

Practice Quiz 4 ANSWERS

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

Answer: True

1.2

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

True False

Answer: 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

Answer: True

1.4

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

True False

Answer: 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

Answer: True

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

Answer: True

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

Answer: 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

Answer: 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 3GHz scalar processing core which executes only one instruction per cycle, and that everything is in cache.

```
1 #define ROWS 1000
2 #define COLS 1000
3
4 typedef int32_t element_t;
5
6 void get_row_sums(element_t M[ROWS][COLS],
7                  element_t row_sums[ROWS]) {
8     for (int32_t i = 0; i < ROWS; i++) {
9         element_t sum = 0;
10        for (int32_t j = 0; j < COLS; j++) {
11            sum += M[i][j];
12        }
13        row_sums[i] = sum;
14    }
15 }
```

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.

Answer: A

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.

Answer: E

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.

Answer: E

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.

```
1 void insertion_sort (int * A, int N) {  
2     for (int j = 0; j < N; j++) {  
3         int insert = A[j];  
4         int slot = j;  
5         while (slot > 0 && insert < A[slot-1]) {  
6             A[slot] = A[slot-1];  
7             slot--;  
8         }  
9         A[slot] = insert;  
10    }  
11 }
```

3.1 Sorted

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

Answer: A

3.2 Reverse Sorted

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

Answer: B

3.3 Random Order

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

Answer: B

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.

Answer: D

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*.

```
1 void encode(uint32_t* In, int32_t N, unsigned char* Out) {
2     for(int32_t i=0; i < N; i++) {
3         uint32_t x = In[i];
4         while(x) {
5             char byte = _____ (A) _____;
6             x = _____ (B) _____;
7             if(x)
8                 { _____ (C) _____ };
9             *Out++ = byte;
10        }
11    }
12 }
```

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 | _____ x << 7 |
| _____ byte & 0x8 | _____ x = x << 7 |
| _____ byte & ~0x80 | _____ x = x >> 1 |
| _____ byte &= 0x80 | _____ x >> 8 |
| _____ byte ^ 0x80 | _____ x ^ 0x7f |
| _____ byte = 0x80 Answer: C | _____ x ^ 0x80 |
| _____ byte | _____ x 0x7f |
| _____ x >> 7 Answer: B | _____ x 0x80 |
| _____ x & 0x7f Answer: A | _____ x++ |
| _____ x & 0x7 | _____ x-7 |
| _____ x & 0x80 | _____ 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.

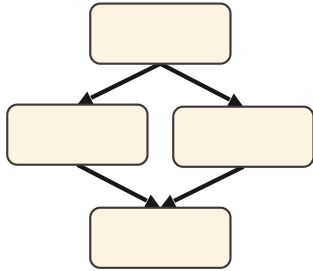
```
1  int value1();
2  int value2();
3
4  __attribute__((const))
5  int bar(bool p) {
6      int val;
7      if (p)
8          val = value1();
9      else
10         val = value2();
11     return val;
12 }
13
14 void foo(int *restrict Y,
15          const int *restrict X,
16          int n, bool p) {
17     for (int i = 0; i < n; ++i)
18         Y[i] += bar(p) * X[i];
19 }
```

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:

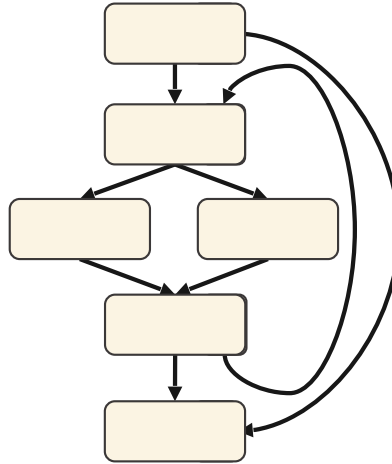
- A The function `bar` with no optimization.
- B The function `foo` with no optimization.
- C The function `foo` with function inlining and no other optimizations.
- D The function `foo` with loop unrolling and no other optimizations.
- E The function `foo` with code hoisting followed by function inlining 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.

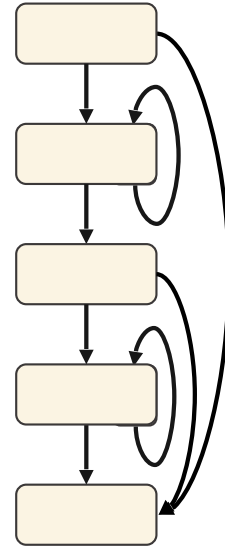
A B C D E None



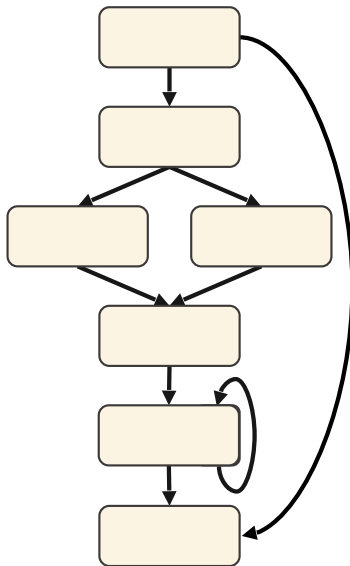
A B C D E None



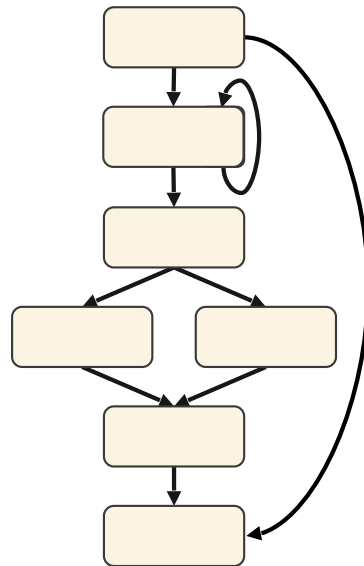
A B C D E None



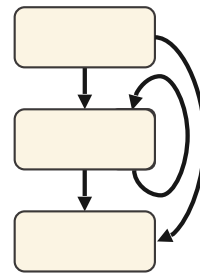
A B C D E None



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.

```

1  define i32 @bar(i1 zeroext) #0 {
2      br i1 %0, label %2, label %4
3
4      ; <label>:2:                                ; preds = %1
5      %3 = call i32 (...) @value1() #3
6      br label %6
7
8      ; <label>:4:                                ; preds = %1
9      %5 = call i32 (...) @value2() #3
10     br label %6
11
12     ; <label>:6:                                ; preds = %4, %2
13     %.0 = phi i32 [ %3, %2 ], [ %5, %4 ]
14     ret i32 %.0
15 }
16
17 define void @foo(i32* noalias, i32* noalias, i32, i1 zeroext) #2 {
18     %5 = icmp sgt i32 %2, 0
19     br i1 %5, label %4, label %._ret_edge
20
21     ; <label>:6:                                ; preds = %4, %6
22     %.01 = phi i32 [ 0, %4 ], [ %16, %6 ]
23     %7 = call i32 @bar(i1 zeroext %3) #4
24     %8 = sext i32 %.01 to i64
25     %9 = getelementptr inbounds i32, i32* %1, i64 %8
26     %10 = load i32, i32* %9, align 4
27     %11 = mul nsw i32 %7, %10
28     %12 = sext i32 %.01 to i64
29     %13 = getelementptr inbounds i32, i32* %0, i64 %12
30     %14 = load i32, i32* %13, align 4
31     %15 = add nsw i32 %14, %11
32     store i32 %15, i32* %13, align 4
33     %16 = add nsw i32 %.01, 1
34     %17 = icmp slt i32 %16, %2
35     br i1 %17, label %6, label %._ret_edge
36
37     ._ret_edge:                                ; preds = %6, %4
38     ret void
39 }

```

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.

```
1 void upsweep(int64_t* A, int64_t N) {
2     for (int64_t d = 1; d < N; d*=2) {
3         cilk_for (int64_t k = 0; k < N; k += 2*d) {
4             A[k + 2*d - 1] = A[k + d - 1] + A[k + 2*d - 1];
5         }
6     }
7 }
8
9 void downsweep(int64_t* A, int64_t N) {
10     A[N] = A[N - 1] //total sum
11     A[N - 1] = 0;
12     for (int64_t d = N / 2; d >= 1; d = d / 2) {
13         cilk_for(int64_t i = 0; i < N; i += 2*d) {
14             int64_t temp = A[i + d - 1];
15             A[i + d - 1] = A[i + 2*d - 1];
16             A[i + 2*d - 1] = temp + A[i + 2*d - 1];
17         }
18     }
19 }
20
21 void prefix_sum(int64_t* A, int64_t N) {
22     upsweep(A, N);
23     downsweep(A, N);
24 }
```

6.1

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

Answer: $\Theta(N)$

6.2

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

Answer: $\Theta(\log^2 N)$

Consider the following algorithm for computing a histogram on a length- N array A of integers in the range $\{0, 1, \dots, k-1\}$, where $k \leq N$. 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 $A_0, A_1, \dots, A_{N/k-1}$.
2. Initialize each element of the matrix H to 0.
3. For all $r = 0, 1, \dots, N/k - 1$, 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 $c = 0, 1, \dots, k - 1$, perform a prefix sum on the c th column of H , that is, on the values $H[r][c]$ for all $0 \leq r < N/k$. 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 c th column of H with N/k entries in parallel.

```

1 // H has dimension (n/k)+1 by k+1
2 void histogram(int64_t* A, int64_t** H, int64_t N, int64_t k) {
3     cilk_for (int64_t r = 0; r < N/k; r++) {
4         for (int64_t c = 0; c < k; c++) {
5             H[r][c] = 0;
6         }
7         for (int64_t c = 0; c < k; c++) {
8             H[r][A[r * N/k + c]]++;
9         }
10    }
11    cilk_for (int64_t c = 0; c < k; c++) {
12        vertical_prefix_sum(H, c, N/k);
13    }
14    //resulting histogram is stored in H[N/k]
15 }
```

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.

Answer: $\Theta(N) + kW_{ps}(N/k)$

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.

Answer: $\Theta(k + \lg(N/k)) + S_{ps}(N/k)$

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 \$0xffff, %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)

dynamic address - (%rsi)