(|X|)))+1 and X = -128
        See Lecture 3 Slide 11 Method 1
        (~(U2B(|-128|)))+1
        (~(U2B(128)))+1
        (~(10000000))+1
        -> someParam is a char -> w = 8 bits
        (01111111) + 1 = 10000000
        Check: B2T(10000000) -> -2^7 -> -128
-128 -> T2B(X) -> U2B(X + 2^w)
        See Lecture 3 Slide 11 Method 2
        U2B(-128 + 2^w) -> U2B(-128 + 2^8)
                        -> U2B(-128 + 256)
                        -> U2B(128) -> 10000000
```

e. None of the above

4. Consider the following syntactically correct C code fragment:

```
short count = 0xACE;
printf( "count = %hhi\n", (char) count );
```

What is printed on the computer monitor screen when the above C code fragment has executed on our *target machine*?

a. **`count = -50`**

```
count = 0xACE = 0000 1010 1100 1110
(char) count = 0xCE = 1100 1110
```

```
hhi -> signed char numerical output
    -> 1100 1110 = -2^7 + 2^6 + 2^3 + 2^2 + 2^1
                 = -128 + 64 + 8 + 4+ 2 = -50
-> See Lecture_4_Demo.c
```

b. `count = 0xCE`

c. `count = 206`

d. `count = 0xACE`

e. None of the above

5. Consider the following syntactically correct C code fragment:

```
short aShort = -2;
char aChar = aShort;
short sumS = 0xABBB + (short) aChar + 1;
```

Which statement below is true about the above C code fragment once it has executed on our *target machine*, but has not yet exited the scope of the variables `aShort`, `aChar` and `sumS`?

a. `sumS` contains the hex value `0xABBA`

b. `aChar == aShort`

c. Statements a. and b. are true.

```
aShort = -2 = 0xFFFE
aChar = aShort -> aChar = -2 = 0xFE -> 1111 1110
1111 1110 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1
          = -128 + 64 + 32 + 16 + 8 + 4+ 2 = -2
So, Statement a. => aChar (-2) == aShort (-2) is TRUE
(short) aChar = 0xFFFE -> 1111 1111 1111 1110 = -2
(short) aChar + 1 = -2 + 1 = -1 -> 0xFFFF
```

```
0xFFFF = 1111 1111 1111 1111
0xABBB (1010 1011 1011 1011) + 0xFFFF -> 0xABBA
short sumS = 0xABBB + (short) aChar + 1;
```

So, Statement b.

=> `sumS` contains the hex value `0xABBA is TRUE`

So, Statements a. and b. are true.

d. Only the statement b. is true.

e. None of the above

6. Consider the following C code fragment:

```
char char1 = 101;
char char2 = ________ ;
char sumOfChar = char1 + char2;
```

Which value must be assigned to `char2` in order for the sum of `char1` and `char2` to create a positive overflow?

a. No numbers would create a positive overflow when added to `101`.

b. `42`

```
char1 = 101;
range of char -> [-128 .. 127]
For char1 + char2 > 127 i.e., create a positive
overflow, char2 > 127 - char1 (101) -> char2 > 26
So char2 = 42 satisfies the above condition
char1 (101) + char2 (42) = 143 > 127
char1 (101) + char2 (26) = 127 -> still within the
range (on positive side of range)
```

```
char1 (101) + char2 (-203) = -102 -> still within
the range (on negative side of range)
```

c. `26`

d. `-230`

e. None of the above

7. Consider the following syntactically correct C code fragment:

```
unsigned int x = 0xDECAF000;
unsigned short y = 0xCAFE;
if ( x > y ) printf("Caf ");
if ( x < (signed short) y ) printf("Decaf ");
if ( (unsigned char) x > y ) printf("Latte ");
```

What is printed on the computer monitor screen when the above C code fragment has executed on our *target machine*?

a. `Caf Decaf Latte`

b. `Caf Latte`

c. `Caf`

d. `Decaf`

e. None of the above

```
unsigned int x = 0xDECAF000;
unsigned short y = 0xCAFE;
if ( x > y ) printf("Caf ");
   promoting y to 32 bits as an unsigned i.e.
   padding with 0's: 0xDECAF000 > 0x0000CAFE
   without a calculator, we can see that these 32
   bits 0xDECAF000, interpreted as an unsigned
```

value, will be > than 0x0000CAFE, also interpreted as an unsigned value

So, Caf is printed on the computer monitor screen of *target machine*.

```
if ( x < (signed short) y ) printf("Decaf ");
```

casting y to 16 bits as a signed i.e. interpreting 0xCAFE as a signed value and promoting it to 32 bits still as a signed i.e. padding with 1's: 0xFFFFCAFE

0xDECAF000 < 0xFFFFCAFE

without a calculator, we can see that these 32 bits 0xDECAF000, interpreted as a signed value, will represent a larger negative value than 0xFFFFCAFE, also interpreted as a signed value

Remember from previous questions that 0xFE = 0xFFFE = 0xFFFFFFFE = -2

So, Decaf is printed on the computer monitor screen of *target machine*.

```
if ( (unsigned char) x > y ) printf("Latte ");
```

casting x (0xDECAF000) to a char -> 8 bits, we get 0x00 = 0 (unsigned value)

promoting it to 16 bits still gives us 0

0x0000 > 0xCAFE

without a calculator, we can see that this is not the case.

So, Latte is NOT printed on the computer monitor screen of *target machine*.

`The answer (Caf Decaf) is not one of the options.`

8. Which range of values can be stored in the variable `y` declared in the C fragment code of Question 7 above?

   a. $[0 .. 2^{16}]$
   b. $[-128 .. 127]$
   c. $[0 .. 2^{16-1}]$
   d. $[0 .. 2^{15} - 1]$
   e. None of the above

```
unsigned short y = 0xCAFE;
y is declared as an unsigned short -> 16 bits
so range of unsigned short is [0 .. 2^16 -1]
which is not one of the options sbove.
```

Part 2 – The weight of each question is indicated in [ ] – Write your answer below each question unless instructed otherwise.

1. [Total marks: 15] Consider the following function `mystery` written in x86-64 assembly code, where the parameter `x` is in the register `%edi` and the parameter `y` is in the register `%esi`:

```
            .globl   mystery power              Line 1
power mystery: # x -> %edi, y -> %esi           Line 2
            xorl     %eax,    %eax              Line 3
            movl     $1,      %r8d              Line 4
loop .L3:                                       Line 5
            addl     $1,      %eax              Line 6
            imull    %edi,    %r8d              Line 7
            cmpl     %eax,    %esi              Line 8
            jne      .L3 loop                   Line 9
            movl     %r8d, %eax                 Line 10
            ret                                 Line 11
```

a. [3 marks] If we call this function as follows: `mystery( x, y )` where `x = 2` and `y = 3`, what value will it return?

Answer: **8**

b. [2 marks] Replace the label `.L3` with a more descriptive label name. Do this replacement in the above code.

Possible answer: **`loop`**

c. [2 marks] Rename this function with a more descriptive name. Do this renaming in the above code.

Possible answer: **`power`** or **`pow`** `xʸ -> x is raised to the power y`

d. [2 marks] What is the data type of the parameters and the return value of this function? Express your answer using C data types.

Answer: `int or unsigned int`

e. [2 marks] Replace `Line 3` with another equivalent x86-64 instruction, i.e., an x86-64 instruction that will produce the same result, but is not a `xor*` instruction.

Answer: `Line 3: xorl  %eax, %eax`

The purpose of this instruction is to "zero" the register %eax.

Possible replacement: `Line 3: movl  $0,  %eax`

f. [2 marks] Replace `Line 6` with another equivalent x86-64 instruction, i.e., an x86-64 instruction that will produce the same result, but is not an `add*` instruction.

Answer: `Line 6: addl  $1,  %eax`

The purpose of this instruction is to increment the value of the register %eax by 1.

Possible replacements: `Line 6: incl  %eax`

`Line 6: leal  1(%eax), %eax`

g. [2 marks] On `Line 6` and `Line 8`, the register `%eax` is used for a different purpose than holding the return value. For what purpose is the register `%eax` used on those two lines?

Answer: On `Line 6` and `Line 8`, the register `%eax` is used as a loop increment or a loop counter, expressing the number of times the loop executes. At every loop iteration, `%eax` is incremented by 1 (`Line 6`) then compare to `y` (`Line 8`). The loop terminates when `%eax` equals the value of `y` (the power to which we are raising `x`).

2. [Total marks: 14] Consider a floating point number encoding scheme based on the IEEE floating point format and defined as follows:

- It uses 7 bits.
- There is one sign bit `s`.
- The `exp` can be any number in the range [0 .. 31].

From the above, we gather:

- the format is: `s exp frac`
- `exp` is 5 bits (k = 5) and
- `frac` is 1 bit1 (1 bit for `s` + 5 bits for `exp` + 1 for `frac` = 7 bits)
- since this floating point number encoding scheme is based on the IEEE floating point format, the following equations hold:
    - $V = (-1)^s M 2^E$
    - $E = exp - bias$ (for normalized numbers)
    - $M = 1 + frac$ (for normalized numbers)
    - $E = 1 - bias$ (for denormalized numbers)
    - $M = frac$ (for denormalized numbers)
    - $bias = 2^{k-1} - 1$ (for normalized and denormalized numbers)

a. [2 marks] Compute the bias of this IEEE-like floating point number encoding scheme described above and show your work:

Answer: $k = 5$

$$bias = 2^{k-1}-1 = 2^{5-1} - 1 = 2^4 - 1 = 16 - 1 = 15$$

b. [7 marks] Encode the value `24.5`$_{10}$ using this IEEE-like floating point number representation described in this question. Show all your work. Clearly show the resulting bit pattern and label its three sections `s`, `exp`, `frac`.

Answer: Step 1) 24.5_{10} is a positive number so s = 0

Step 2) R2B(24.5_{10})

=> 24 – 16 (2^4) = 8 and 0.5 - 0.5 (2^{-1}) = 0

8 - 8 (2^3) = 0

24.5_{10} => 11000.1_2

Step 3) normalize 11000.1_2 => $1.10001_2 \times 2^4$

Step 4) Using $V = (-1)^s M 2^E$

1) E= exp – bias => exp = E + bias

=> exp = 4_{10} + 15_{10} = 19_{10} since E = 4_{10}

=> U2B(19_{10}) = 10011_2 (k = 5)

2) M = 1 + frac => frac = M - 1

=> frac = 1.10001_2 – 1 => 0.10001_2 since M = 1.10001_2

=> frac = 10001_2 (ignoring "0.")

but since frac only has 1 bit, 10001_2 cannot be stored in frac, so we need to round frac .**1**~~0001~~$_2$ => .**1**$_2$

=> MSBit is the rounding position (in **blue**)

=> since the value of the rest of the bits (0001_2 = 0.03125 (2^{-5}) - see table below) < ½ the worth of rounding position (½ of 0.5 = 0.25), then we round down which means we only discard the bits 0001_2 from .**1**0001_2)

Step 5) Using the format: s exp frac

the resulting bit pattern encoding 24.5_{10} in the IEEE-like floating point number representation described in this question is: 0 10011 1

s exp frac

c. [2 marks] Write the "range" (non-contiguous) of real numbers (excluding +/- infinity and NAN) that can be encoded using this IEEE-like floating-point representation described in this question. Express this range using the bit patterns (not the actual real numbers).

Answer: "range" (non-contiguous) of real numbers (excluding +/- infinity and NAN) that can be encoded using this IEEE-like floating-point representation described in this question (expressed using the bit patterns):

[1 11110 1 .. 0 11110 1]

d. [3 marks] Can 65536_{10} be encoded as a normalized number in this IEEE-like floating point representation? Briefly explain why/why not.

Hint: Use your range in the above question and the table below.

Answer: 65536_{10} cannot be encoded as a normalized number in this IEEE-like floating point representation because expressed as $V = (-1)^s M 2^E$

65536_{10} is $V = (-1)^0 1.0 \, 2^{16} = 2^{16}$ (see table below)

E= exp – bias => exp = E + bias

exp = $16_{10} + 15_{10} = 31_{10}$ since E = 16_{10}

and U2B(31_{10}) = 11111_2 (k = 5) which is outside the range for exp as indicated in the range given as the answer to the above question:

[1 11110 1 .. 0 11110 1]

When exp = 11111_2 for k = 5, it indicates overflow, i.e., one of the special cases.

Table of Powers of 2

Power of 2^x	Value	Power of 2^x	Value	
0	1			
1	2	-1	1/2	0.5
2	4	-2	1/4	0.25
3	8	-3	1/8	0.125
4	16	-4	1/16	0.0625
5	32	-5	1/32	0.03125
6	64	-6	1/64	0.015625
7	128	-7	1/128	0.0078125
8	256	-8	1/256	0.00390625
9	512	-9	1/512	0.001953125
10	1024	-10	1/1024	0.0009765625
11	2048			
12	4096			
13	8192			
14	16384			
15	32768			
16	65536			

Table of x86-64 Jumps

Instruction	Condition	Description
jmp	always	Unconditional jump
je / jz	ZF	Jump if equal / zero
jne / jnz	~ZF	Jump if not equal / not zero
js	SF	Jump if negative
jns	~SF	Jump if nonnegative
jo	OF	Jump if overflow
jno	~OF	Jump if not overflow
jg / jnle	~(SF ^ OF) & ~ZF	Jump if greater (signed >)
jge / jnl	~(SF ^ OF)	Jump if greater or equal (signed ≥)
jl / jnge	SF ^ OF	Jump if less (signed <)
jle / jng	(SF ^ OF) | ZF	Jump if less or equal (signed ≤)
ja / jnbe	~CF & ~ZF	Jump if greater (unsigned >)
jae / jnb	~CF	Jump if greater or equal (unsigned ≥)
jb / jnae / jc	CF	Jump if less (unsigned <)
jbe / jna / jnc	CF | ZF	Jump if less or equal (unsigned ≤)

Table of x86-64 Registers

64-bit (quad)	32-bit (double)	16-bit (word)	8-bit (byte)		
63..0	31..0	15..0	15..8	7..0	
rax	eax	ax	ah	al	Return value
rbx	ebx	bx	bh	bl	Callee saved
rcx	ecx	cx	ch	cl	4th arg
rdx	edx	dx	dh	dl	3rd arg
rsi	esi	si		sil	2nd arg
rdi	edi	di		dil	1st arg
rbp	ebp	bp		bpl	Callee saved
rsp	esp	sp		spl	Stack pointer
r8	r8d	r8w		r8b	5th arg
r9	r9d	r9w		r9b	6th arg
r10	r10d	r10w		r10b	Caller saved
r11	r11d	r11w		r11b	Caller saved
r12	r12d	r12w		r12b	Callee saved
r13	r13d	r13w		r13b	Callee saved
r14	r14d	r14w		r14b	Callee saved
r15	r15d	r15w		r15b	Callee saved