University of Washington – Computer Science & Engineering

Winter 2018 | Instructor: Mark Wyse | February 5, 2018

CSE 351 Midterm Exam

Last Name:	SOLUTIONS
First Name:	
UW Student ID Number:	
UW NetID (username):	
Academic Integrity Statement: All work on this exam is my own. I had no prior knowledge of the exam contents, nor will I share the contents with others in CSE 351 who haven't taken it yet. Violation of these terms may result in a failing grade. (**please sign**)	

Do not turn the page until 11:30 am.

Instructions

- This exam contains **10** pages, including this cover page, and 2 reference pages.
- Show scratch work for partial credit but answer in the blanks and boxes provided.
- The last page is a reference sheet. Please detach it from the rest of the exam.
- This exam is **closed book and closed notes** (no laptops, tablets, smartphones, wearable devices, or calculators).
- Please silence/turn-off all cell phones, mobile devices, or other noise-making devices. Remove all hats, headphones, and watches.
- You have 50 minutes to complete this exam.

Advice

- Read each question carefully.
- Read *all* questions first and start where you feel most confident.
- Relax and breathe; you are here to learn.

Question	1	2	3	4	5	Total
Points Possible	20	14	20	24	10	**88**
Points Earned						

Question 1: Number Representation [20 pts.]

(A) Complete the following table, assuming an 8-bit, Two's Complement representation. *Remember to use the appropriate bit widths for the hex and binary columns.* [6 pts.]

Decimal (base 10)	Hexadecimal (base 16)	Binary
1	**0x 01**	**0b 0000 0001**
103	**0x 67**	**0b 0110 0111**
-39	0x D9	**0b 1101 1001**
38	**0x 26**	0b 0010 0110

(B) Consider the table below where each row contains two 8-bit integral constants that will be compared using the **<**, **>**, or **==** comparison. Determine which comparison makes the expression: ***Left Constant* (<, >, ==) *Right Constant*** evaluate to ***True***. Also state the type of comparison that is performed (***signed*** or ***unsigned***) assuming we use the same type promotion and casting rules as C does. [8 pts.]

Left Constant	Order (<, >, ==)	Right Constant	Comparison Type
1	**>**	**0**	**signed**
(int) 15U	**==**	15	**signed**
(unsigned) -1	**>**	-2	**unsigned**
(unsigned) -128	**>**	127	**unsigned**
127	**>**	(int) 128U	**signed**

(C) Given the 4-bit bit vector 0b1101, what is its value in decimal (base 10)? *Circle your answer.* [2 pts.]

a. 13
b. -3
c. -5
d. Undefined. Need to specify if we want unsigned, sign & magnitude, two's complement, etc.

(D) In the C programming language, unsigned overflow is well defined. *Circle your answer.* [2 pts.]

a. True
b. False

(E) In the C programming language, signed overflow is well defined. *Circle your answer.* [2 pts.]

a. True
b. False. C allows for a variety of signed integer representations, and thus signed overflow results in undefined behavior

Question 2: Pointers & Memory [14 pts.]

For this problem, assume we are executing on a 64-bit x86-64 machine (**little endian**). The current state of memory (values in hex) is shown below.

```
int *x = 0x00;
long *y = 0x10;
unsigned short *z = 0x18;
```

Memory Address	+0	+1	+2	+3	+4	+5	+6	+7
0x00	ac	ab	dc	ff	0a	a8	11	fa
0x08	de	ad	ac	ae	32	5a	42	ff
0x10	de	ad	be	ef	10	ab	cd	00
0x18	bb	ff	ee	cc	00	11	22	33
0x20	01	00	02	00	08	00	0f	00
0x28	11	11	00	10	01	11	22	17

(A) Fill in the type and value (in hex) for each of the following C expressions. *Remember to use the appropriate bit widths.* [8 pts.]

Expression (in C)	**Type**	**Value (in hex)**
`z`	unsigned short *	0x 0000 0000 0000 0018
`*x`	**int**	0x ffdc abac
`x+3`	**int** *	0x 0000 0000 0000 000c
`*(y-1)`	**long**	0x ff42 5a32 aeac adde
`z[3]`	**unsigned short**	0x 3322

(B) What are the values (in hex) stored in each register shown after the following x86-64 instructions are executed? We are still using the state of memory shown above in part a. *Remember to use the appropriate bit widths.* [6 pts.]

```
movb    (%rsi), %cl
leaq    16(%rsi, %rsi, 4), %rcx
movswl  -10(%rsi, %rax, 4), $r8d
```

Register	**Value (in hex)**
`%rax`	0x 0000 0000 0000 0008
`%rsi`	0x 0000 0000 0000 0018
`%cl`	0x bb
`%rcx`	0x 0000 0000 0000 0088
`%r8d`	0x 0000 1722

Question 3: C Programming & x86-64 Assembly [20 pts.]

Consider the following x86-64 assembly and (mostly blank) C code. The C code is in a file called foo.c and contains a `main` function and a mystery function, `foo`. The function `foo` takes one input and returns a single value. *Fill in the missing C code that is equivalent to the x86-64 assembly for the function* `foo`*. You can use the names of registers (without the %) for C variables.* [18 pts.]

Hint*: the function* `foo` *contains a* ***for loop****. There are more blank lines in the C Code than should be required for your solution.*

x86-64 Assembly: function `foo`

```
  .text
  .globl foo
  .type foo, @function
foo:
  jmp .L2
.L4:
  testb $1, %dil
  je .L3
  movslq %edi, %rdx
  addq %rdx, %rax
.L3:
  subl $3, %edi
.L2:
  testl %edi, %edi
  jg .L4
  ret
```

```
int main(int argc, char **argv) {
  long r = foo(10);
  printf("r: %ld\n", r);
  return 0;
}
```

C Code: file foo.c

```
#include <stdio.h> // for printf

long foo(int x) {
  long sum;
  for (int i = x; i > 0; i = i-3) {
    if (i & 0x1) {
      sum += i;
    }
  }
  return sum;

}
```

Note: variable names may be different in students' answers (e.g., use rax instead of sum).

Follow up: Assume the code in main is correct and has no errors. However, the provided x86-64 code for function `foo` has a single correctness error. *What is the error, and when might this error cause a problem with the execution of* `foo`*? Answer in one or two short English sentences.* [2 pts.]

The variable "sum" (or the variable we return from foo) is never initialized. Thus, it will hold a random value prior to the loop, and the execution of foo will always be incorrect (unless the variable happens to have the value 0 prior to loop execution).

Question 4: Procedures & The Stack [24 pts.]

Consider the following x86-64 assembly and C code for the recursive function `rfun`.

```
// Recursive function rfun
long rfun(char *s) {
  if (*s) {
    long temp = (long)*s;
    s++;
    return temp + rfun(s);
  }
  return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
  char *s = "CSE351";
  long r = rfun(s);
  printf("r: %ld\n", r);
}
```

```
00000000004005e6 <rfun>:
  4005e6: 0f b6 07               movzbl (%rdi),%eax
  4005e9: 84 c0                  test   %al,%al
  4005eb: 74 13                  je     400600 <rfun+0x1a>
  4005ed: 53                     push   %rbx
  4005ee: 48 0f be d8            movsbq %al,%rbx
  4005f2: 48 83 c7 01            add    $0x1,%rdi
  4005f6: e8 eb ff ff ff         callq  4005e6 <rfun>
  4005fb: 48 01 d8               add    %rbx,%rax
  4005fe: eb 06                  jmp    400606 <rfun+0x20>
  400600: b8 00 00 00 00         mov    $0x0,%eax
  400605: c3                     retq
  400606: 5b                     pop    %rbx
  400607: c3                     retq
```

(A) How much space (in bytes) does this function take up in our final executable? [2 pts.]

34 Bytes

Count all bytes (middle column) or subtract address of first instruction (0x4005e6) from last instruction (0x400607), then add 1 byte for the retq instruction.

(B) The compiler automatically creates labels it needs in assembly code. How many labels are used in `rfun` (including the procedure itself)? [2 pts.]

3

The addresses 0x4005e6, 0x400600 (Base Case), 0x400606 (Exit)

(C) In terms of the C function, what value is being saved on the stack? [2 pts.]

`*s`

The movsbq instruction at 0x4005ee puts *s into %rbx, which is pushed onto the stack by the pushq instruction at 0x4005ed.

(D) What is the return address to `rfun` that gets stored on the stack during the recursive calls (in hex)? [2 pts.]

`0x4005fb`

(E) Assume main calls `rfun` with `char *s = "CSE351"` and then prints the result using the `printf` function, as shown in the C code above. Assume `printf` does not call any other procedure. Starting with (and including) `main`, how many total stack frames are created, and what is the maximum depth of the stack? [2 pts.]

Total Frames: **9**	Max Depth: **8**

```
main -> rfun(s) -> rfun(s+1) -> rfun(s+2) -> rfun(s+3) -> rfun(s+4) -> rfun(s+5) -> rfun(s+6)
     -> printf()
```

`The recursive call to rfun(s+6), which handles the null-terminator in the string (base case), still creates a stack frame since we consider the return address pushed to the stack during a procedure call to be part of the callee's stack frame.`

(F) Assume main calls `rfun` with `char *s = "CSE351"`, as shown in the C code. After `main` calls `rfun`, we find that the return address to `main` is stored on the stack at address `0x7fffffffdb38`. On the first call to `rfun`, the register `%rdi` holds the address `0x4006d0`, which is the address of the input string "CSE351" (i.e. `char *s == 0x4006d0`). Assume we stop execution prior to executing the `movsbq` instruction (address `0x4005ee`) during the **fourth** call to `rfun`. [14 pts.]

*For each address in the stack diagram below, fill in both the **value** and a **description** of the entry.*

*The **value** field should be a hex value, an expression involving the C code listed above (e.g., a variable name such as `s` or `r`, or an expression involving one of these), a literal value (integer constant, a string, a character, etc.), "unknown" if the value cannot be determined, or "unused" if the location is unused.*

*The **description** field should be one of the following: "Return address", "Saved %reg" (where reg is the name of a register), a short and descriptive comment, "unused" if the location is unused, or "unknown" if the value is unknown.*

Memory Address	Value	Description
0x7fffffffdb48	unknown	%rsp when main is entered
0x7fffffffdb38	0x400616	Return address to main
0x7fffffffdb30	unknown	original %rbx
0x7fffffffdb28	0x4005fb	Return address
0x7fffffffdb20	*s, "C", 0x43	Saved %rbx
0x7fffffffdb18	0x4005fb	Return address
0x7fffffffdb10	*s, *(s+1), "S", 0x53	Saved %rbx
0x7fffffffdb08	0x4005fb	Return address
0x7fffffffdb00	*s, *(s+2), "E", 0x45	Saved %rbx

Question 5: Fun Stuff [10 pts.]

(A) Assume we are executing code on a machine that uses k-bit addresses, and each addressable memory location stores b-bytes. *What is the total size of the addressable memory space on this machine?* [2 pts.]

(2^k) * b

(B) In C, who/what determines whether local variables are allocated on the stack or stored in registers? *Circle your answer.* [2 pts.]

Programmer **Compiler** Language (C) Runtime Operating System

(C) Assume procedure P calls procedure Q and P stores a value in register `%rbp` prior to calling Q. *True or False: P can safely use the register `%rbp` after Q returns control to P. Circle your answer.* [2 pts.]

a. True. %rbp is a callee saved register.
b. False

(D) Assume we are implementing a new CPU that conforms to the x86-64 instruction set architecture (ISA). *Answer the following questions, in one or two English sentences, regarding this new CPU.* [4 pts.]

a. In modern x86-64 CPUs, a new add operation can be executed every cycle. However, for our new CPU, we realize that we can save power by implementing the add operation such that we can execute a new add only once every three cycles. *Is our new CPU still a valid x86-64 implementation?*

Yes. The x86-64 architecture/specification says nothing about how fast any operation must execute in hardware.

b. In our new CPU implementation, we decide to change the width of register %rsp to be 48-bits, since most modern x86-64 CPUs only use 48-bit physical addresses, but we still use the name %rsp. *Is our CPU still a valid x86-64 implementation?*

No. The x86-64 architecture/specification determines the number and size of registers available to the programmer/compiler. Changing this in our implementation violates the architecture.

CSE 351 Reference Sheet (Midterm)

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
1	2	4	8	16	32	64	128	256	512	1024

IEEE 754 FLOATING-POINT STANDARD

Value: ±1 x Mantissa x 2^{Exponent}

Bit fields: $(-1)^S$ x 1.M x $2^{(E-\text{bias})}$

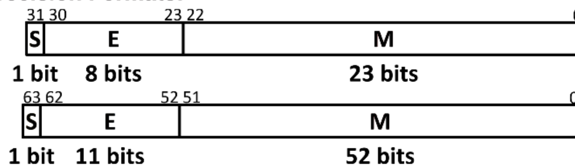
where Single Precision Bias = 127,
Double Precision Bias = 1023.

IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

IEEE Single Precision and Double Precision Formats:



Assembly Instructions

`mov a, b`	Copy from `a` to `b`.
`movs a, b`	Copy from `a` to `b` with sign extension. Needs *two* width specifiers.
`movz a, b`	Copy from `a` to `b` with zero extension. Needs *two* width specifiers.
`leaq a, b`	Compute address and store in `b`. *Note:* the scaling parameter of memory operands can only be 1, 2, 4, or 8.
`push src`	Push `src` onto the stack and decrement stack pointer.
`pop dst`	Pop from the stack into `dst` and increment stack pointer.
`call <func>`	Push return address onto stack and jump to a procedure.
`ret`	Pop return address and jump there.
`add a, b`	Add `a` to `b` and store in `b` (and sets flags).
`sub a, b`	Subtract `a` from `b` (compute `b-a`) and store in `b` (and sets flags).
`imul a, b`	Multiply `a` and `b` and store in `b` (and sets flags).
`and a, b`	Bitwise AND of `a` and `b`, store in `b` (and sets flags).
`sar a, b`	Shift value of `b` *right* (*arithmetic*) by `a` bits, store in `b` (and sets flags).
`shr a, b`	Shift value of `b` *right* (*logical*) by `a` bits, store in `b` (and sets flags).
`shl a, b`	Shift value of `b` *left* by `a` bits, store in `b` (and sets flags).
`cmp a, b`	Compare `b` with `a` (compute `b-a` and set condition codes based on result).
`test a, b`	Bitwise AND of `a` and `b` and set condition codes based on result.
`jmp <label>`	Unconditional jump to address.
`j* <label>`	Conditional jump based on condition codes (*more on next page*).
`set* a`	Set byte based on condition codes.

Conditionals

Instruction		Condition Codes	(op) s, d	test a, b	cmp a, b
je	"Equal"	ZF	d (op) s == 0	b & a == 0	b == a
jne	"Not equal"	~ZF	d (op) s != 0	b & a != 0	b != a
js	"Sign" (negative)	SF	d (op) s < 0	b & a < 0	b-a < 0
jns	(non-negative)	~SF	d (op) s >= 0	b & a >= 0	b-a >= 0
jg	"Greater"	~(SF^OF) & ~ZF	d (op) s > 0	b & a > 0	b > a
jge	"Greater or equal"	~(SF^OF)	d (op) s >= 0	b & a >= 0	b >= a
jl	"Less"	(SF^OF)	d (op) s < 0	b & a < 0	b < a
jle	"Less or equal"	(SF^OF) | ZF	d (op) s <= 0	b & a <= 0	b <= a
ja	"Above" (unsigned >)	~CF & ~ZF	d (op) s > 0U	b & a < 0U	b > a
jb	"Below" (unsigned <)	CF	d (op) s < 0U	b & a > 0U	b < a

Registers

Name	Convention	Name of "virtual" register		
		Lowest 4 bytes	Lowest 2 bytes	Lowest byte
%rax	Return value – **Caller** saved	%eax	%ax	%al
%rbx	**Callee** saved	%ebx	%bx	%bl
%rcx	Argument #4 – **Caller** saved	%ecx	%cx	%cl
%rdx	Argument #3 – **Caller** saved	%edx	%dx	%dl
%rsi	Argument #2 – **Caller** saved	%esi	%si	%sil
%rdi	Argument #1 – **Caller** saved	%edi	%di	%dil
%rsp	Stack Pointer	%esp	%sp	%spl
%rbp	**Callee** saved	%ebp	%bp	%bpl
%r8	Argument #5 – **Caller** saved	%r8d	%r8w	%r8b
%r9	Argument #6 – **Caller** saved	%r9d	%r9w	%r9b
%r10	**Caller** saved	%r10d	%r10w	%r10b
%r11	**Caller** saved	%r11d	%r11w	%r11b
%r12	**Callee** saved	%r12d	%r12w	%r12b
%r13	**Callee** saved	%r13d	%r13w	%r13b
%r14	**Callee** saved	%r14d	%r14w	%r14b
%r15	**Callee** saved	%r15d	%r15w	%r15b

Sizes

C type	x86-64 suffix	Size (bytes)
char	b	1
short	w	2
int	l	4
long	q	8