6.172 Fall 2014

Quiz 1

Given: In class on Thursday, October 9, 2014

Name: _____________________________________________

CSAIL username: ____________________________________

Instructions

• DO NOT open this quiz booklet until you are instructed to do so.
• This quiz booklet contains 14 pages, including the x86 quick reference at the end.
• You have 80 minutes to earn 100 points.
• This quiz is closed book, but you may use one handwritten, double-sided 8 1/2” ×11” crib sheet and the Master Theorem Cheat Sheet.
• When the quiz begins, please write your name and username on this coversheet, and write your name on the top of each page, since the pages may be separated for grading.
• Wrong answers will be penalized on the true/false questions, and so do not guess unless you are reasonably sure.
• Good luck!

<table>
<thead>
<tr>
<th>Question</th>
<th>Parts</th>
<th>Points</th>
<th>Score</th>
<th>Grader</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Name on all pages</td>
<td>14</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Bit hacks</td>
<td>4</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Find all</td>
<td>3</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Find one</td>
<td>2</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Parallel apply</td>
<td>4</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Parallel symmetric force</td>
<td>2</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. C to Assembly</td>
<td>4</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Section 1: Bithacks

Question 1.1 For each of the assertions in the following the code, circle T if the assertion would always succeed, and circle F if the assertion might fail. Remember, wrong answers will be penalized, and so do not guess unless you are reasonably sure.

```c
void bithack(int64_t x) {
    int64_t cmp = x < 0;
    int64_t r = (x ^ -cmp) + cmp;

    assert(r == -x); // A. T F
    assert(r == max(x, -x)); // B. T F
    assert(r == min(x, -x)); // C. T F
    assert(r == (1 - x)); // D. T F
}
```

Question 1.2 For each of the assertions in the following the code, circle T if the assertion would always succeed, and circle F if the assertion might fail.

```c
void bithack(uint64_t x) {
    uint64_t r;
    r = x | (x >> 1);
    r |= r >> 2;
    r |= r >> 4;
    r |= r >> 8;
    r |= r >> 16;
    r |= r >> 32;
    r -= r >> 1;

    assert(r >= x); // A. T F
    assert(r <= x); // B. T F
    assert(r >= x/2); // C. T F
    assert(r <= x/2); // D. T F
}
```
**Question 1.3** For each of the assertions in the following the code, circle T if the assertion would always succeed, and circle F if the assertion might fail.

```c
void bithack(uint64_t x) {
    const MASK = 0x00000000FFFFFFFFUL;
    uint64_t r = x;
    r ^= r >> 32;
    r ^= r << 32;
    r ^= r >> 32;

    assert((r & MASK) == (x & MASK)); // A. T F
    assert((r & MASK) == ((x & ~MASK) >> 32)); // B. T F
    assert(((r & ~MASK) >> 32) == (x & MASK)); // C. T F
    assert((r & ~MASK) == (x & ~MASK)); // D. T F
}
```

**Question 1.4** This bit trick resembles the bit trick from lecture for computing $2^\lceil \log n \rceil$, but all right shifts in that trick have been replaced with rightward bit rotations. For example, 0xDEADBEEF >> 12 yields 0x000DEADB, and rightrotate(0xDEADBEEF, 12) produces 0xEEFDEADB. For each of the assertions in the following the code, circle T if the assertion would always succeed, and circle F if the assertion might fail.

```c
void bithack(uint64_t x) {
    uint64_t r = x-1;
    r |= rightrotate(r, 1);
    r |= rightrotate(r, 2);
    r |= rightrotate(r, 4);
    r |= rightrotate(r, 8);
    r |= rightrotate(r, 16);
    r |= rightrotate(r, 32);
    r++;

    assert(r < 0); // A. T F
    assert(r < 1); // B. T F
    assert(r < 2); // C. T F
    assert(r < 4); // D. T F
}
```
Section 2: Find All

Question 2.1 Consider the following snippet of code that traverses a DAG (directed acyclic graph) and marks each node as visited in parallel:

```c
void visit_dag (struct Node node) {
    if (node.visited)
        return;
    node.visited = 1;
    for (child = node.first; child = child->next; child != NULL)
        cilk_spawn visit_dag(child);
}
```

What will happen when this program is executed? For each statement below, circle T if the statement is true and F if the statement is false.

A) T F The program will deadlock.
B) T F The program has a race condition.
C) T F The program will execute infinitely without termination.
D) T F The program will execute without any problem.

Question 2.2 Consider the following compiler transformation of a loop. For each statement below, circle T if the statement is true and F if the statement is false.

```c
int X[N];
for (int i = 0; i < N; i++) {
    X[i] = 41*i + 15;
}
```

```c
int X[N];
int v = 15;
for (int i = 0; i < N; i++) {
    X[i] = v;
    v = v + 41;
}
```

This transformation...

A) T F ...should be performed whenever possible, as it is always a win.
B) T F ...may lead to slowdowns due to increased register pressure.
C) T F ...may stop parallelization due to a loop-carried dependence.
D) T F ...may lead to a slowdown due to the extra assignment.
**Question 2.3** For each statement below, circle T if the statement is true and F if the statement is false.

An out-of-order superscalar processor can…

A) T F …eliminate the penalty of the Read-After-Write (RAW) dependences by using a write buffer.

B) T F …eliminate the penalty of the Write-After-Read (WAR) dependence by register renaming.

C) T F …eliminate the penalty of the control dependences by branch prediction and speculation.

D) T F …eliminate instructions with unused results by dead code elimination.

### Section 3: Find one

**Question 3.1** In order to evaluate the performance of multiple computers, you are asked to select a benchmark suite and a test harness. You are given the user-time measurements of running four benchmarks with its test harness on one machine. Each row in the table below provides the data obtained for one test.

<table>
<thead>
<tr>
<th></th>
<th>10.8</th>
<th>10.8</th>
<th>22.4</th>
<th>10.9</th>
<th>10.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mark A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mark B</td>
<td>8.5</td>
<td>7.3</td>
<td>6.8</td>
<td>9.6</td>
<td>8.2</td>
</tr>
<tr>
<td>Mark C</td>
<td>0.1</td>
<td>0.3</td>
<td>0.2</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>Mark D</td>
<td>8.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Your should choose the benchmark… (select only one)

A) …Mark A, because it has the smallest variation after eliminating outliers.

B) …Mark B, because it has only a small variation.

C) …Mark C, because it can run quickly, as each run only takes a little time.

D) …Mark D, because it is the most accurate with zero variation.

E) You should choose no benchmark, because user time does not correlate to performance.
Question 3.2 A program execution can be represented by a dag in which each node is a program statement and an arrow between two nodes indicates sequential dependence. If there are no paths between two nodes, the nodes can run in parallel. Which of the four programs below leads to the following dag? (Circle the letter of the most accurate answer.)

A. int fib(x) {
    if(x < 2)
        return x;
    int a = cilk_spawn fib(x-1);
    cilk_sync;
    int b = fib(x-2);
    return a + b;
}

B. int fib(x) {
    if(x < 2)
        return x;
    int a = cilk_spawn fib(x-1);
    int b = fib(x-2);
    cilk_sync;
    return a + b;
}

C. int fib(x) {
    if(x < 2)
        return x;
    int a = fib(x-1);
    int b = cilk_spawn fib(x-2);
    cilk_sync;
    return a + b;
}

D. int fib(x) {
    if(x < 2)
        return x;
    int a = cilk_spawn fib(x-1);
    int b = cilk_spawn fib(x-2);
    cilk_sync;
    return a + b;
}

E. None of the above.
Section 4: Parallel apply

Consider the following pseudocode Cilk snippet that applies a serial function \( f \) to every node in complete 4-ary tree:

```c
void apply(func_ptr f, Struct node x) {
    f(x);
    if (is_leaf(x)) return;
    cilk_spawn apply(f, x.child[0]);
    cilk_spawn apply(f, x.child[1]);
    cilk_spawn apply(f, x.child[2]);
    cilk_spawn apply(f, x.child[3]);
    cilk_sync;
}
```

**Question 4.1** Write a recurrence for the work \( T_1(n) \) of `apply()` in terms of \( n \), the number of nodes in the tree, and \( F(n) \), the running time of \( f(n) \).

**Question 4.2** Similarly, write a recurrence for the span \( T_\infty(n) \) of `apply()`.

**Question 4.3** If \( F(n) = O(1) \), what is the work of `apply()` in terms of \( n \)? What is the span? What is the parallelism?

Work = 

Span = 

Parallelism = 
Question 4.4 If $F(n) = \Theta(n)$, what are the work, span, and parallelism?

Work = ________________________________

Span = ________________________________

Parallelism = __________________________
Section 5: Parallel symmetric force

Consider the following parallel code for simulating the gravitational interaction between massive objects in 1 dimension.

const double G = 6.674E-11; // Gravitational constant

typedef struct {
    double mass; // Object mass
    double x; // Object position
    double f; // Force on object
} object_t;

// Returns the gravitational force on a due to b.
double force(const object_t a, const object_t b) {
    double f = G * a.mass * b.mass / ((a.x - b.x) * (a.x - b.x));
    if (a.x > b.x) return -f;
    return f;
}

// This function computes forces on all objects
void calc_all_forces(object_t A[], int n) {
    cilk_for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i != j) {
                A[i].f += force(A[i], A[j]);
            }
        }
    }
}

Question 5.1 What are the work, span, and parallelism of the above code?

Work = ________________________________________________

Span = ________________________________________________

Parallelism = __________________________________________
Question 5.2 Now let’s optimize the update_objects() function. Because $\text{force}(a,b) = -\text{force}(b,a)$, we need only compute force() once per pair. Use that observation to fill in the blanks in the following code.

cilk_for (int i = 0; i < n; ++i) {

  for (int j = ______; j < ______; ++j) {

    ______________________________

    ______________________________

    ______________________________

    }
Section 6: C to Assembly
Below are four descriptions and four snippets of assembly (A to D, in the next two pages). Match the descriptions to the assembly by writing which letter the assembly corresponds to.

Question 6.1 Transpose a matrix. (select one)
A) Code snippet A
B) Code Snippet B
C) Code Snippet C
D) Code Snipped D
E) None of the code snippets

Question 6.2 Compute the $i^{th}$ Fibonacci element and insert it into position $(i \pmod{128}, i/128)$ in a matrix, where the $0^{th}$ element is 0, the $1^{st}$ is 1, the $2^{nd}$ is 1, .... (select one)
A) Code snippet A
B) Code Snippet B
C) Code Snippet C
D) Code Snipped D
E) None of the code snippets

Question 6.3 Perform matrix multiplication modulo some value. (select one)
A) Code snippet A
B) Code Snippet B
C) Code Snippet C
D) Code Snipped D
E) None of the code snippets

Question 6.4 Compute the sum of squares of each element in the matrix, modulo some value. (select one)
A) Code snippet A
B) Code Snippet B
C) Code Snippet C
D) Code Snipped D
E) None of the code snippets
A:
xorl %esi, %esi
xorl %edx, %edx
.L1:
xorl %ecx, %ecx
.L2:
    movq C(%rsi,%rcx,8), %rax
    addq $1, %rcx
    imulq %rax, %rax
    addq %rax, %rdx
    andl $1023, %edx
    cmpq $128, %rcx
    jne .L2
    addq $1024, %rsi
    cmpq $131072, %rsi
    jne .L1

B:
xorl %edi, %edi
.L1:
xorl %r8d, %r8d
.L2:
    movq C(%rdi,%r8), %rdx
    leaq B(%r8), %rsi
    xorl %eax, %eax
.L3:
    movq (%rsi), %rcx
    addq $1024, %rsi
    imulq A(%rdi,%rax,8), %rcx
    addq $1, %rax
    addq %rcx, %rdx
    andl $1023, %edx
    cmpq $128, %rax
    jne .L3
    movq %rdx, C(%rdi,%r8)
    addq $8, %r8
    cmpq $1024, %r8
    jne .L2
    addq $1024, %rdi
    cmpq $131072, %rdi
    jne .L1
C:

cmovq $0, B(%rip)
cmovq $1, B+8(%rip)
cmovl $2, %eax
cmovl $1, %edx
cxorl %ecx, %ecx
cjmp .L2
.L1:
cmovq %rcx, %rdx
cmovq %rsi, %rcx
.L2:
pleaq (%rdx,%rcx), %rsi
cmovq %rax, %rdx
cmovq %rax, %rdi
cand $127, %edx

.D:
cshrq $7, %rdi
caddq $1, %rax
csalq $7, %rdx
caddq %rdi, %rdx
ccmpq $16384, %rax

movq %rsi, B(,%rdx,8)
cjne .L1

D:
cmovl $A, %r10d
cmovl $1, %r8d
cmovq %r10, %rdi
cmovq %r10, %r9
.L9:
ccmpq $1, %r8
cje .L12

movq %r9, %rdx
cmovq %r10, %rax
.L13:
cmovq (%rdx), %rsi
cmovq (%rax), %rcx
caddq $8, %rax
caddq $1024, %rdx

cmovq %rsi, -8(%rax)
cmovq %rcx, -1024(%rdx)
cmpq %rdi, %rax

.jne .L13
cmpq $128, %r8
cje .L8
.L12:
caddq $1, %r8
caddq $1024, %r10
caddq $8, %r9
caddq $1032, %rdi
cjmp .L9

.L8:
# Intel x86 Assembly Language Cheat Sheet

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Copying Data</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mov src, dest</td>
<td>Copy src to dest</td>
<td>mov $10, %eax</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movw %eax, (2000)</td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add src, dest</td>
<td>dest = dest + src</td>
<td>add $10, %esi</td>
</tr>
<tr>
<td>sub src, dest</td>
<td>dest = dest – src</td>
<td>sub %eax, %ebx</td>
</tr>
<tr>
<td>mul reg</td>
<td>edx:eax = eax * reg</td>
<td>mul %esi</td>
</tr>
<tr>
<td>div reg</td>
<td>edx = edx:eax mod reg</td>
<td>div %edi</td>
</tr>
<tr>
<td></td>
<td>eax = edx:eax – reg</td>
<td></td>
</tr>
<tr>
<td>inc dest</td>
<td>Increment destination</td>
<td>inc %eax</td>
</tr>
<tr>
<td>dec dest</td>
<td>Decrement destination</td>
<td>dec (0x1000)</td>
</tr>
<tr>
<td><strong>Function Calls</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call label</td>
<td>Push eip, transfer control</td>
<td>call format_disk</td>
</tr>
<tr>
<td>ret</td>
<td>Pop eip and return</td>
<td>ret</td>
</tr>
<tr>
<td>push item</td>
<td>Push item (constant or register) to stack</td>
<td>pushl $32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push %eax</td>
</tr>
<tr>
<td>pop [reg]</td>
<td>Pop item from stack; optionally store to register</td>
<td>pop %eax</td>
</tr>
<tr>
<td></td>
<td></td>
<td>popl</td>
</tr>
<tr>
<td><strong>Bitwise Operations</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and src, dest</td>
<td>dest = src &amp; dest</td>
<td>and %ebx, %eax</td>
</tr>
<tr>
<td>or src, dest</td>
<td>dest = src</td>
<td>dest</td>
</tr>
<tr>
<td>xor src, dest</td>
<td>dest = src ^ dest</td>
<td>xor $0xffffffff, %ebx</td>
</tr>
<tr>
<td>shl count, dest</td>
<td>dest = dest &lt;&lt; count</td>
<td>shl $2, %eax</td>
</tr>
<tr>
<td>shr count, dest</td>
<td>dest = dest &gt;&gt; count</td>
<td>shr $4, (%eax)</td>
</tr>
<tr>
<td>sal count, dest</td>
<td>same as shl, but shifted bits will be the sign bit</td>
<td></td>
</tr>
<tr>
<td><strong>Conditionals and Jumps</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp arg1, arg2</td>
<td>Compare arg1 to arg2; must immediately precede any of the conditional jump instructions</td>
<td>cmp $0, %eax</td>
</tr>
<tr>
<td>je label</td>
<td>Jump to label if arg1 == arg2</td>
<td>je endloop</td>
</tr>
<tr>
<td>jne label</td>
<td>Jump to label if arg1 != arg2</td>
<td>jne loopstart</td>
</tr>
<tr>
<td>jg label</td>
<td>Jump to label if arg2 &gt; arg1</td>
<td>jg exit</td>
</tr>
<tr>
<td>jge label</td>
<td>Jump to label if arg2 &gt;= arg1</td>
<td>jge format_disk</td>
</tr>
<tr>
<td>jl label</td>
<td>Jump to label if arg2 &lt; arg1</td>
<td>jl error</td>
</tr>
<tr>
<td>jle label</td>
<td>Jump to label if arg2 &lt;= arg1</td>
<td>jle finish</td>
</tr>
<tr>
<td>test reg, imm</td>
<td>Bitwise compare of register and constant; must immediately precede the jz or jnz instructions</td>
<td>test $0xffff, %eax</td>
</tr>
<tr>
<td>jz label</td>
<td>Jump to label if bits were not set (&quot;zero&quot;)</td>
<td>jz looparound</td>
</tr>
<tr>
<td>jnz label</td>
<td>Jump to label if bits were set (&quot;not zero&quot;)</td>
<td>jnz error</td>
</tr>
<tr>
<td>jmp label</td>
<td>Unconditional relative jump</td>
<td>jmp exit</td>
</tr>
<tr>
<td>jmp *reg</td>
<td>Unconditional absolute jump; arg is a register</td>
<td>jmp %eax</td>
</tr>
<tr>
<td>jmp segment, offs</td>
<td>Unconditional absolute far jump</td>
<td>jmp $0x10,$0</td>
</tr>
<tr>
<td><strong>Miscellaneous</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lea addr, dest</td>
<td>move the address calculated to the dest</td>
<td>lea 23(%eax, %ebx, 8), %ecx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arguments to instructions: Note that it is not possible for both src and dest to be memory addresses.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant (decimal or hex):</td>
<td>$10 or $0xff</td>
<td>Fixed address:</td>
</tr>
<tr>
<td>Register: %eax</td>
<td>%ebx</td>
<td>%ecx</td>
</tr>
<tr>
<td>Dynamic address:</td>
<td>%eax</td>
<td>or</td>
</tr>
<tr>
<td>Base indexed scale displacement</td>
<td>172(%rdi, %rdx, 8) = %rdi+8*%rdx+172</td>
<td></td>
</tr>
<tr>
<td>32-bit registers: %eax, %ebx, %ecx, %edx, %esi, %edi, %esp, %ebp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit registers: %ax, %bx, %cx, %dx, %si, %di, %esp, %ebp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-bit registers: %al, %ah, %bl, %bh, %cl, %ch, %dl, %dh</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>