UCLA Computer Science 33 (Fall 2016)
Midterm 1
100 minutes, 100 points, open book, open notes.
Use a separate sheet of paper for each answer, except for problem 1 where you should simply write the answer on your copy of the exam.
Put a big problem number at each sheet's top.
Turn in your sheets in increasing numeric order.

Name: Tyler Carson Student ID: 304675148

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 18 | 4 | 5 | 4 | 5 | 2/4 | 8 | 0 | 1 | 8 |

59

Consider the following C program and x86-64 assembly-language code in gcc -S format:

```
int a (int m) { return m << 31 << 2; }                L5
int b (int m) { return m >> 31 >> 2; }                L4
int c (int m, int n) { return m >> n >> 2; }          L3
int d (int m, int n) { return m << n >> 2; }
int e (int m, int n) { return n >> m >> 2; }          L2
int f (int m, int n) { return n << m >> 2; }          L9
int g (int m, int n) { return m >> 2 >> n; }          L1
int h (int m, int n) { return m << 2 >> n; }          L6
int i (int m, int n) { return n >> 2 >> m; }          L8
int j (int m, int n) { return n << 2 >> m; }          L7
```

```
L1:
        movl    %edi, %eax        eax = m
        movl    %esi, %ecx        ecx = n
        sarl    $2, %eax          m >> 2 >> n
        sarl    %cl, %eax
        ret
L2:
        movl    %edi, %ecx        ecx = m
        sarl    %cl, %esi         n >> m
        movl    %esi, %eax        eax = esi
        sarl    $2, %eax
        ret                       m >> m >> 2
```

```
L3:
        movl    %esi, %ecx        ecx = n
        sarl    %cl, %edi         m >> n
        movl    %edi, %eax        eax = m >> n
        sarl    $2, %eax
        ret                       m >> n >> 2
L4:
        movl    %edi, %eax        shift til sign bit
        sarl    $31, %eax
        ret
L5:
        xorl    %eax, %eax        shift all away
        ret
L6:
        leal    0(,%rdi,4), %eax  eax = m*4
        movl    %esi, %ecx        ecx = n
        sarl    %cl, %eax         (m<<2)>>n
        ret
L7:
        leal    0(,%rsi,4), %eax  ~~eax = 4*~~ n << 2
        movl    %edi, %ecx        ecx = edi
        sarl    %cl, %eax         eax >> ecx
        ret                       n << 2 >> m
L8:
        movl    %esi, %eax        eax = n           n >> 2 >> m
        movl    %edi, %ecx        ecx = m
        sarl    $2, %eax          eax >>= 2
        sarl    %cl, %eax         eax >>= m
        ret
L9:
        movl    %edi, %ecx        ecx = m           n << m >> 2
        sall    %cl, %esi         esi = n << m
        movl    %esi, %eax        eax = esi
        sarl    $2, %eax          eax >> 2
        ret
```

This code is covered in questions 1 through 5 on the last page.

Consider also the following x86-64 assembly-language code, in gcc -S format.

```
F:
        movl    %edi, %eax
        andl    $1, %eax
        ret

G:
        pushq   %rbp        base ptr
        leaq    1(%rdi), %rax
        pushq   %rbx
        subq    $8, %rsp
        cmpq    $1, %rax
        jbe     .L4
        movq    %rdi, %rbx
        xorl    %ebp, %ebp
.L3:
        movq    %rbx, %rdi
        sarq    %rdi
        xorq    %rbx, %rdi
        sarq    $2, %rbx
        call    f
        movzbl  %al, %eax
        addl    %eax, %ebp
        leaq    1(%rbx), %rax
        cmpq    $1, %rax
        ja      .L3
.L6:
        addq    $8, %rsp
        movl    %ebp, %eax
        popq    %rbx
        popq    %rbp
        ret
.L4:
        xorl    %ebp, %ebp
        jmp     .L6
```

Hint: 'sarq %rdi' is equivalent to 'sarq $1, %rdi'.

This code is covered in questions 6 through 10 on the last page.

[this column is intentionally blank]

1 (18 minutes). Each assembly-language function (L1 through L9) corresponds to a C function (a through j). Write the letter of the C function next to the corresponding assembly-language function. Since there are ten C functions and nine assembly-language functions, one C function should be unused.

2 (5 minutes). Give assembly-language code for the C function that was unused in the previous question.

3 (10 minutes). The source code contains 3 instances of '<< 2' and 7 instances of '>> 2'. However, the assembly-language code contains no instances of 'sall $2,...' and only 5 instances of 'sarl $2,...'. Explain why the compiler can omit each "missing" shift instruction.

4 (10 minutes). Suppose we replace each 32-bit instruction in the above assembly-language code by the corresponding 64-bit instruction. For example, we replace "movl %esi, %eax" by "movq %rsi, %rax" and we replace "sarl %cl, %eax" by "sarq %cl, %rax". What correctness bugs, if any, would this introduce in L1 through L9 as implementations of their corresponding C functions? Briefly explain. Do not worry about performance.

5 (6 minutes). Explain why the following code is not a valid alternate implementation for the C function that matches L8:

```
L8-wrong:
        leal    2(%rdi), %ecx
        movl    %esi, %eax
        sarl    %cl, %eax
        ret
```

6a (2 minutes): Explain from the caller's point of view what F does.

6b (4 minutes): Give C source code that corresponds to F. Briefly justify the types that you use.

7a (5 minutes): Suppose we remove all the pushq and popq instructions from G. Explain what (if anything) could go wrong, from the point of view of G's caller.

7b (10 minutes): Suppose we remove the addq and subq instructions from G. Explain what (if anything) could go wrong, from the point of view of G's caller, and explain whether your answer depends on the internal behavior of F.

8 (12 minutes): Speed up G by inlining the body of F; that is, assume the internal behavior of F and use this assumption to minimize the number of instructions executed by G. Your modified version of G should not call F.

Suppose you are working for a startup called Reduced Intel, which has licensed the x86-64 architecture (and the name "Intel"!) from Intel, and which is building processors that are simpler and faster than x86-64 processors.

9 (6 minutes): You are considering removing the 'call' instruction. Rewrite G so that it calls F according to the usual x86-64 calling conventions, but does not use the 'call' instruction. Do not assume anything about F's internal behavior.

10 (12 minutes): You are also considering removing all jump instructions; that is, you will remove unconditional jumps 'jmp' and all conditional jumps like 'jbe'. Rewrite G so that it does not use any jump or call instructions. As before, your rewrite should not assume anything about F's internal behavior.

Tyler Carson

(2)

```
movl %edi %eax      ; eax = m
movl %esi %ecx      ; ecx = n
sall %ecx %eax      ; shift m << n
sarl $2 %eax
ret
```

%cl

(3) sall $2, ... instructions can be replaced by an instruction leal, w/j which is faster and more efficient than an arithmetic shift operation. In cases of sarl $2 being ommitted, the compiler can often optimize away instructions by reordering instructions to achieve the same effect as the higher level code.

h + j?

(4) In functions L4 and L5, (corresponding to b and a), the optimizations would no longer have the desired effect. L4 performs only one right shift of 31 bits. this is sufficient for completely extending the MSBit in a 32-bit reg, but not for a 64-bit register. L5 sets the return value to 0 with XOR, which is a sufficient optimization for (m<<31<<2), given m has 32 bits, but does not necessarily return the correct result if m has 64 bits.

4

(5) $\overset{m}{rdi+2}$ into ecx
$\overset{n}{esi}$ into eax
shift n by (m+2)

} assembly psuedo

- 2(rdi) produces m+2. this means that the end result is something like $n >> m << 1$, which is in equivalent to the function produced by L8

the original function L8 first shifts the value of n by 2, then shifts it by m.

5

6a

```
movl  %edi, %eax   ; arg 1 to ret-reg
andl  $ , %eax     ; arg1 & 1
ret
```

2

this function, from the caller's perspective, returns 1 if the input value is odd, and 0 otherwise

6b

```
int F(int x){
    return x & 1;
}
```

4

I chose type int because the assembly operations are suffixed with l, and take place in 32-bit registers. Int is a 32-bit type, so the assembly generated by my function should have the same suffixes and registers as the provided code.

(7a) - base ptr not saved

- rbx not saved (lost ∵ callee saved)

-

From the caller's perspective, the function G leaves a lot of problems with the stack. first: the base pointer, %rbp, is not saved (required of a called function). this means that once the function G returns the stack frame for the caller will be incorrect. 2nd: the value in %rbx is lost during execution of G. %rbx is a callee save register, meaning that the caller relies on the callee to preserve the value of %rbx by pushing / popping w/ the stack.

(7b) by erasing the addq / subq instructions from the function G, we inadvertently remove all arithmetic involving the stack pointer. The function should have completely ill-defined behavior when it tries to use local variables on the stack. The function should be pretty broken whether or not it depends on the internal behavior of f

8

```
G:      pushq   %rbp
        leaq    1(%rdi) %rax
        pushq   %rbx
        subq    $12, rsp        ; allocate enough space for
                                  extra int on stack
        cmpq    $1 , rax
        jbe     .L4
        movq    %rdi , %rbx
        xorl    %ebp  %ebp
.L3:
        movq    %rbx  %rdi
        sarq    % rdi
        xorq    %rbx   %rdi
        sarq    $2 , %rbx
        movl    8(rsp), %edx    ; grab value for F function
                                  from where I hope it is
        andl    $1  , %edx
        movl    %edx , %rax  0  ; move the resulting value
        movzbl  %al , %eax        into %rax, where it would
        addl    %eax, %ebp        otherwise be located
        leaq    1(%rbx), %rax
        cmpq    $1 , %rax
        ja      .L3
.L6:
        addq    $8 , %rsp
        movl    %ebp , %eax
        popq    %rbx
        popq    %rbp
        ret
.L4:
        xorl    %ebp, %ebp
        jmp     .L6
```

9

G: { Same from line 1 → .L3

.L3 movq %rbx %rdi
sarq %rdi
xorq %rbx %rdi
sarq $7 %rbx

movzbl %al %eax
addl %eax %ebp
leaq 1(%rbx), %rax
cmpq $1, %rax
ja L3

{ same 'til end

Idea is perform all necessary caller saves, then push the return address on top of the parameters on the stack, then jump to ~~the location~~ of F

1

(10)

In place of the call operation, the program would save all caller-save registers appropriately, then push the return address onto the stack, then set the instruction pointer %rip to the location of F using perhaps a leaq instruction. Once F was done executing, the stack should have the return-address for the G function, and the program should be able to continue normally.

This same technique of manipulating the instruction pointer manually would also have to be used to replace the conditional jumps through-out G.

8