UCLA Computer Science 33 (Spring 2017)
Midterm 2, 100 minutes, 100 points, open book, open notes.
Put your answers on the exam, and put your name and student ID at the top of each page.

TA: Shikhar Malhotra

Name: Jonathan Chu Student ID: 004832220

1 a-c	1 d-e	2 a	2 b	2 c	3	4	5 a-c	5 d	total

1a (6 minutes). Write two C functions with the following APIs:

```
unsigned f2u (float x);
float u2f (unsigned y);
```

f2u(X) should yield an unsigned integer Y that has the same 32-bit representation as X. For example, f2u(-0.1f) should yield 0xbdcccccd, because -0.1f has a sign bit 1, an exponent field 0x7b, and a fraction field 0x4ccccd, and ((1u<<31) | (0x7b<<23) | 0x4ccccd) == 0xbdcccccd. The u2f function should be the reverse operation, i.e., f2u(u2f(Y)) == Y should be true for all unsigned values Y.

```
unsigned f2u (float x) {
  union {
    unsigned u;
    float f;
  } un;

  un.f = x;
  return un.u;
}
```

```
float u2f (unsigned y) {
  union {
    unsigned u;
    float f;
  } un2;

  un2.u = y;
  return un2.f;
}
```

+6

6

1b (3 minutes). Does the C expression u2f(f2u(X)) == X yield 1 for all float values X? If so, briefly justify why; if not, give a counterexample.

Yes it does because no values of X will cause u2f or f2u to work improperly. NaNs and infinity will still have their bits preserved, and the expression will yield true.

0

1c (3 minutes). Give two unsigned values Y1 and Y2 such that the C expression (Y1 != Y2 && u2f(Y1) == u2f(Y2)) yields 1.

0

Y1's bit representation: 01111111100000000000000000000000

Y2's bit representation: 01111111000000000000000000000000

Y1 and Y2 will both be NaN's, but with different bit representations.

Name: Jonathan Chen No. 487 Student ID: 004832220

1d (4 minutes). Which of the following machine-language functions, if any, are plausible x86-64 implementations of f2u and of u2f, respectively?

8

A:
```
movl  %edi, -4(%rsp)
movss -4(%rsp), %xmm0
ret
```
u2f

B:
```
movl  %edi, 4(%rsp)
movss 4(%rsp), %xmm0
ret
```

C:
```
movss %xmm0, -4(%rsp)
movl  -4(%rsp), %eax
ret
```
f2u

D:
```
pxor      %xmm0, %xmm0
movl      %edi, %edi
cvtsi2ssq %rdi, %xmm0
ret
```

2

E:
```
cvttss2siq %xmm0, %rax
ret
```

F:
```
movd %xmm0, %eax
ret
```

1e (9 minutes). For each machine-language function (A)-(F) that is not a valid implementation of either u2f or f2u, explain why not. If there is some other C-language function that the machine-language function is a valid implementation of, give such a function; if not, explain why not.

6

B overwrites a value above the stack pointer, potentially overwriting important information. ✓

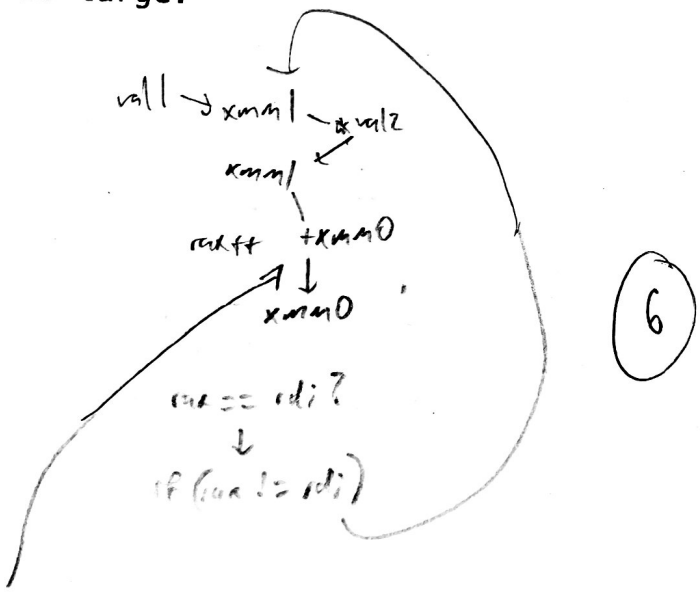
D and E both have correct instructions which will not work because we are trying to preserve the bits, not the value.

Name: Jonathan Cho Student ID: 004832220

Consider the following x86-64 function foo:

```
foo:
        testq %rdi, %rdi
        je    .L4
        xorpd %xmm0, %xmm0
        xorl  %eax, %eax
.L3:
        movsd (%rsi,%rax,8), %xmm1
        mulsd (%rdx,%rax,8), %xmm1
        incq  %rax
        cmpq  %rax, %rdi
        addsd %xmm1, %xmm0
        jne   .L3
        ret
.L4:
        xorpd %xmm0, %xmm0
        ret
```

2a (11 minutes). Construct a data-flow representation of the micro-operations for the inner loop of the function foo. Identify any critical paths that are likely to be a performance bottleneck when %rdi is large.



6

bottleneck because xmm1's value needs to be updated before the instructions movsd and mulsd can be executed, since the value in xmm1 is used for those instructions. This disallows parallelism in that part of the program.

Name: Jonathan Chu Student ID: 99[illegible]

2b (11 minutes). Suppose the function foo is executing on a Kaby Lake processor. Kaby Lake processors have the same set of functional units as the Haswell. How well will the inner loop be parallelized on this processor, using instruction-level paralellism? Briefly justify your answer by appealing to your answer to the previous subquestion.

Parallelism between iterations of the loop will not occur very well because the value in xmm1 is used towards the end of the loop and at the beginning of the loop. Thus, the operations at the beginning of the loop body cannot be executed in parallel with the operations at the beginning because the value of xmm1 cannot be changed while it is being used at the end of the previous loop iteration.

Otherwise, the instructions in the loop can run in parallel more effectively since they are separate operations involving separate values.

0

3/6

Name: [illegible] Student ID: [illegible]

2c (11 minutes). Suppose you have several chips that implement the same x86-64 instruction set. Each chip has just one cache for RAM. The cheapest chip uses a direct-mapped cache; the next-cheapest one uses a 2-way set-associative cache; the next-cheapest one a 4-way set-associative cache, and so on for 8-way and 16-way. All the caches use a 64-byte cache line and they all contain 1 MiB of data. If you are interested in executing the function foo on many small arrays (with typical size 256 bytes), which of these chips will you be most interested in buying, if you want to maximize the bang for your buck? Briefly justify your answer.

(5)

We would want to buy the chip with the smallest set but still has enough memory to hold a 256 byte array.

$$\frac{256 \text{ bytes}}{64 \text{ bytes}} = 4$$

We would want a 4-way set-associative cache because its cache sets have 4*64 = 256 bytes, just enough to each hold a small array. This option would minimize wasted cache space and maximize efficiency retrieving cache values.

2-way for two arrays of Foo

Name: Jonathan Chu Student ID: 004832220

3 (11 minutes). For each of the following forms of machine parallelism, give an example of an application that will likely work better with this form of parallelism than with any of the other forms listed. Briefly justify your answers.

1) instruction level parallelism
2) multiplexing
3) process parallelism
4) SIMD
5) thread parallelism

10

1) ILP
ILP would work with an application with few conditional branches and loops because processors would not have to run the risk of incorrect predictions/branches chosen, eliminating mispredictions in ILP.

2) Multiplexing
An application that must interact through different types of events, say through both user input and interaction with other applications or the network, would run best with multiplexing because multiplexing would allow for efficient responses to different types of unpredictable events.

3) Process
Process parallelism would be effective for an application that frequently receives network requests, multiple at once, because child processes could accommodate each request.
2

4) SIMD
An application that has large data structures, such as arrays, in which several consecutive values must be modified would run best with SIMD because SIMD allows multiple values to be changed in one instruction instead of many

5) Thread
Thread parallelism would be most effective for an application with several separate but small, light operations to be executed because separate threads could be used to concurrently execute them.
2

specific examples?

Name: Jonathan Chen Student ID: [illegible]

9

4 (10 minutes). Suppose you are designing a computer with two caches: a smaller L1 cache (one per core), and a larger L2 cache (shared among all the cores). You are trying to decide whether the caches should be *exclusive*, i.e., a data word is never in both L1 and L2, or *strictly inclusive*, i.e., every data word cached in L1 is also cached in L2. Give pros and cons of both approaches.

exclusive:

pros:

- would not waste time checking for duplicates between caches
- ✓ would use less total space

cons:

- If a frequently used value is already in L2, we can't put it in L1 to speed up its read/write ✓
- may have to search both caches for a value; waste of time

strictly inclusive:

pros:

- ✓ all values are backed up in L2
- never have to check if value is already stored before storing in L2

cons:

- ✓ we are wasting the space in L2
- wasting time writing to L2 when just L1 is enough in many cases

Name: Jonathan Cho Student ID: 00858220

5. Consider the following assembly-language program XYZ. The 'puts' function is a standard function declared in <stdio.h> with signature 'int puts(const char *s);'; it outputs its argument string to standard output with a newline appended, and returns a nonnegative integer on success and a negative integer on failure.

```
 1    f:
 2          subq    $8, %rsp
 3          call    puts
 4          xorl    %eax, %eax
 5          addq    $8, %rsp
 6          ret
 7          .globl  main
 8    main:
 9          pushq   %rbx
10          leaq    8(%rsi), %rbx
11          subq    $16, %rsp
12          jmp     .L10
13    .L6:
14          xorl    %esi, %esi
15          addq    $8, %rbx
16          movl    $f, %edx
17          movq    %rsp, %rdi
18          call    pthread_create
19    .L10:
20          movq    (%rbx), %rcx
21          testq   %rcx, %rcx
22          jne     .L6
23          addq    $16, %rsp
24          xorl    %eax, %eax
25          popq    %rbx
26          ret
```

5

5a (2 minutes). What would go wrong if we deleted lines 9 and 25?

5b (3 minutes). What would go wrong if we deleted lines 2 and 5?

5c (5 minutes). This machine code has a bug. What is it?

a) rbx would not live on the stack, so its value would not be saved. 2

b) Whatever was in the bottom 8 bits of the stack would be overwritten and lost. 2

c) There is a return in line 6 that would cause the program to end before executing what it needs to. 1

Name: Jonathan Chan Student ID: [illegible]

5d (11 minutes). Translate program XYZ to the equivalent C code. Your C program should have the same bug that the machine code does.

```
#include "stdio.h"

void f(void) {
    char s;
    printf("%p\n", &s);
}

int main(int a) {
    f(); return 0;
    int *b = &(a+2);
    int c, d;

    while (c != 0) {
        b += 8;
        d = 15;
        pthread_create;
        c = b;
    }
    return 0;
}
```

(3)