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

Sp16 Midterm Q1 Solutions

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)^$\underline{1}$ * 1.$\underline{011}$ * 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)

Sp17 Midterm Q4 Solutions

4. **Pointers, Memory & Registers (14 points)**

Assuming a 64-bit x86-64 machine (little endian), you are given the following variables and initial state of memory (values in hex) shown below:

Address	+0	+1	+2	+3	+4	+5	+6	+7
0x00	AB	EE	1E	AC	D5	8E	10	E7
0x08	F7	84	32	2D	A5	F2	3A	CA
0x10	83	14	53	B9	70	03	F4	31
0x18	01	20	FE	34	46	E4	FC	52
0x20	4C	A8	B5	C3	D0	ED	53	17

```
int* ip = 0x00;
short* sp = 0x20;
long* yp = 0x10;
```

a) Fill in the type and value for each of the following C expressions. If a value cannot be determined from the given information answer UNKNOWN.

Expression (in C)	Type	Value (in hex)
`yp + 2`	long*	0x20
`*(sp – 1)`	short	0x52FC
`ip[5]`	int	0x31F40370
`&ip`	int**	UNKNOWN

b) Assuming that all registers start with the value 0, except `%rax` which is set to 0x4, fill in the values (in hex) stored in each register after the following x86 instructions are executed. *Remember to give enough hex digits to fill up the width of the register name listed.*

```
movl 2(%rax), %ebx

leal (%rax,%rax,2), %ecx

movsbl 4(%rax), %edi

subw (,%rax,2), %si
```

Register	Value (in hex)
`%rax`	`0x0000 0000 0000 0004`
`%ebx`	`0x84f7 e710`
`%ecx`	`0x0000 000c`
`%rdi`	`0x0000 0000 ffff fff7`
`%si`	`0x7B09`

Sp17 Midterm Q5 Solutions

5. Stack Discipline (15 points)

Examine the following recursive function:

```
long sunny(long a, long *b) {
  long temp;
  if (a < 1) {
    return *b - 8;
  } else {
    temp = a - 1;
    return temp + sunny(temp - 2, &temp);
  }
}
```

Here is the x86_64 assembly for the same function:

```
0000000000400536 <sunny>:
  400536:       test   %rdi,%rdi
  400539:       jg     400543 <sunny+0xd>
  40053b:       mov    (%rsi),%rax
  40053e:       sub    $0x8,%rax
  - - - - - - - - - - - - - - - - - - - - - -
  400542:       retq
  400543:       push   %rbx
  400544:       sub    $0x10,%rsp
  400548:       lea    -0x1(%rdi),%rbx
  40054c:       mov    %rbx,0x8(%rsp)
  400551:       sub    $0x3,%rdi
  400555:       lea    0x8(%rsp),%rsi
  40055a:       callq  400536 <sunny>
  40055f:       add    %rbx,%rax
  400562:       add    $0x10,%rsp
  400566:       pop    %rbx
  400567:       retq
```

Breakpoint

We call `sunny` from `main()`, with registers %`rsi` = `0x7ff`…`ffad8` and %`rdi` = 6. The value stored at address `0x7ff`…`ffad8` is the long value 32 (0x20). We set a breakpoint at "`return *b - 8`" (i.e. we are just about to return from `sunny()` without making another recursive call). We have executed the `sub` instruction at `40053e` but have not yet executed the `retq`.

Fill in the register values on the next page and **draw what the stack will look like when the program hits that breakpoint**. Give both a description of the item stored at that location and the value stored at that location. 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 or decimal. Unless preceded by `0x` we will assume decimal. It is fine to use `f`…`f` for sequences of `f`'s as shown above for %`rsi`. Add more rows to the table as needed. Also, fill in the box on the next page to include the value this call to `sunny` will *finally* return to `main`.

Register	Original Value	Value at Breakpoint
rsp	0x7ff…ffad0	0x7ff…ffa90
rdi	6	0
rsi	0x7ff…ffad8	0x7ff…ffaa0
rbx	4	2
rax	5	-6

DON'T FORGET

What value is **finally** returned to **main** by this call? **1**

Memory address on stack	Name/description of item	Value
0x7fffffffffffad8	Local var in main	0x20
0x7fffffffffffad0	Return address back to main	0x400827
0x7fffffffffffac8	Saved %rbx	*4*
0x7fffffffffffac0	*temp*	*5*
0x7fffffffffffab8	*Unused*	---------------
0x7fffffffffffab0	Return address to sunny	0x40055f
0x7fffffffffffaa8	Saved %rbx	5
0x7fffffffffffaa0	*temp*	2
0x7fffffffffffa98	*Unused*	---------------
0x7fffffffffffa90	Return address to sunny	0x40055f
0x7fffffffffffa88		
0x7fffffffffffa80		
0x7fffffffffffa78		
0x7fffffffffffa70		
0x7fffffffffffa68		
0x7fffffffffffa60		

Au16 Midterm Q5 Solutions

Question 5: The Stack [12 pts]

The recursive factorial function `fact()` and its x86-64 disassembly is shown below:

```
int fact(int n) {
     if(n==0 || n==1)
          return 1;
     return n*fact(n-1);
```

```
000000000040052d <fact>:
  40052d:  83 ff 00          cmpl   $0, %edi
  400530:  74 05             je     400537 <fact+0xa>
  400532:  83 ff 01          cmpl   $1, %edi
  400535:  75 07             jne    40053e <fact+0x11>
  400537:  b8 01 00 00 00    movl   $1, %eax
  40053c:  eb 0d             jmp    40054b <fact+0x1e>
  40053e:  57                pushq  %rdi
  40053f:  83 ef 01          subl   $1, %edi
  400542:  e8 e6 ff ff ff    call   40052d <fact>
  400547:  5f                popq   %rdi
  400548:  0f af c7          imull  %edi, %eax
  40054b:  f3 c3             rep ret
```

(A) Circle one: [1 pt] `fact()` is saving `%rdi` to the Stack as a [✓ **Caller**] // **Callee**

(B) How much space (in bytes) does this function take up in our final executable? [2 pt]

Count all bytes (middle columns) or subtract address of next instruction (0x40054d) from 0x40052d.

32 B

(C) **Stack overflow** is when the stack exceeds its limits (i.e. runs into the Heap). Provide an argument to `fact(n)` here that will cause stack overflow. [2 pt]

Any negative int

We did mention in the lecture slides that the Stack has 8 MiB limit in x86-64, so since 16B per stack frame, credit for anything between 2^{19} and TMax (2^{31}-1).

(D) If we use the `main` function shown below, answer the following for the execution of the entire program: [4 pt]

```
void main() {
      printf("result = %d\n",fact(3));
}
```

Total frames created: **5**	Maximum stack frame depth: **4**

main → fact(3) → fact(2) → fact(1)
→ printf

(E) In the situation described above where `main()` calls `fact(3)`, we find that the word `0x2` is stored on the Stack at address `0x7fffdc7ba888`. At what address on the Stack can we find the return address to `main()`? [3 pt]

`0x7fffdc7ba8a0`

Only `%rdi` (current `n`) and return address get pushed onto Stack during `fact()`.

Address	**Contents**
	<Rest of Stack>
`0x7fffdc7ba8a0`	Return addr to `main()`
`0x7fffdc7ba898`	Old `%rdi` (n=3)
`0x7fffdc7ba890`	Return addr to `fact()`
`0x7fffdc7ba888`	Old `%rdi` (n=2)
`0x7fffdc7ba880`	Return addr to `fact()`

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.

Wi17 Midterm Q3 Solutions

3. Assembly and C (30 points)

Consider the following x86-64 assembly, (partially blank) C code, and memory listing. Addresses and values are 64-bit.

```
foo:
  movl $0, %eax

L1:
  testq %rdi, %rdi
  je L2
  movq (%rdi), %rdi
  addl $1, %eax
  jmp L1

L2:
  ret
```

```
int foo(long *p) {
    int result = 0;
    while (p != NULL) {
        // cast p, then deref
        p = *(long**)p;
        result = result + 1;
    }
    return result;
}
```

Address	Value
0x1000	0x1030
0x1008	0x1020
0x1010	0x1000
0x1018	0x0000
0x1020	0x1030
0x1028	0x1008
0x1030	0x0000
0x1038	0x1038
0x1040	0x1048
0x1048	0x1040

(a) Given the assembly of `foo`, fill in the blanks of the C version.

(b) Trace the execution of the call to `foo((long*)0x1000)` in the table to the right. Show which instruction is executed in each step until `foo` returns. In each space, place the **the assembly instruction** and the values of the appropriate registers **after that instruction executes**. *You may leave those spots blank when the value does not change.* You might not need all steps listed on the table.

Instruction	`%rdi` (hex)	`%eax` (decimal)
`movl`	`0x1000`	0
`testq`		
`je`		
`movq`	`0x1030`	
`addl`		1
`jmp`		
`testq`		
`je`		
`movq`	`0x0`	
`addl`		2
`jmp`		
`testq`		
`je`		
`ret`		

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

It returns the depth of the pointer chain from `p` by counting how many times it can be dereferenced before it's `NULL`.

Name:____

Wi16 Midterm Q4 Solutions

4. (**9** points) Computer-Architecture Design

(a) In roughly one English sentence, give a reason that it is better to have *fewer* registers in an instruction-set architecture.

(b) In roughly one English sentence, give a reason that it is better to have *many* registers in an instruction-set architecture.

(c) Yes or no: If we decided to change the x86-64 calling convention to make `%rbx` caller-saved, would the implementation of the CPU need to change?

Solution:

(a) We can implement the CPU with faster access to the registers, and we can design instruction encodings with fewer bits for identifying a register. (One reason is enough for full credit.)

We also gave partial credit for saying there are fewer registers to save across a function call. This really is not a correct answer because unused registers do not take any effort to save, but the intuition on an open-ended question is good. And there is a related issue when different programs/threads take turns executing.

(b) It is easier for humans or the compiler to write code without having to use the slower and harder-to-use memory on the stack for temporary variables.

(c) No (it's just a convention)