Practice Quiz 4

Name: ____

Instructions

- DO NOT open this quiz booklet until you are instructed to do so.
- This quiz booklet contains 13 pages, including this one. You have 80 minutes to earn 80 points.
- This quiz is closed book, but you may use one handwritten, double-sided 8 1/2″ × 11″ crib sheet and the Master Method card handed out in lecture.
- When the quiz begins, please write your name on this coversheet, and write your name on the top of each page, since the pages may be separated for grading.
- Some of the questions are true/false, and some are multiple choice. You need not explain these answers unless you wish to receive partial credit if your answer is wrong. For these kinds of questions, **incorrect answers will be penalized, so do not guess unless you are reasonably sure**.
- Good luck!

Number	Question	Parts	Points	Score	Grader
0	Name on Every Page	13	2		
1	True or False	8	16		
2	Back Of The Envelope	3	10		
3	Branch Prediction	3	9		
4	Variable Byte Compression	2	11		
5	LLVM	6	12		
6	Algorithm Analysis	4	20		
	Total		80		

1 True or False (8 parts, 16 points)

Incorrect answers will be penalized, so do not guess unless you are reasonably sure. You need not justify your answer unless you want to leave open the possibility of receiving partial credit if your answer is wrong.

1.1

Because thermal limits prevented CPU manufacturers from increasing clock frequencies significantly, they began to produce multicore microprocessors.

True **False**

1.2

Inlining a function tends to improve performance by reducing the number of instruction cache misses.

True **False**

1.3

Suppose that a recursive function performs work on a problem of size N according to the recurrence $T(N) = 2T(N/3) + \Theta(N \lg N)$. Then coarsening the recursion is unlikely to significantly improve performance.

True **False**

1.4

Compilers eliminate data dependencies through register renaming by replacing x86 logical registers with physical registers.

True **False**

1.5

There can never be a true-, anti-, or output-data dependence between the following two lines of code:

```
movl %eax , 8(%esi)
lea 24(%esi, %edi, 8) , %ecx
```

True **False**

1.6

When using the MSI (modified, shared, invalid) cache coherence protocol, the state of a cache line in a processor's cache may transition directly from shared to invalid.

True **False**

1.7

If a memory location in a program is read by two logically parallel instructions, then a read-read determinacy race exists.

True **False**

1.8

A greedy scheduler schedules a computation with work T_1 and span T_∞ in time $T_p \leq \max(T_1/P, T_\infty)$ on a P-processor ideal parallel computer.

True **False**

2 Back Of The Envelope Calculations (3 Part, 10 Points)

Perform a back-of-the-envelope calculation for the work of a serial execution of `get_row_sums` on a 1000-by-1000 matrix. Assume that you have one 3 GHz scalar processing core which executes only one instruction per cycle, and that everything is in cache.

```
#define ROWS 1000
#define COLS 1000

typedef int32_t element_t;

void get_row_sums(element_t M[ROWS][COLS],
                  element_t row_sums[ROWS]) {
  for (int32_t i = 0; i < ROWS; i++) {
    element_t sum = 0;
    for (int32_t j = 0; j < COLS; j++) {
      sum += M[i][j];
    }
    row_sums[i] = sum;
  }
}
```

2.1

How long does `get_row_sums` take assuming that vectorization is disabled? Please circle the letter of your answer.

A 0.00001–0.001 seconds.
B 0.001–0.1 seconds.
C 0.1–10 seconds.
D 10–1000 seconds.
E None of the above.

2.2

After the code is compiled with vectorization enabled (with 128-bit vector registers), what is the performance of the program compared to the original program? Please circle the letter of your answer.

- **A** More than twice as slow as the code without vectorization.
- **B** Slower, but less than twice as slow as the code without vectorization.
- **C** About the same as the code without vectorization.
- **D** Faster, but less than twice as fast as the code without vectorization.
- **E** More than twice as fast as the code without vectorization.

2.3

With vectorization enabled (with 128-bit vector registers), what is the performance of the program if you change `element_t` from `int32_t` to `int8_t`? Please circle the letter of your answer.

- **A** More than twice as slow as the vectorized code with `int32_t`'s.
- **B** Slower, but less than twice as slow as the vectorized code with `int32_t`'s.
- **C** About the same as the vectorized code with `int32_t`'s.
- **D** Faster, but less than twice as fast as the vectorized code with `int32_t`'s.
- **E** More than twice as fast the vectorized code with `int32_t`'s.

3 Branch Prediction (3 parts, 9 points)

Using asymptotic Θ-notation, give the expected number of branch misses for the following sorting algorithm on an array of N distinct integers on (1) a sorted input (increasing order); (2) a reverse sorted input (decreasing order); and (3) a randomly ordered input.

```
void insertion_sort (int * A, int N) {
  for (int j = 0; j < N; j++) {
    int insert = A[j];
    int slot = j;
    while (slot > 0 && insert < A[slot-1]) {
      A[slot] = A[slot-1];
      slot--;
    }
    A[slot] = insert;
  }
}
```

3.1 Sorted

A $\Theta(1)$
B $\Theta(N)$
C $\Theta(N \lg N)$
D $\Theta(N^2)$
E None of the above.

3.2 Reverse Sorted

A $\Theta(1)$
B $\Theta(N)$
C $\Theta(N \lg N)$
D $\Theta(N^2)$
E None of the above.

3.3 Random Order

A $\Theta(1)$
B $\Theta(N)$
C $\Theta(N \lg N)$
D $\Theta(N^2)$
E None of the above.

4 Variable Byte Compression (2 parts, 11 points)

Byte codes are used as a way to compress sequences of positive integers of varying magnitudes. Each integer is represented as a series of bytes, where the most significant bit of a byte is called the ***continuation*** bit, and the remaining 7 bits are the ***payload***. To encode an integer, we take its binary representation ignoring leading zeros and group the remaining bits in 7-bit payloads. The remaining payloads are placed in bytes, with the least significant bits being in the first byte and the most significant bits being in the last byte. The continuation bit in each byte is set to `1` except for the last byte where it is set to `0`.

For example, to encode the integer 6172, we inspect its binary representation (without the leading `0`'s), which is `0b1100000011100`. We create two payloads, $\langle$`0011100`$\rangle$ and $\langle$`0110000`$\rangle$, and place them into bytes with the first byte's continuation bit set to `1` and the second byte's continuation bit set to `0`. The resulting bytes that encode the integer 6172 are `0b10011100` and `0b00110000`, where the first byte encodes the lower-order bits and the second byte encodes the higher-order bits.

4.1

Suppose that the sequence of bytes that resulted from encoding some integer x was `0b11001010`, `0b10001011`, and `0b00101100`. What is x in hexadecimal notation ignoring the leading zeros?

A `0x1645CA`

B `0x160BCA`

C `0x2C8BCA`

D `0xB05CA`

E None of the above.

4.2

The following program encodes an array `In` of N positive integers into a sequence of bytes, stored in the array `Out`. Assume that sufficient memory has been allocated for `Out`.

```
void encode(uint32_t* In, int32_t N, unsigned char* Out) {
   for(int32_t i=0; i < N; i++) {
     uint32_t x = In[i];
     while(x) {
       char byte = ______(A)_______;
       x = ______(B)_______;
       if(x)
         {______(C)_______;}
       *Out++ = byte;
     }
   }
}
```

For each blank in the code, write its label (A, B, or C) next to the expression that best fits. (*Hint:* Some blanks can take more than one expression, but only one is "best.")

____ `byte & 0x80`

____ `byte & 0x8`

____ `byte & ~0x80`

____ `byte &= 0x80`

____ `byte ^ 0x80`

____ `byte |= 0x80`

____ `byte`

____ `x >> 7`

____ `x & 0x7f`

____ `x & 0x7`

____ `x & 0x80`

____ `x << 7`

____ `x = x << 7`

____ `x = x >> 1`

____ `x >> 8`

____ `x ^ 0x7f`

____ `x ^ 0x80`

____ `x | 0x7f`

____ `x | 0x80`

____ `x++`

____ `x-7`

____ `x`

5 LLVM (6 parts, 12 points)

This question explores your understanding of control-flow graphs and Bentley optimizations. Consider the following complete C source file.

```c
int value1();
int value2();

__attribute__((const))
int bar(bool p) {
  int val;
  if (p)
    val = value1();
  else
    val = value2();
  return val;
}

void foo(int *restrict Y,
         const int *restrict X,
         int n, bool p) {
  for (int i = 0; i < n; ++i)
    Y[i] += bar(p) * X[i];
}
```

When compiling this C code, LLVM can perform optimizations involving the functions `foo` and `bar` defined in this file, but not on the functions `value1` and `value2`, which are only declared in this file. Suppose that LLVM compiles the following functions with the specified optimizations:

A The function `bar` with no optimization.

B The function `foo` with no optimization.

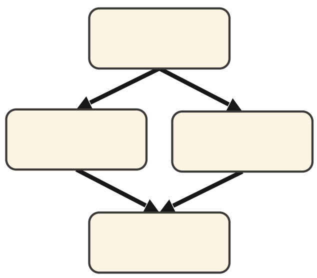
C The function `foo` with function inlining and no other optimizations.

D The function `foo` with loop unrolling and no other optimizations.

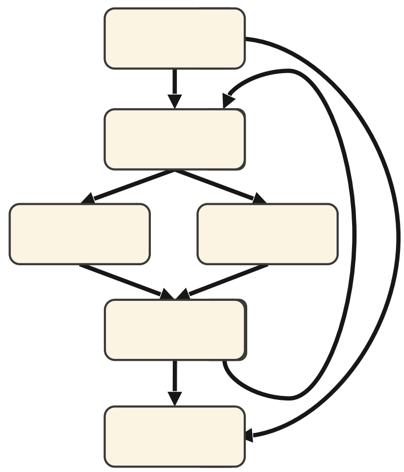
E The function `foo` with code hoisting followed by function inlining and no other optimizations.

The next page contains several pictures of control-flow graphs. For each control-flow graph, circle either the unique letter of the function-and-optimization scenario from above that it corresponds to, or circle “None” if it does not correspond to any of the scenarios. While not strictly necessary to solve this question, the LLVM IR is provided on Page 11 for your reference.

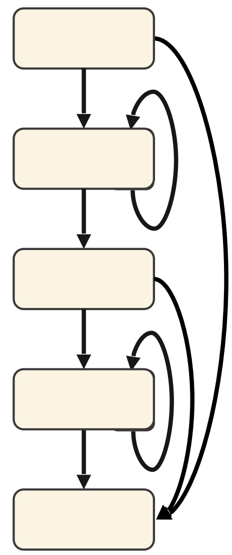
A B C D E None



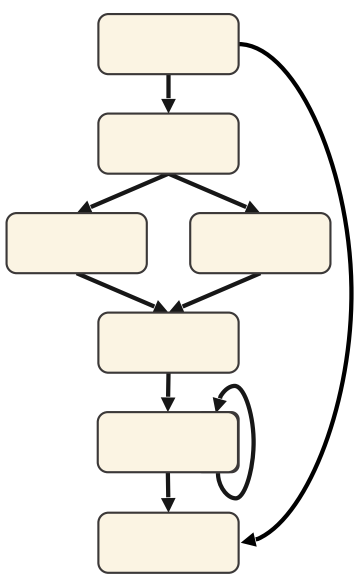
A B C D E None



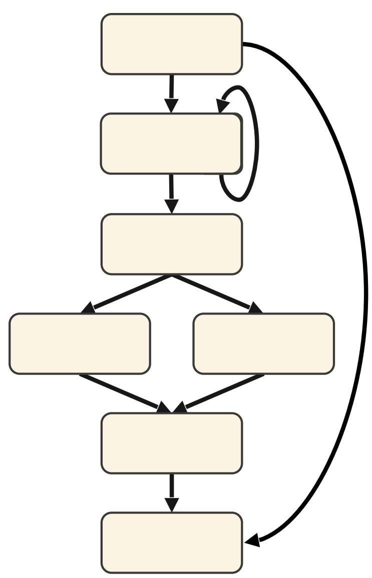
A B C D E None



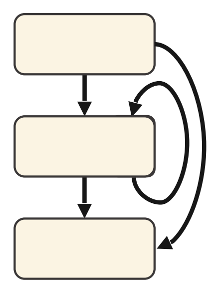
A B C D E None



A B C D E None



A B C D E None



Here is the LLVM IR for the two functions `bar` and `foo` without function inlining, loop unrolling, or code hoisting.

```
define i32 @bar(i1 zeroext) #0 {
  br i1 %0, label %2, label %4

; <label>:2:                                       ; preds = %1
  %3 = call i32 (...) @value1() #3
  br label %6

; <label>:4:                                       ; preds = %1
  %5 = call i32 (...) @value2() #3
  br label %6

; <label>:6:                                       ; preds = %4, %2
  %.0 = phi i32 [ %3, %2 ], [ %5, %4 ]
  ret i32 %.0
}

define void @foo(i32* noalias, i32* noalias, i32, i1 zeroext) #2 {
  %5 = icmp sgt i32 %2, 0
  br i1 %5, label %4, label %._ret_edge

; <label>:6:                                       ; preds = %4, %6
  %.01 = phi i32 [ 0, %4 ], [ %16, %6 ]
  %7 = call i32 @bar(i1 zeroext %3) #4
  %8 = sext i32 %.01 to i64
  %9 = getelementptr inbounds i32, i32* %1, i64 %8
  %10 = load i32, i32* %9, align 4
  %11 = mul nsw i32 %7, %10
  %12 = sext i32 %.01 to i64
  %13 = getelementptr inbounds i32, i32* %0, i64 %12
  %14 = load i32, i32* %13, align 4
  %15 = add nsw i32 %14, %11
  store i32 %15, i32* %13, align 4
  %16 = add nsw i32 %.01, 1
  %17 = icmp slt i32 %16, %2
  br i1 %17, label %6, label %._ret_edge

._ret_edge:                                       ; preds = %6, %4
  ret void
}
```

6 Algorithm Analysis (4 parts, 20 points)

Recall the prefix sum algorithm from Homework 5, and consider the following alternative parallel prefix sum algorithm, where A is the input array with N elements. The array element $A[N+1]$ is also allocated to store the total sum. Assume that N is an exact power of 2.

Note that you do not need to understand why the code is correct — just its structure.

```
void upsweep(int64_t* A, int64_t N) {
    for (int64_t d = 1; d < N; d*=2) {
        cilk_for (int64_t k = 0; k < N; k += 2*d) {
            A[k + 2*d - 1] = A[k + d - 1] + A[k + 2*d - 1];
        }
    }
}

void downsweep(int64_t* A, int64_t N) {
    A[N] = A[N - 1] //total sum
    A[N - 1] = 0;
    for (int64_t d = N / 2; d >= 1; d = d / 2) {
        cilk_for(int64_t i = 0; i < N; i += 2*d) {
            int64_t temp = A[i + d - 1];
            A[i + d - 1] = A[i + 2*d - 1];
            A[i + 2*d - 1] = temp + A[i + 2*d - 1];
        }
    }
}

void prefix_sum(int64_t* A, int64_t N) {
    upsweep(A, N);
    downsweep(A, N);
}
```

6.1

What is the work of `prefix_sum`? Give your answer in terms of N using Θ-notation.

6.2

What is the span of `prefix_sum`? Give your answer in terms of N using Θ-notation.

Consider the following algorithm for computing a histogram on a length-N array A of integers in the range $\{0,1,\ldots,k-1\}$, where $k \le N$. The algorithm uses a two-dimensional array H, whose dimensions are $(N/k)+1$ rows by $k+1$ columns. Assume for simplicity that N is divisible by k. Here is the algorithm:

1. Partition the array into N/k length-k subarrays $A_0, A_1, \ldots, A_{N/k-1}$.
2. Initialize each element of the matrix H to 0.
3. For all $r = 0,1,\ldots,N/k-1$, sequentially count the number of occurrences of each integer in A_r, and store the number of occurrences of each integer c in $H[r][c]$ (which fills in a row of H). Each row is processed sequentially but different rows can be processed in parallel.
4. For each $c = 0,1,\ldots,k-1$, perform a prefix sum on the cth column of H, that is, on the values $H[r][c]$ for all $0 \le r < N/k$. Different columns can be processed in parallel. $H[N/k][c]$ now stores the number of integers of value c in A.

The code for this algorithm is shown below. Assume that `vertical_prefix_sum(H, c, N/k)` computes the prefix sum on the cth column of H with N/k entries in parallel.

```
// H has dimension (n/k)+1 by k+1
void histogram(int64_t* A, int64_t** H, int64_t N, int64_t k) {
  cilk_for (int64_t r = 0; r < N/k; r++) {
    for (int64_t c = 0; c < k; c++) {
      H[r][c] = 0;
    }
    for (int64_t c = 0; c < k; c++) {
      H[r][A[r * N/k + c]]++;
    }
  }
  cilk_for (int64_t c = 0; c < k; c++) {
     vertical_prefix_sum(H, c, N/k);
  }
  //resulting histogram is stored in H[N/k]
}
```

6.3

What is the asymptotic work of `histogram`? Give your answer using Θ-notation in terms of N, k, and $W_{ps}(N)$, where $W_{ps}(N)$ is the work of prefix sum on N elements.

6.4

What is the asymptotic span of the `histogram`? Give your answer using Θ-notation in terms of N, k, and $S_{ps}(N)$, where $S_{ps}(N)$ is the span of prefix sum on N elements.

Intel x86 Assembly Language Cheat Sheet

Instruction	Effect	Example
Data movement		
mov src, dest	Copy src to dest	mov $10,%eax
Arithmetic		
add src, dest	Dest = dest + src	add $10, %esi
mul reg	edx:eax = eax * reg (colon means the result spans across two registers)	mul %esi
div reg idiv reg	edx = edx:eax mod reg eax = edx:eax / reg	div %edi
inc dest	Increment destination	Inc %eax
dec dest	Decrement destination	dec (%esi)
sbb arg1, arg2	If CF = 1, (this is set by cmp instruction; refer cmp) arg2 = arg2 – (arg1 + 1) else arg2 = arg2 – arg1	sbb %eax, %ebx
Function Calls		
call label	Push eip, transfer control	call _fib
ret	Pop eip and return	ret
push item	Push item (constant or register) to stack	pushl $32 pushl %eax
pop [reg]	Pop item from stack; optionally store to register	pop %eax popl
Bitwise Operations		
and src,dest	Dest = src & dest	and %ebx, %eax
or src, dest	Dest = src | dest	orl (0x2000), %eax
xor src, dest	Dest = src ^ dest	xor $0xffffff, %eax
shl count, dest	Dest = dest << count	shl $2, %eax
shr count, dest	Dest = dest >> count	shr $4, (%eax)
sal count, dest	Same as shl, shifted bits will be the sign bit	
Conditionals and jumps		
cmp arg1, arg2	If arg1 > arg2 sets CF=1 (carry flag =1) This compares arg1 and arg2; you can use any conditionals jumps below to act upon the result of this comparison	cmp $0, %eax
test reg,imm/reg	Bitwise and of register and constant/register; the next jump command uses the result of this; consider this essentially as same as compare	test %rax, %rcx
je label	Jump to label if arg2 = arg1	je endloop
jne label	Jump to label if arg2 != arg1	jne loopstart
jg label / ja label	Jump to label if arg2 > arg1	jg exit / ja exit
jge label	Jump to label if arg2 >= arg1	jge format_disk
jl label	Jump to label if arg2 < arg1	jl error
jle label	Jump to label if arg2 <= arg1	jle finish
jz label	Jump to label if bits were not set	jz looparound
jnz label	Jump to label if bits were set	jnz error
jump label	Unconditional jump	jmp exit
Miscellaneous		
nop	No-op	nop
lea addr, dest	Move the address calculated to the dest	lea 23(%eax, %ecx,8),%eax
cqto	%rdx:%rax← sign-extend of %rax.	cqto

suffixes b=byte(8), w=word(16), l=long(32), q=quad(64)
base indexed scale displacement 172(%rdi, %rdx,8) = %rdi + 8 * %rdx + 172
Note that not both src and dest can be memory operands at the same time.

register - %eax　　　　fixed address – (0x1000)
constant - $10　　　　dynamic address – (%rsi)