CSE 351 – Midterm Exam – Spring 2016

May 2, 2015

Solution

Name: ________________________________

UWNetID: ________________________________

Please do not turn the page until 11:30.

Instructions

- The exam is closed book, closed notes (no calculators, no mobile phones, no laptops, no futuristic Google Glasses or HoloLenses).
- **Please stop promptly at 12:20.**
- There are 100 points total, divided unevenly among **5** problems (each with multiple parts).
- The exam is **printed double-sided.** If you separate any pages, be sure to print your name at the top of each separated page so we can match them up.
- Useful **reference material** can be found on the last 2 pages of the exam. Feel free to tear it off.

Advice

- Read questions carefully before starting.
- Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what is your final answer.
- Questions are not necessarily in order of difficulty. Skip around or read ahead. Make sure you get to all the questions.
- Relax. You are here to learn.

Problem	Points	Score
1. Number Representation	20	
2. C to Assembly	25	
3. Computer Architecture	10	
4. Stack Discipline	30	
5. Pointers and Memory	15	

Name: ______________________________

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.

Name: ______________________________

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?

sign	exp			frac	
1	1	0	1	0	1

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

(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?

sign	exp			frac	
0	0	0	0	0	0

0.0 (it is a denormalized case)

Name: ______________________________

2. C to Assembly (25 pts)

Imagine we're designing a new, super low-power computing device that will be powered by ambient radio waves (that part is actually a real research project). Our imaginary device's CPU supports the x86-64 ISA, but its general-purpose integer multiply instruction (`imul`) is very bad and consumes lots of power. Luckily, we have learned several other ways to do multiplication in x86-64 in certain situations. To take advantage of these, we are designing a custom multiply function, `spmult`, that checks for specific arguments where we can use other instructions to do the multiplication. But we need your help to finish the implementation.

Fill in the blanks with the correct instructions or operands. It is okay to leave off size suffixes. *Hint:* there are reference sheets with x86-64 registers and instructions at the end of the exam.

```
long spmult(long x, long y) {
  if (y == 0)        return 0;
  else if (y == 1)   return x;
  else if (y == 4)   return x * 4;
  else if (y == 5)   return x * 5;
  else if (y == 16)  return x * 16;
  else               return x * y;
}
```

```
spmult(long, long):
        testq   %rsi, %rsi
        je      .L3
        cmpq    $1, %rsi
        je      .L4
        cmpq    $4, %rsi
        jne     .L1
.case4:
        leaq    0(,%rdi,4), %rax
        ret
.L1:
        cmpq    $5, %rsi
        jne     .L2
        leaq    (%rdi,%rdi,4), %rax
        ret
.L2:
        cmpq    $16, %rsi
        jne     .else
        movq    %rdi, %rax
        salq    $4, %rax
        ret
.L3:
        movq    $0, %rax
        ret
.L4:
        movq    %rdi, %rax
        ret
.else:  # fall back to multiply
        movq    %rsi, %rax
        imulq   %rdi, %rax
        ret
```

3. Computer Architecture Design (10 pts)

In the previous question, we designed a new multiply function optimized for an imaginary low-power CPU implementing the **x86-64 ISA**. The questions in this section consider various design choices facing the engineers of that CPU.

(a) We designed a new multiply function because our low-power x86-64 CPU has a power-hungry implementation of `imul`. *Would it have been okay for the designers of the chip to simply not implement `imul`? Briefly explain why or why not (roughly one English sentence).* (4 pts)

No, the designers would have to implement `imul` somehow, otherwise it wouldn't conform to the x86-64 interface, and programs written in x86-64 would crash on it. However, that does not mean the implementation can't be really terrible for certain instructions.

(b) Faster registers consume more power. What if the designers decided to make half of the registers slower (probably r8-r15 because they're used less often)? *Would this still be a valid x86-64 implementation? Explain briefly.* (3 pts)

Yes, the architecture/specification says nothing about how fast anything is.

(c) Bigger registers consume more power. What if the designers wanted to make the registers smaller, only 4-bytes wide (but still call them `%r_`). *Would this still implement the x86-64 ISA? Explain briefly.* (3 pts)

No, if you make the registers smaller then you can't hold 8-byte pointers (or 8-byte longs).

Name: ______________________________

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;
  - - - - - - - - - - - - - - - - -     <----  Breakpoint
  return sum;
}
```

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
  - - - - - - - - - - - - - - - - - - - - - - -     <----  Breakpoint
  0x400563:  ret
```

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.

Name: ______________________________

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.

Name: ______________________________

5. Pointers and Memory (15 pts)

For this section, refer to this 8-byte aligned diagram of memory, with addresses increasing top-to-bottom and left-to-right (address 0x00 at the top left). When answering the questions below, don't forget that x86-64 machines are little-endian. If you don't remember exactly how endianness works, you should still be able to get significant partial credit.

```
int*  x = 0x10;
long* y = 0x20;
char* s = 0x00;
```

Memory Address	+0	+1	+2	+3	+4	+5	+6	+7
0x00	aa	bb	cc	dd	ee	ff	00	11
0x08	00	00	00	00	00	00	00	00
0x10	ab	01	51	f0	07	06	05	04
0x18	de	ad	be	ef	10	00	00	00
0x20	ba	ca	ff	ff	1a	2b	3c	4d
0x28	a0	b0	c0	d0	a1	b1	c1	d1

(a) Fill in the type and value for each of the following C expressions:
(.5pts for each type, 2pts for each value)

Expression (in C)	Type	Value (in hex)
`*x`	int	0xf05101ab
`x+1`	int*	0x14
`*(y-1)`	long	0x00000010efbeadde
`s[4]`	char	0xEE

(b) Assume that all registers start with the value 0, except `%rax` which is set to 8. Determine what the final values of each of these registers will be *after* executing the following instructions:

Instruction	Points	Register	Value
	(1 pt)	`%rax`	8
`movb %al, %bl`	(2 pts)	`%bl`	8 or 0x8
`leal 2(%rax), %ecx`	(2 pts)	`%ecx`	10 or 0xa
`movsbw (,%rax,4), %dx`		`%dx`	65466 or 0xffba

End of exam!

Name: ______________________________

References

Powers of 2

$2^0 = 1$ $2^{-1} = 0.5$
$2^2 = 4$ $2^{-2} = 0.25$
$2^3 = 8$ $2^{-3} = 0.125$
$2^4 = 16$ $2^{-4} = 0.0625$
$2^6 = 64$ $2^{-5} = 0.03125$
$2^8 = 256$
$2^{10} = 1024$

Hex conversions

`0x00 = 0`

`0xA  = 0xa = 0b1010 10`

`0xF  = 0xf = 15`

`0x10 = 16`

`0x20 = 32`

Assembly Instructions

`mov a,b`	Copy from `a` to `b`
`movs a,b`	Copy from `a` to `b` with sign extension.
`movz a,b`	Copy from `a` to `b` with zero extension.
`lea 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)
`imul a,b`	Multiply `a` by `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 `a` and `b` and set condition codes based on result.
`jmp <label>`	Jump to address
`j_ <label>`	Conditional jump based on condition codes (*more on next page*)
`set_ a`	Set byte based on condition codes.

Conditionals

		cmp b,a	test a,b
je	"Equal"	a == b	a & b == 0
jne	"Not equal"	a != b	a & b != 0
js	"Sign" (negative)		a & b < 0
jns	(non-negative)		a & b >= 0
jg	"Greater"	a > b	a & b > 0
jge	"Greater or equal"	a >= b	a & b >= 0
jl	"Less"	a < b	a & b < 0
jle	"Less or equal"	a <= b	a & b <= 0
ja	"Above" (unsigned >)	a > b	
jb	"Below" (unsigned <)	a < b	

Sizes

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

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