Travis McGaha & Joel Ramirez
CIS2400 Fall 2024

CIS2400 Midterm Exam Solutions

This is a closed calculator/computer exam. You may consult one double-sided sheet of notes. No other references are allowed.

If you provide an incorrect answer, any explanation about how you arrived at your solution may lead to partial credit.

Full Name [print]: ______________________________

PennID: ______________________________

Exam Details & Instructions

- The exam consists of 5 questions (and a short bonus 6th question) worth 100 points total.
- You have 90 minutes to complete the exam.
- Two reference sheets will be provided. Do not write answers on them; they will not be graded.
- The exam is closed book. No textbooks, phones, laptops, wearable devices, or any other electronics are allowed beyond the permitted notes.
- You may use one 8.5” x 11” double-sided sheet of paper for notes.
- Turn off and put away all electronic devices and noise-making items.
- Remove all hats, headphones, and watches.

Pledge: I neither cheated nor helped anyone else cheat on this exam. All answers are my own. Violating this pledge may result in a failing grade.

Sign Here: ______________________________

Advice

- There are 5 questions (with 11 parts total). Budget your time to complete all of them.
- Don’t worry if there is more space than you need for answers. Extra space is provided just in case.
- Take a deep breath and relax. A bad grade on this exam is not the end of the world. You can improve your score with the Midterm ”Clobber” Policy (see the course syllabus).

Please initial or write your PennID on each page to ensure they can be matched if they become separated.

The last page is blank for extra space. If you use it, indicate clearly where your answer continues and include your full name and PennID at the top.

1 Memory Diagram and Structs [12 Points]

Consider the following struct definitions, which define a `student` struct containing the student's ID number, height, and enrolled classes:

```c
typedef struct {
  int feet;
  int inches;
} height_t;

typedef struct student_st {
  int        id_number;
  height_t   height;
  size_t     num_classes;
  int*       classes;
} student;

student create_travis() {
  student travis;                     // line  1
                                      // line  2
  travis.id_number = 2030;            // line  3
                                      // line  4
  travis.height.feet = 5;             // line  5
  travis.height.inches = 11;          // line  6
                                      // line  7
  travis.num_classes = 2;             // line  8
  int course_nums[] = {2400, 2620};   // line  9
  travis.classes = course_nums;       // line 10
                                      // line 11
  return travis;                      // line 12
}
```

If we had a `main` function that called `create_travis()` and printed the result:

```c
int main() {
  student travis = create_travis();
  // various print statements would go here
}
```

We might expect the following output:

```
travis {
  id_number = 2030
  height {
    .feet =   5
    .inches = 11
  }
  num_classes = 2
  classes = {2400, 2620}
}
```

Problem continues onto the next page

Interpretation

However, if we actually printed the result of `create_travis()`, the output would not match the expected result. What would be wrong with the printed output and why?

// You don't need to use all of this space

Solution. The issue with the printed result of `create_travis()` arises because the variable `course_nums` is defined within the `create_travis` function. When the function returns, the `course_nums` array goes out of scope as it exists within the stack memory of that function call. This means that the memory it occupies could be reused, leading to undefined behavior. Specifically, the `classes` pointer in `travis` will point to an invalid memory location once the function exits, causing the program to print unintended values or potentially crash.

2 Hamming Weight Calculation Using Bit Manipulations [12 Points]

Write a function in C that calculates and returns the amount of 1 bits (the Hamming weight) in the binary representation of an unsigned integer.

Assume that an `unsigned int` is 32 bits.

For example, if the input was 0xFFFFFFFF (all 1's) then 32 should be returned.

If the input was 7 (0000 0000 [20 more 0's]... 0111) then the function should return 3.

Solution. To calculate the Hamming weight (number of 1's) in the binary representation of an unsigned integer, we can repeatedly check the least significant bit, add it to our count, and then right-shift the integer by one position.

```c
unsigned int hamming_weight(unsigned int x) {
    unsigned int count = 0;
    while (x > 0) {
        count += (x & 1);
        x >>= 1;
    }
    return count;
}
```

3 Two's Complement Galore [17 Points]

You are given the following two 8-bit hexadecimal numbers which are declared as chars:

`A = 0xEB,   B = 0x3E`

For convenience, the range of signed chars is from −128 to 127.

(a) Interpretation [3 Points]

Convert both `A` and `B` from hexadecimal to binary. Clearly label each.

Solution.

`A = 0xEB` in binary is 1110 1011.

`B = 0x3E` in binary is 0011 1110.

(b) Addition [6 Points]

We want to take the two `char`s `A` and `B` from the previous part, add them together and store the result in another `char` variable. E.g.

```
char sum = A + B;
```

Compute the resulting character `sum` that we get from adding `A` and `B` using two's complement arithmetic. Use your results from part (**a**) and show your steps for the binary addition, carry's, etc. Clearly label the bits that would make up the `char sum` .

Solution.

Align the binary values of `A` and `B` and perform binary addition bit by bit, from right to left, carrying over any extra 1's.

$$\begin{array}{r} 11101011 \\ +00111110 \\ \hline 100101001 \end{array}$$

The result is 1 0001 1001 in binary, but we only have 8 bits, so *0b00101001* is the final answer. Other representations of this answer are `0x29` and `41`.

Assume your system supports 4 bit signed numbers who's type is called a `half char`. You are given the following two 4-bit binary numbers declared as `half chars`.

`C = 0b1011,   D = 0b0110`

(c) Casting from `half char` to `char` [8 Points]

Now, assume you want to cast these two signed `half chars`, C and D, to `chars` (which we'll assume are 8 bits in size). When casting from 4-bits `half char` to a 8-bits `char`, the system will copy the four bits into the bottom four bits of the result and then "fill in" the upper 4 bits.

```
half char C = 0b1011
char full_C = (char) C;
// full_c will be 0b XXXX 1011. Note the bottom four bits are the same as the original.
// We leave the value of the upper 4 bits hidden since
// you need to figure that out for this question.
```

When filling in the upper 4-bits, it needs to make sure that the decimal value remains the same in two's complement form whether it is 4-bits or 8-bits.

What should the 8-bit binary representation be after this cast of C and D? In other words, what should the upper 4 bits be filled with for C and D respectively. Please justify your answer.

Solution. When casting from a 4-bit signed `half char` to an 8-bit `char`, we must extend the sign bit to ensure that the value remains the same in two's complement form. This process is known as *sign extension*. In sign extension, the upper bits of the new, larger type are filled with copies of the sign bit (the leftmost bit of the original binary number) to preserve the original number's value and sign. This lines up with what we've learned in RISC-V and the difference between the instructions `ls` and `lsu`.

To extend each 4-bit number to 8 bits:

- For `C = 0b1011` (negative): - The leftmost bit is 1, so we fill the upper 4 bits with 1's to preserve the negative value. - This gives us `C` in 8-bit form: 1111 1011.
- For `D = 0b0110` (non-negative): - The leftmost bit is 0, so we fill the upper 4 bits with 0's to maintain the non-negative value. - This gives us `D` in 8-bit form: 0000 0110.

The 8-bit binary representations after casting are:

`C = 1111 1011,   D = 0000 0110`

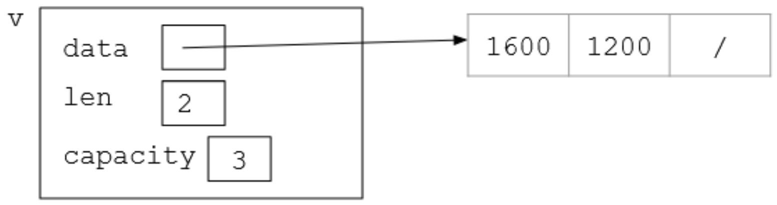
4 Dynamic Array Implementation in C [23 Points]

In this question, you will implement a dynamic array (vector) structure in C. Below is the definition of a vector struct, along with some partially implemented functions.

```
typedef struct vec_st {
    int* data;        // Inner array that the vector manages.
    size_t len;       // Number of elements currently stored in the vector.
    size_t capacity;  // Maximum number of elements the inner array can hold before resizing.
} vec;
```

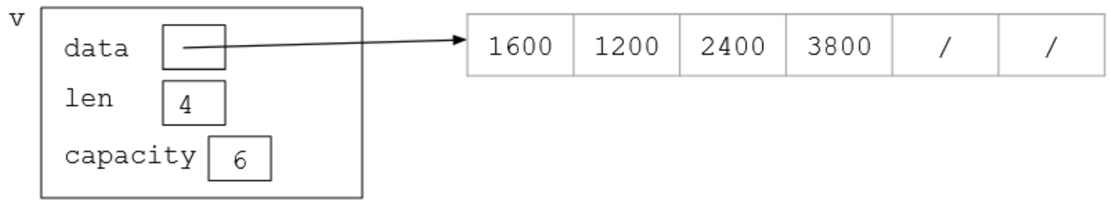
A vector is a type of "list" data structure that maintains an underlying array. When an element is added to the vector, it is added to the end of the underlying array. If the we needed to add more elements to the vector but have run out of space, we resize its underlying array by doubling the capacity of the array. After, we add the new element.

For example, this vector has a capacity of three but only contains 2 integers.



If we push **2400** onto the end of the vector via `vec_push(&v, 2400)`, then the last box on the right with a / (in the figure above) now holds the value 2400. `len` is updated to 3 while `capacity` stays the same.

If we push another value, **3800**, onto the end of our vector, we would need to resize the array to support holding more elements. We double the size of our array when we need to resize so `len` is updated to 4 while `capacity` is increased to 6. This results in the following:



Problem continues onto the next page

Below are three core functions that are used to manage the vector.

```c
// Creates a new vector with a specified initial capacity.
// Assume the initial capacity is greater than 0.
vec vec_new(size_t initial_capacity) {
    vec result;
    result.data = malloc(sizeof(int) * initial_capacity);
    result.len = 0;
    result.capacity = initial_capacity;
    return result;
}

// Gets the integer value stored at the specified index in the vector.
// This operation does not change the length or capacity of the vector.
int vec_get(vec* this, size_t index) {
    return this->data[index];
}

// Sets the integer value at the specified index to a new value.
// This operation does not change the length or capacity of the vector.
void vec_set(vec* this, size_t index, int element) {
    this->data[index] = element;
}
```

You do not need to read this code, but we provide it as a reference in-case it helps understand how one would use a vector. If you are comfortable with the idea of what a vector is, then feel free to move on to the next page (where the problem continues)

```c
int main() {
    vec v = vec_new(3); // Create a new vector with an initial capacity of 3.
    vec_push(&v, 1600);
    vec_push(&v, 1200);
    vec_push(&v, 2400);

    // v.len should be 3 (three elements have been added).
    // v.capacity should still be 3 (it has not been resized yet).

    vec_push(&v, 3800);

    // v.len should be 4 (a new element was added).
    // v.capacity should be 6 (the vector resized to double its capacity).

    vec_set(&v, 0, 1100); // Set the value at index 0 to 1100.

    for (size_t i = 0; i < v.len; i++) { // Loop through and print all values in the vector.
        printf("%d\n", vec_get(&v, i));
    }
    free(v.data); // Free the memory allocated for the vector.
}
```

(a) Implement the vec_push function [15 Points]

The `vec_push` function adds a new element to the end of the vector. If the vector's internal array does not have enough capacity, then the array is resized with double the capacity before adding the new element. Implement this function in C. You are not allowed to use `realloc`. You may assume functions like `malloc` always succeeds and that `this` is a valid vector.

Solution.

```c
void vec_push(vec* this, int element) {
    if (this->len == this->capacity) {
        size_t new_capacity = this->capacity * 2;
        int* new_data = malloc(new_capacity * sizeof(int));
        for (size_t i = 0; i < this->len; i++) {
            new_data[i] = this->data[i];
        }
        free(this->data);
        this->data = new_data;
        this->capacity = new_capacity;
    }
    this->data[this->len++] = element;
}
```

`We also noticed other students who created this function using the other methods in the problem. These were accepted as correct solutions. Here is one possible solution.`

Solution.

```c
void vec_push(vec* this, int element) {
    if (this->len == this->capacity) {
        // Create a new vector with double the current capacity
        vec new_vec = vec_new(this->capacity * 2);

        // Copy elements from the old array to the new array using vec_get and vec_set
        for (size_t i = 0; i < this->len; i++) {
            vec_set(&new_vec, i, vec_get(this, i));
        }

        // Free the old array and update the vector's data and capacity
        free(this->data);
        this->data = new_vec.data;
        this->capacity = new_vec.capacity;
    }

    // Set the new element at the next available position
    vec_set(this, this->len++, element);
}
```

(b) Pointer Passing [8 Points]

The `vec_get` function currently takes a pointer to a `vec` structure. Could this function be **rewritten** to take the vector by value (i.e., `vec` instead of `vec *`) and still behave correctly? Justify your answer.

// You do not need to use all of this space

Solution. If we pass `vec` by value in the `vec_get` function, the `vec` structure itself is copied, but this copies everything such as `len`, `capacity`, and the `data` pointer. Importantly, the `data` pointer within the structure still points to the original array in memory where the elements are stored so we can still retrieve values within the correct array.

5 Bit Error Detection [35 Points]

You are given a 3-bit sequence, $I_2I_1I_0$, where I_2 is a parity bit used to verify whether the number of 1s in the last two bits, (I_1 and I_0), is even or odd.

The rules for interpreting the parity bits are as follows for valid sequences:

- If I_2 is 0, the number of 1s in I_1I_0 should be **even** (0 counts as even in this case).
- If I_2 is 1, the number of 1s in I_1I_0 should be **odd**.

If the sequence does not follow the rules described above, it is considered an invalid sequence of bits. (An example of an invalid sequence is 111).

(a) Logic Expression [5 Points]

Write a **boolean logic expression** that checks the validity of the 3-bit sequence. The expression should evaluate to **true** if the sequence is valid, **false** otherwise. Do not use ∩ or ∪ notation nor code.

Solution.

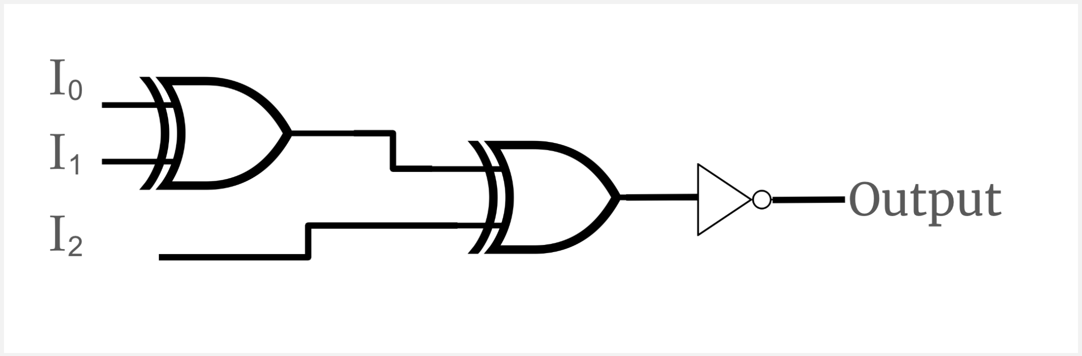
$$\sim (I_2 \hat{} I_1 \hat{} I_0)$$

There are of course much longer solutions that we accepted. But this was the shortest possible.

(b) Detecting Errors Using Logic Gates [10 Points]

Based on your expression in part (a), create a **gate-level logic circuit** that takes in I_0, I_1, and I_2 and outputs **1** if the sequence is valid, **0** otherwise.

Solution.



You could use a negated XOR gate as the final gate, but here is a more detailed solution. Many different approaches would have received credit.

(c) CMOS [10 Points]

Design a proper CMOS circuit which takes in all 3 bits of the input I and produces the output bit. You can assume that you also have access to negated versions of all of the input bits. Your solution must be a single CMOS circuit consisting of complementary pull up and pull down transistor networks. It should not involve cascading multiple CMOS circuits. Please label the inputs and outputs of your circuit clearly on your schematic.

Solution. Although our solution from part A is concise, it doesn't allow for straightforward "xor" blocks at the CMOS level, based on our current knowledge of CMOS design. Therefore, we'll need to use a version of part A that relies solely on OR, AND, and NOT gates to implement the solution in CMOS. We can adopt a quasi-PLA form for the Pull-Down Network (PDN) and then translate it directly into the Pull-Up Network (PUN).

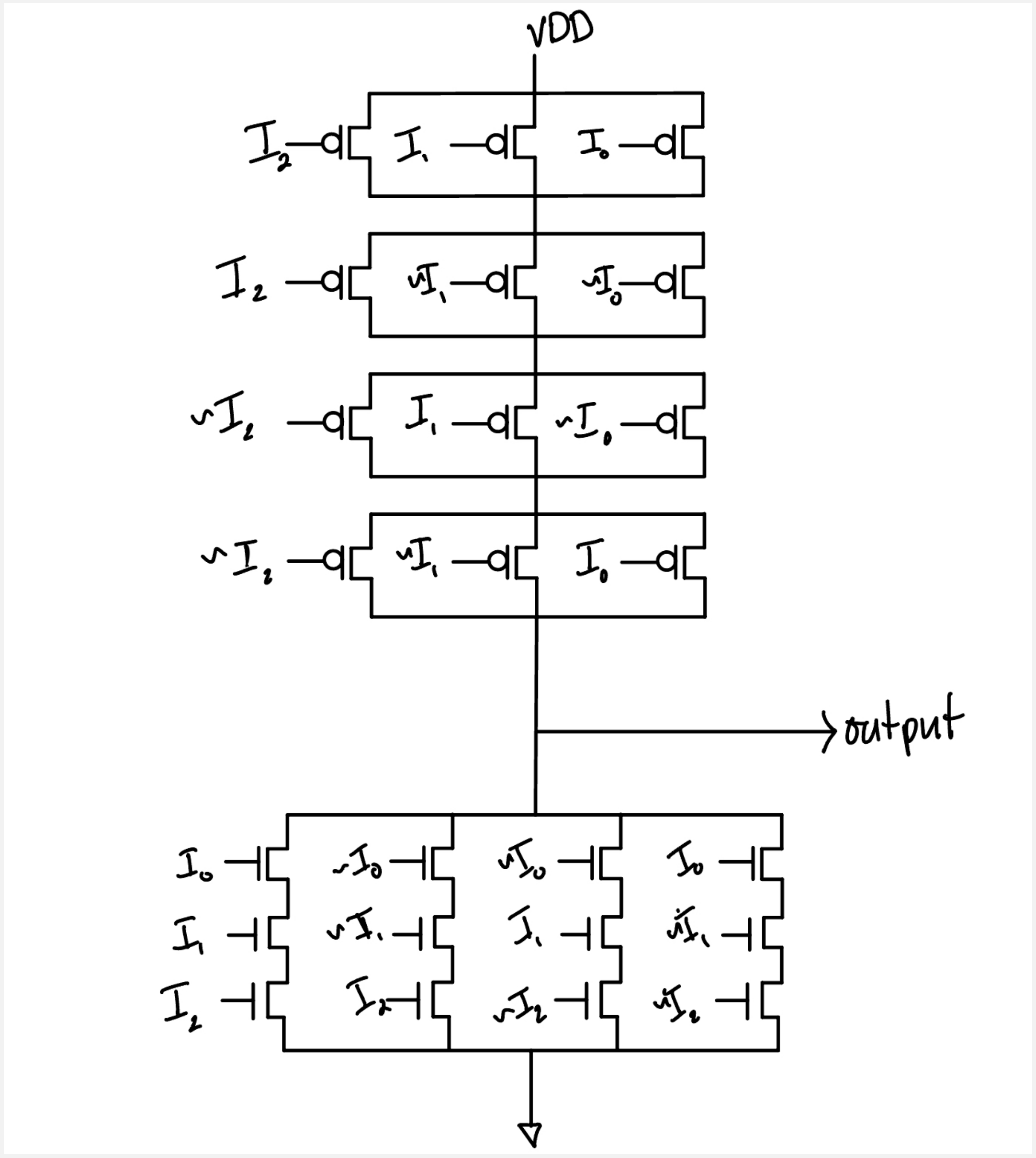
First, let's determine the expression for when the PDN allows a signal to pass through. In other words, we need to identify the conditions that produce an invalid sequence of bits.

$$(I_2 \,\&\, I_1 \,\&\, I_0) \;|\; (I_2 \,\&\, \neg I_1 \,\&\, \neg I_0) \;|\; (\neg I_2 \,\&\, I_1 \,\&\, \neg I_0) \;|\; (\neg I_2 \,\&\, \neg I_1 \,\&\, I_0)$$

From this, we can see an immediate relationship with the PUN's corresponding boolean expression.

$$(\neg I_2 \,|\, \neg I_1 \,|\, \neg I_0) \,\&\, (\neg I_2 \,|\, I_1 \,|\, I_0) \,\&\, (I_2 \,|\, \neg I_1 \,|\, I_0) \,\&\, (I_2 \,|\, I_1 \,|\, \neg I_0)$$

And finally, here is the entire network.


VDD
I_2
I_1
I_0
I_2
$\sim I_1$
$\sim I_0$
$\sim I_2$
I_1
$\sim I_0$
$\sim I_2$
$\sim I_1$
I_0
output
I_0
I_1
I_2
$\sim I_0$
$\sim I_1$
I_2
$\sim I_0$
I_1
$\sim I_2$
I_0
$\sim I_1$
$\sim I_2$

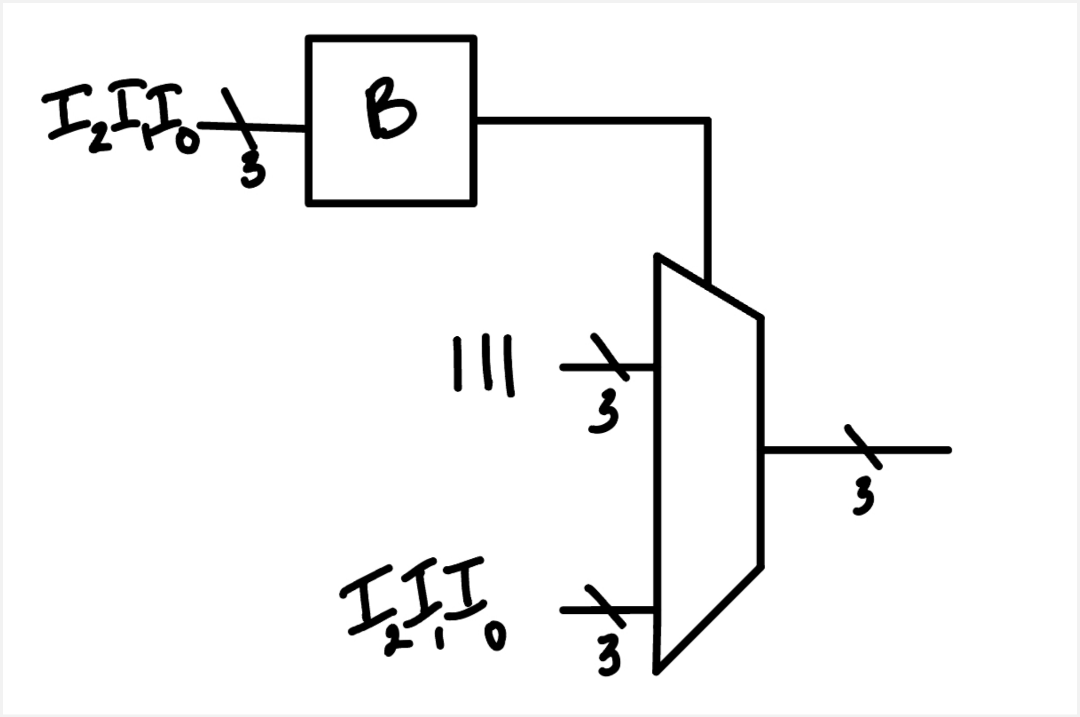
(d) Error Handling with a Multiplexer [10 Points]

Let's decide how to handle data transmission based on the result of the error detection logic circuit. The system must meet the following conditions:

- If no error is detected, the original 3-bit data should be output.
- If an error is detected, output a sequence of 1s (e.g., 111).

Draw a logic circuit diagram using a multiplexer (MUX) and any necessary logic gates to select between transmitting the original 3-bit data or the error signal based on the conditions described above. **You may assume you have access to the solution from part (b) as a box labeled "B" in your diagram.**

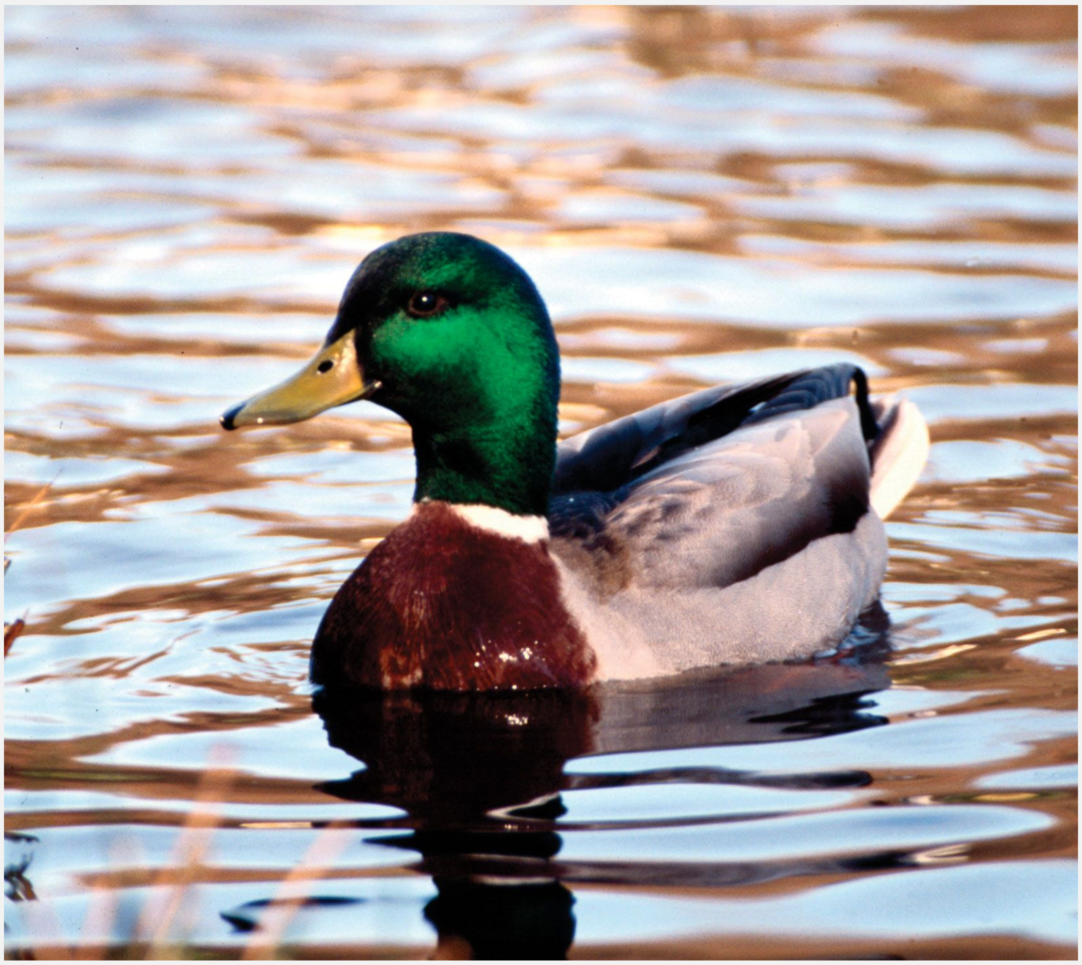
Solution. And, finally, the last diagram of the midterm.



6 Bonus Question [1 Point: all submissions will get this 1 point]

Put anything you'd like the course staff to see here. This can be nothing, a piece of art, a message to members of the staff, anything you like! If you can't come up with anything, then a suggestion would be to write down your favorite aspect(s)/topic(s) of the course so far!

Solution.



Scratch Work Page