Name: ______________________________

Sp16 Midterm Q1 Solutions

1. Number Representation (20 pts)

Consider the binary value **110101_2**:

(a) Interpreting this value as an **unsigned 6-bit integer**, what is its value in **decimal**?

2^5+2^4+2^2+2^0 = 32 + 16 + 4 + 1 = 53

(b) If we instead interpret it as a **signed (two's complement) 6-bit integer**, what would its value be in decimal?

-2^5 + 2^4 + 2^2 + 2^0 = -32 + 16 + 4 + 1 = -11

(most significant bit becomes "negatively weighted")

(c) Assuming these are all signed two's complement 6-bit integers, compute the result (leaving it in binary is fine) of each of the following additions. For each, indicate if it resulted in *overflow*.

Note: TMIN = -32

9	001001	−15	110001		011001		101111
−10	+ 110110	−5	+ 111011		+ 001100		+ 011111

Result:

111111	~~1~~ 101100	100101	~~1~~ 001110

Overflow?

No	No	Yes	No

Overflow only occurs for signed addition if the result comes out wrong. The easiest way to determine this is by looking at the signs: if 2 positive values result in a negative result, or 2 negatives result in a positive, then overflow must have occurred.

Now assume that our fictional machine with 6-bit integers also has a 6-bit IEEE-like floating point type, with 1 bit for the sign, 3 bits for the exponent (`exp`) with a *bias* of 3, and 2 bits to represent the mantissa (`frac`), not counting implicit bits.

(d) If we reinterpret the bits of our binary value from above as our 6-bit floating point type, what value, in decimal, do we get?

1	1	0	1	0	1
sign	exp			frac	

-1.01_2 * 2^(4+1-3) = -1.01_2 * 2^2 = -101_2 = -5

(e) If we treat 110101_2 as a *signed integer*, as we did in **(b),** and then *cast* it to a 6-bit floating point value, do we get the correct value in decimal? (That is, can we represent that value in our 6-bit float?) If yes, what is the binary representation? If not, why not? (and in that case you do *not* need to determine the rounded bit representation)

No, we cannot represent it exactly because there are not enough bits for the mantissa.

To determine this, we have to find out what the mantissa would be once we are in "sign-and-magnitude" style: 110101 (-11) → 001011 (+11). In normalized form, this would be: (-1)^1 * 1.011 * 2^3, which means `frac` would need to be 011, which doesn't fit in 2 bits.

(f) Assuming the same rules as standard IEEE floating point, what value (in decimal) does the following represent?

0	0	0	0	0	0
sign	exp			frac	

0.0 (it is a denormalized case)

Sp15 Midterm Q1 Solutions

1 Number Representation(10 points)

Let `x=0xE` and `y=0x7` be integers stored on a machine with a word size of **4bits**. Show your work with the following math operations. **The answers—including truncation—should match those given by our hypothetical machine with 4-bit registers.**

A. (2pt) What hex value is the result of adding these two numbers?

In hex: 0xE + 0x7 = 0x15 → 0x5
In binary converted back to hex: 0xE + 0x7 = 1110 + 0111 = 10101 → 0101 = 0x5
Half credit for not truncating to the appropriate value.

B. (2pt) Interpreting these numbers as unsigned ints, what is the decimal result of adding $x + y$?

In unsigned decimal: 0xE + 0x7 = 14 + 7 = 21 % 16 = 5
Half credit for not truncating to the appropriate value or incorrect conversion.
No credit for computing in signed decimal

C. (2pt) Interpreting x and y as two's complement integers, what is the decimal result of computing $x - y$?

In signed decimal: 0xE - 0x7 =¿ -2 - 7 = -9 → 7
Half credit for not truncating to the appropriate value, or incorrect conversion.
No credit for computing in unsigned decimal

D. (2pt) In one word, what is the phenomenon happening in 1B?

Overflow.

E. (2pt) Circle all statements below that are **TRUE** on a **32-bit architecture**:
Half point each.

- It is possible to lose precision when converting from an int to a float. True
- It is possible to lose precision when converting from a float to an int. True
- It is possible to lose precision when converting from an int into a double. False
- It is possible to lose precision when converting from a double into an int. True

UW NetID: abcde

Wi19 Midterm Q2 Solutions

Question 2: Pointers **(30 total points)**

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

Word Addr	**+0**	**+1**	**+2**	**+3**	**+4**	**+5**	**+6**	**+7**
0x00	BD	28	ED	02	35	72	3A	AF
0x08	66	6F	B1	E9	00	FF	5D	4D
0x10	86	06	04	30	64	31	8C	B3
0x18	63	78	1E	1C	25	34	EE	93
0x20	42	6C	65	67	DE	AD	BE	EF
0x28	CA	FE	D0	0D	1E	93	FA	CE

(a) (16 points) Write the value **in hexadecimal** of each expression within the commented lines at their respective state in the execution of the given program. Write UNKNOWN in the blank if the value cannot be determined.

```
int main(int argc, char** argv) {
    char *charP;
    short *shortP;
    int *intP = 0x00;
    long *longP = 0x28;

    // The value of intP is:              0x 00 00 00 00 00 00 00 00

    // *intP                              0x 02 ED 28 BD

    // &intP                              0x UNKNOWN

    // longP[-2]                          0x 93 EE 34 25 1C 1E 78 63

    charP = 0x20;
    shortP = (short *) intP;
    intP++;
    longP--;

    // *shortP                            0x 28 BD

    // *intP                              0x AF 3A 72 35

    // *((int*) longP)                    0x 67 65 6C 42

    // (short*) (((long*) charP) - 2)     0x 10
}
```

Au16 Midterm Q2 Solutions

Question 2: Pointers & Memory [12 pts]

For this problem we are using a 64-bit x86-64 machine (**little endian**). The initial state of memory (values in hex) is shown below:

Word Addr	**+0**	**+1**	**+2**	**+3**	**+4**	**+5**	**+6**	**+7**
0x00	AC	AB	03	01	BA	5E	BA	11
0x08	5E	00	AB	0C	BE	A7	CE	FA
0x10	1D	B0	99	DE	AD	60	BB	40
0x18	14	CD	FA	1D	D0	41	ED	77
0x20	BA	B0	FF	20	80	AA	BE	EF

```
char* cp = 0x12
short* sp = 0x0C
unsigned* up = 0x2C
```

(A) What are the values (in hex) stored in each register shown after the following x86 instructions are executed? Remember to use the appropriate bit widths. [6 pt]

```
leaw (%rsi, %rdi), %ax
movb 8(%rdi), %bl
movswl (,%rdi,8), %ecx
```

Register	**Value (hex)**
%rdi	0x0000 0000 0000 0004
%rsi	0x0000 0000 0000 0000
%ax	**0x0004**
%bl	**0xBE**
%rcx	**0x0000 0000 FFFF B0BA**

`movb` instruction pulls byte from memory at address 8+4 = 12 = 0x0C.

`movswl` instruction pulls 2 bytes from memory starting at addresses 8*4 = 32 = 0x20.

Remember little-endian! Then sign extended to 32 bits, zero out top 32 bits.

(B) It's a memory scavenger hunt! Complete the C code below to fulfill the behaviors described in the comments using pointer arithmetic. [6 pt]

```
long v1 = (long) *(cp + __3__);   // set v1 = 0x60

unsigned* v2 = up + __5__;        // set v2 = 64

int v3  = *(int *)(sp + __1__);   // set v3 = 0xB01DFACE
```

`v1`: Byte 0x60 is at address 0x15. `0x15 - cp` = 3.

`v2`: No dereferencing, just pointer arithmetic (scaled by `sizeof(unsigned)=4`).

`up` = 0x2C = 44. To get to 64, need to add 20 (5 by pointer arithmetic).

`v3`: The correct bytes can be found (in little-endian order) in addresses 0x0E-0x11.

Want `(0x0E - sp)/sizeof(short)` = 1.

Wi18 Midterm Q5 Solutions

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.

Au16 Midterm Q3 Solutions

Question 3: Computer Architecture Design [8 pts]

Answer the following questions in the boxes provided with a **single sentence fragment**. Please try to write as legibly as possible.

(A) Why can't we upgrade to more registers like we can with memory? [2 pt]

Registers are part of the CPU (and the architecture) and are not modular like RAM.

(B) Why don't we see new assembly instruction sets as frequently as we see new programming languages? [2 pt]

Hard to implement/get adopted – need to build new hardware. (by comparison, a new programming language only needs a new compiler – software)

(C) Name one reason why a program written in a CISC language might run slower than the same program written in a RISC language and one reason why the reverse might be true: [4 pt]

CISC slower:	RISC slower:
Complicated instructions take longer to execute (fewer instructions, but each is slower).	Need more instructions to do complicated computations (faster instructions, but more numerous).

Sp19 Midterm Q3 Solutions

3. C and Assembly (11 points total)

You are given the following x86-64 assembly function:

```
mystery:
        movl    $0, %edx
        movl    $0, %eax
.L3:
        cmpl    %esi, %edx
        jge     .L1
        movslq  %edx, %rcx
        addl    (%rdi,%rcx,4), %eax
        addl    $1, %edx
        jmp     .L3
.L1:
        rep ret
```

a) (1 pt) What variable type would **`%rdi`** be in the corresponding C program?

`int*`

b) (1 pt) What variable type would **`%rsi`** be in the corresponding C program?

`int`

c) (7 pts) Fill in the missing C code that is equivalent to the x86-64 assembly above:

```
______int_____ mystery( (answer to a) rdi, (answer to b) rsi) {

     ___ int _____ eax = ___0 ;________

     for (int edx = 0; edx < rsi; edx++) {

          eax += rdi[edx];
     }



     return eax;
}
```

d) (2 pts) In 1 sentence, describe what this function is doing?

Summing the first rsi elements of the int array starting at rdi

Wi15 Midterm Q2 Solutions

2. Assembly and C (20 points)

Consider the following x86-64 assembly and C code:

```
<do_something>:
    cmp    $0x0,%rsi
    jle     <end>
    xor    %rax,%rax
    sub    $0x1,%rsi

<loop>:
    lea    (%rdi,%rsi,  2  ),%rdx
    add    (%rdx),%ax
    sub    $0x1,%rsi
    jns    <loop>

<end>:
    retq


short do_something(short* a, int len) {
    short result = 0;
    for (int i = len - 1; i >= 0 ;  i-- ) {
        result += a[i] ;
    }
    return result;
}
```

(a) Both code segments are implementations of the unknown function `do_something`. Fill in the missing blanks in both versions. (Hint: `%rax` and `%rdi` are used for `result` and `a` respectively. `%rsi` is used for both `len` and `i`)

(b) Briefly describe the value that `do_something` returns and how it is computed. Use only variable names from the C version in your answer.

`do_something` returns the sum of the shorts pointed to by `a`. It does so by traversing the array backwards.

Sp14 Midterm Q4 Solutions

4. Stack Discipline (30 points)

The following function recursively computes the greatest common divisor of the integers a, b:

```
int gcd(int a, int b) {
     if (b == 0) {
          return a;
     } else {
          return gcd(b, a % b);
     }
}
```

Here is the x86_64 assembly for the same function:

```
4006c6 <gcd>:
4006c6:   sub       $0x18, %rsp
4006ca:   mov       %edi, 0x10(%rsp)
4006ce:   mov       %esi, 0x08(%rsp)
4006d2:   cmpl      $0x0, %esi
4006d7:   jne       4006df <gcd+0x19>
4006d9:   mov       0x10(%rsp), %eax
4006dd:   jmp       4006f5 <gcd+0x2f>
4006df:   mov       0x10(%rsp), %eax
4006e3:   cltd
4006e4:   idivl     0x08(%rsp)
4006e8:   mov       0x08(%rsp), %eax
4006ec:   mov       %edx, %esi
4006ee:   mov       %eax, %edi
4006f0:   callq     4006c6 <gcd>
4006f5:   add       $0x18, %rsp
4006f9:   retq
```

Note: **`cltd`** is an instruction that sign extends %eax into %edx to form the 64-bit signed value represented by the concatenation of [%edx | %eax].

Note: **`idivl <mem>`** is an instruction divides the 64-bit value [%edx | %eax] by the long stored at <mem>, storing the quotient in %eax and the remainder in %edx.

A. Suppose we call gcd(144, 64) from another function (i.e. main()), and set a breakpoint just before the statement “return a”. When the program hits that breakpoint, what will the stack look like, starting at the top of the stack and going all the way down to the saved instruction address in main()? Label all return addresses as "ret addr", label local variables, and leave all unused space blank.

Memory address on stack	**Value (8 bytes per line)**	
0x7fffffffffffad0	Return address back to main	<-%rsp points here at start of procedure
0x7fffffffffffac8	*1st of 3 local variables on stack (argument a = 144)*	
0x7fffffffffffac0	*2nd of 3 local variables on stack (argument b = 64)*	
0x7fffffffffffab8	*3rd of 3 local variables on stack (unused)*	
0x7fffffffffffab0	Return address back to gcd(144, 64)	
0x7fffffffffffaa8	*1st of 3 local variables on stack (argument a = 64)*	
0x7fffffffffffaa0	*2nd of 3 local variables on stack (argument b = 16)*	
0x7fffffffffffa98	*3rd of 3 local variables on stack (unused)*	
0x7fffffffffffa90	Return address back to gcd(64,16)	
0x7fffffffffffa88	*1st of 3 local variables on stack (argument a = 16)*	
0x7fffffffffffa80	*2nd of 3 local variables on stack (argument b = 0)*	
0x7fffffffffffa78	*3rd of 3 local variables on stack (unused)*	*<-%rsp at “return a” in 3rd recursive call*
0x7fffffffffffa70		

B. How many total bytes of local stack space are created in each frame (in decimal)?

______32_______ *24 allocated explicitly and 8 for the return address.*

C. When the function begins, where are the arguments (a, b) stored?

They are stored in the registers %rdi and %rsi, respectively.

D. From a memory-usage perspective, why are iterative algorithms generally preferred over recursive algorithms?

Recursive algorithm continue to grow the stack for the maximum number of recursions which may be hard to estimate.

Name: ______________________________

Sp16 Midterm Q4 Solutions

4. Stack Discipline (30 pts)

Take a look at the following recursive function written in C:

```
long sum_asc(long * x, long * y) {
  long sum = 0;
  long v = *x;
  if (v >= *y) {
    sum = sum_asc(x + 1, &v);
  }
  sum += v;
  return sum;
}
```

Breakpoint

Here is the x86-64 disassembly for the same function:

```
0000000000400536 <sum_asc>:
  0x400536:  pushq  %rbx
  0x400537:  subq   $0x10,%rsp
  0x40053b:  movq   (%rdi),%rbx
  0x40053e:  movq   %rbx,0x8(%rsp)
  0x400543:  movq   $0x0,%rax
  0x400548:  cmpq   (%rsi),%rbx
  0x40054b:  jl     40055b <sum_asc+0x25>
  0x40054d:  addq   $0x8,%rdi
  0x400551:  leaq   0x8(%rsp),%rsi
  0x400556:  callq  400536 <sum_asc>
  0x40055b:  addq   %rbx,%rax
  0x40055e:  addq   $0x10,%rsp
  0x400562:  popq   %rbx
  0x400563:  ret
```

Breakpoint

Suppose that `main` has initialized some memory in its stack frame and then called `sum_asc`. We set a breakpoint at "`return sum`", which will stop execution right before the first return (from the deepest point of recursion). That is, we will have executed the popq at `0x400562`, but not the `ret`.

(a) ***On the next page:* Fill in the state of the registers and the contents of the stack (in memory) <u>when the program hits that breakpoint</u>.** For the contents of the stack, give both a description of the item stored at that location as well as the value. If a location on the stack is not used, write "unused" in the Description for that address and put "---" for its Value. You may list the Values in hex (prefixed by `0x`) or decimal. Unless preceded by `0x`, we will assume decimal. It is fine to use `ff...` for sequences of `f`'s, as we do for some of the initial register values. Add more rows to the table as needed. (20 pts)

Name: ______________________________

Register	Original Value	Value at Breakpoint
%rsp	0x7ff..070	**0x7ff..050**
%rdi	0x7ff..080	**0x7ff..088**
%rsi	0x7ff..078	**0x7ff..060**
%rbx	2	**7**
%rax	42	**2**

Memory Address	Description of item	Value at Breakpoint
0x7fffffff090	Initialized in main to: 1	1
0x7fffffff088	Initialized in main to: 2	2
0x7fffffff080	Initialized in main to: 7	7
0x7fffffff078	Initialized in main to: 3	3
0x7fffffff070	Return address back to main	0x400594
0x7fffffff068	**Original %rbx value**	**2**
0x7fffffff060	**Temporary variable v or %rbx**	**7**
0x7fffffff058	***Unused***	**---**
0x7fffffff050	**Return address back to sum_asc**	**0x40055b**
0x7fffffff048	**Previous value of %rbx (v from first call)**	**7**
0x7fffffff040	**Temporary variable v or %rbx**	**2**
0x7fffffff038	***Unused***	**---**
0x7fffffff030		
0x7fffffff028		
0x7fffffff020		
0x7fffffff018		
0x7fffffff010		
0x7fffffff008		
0x7fffffff000		

Grading Rubric

Registers (6 pts)

- %rsp: (2) *(-1 if only missing last pop)*
- %rdi: (1)
- %rsi: (1)
- %rbx: (1)
- %rax: (1)

Stack (14 pts)

Generally, 1 pt for each stack frame where correct desc/value appears.

- saved %rbx: desc (2), value (2)
- temp "v"/"rbx": desc (2), value (2)
- unused space: (2) *second unused optional*
- return address desc (2), value (2)

Additional questions about this problem on the next page.

Continue to refer to the `sum_asc` code from the previous 2 pages.

(b) What is the purpose of this line of assembly code: `0x40055e:   addq   $0x10,%rsp`? Explain briefly (at a high level) something bad that could happen if we removed it. (5 pts)

This resets the stack pointer to deallocate temporary storage. If we didn't increment here, we wouldn't pop the correct return address or the right value of %rbx.

Note that this would not lead to slow stack overflow due to leaking memory – the first ret would most likely crash because it got the wrong return address; it is highly unlikely that it could continue to execute successfully long enough for this leak to be a problem.

(c) Why does this function `push %rbx` at `0x400536` and `pop %rbx` at `0x400562`? (5 pts)

The register `%rbx` is a callee-saved register, so if we use it, it is our responsibility to set it back to what it was before we return from the function.

We gave some points for people recognizing that the two have to be matched for everything else on the stack to work out (similar to the reasoning for deallocation above), but if that were the only reason, then we could have just left both of the instructions out.