Computer Architecture, Winter 2019
Final Exam. November 18th, 2019. 14:00-17:00.

Full Name: **Solution Key**

Username: Dr.Evil

Password: i_love_tolh

- Make sure that your exam is not missing any sheets, then write your full name on the front and your initials in the top right corner on every sheet.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- **NOTE:** A grade of 10 or higher will be given to those with 77 or more points, you can get a grade of 11 in this exam. This means you have a leeway for 8 points worth of questions.
- The exam has a maximum score of 85 points. You need 37 points to receive a passing grade on the exam. (A consequence of the previous point)
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems. Wow, you're actually reading this, please write your initials here.
- This exam is **CLOSED BOOK**. You may bring one sheet of A4 paper as notes (cheat-sheet) and also have them tattooed on up to 70% of your body, a non-programmable calculator, and milk chocolate to share with the instructor and the lovely staff who will be continuously clicking their pens during the exam. **Please hand in your single A4 cheat-sheet with your exam.**
  Good luck!

Question	Points	Score
1. Multiple Choice	12	
2. Two's complement numbers	8	
3. Floating point numbers	10	
4. Pointers	8	
5. Linux command line	7	
6. The Cache	10	
7. Assembly loop	10	
8. Assembly matrix	10	
9. The stack	10	
Total:	85	

Multiple Choice

1. Choose the **ONE** right answer for each question.

   (a) (1 point) Given the following settings: `%rax = 0x0000000012345678` and `%rbx = 0x1122334455667788`. What will be the value of register `%rax` be after this command is executed `movsbl %bl, %eax` ?

   A. `%rax = 0x00000000ffffffff`
   B. `%rax = 0x00000000ffffff88`
   C. `%rax = 0xffffffffffffff88`
   D. `%rax = 0x0000000012345688`
   E. `%rax = 0x0000000000000088`

   (b) (1 point) Which of the following is **true** about calling a function in assembly:

   A. Before calling a function, the *caller* must first push, and later pop, all "caller save" registers.
   B. The first thing the *callee* must do is to push, and later pop, all "callee save" registers.
   C. The *callee* only needs to push, and later pop, those "callee save" registers he will use.
   D. The *callee* must push the `%rax` register before calling `ret`.

   (c) (1 point) How much virtual memory does each process have in a 64-bit system? (Process is a running program)

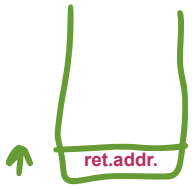
   A. 2 GiB (Gibibyte)
   B. 4 GiB (Gibibyte)
   C. 8 GiB (Gibibyte)
   D. 16 EiB (Exbibyte)

   (d) (1 point) Which one of these options represents `0x12345678` in little endian form?

   A. 0x12 0x34 0x56 0x78
   B. 0x78 0x56 0x34 0x12
   C. 0x87 0x65 0x43 0x21
   D. 00010010000110100010101100111000

(e) (1 point) On a 64-bit system, if `%rsp` has the value `0x7fffff0000` immediately before a `ret` instruction, what is the value of `%rsp` immediately after the `ret`?

A. `0x7ffffefff8`
B. `0x7fffff0000`
C. `0x7fffff0004`
D. `0x7fffff0008`
E. The return address



(f) (1 point) If you buy a HDD that says it is 3.0 TB on the box, how many TiB is it actually?

A. Does not apply to hard disks.
B. 2.73 TiB
C. 3.0 TiB
D. 3.27 TiB

(g) (1 point) Let int `x = -31/8` and `int y = -31 >> 3`. What are the values of `x` and `y`?

A. `x = -3, y = -3`
B. `x = -4, y = -4`
C. `x = -3, y = -4`
D. `x = -4, y = -3`

(h) (1 point) On what value types does C perform logical right shifts? Assume you are using the gcc compiler on a x86-64 CPU.

A. signed types
B. unsigned types
C. signed and unsigned types
D. C does not perform logical right shifts

(i) (1 point) Each HDD platter has two sides and each side is divided up further. What is the order of that division and what are the segments called?

A. Tracks and each track as many sectors
B. Tracks and each track has many zones
C. Pages and each page has many sectors
D. Pages and each page has many tracks
E. Ricks and each Rick has many Morty's

(j) (1 point) The x86 instruction `test` is best described as which of the following:

A. Same as `sub`.
B. Same as `sub`, but doesn't keep the result (only sets flags).
C. Same as `and`.
D. Same as `and`, but doesn't keep the result (only sets flags).

(k) (1 point) Consider an `long* a` and an `long n`. If the value of `%rcx` is `a` and the value of `%rdx` is n, which of the following assembly snippets best corresponds to the C statement return `a[n]`?

A.
```
ret (%rcx,%rdx,8)
```
B.
```
leaq (%rcx,%rdx,8),%rax
ret
```
C.
```
movq (%rcx,%rdx,8),%rax
ret
```
D.
```
movl (%rcx,%rdx,4),%eax
ret
```

(l) (1 point) Given the function foo that is implemented as follows

```
int foo(int x, int y) {
  int ret = ???
  return ret;
}
```

and the assembly is created is

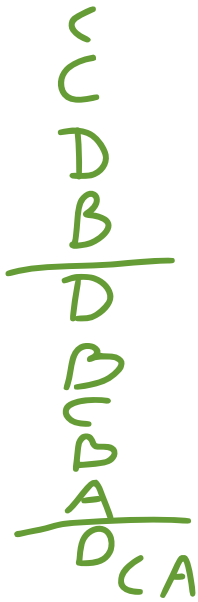
```
foo:
  leal (%rsi, %rdi, 2), %eax
  ret
```

What is the missing C code for `int ret = ?`

A. `2 * x + y;`
B. `x + 2 * y;`
C. `x + y + 2;`
D. `*(x + 2 * y);`

(m) (0 points) How did Chuck Norris solve DataLab?

A. A round house kick to a byte gave him all the bits he needed.
B. After counting to infinity he started sorting the NaNs.
C. He scared all the puzzles into returning valid answers.
D. Who's Chuck Norris?
E. I had to check again to see if this question was worth anything
F. Why am I still staring at this question?



Two's complement numbers

2. (8 points) Two's complement numbers.

   (a) For this problem, assume the following:

   - We are running code on an 8 bit machine using two's complement arithmetic for signed integers (i.e. `int` is only 8 bits).
   - `short` intergers are half the size of the `int` and thus encoded using only 4 bits.
   - Sign extension is performed whenever a `short` is converted to an `int`.

```
int a = -23;
short sa = (short) a;
int b = sa;
unsigned int uc = a;
unsigned int ud = -5 + 3U;
int e = TMax + TMin;
int f = (2 << 2) >> 3;
```

Expression	Decimal	Binary
Zero	0	0000 0000
(short)	0	0000
(int)	-18	**1110 1110**
(int)	**69**	0100 0101
a	**-23**	**1110 1001**
sa	**-7**	**1001**
b	**-7**	**1111 1001**
uc	**233**	**1110 1001**
ud	**254**	**1111 1110**
e	**-1**	**1111 1111**
f	**4**	**0000 0010**

Floating point numbers

3. Answer the following floating point questions.

   (a) (6 points) Consider an **12-bit** IEEE floating-point representation with:
   Express numerical values as fractions in their lowest terms. (e.g., 277/512)

   - **1 sign bit**
   - **5 exponent bits**
   - **6 mantissa bits**

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

   Fill in the blanks in the following table.

Number	Bit Representation
41/4	**0 10010 010010**
-3/65536	**1 00000 110000**
-Infinity	**0 11111 000000**
11/32768	0 00011 011000
NaN	1 11111 111111
49/2	0 10011 100010

   (b) (2 points) Using the format described above: What is the **bit representation** and **numerical value** of the largest possible floating-point number in the **normalized** case.

Solution:

s	E	M	$A = M \cdot 2^E$	**Answer**
$s = +$	$E = e - bias$	$M = 1 + f$	$A = (1 + 63/64) \cdot 2^{15}$	Binary: **0 11110 111111**
	$E = 30 - 15$	$M = 127 \cdot 2^{-6}$	$A = 127 \cdot 2^{-6} \cdot 2^{15}$	Fraction: **63/1**
	$E = 15$	$M = 127 \cdot 2^{-6}$	$A = 127 \cdot 2^9 = 65024$	

   (c) (2 points) In this format: What is the **bit representation** and **numerical value** of the first positive whole number that can not be represented. I.e. what is the first integer value that can not be encoded.

Solution:

s	E	M	$M \cdot 2^E$	**Answer**
$s = +$	$E = e - bias$	$M = 1 + f$	$A = 1 + (0/64) \cdot 2^7$	Binary: **0 10110 000000**
	$e = E + bias$	$M = 1 + 0/64$	$A = 1 \cdot 2^7$	Fraction: **64/1**
	$e = 7 + 15 = 22$	$M = 1$	$A = 129$	129 gets rounded to 128 or 130 dependir

Pointers

4. (8 points) In this problem, your awesome knowledge of pointers and pointer related arithmetic is tested. Assume an x86 architecture.

```
// The array x starts at memory address 0x08048000
int x[5] = { 0x5, 0x20, 0x40, 0x80, 0x100 };
```

Fill in the following table. Give your answers in hexadecimal.

Denote any unknown variables by `?`.

Expression	Type	Value
a = x[1];	int	0x20
b = &x[0];	int*	0x08048000
c = x[2] + 16;	**int**	**0x50**
d = x + 2;	**int***	**0x08048008**
e = *(x + x[0]);	**int**	?
f = &x[2] + 2;	**int***	**0x08048010**

Linux command line

5. (7 points) Your friend Joe has asked you to help him with a problem at work involving finding a specific pattern inside text-based log files (there are more than one). The pattern to find is an ipaddress, namely *212.20.214.4*. The first thing you try is to use *cat* to look at the content of the first file but there is some permission issue. You use *ls -l ./logs/f1.log* to list file permission to verify this.

(a)
```
[Joe@work ~]$ cat ./logs/f1.log
cat: ./logs/file1.log: Permission denied
[Joe@work ~]$ ls -l ./logs/f1.log
----------. 1 Joe domain users 1405797369 Nov 13 10:50 ./logs/f1.log
```

i. What permission is missing and how would you grant them (what command would you use)?

Solution: chmod -R u+r ./logs/*

ii. What command could you then use to find all lines in the logs that contain the ipaddress?

Solution: grep '212.20.214.4' ./logs/*

iii. Now that you have the output on the screen, how would you send the output into a file called *results.txt* instead of printing it on the screen?

Solution: add > *results.txt* to the grep command.

(b) You are in an interview for a job offer at the Umbrella corporation and are given the following ASCII text file.

```
[god@umb ~]$ cat records.txt
Tue Aug 27 04:56:34 GMT 2017 *OK* user:richard.mackenzie door:d4
Tue Aug 27 07:44:07 GMT 2017 *OK* user:virginia.henderson door:d8
Tue Aug 27 07:47:46 GMT 2017 *OK* user:thomas.mackenzie door:d34
Tue Aug 27 07:51:45 GMT 2017 *OK* user:virginia.henderson door:d16
Tue Aug 27 07:52:49 GMT 2017 *OK* user:lily.sanderson door:d10
[god@umb ~]$ head -n 4 records.txt | tail -n 2 | cut -d '.' -f 2
```

You are told that you will be granted a job at the Umbrella corporation if you are able to determine the output from the Linux command given above, or suffer a painful evil laugh from the interviewer.

What is the output from the command? Explain how you got that answer.

Solution: head -n 4 will print only the first 4 lines tail -n 2 will print only the last two of those first 4 cut will split the two lines on . and print only the second column or:
mackenzie door:d34
henderson door:d16

The Cache

6. Consider the following cache problem.

   You may assume the following:

   - The memory is byte addressable.
   - Memory accesses are to **4-byte words**.
   - Physical addresses are **10 bits wide**.
   - The cache is **2-way set associative**, with a **4-byte block size** and **16 total lines**.
   - The cache allows *dirty* blocks as it uses a **write-back** + a **No-write-allocate** policy combo.

   In the following tables, **all numbers are given in hexadecimal**.
   The content of the cache is as follows:

2-way Set Associative Cache															
	Cache line 0							Cache line 1							
Set	Valid	Dirty	Tag	Byte 0	Byte 1	Byte 2	Byte 3	Valid	Dirty	Tag	Byte 0	Byte 1	Byte 2	Byte 3
0	1	0	10	A1	DA	56	50	1	0	04	43	57	5A	DD
1	1	0	04	22	02	69	01	1	0	12	D8	55	FD	34
2	1	0	14	CD	A0	A5	83	0	0	14	9C	12	9D	FE
3	1	0	0D	2C	95	6A	78	1	0	18	20	38	21	97
4	0	0	09	A1	39	BD	6A	0	0	17	CB	0A	7C	30
5	0	0	18	0A	17	8E	04	1	0	18	9D	10	74	7B
6	1	1	07	47	62	31	04	1	0	0F	5A	25	D2	4E
7	0	0	07	CB	BE	58	4E	1	0	17	3C	BE	DE	0C

There are no answers to write on this page so feel free to rip it out if you like.

Remember to double check your answers.

(a) (2 points)
The box below shows the format of a physical address in bits. Indicate (by labeling the diagram) the fields that would be used to determine the following:

BO The bits used as the block offset within the cache line
SI The bits used as the set index (Also called the cache index)
CT The bits that form the cache tag

9	8	7	6	5	4	3	2	1	0
CT	CT	CT	CT	CT	SI	SI	SI	BO	BO

For the given physical address, start by converting it to binary and fill in the bit-boxes. Second, use the format from (a) and fill in the table where you find the values **in hex** for the Block Offset (BO), the Set Index (SI) and the Cache Tag (CT). Third, use the cache configuration and the provided cache table to identify if we have a cache hit or miss. Finally, if applicable, fill in the value of the data returned (as a single hex number). If a read is a cache miss, enter "-" for "Value returned" and in a write operation this field is not applicable (N/A).

(b) (4 points) Read **2 byte** from **Physical address**: `0x0FC`

9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	1	1	0	0

Parameter	Value
Block Offset (BO)	0x0
Set Index (SI)	0x7
Cache Tag (CT)	0x07
Cache Hit? (Y/N)	N (not valid)
Value returned	– (i.e. N/A)

(c) (4 points) Read **1 int** from **Physical address**: `0x288`

9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	0	0	0

Parameter	Value
Block Offset (BO)	0x0
Set Index (SI)	0x2
Cache Tag (CT)	0x14
Cache Hit? (Y/N)	Y
Value returned	0x83A5A0CD

Assembly loop

7. (10 points) Consider the following assembly representation of a function `foo` containing a `for` loop:

```
function(int):
    movl    $10, %eax            # ret = 10
    movl    $0, %edx             # i = 0
    jmp     .L8
.L6:
    movl    %edi, %ecx           # %edi has num so it's no in %ecx
    shrl    $31, %ecx            # bias addjustment
    addl    %edi, %ecx           # add bias to num
    sarl    %ecx                 # num / 2
    addl    %ecx, %eax           # ret += (num /2)
.L7:
    addl    $1, %edx             # i = i+1 (or i++)
.L8:
    leal    0(,%rdi,4), %ecx     # %ecx = num * 4
    cmpl    %edx, %ecx           # %edx is i, so (num*4 <= i) then loop done
    jle     .L4
    cmpl    %edi, %eax           # no jump so we do the if (num < ret )
    jle     .L6                  # go to the else part of the if
    leal    (%rax,%rdi,2), %eax  # ret += num * 2 (if true)
    jmp     .L7                  # loop to incrementing i
.L4:
    ret                          # done
```

Fill in the blanks to provide the functionality of the loop:

```
int function( int num ) {
  int i;
  int ret = _______________;

  for ( i= _____; ___________________; i= _____ ) {

    if ( ____________________ ) {

        ret += ___________________;
    } else {

        ret += ___________________;
    }
  } // end of for-loop

  return ret;
}
```

Solution:

```
int function( int num ) {
  int i;
  int ret = 10;
  for ( i= 0; i < num*4; i++) {
    if ( num < ret ) {
        ret += num << 1;
    } else {
        ret += num / 2;
    }
  }
  return ret;
}
```

Assembly matrix

8. (10 points) Consider the source code below, where `P` and `Q` are constants declared with `#define`.

```
#define P (secret)
#define Q (secret)

int mat1[P][Q];
int mat2[Q][P];

void copy_element(int i, int j)
{
    mat1[i][j] = mat2[j][i];
}
```

This generates the following assembly code:

```
copy_element:                       # i = %edi j = %esi
    movslq  %edi, %rdi              # sign extend i to 8 bytes
    movslq  %esi, %rax              # sign extend j to 8 bytes and to %rax
    leaq    (%rax,%rax,4), %rcx     # %rcx = j + j*4 = 5j
    leaq    (%rcx,%rcx), %rdx       # %rdx = 5j + 5j = 10j
    addq    %rdi, %rdx              # %rdx = 10j + i
    movl    mat2(,%rdx,4), %edx     # get the value of mat2[j][i]
    movq    %rdi, %rsi              # %rsi = %rdi or put i in %rsi
    salq    $4, %rsi                # %rsi = i << 4 = 16i
    subq    %rdi, %rsi              # %rsi = 16i - i = 15i
    addq    %rax, %rsi              # %rsi = 15i + j
    movl    %edx, mat1(,%rsi,4)     # set the value of mat1[i][j]
    ret                             # all done
```

What are the values of P and Q?

Solution:
P = 10
Q = 15

The stack

9. (10 points) Consider the assembly code of the recursive function seen bellow.

   Assembly:

```
Dump of assembler code for function pcount_r:
=> 0x0000000000400530 <+0>: test   %edi,%edi
   0x0000000000400532 <+2>: je     0x400545 <pcount_r+21>
   0x0000000000400534 <+4>: push   %rbx
   0x0000000000400535 <+5>: mov    %edi,%ebx
   0x0000000000400537 <+7>: and    $0x1,%ebx
   0x000000000040053a <+10>: shr    %edi
   0x000000000040053c <+12>: callq  0x400530 <pcount_r>
   0x0000000000400541 <+17>: add    %ebx,%eax
   0x0000000000400543 <+19>: jmp    0x40054b <pcount_r+27>
   0x0000000000400545 <+21>: mov    $0x0,%eax
   0x000000000040054a <+26>: retq
   0x000000000040054b <+27>: pop    %rbx
   0x000000000040054c <+28>: nopl   0x0(%rax)
   0x0000000000400550 <+32>: retq
End of assembler dump.
```

   As we can see the assembly has been halted at the beginning of the foo() function (at 0x400530). However, this is not the first iteration of this recursive code.

   A memory dump of the stack, using `x/136xb`, can be seen below:

```
0x7fffffffe1a8: 0x41 0x05 0x40 0x00 0x00 0x00 0x00 0x00
0x7fffffffe1b0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffe1b8: 0x41 0x05 0x40 0x00 0x00 0x00 0x00 0x00
0x7fffffffe1c0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffe1c8: 0x6a 0x05 0x40 0x00 0x00 0x00 0x00 0x00
0x7fffffffe1d0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffe1d8: 0x15 0xbb 0xa3 0xf7 0xff 0x7f 0x00 0x00
0x7fffffffe1e0: 0x00 0x00 0x00 0x00 0x20 0x00 0x00 0x00
0x7fffffffe1e8: 0xb8 0xe2 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffe1f0: 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00
0x7fffffffe1f8: 0x51 0x05 0x40 0x00 0x00 0x00 0x00 0x00
0x7fffffffe200: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffe208: 0x2e 0xbb 0x6a 0x35 0x79 0x7a 0x4c 0x05
0x7fffffffe210: 0x40 0x04 0x40 0x00 0x00 0x00 0x00 0x00
0x7fffffffe218: 0xb0 0xe2 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffe220: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffe228: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

Following are the questions you need to answer, write your answers as **hexadecimal numbers**:

(a) What is the value of **%rsp** register when the stack was dumped?

(a) **0x7fffffffe1a8**

(b) What is the **return address** of each recursion call?

(b) **0x400541**

(c) What is the number of the current iteration?

(c) **0x3**

(d) What is the **return address** of the function that started the recursion?

(d) **0x40056a**

(e) What was the value of the **%rbx** register in the function that started the recursion?

(e) **0x0**