CMSC 311: Computer Organization

Fall 2002

Exam 1 Solutions

Exam Given: Monday, October 21, 2002

Instructions: This is a 75 minute exam. This exam is closed book and closed notes. No calculators are permitted on the exam. All computations should be done on paper.

This is a 140 point exam. The exam consists of 4 parts:

- Quiz 1: Data Representation (40 points) pass: 28
- Quiz 2: Combinational Logic (40 points) pass: 28
- Quiz 3: Low level C functions/CPU (20 points) pass: 14
- Challenge ("Drool Kitty") Section (40 points)

SEE PAGE 2 for USEFUL FACTS and CONVENTIONS.

Name	
SID (Last 6 digits)	
Email	

HONOR STATEMENT

The following is an honor statement. This is NOT the same as the "honor pledge". It only states your *awareness* of the following statement. It is the equivalent of saying "I know it is against the law to drive faster than the speed limit" instead of saying "I pledge not to drive faster than the speed limit".

> *I am aware that giving or receiving unauthorized assistance or using unauthorized aids on this exam is considered a violation of academic integrity.*

Please sign below to indicate that you READ and UNDERSTAND what the statement says. Your exam will not be returned to you, unless this is signed.

Signature	

Conventions:

- Circle data/control inputs from outside world. SQUARE outputs to outside world.
- Select (control) lines are written using c. For example, a 3 bit control lines are written $c_2c_1c_0$. c_2 is the most significant bit and c_0 is the least.
- Data input lines are written using x. For example, a 3 bit input lines are written $x_2x_1x_0$. x_2 is the most significant bit and x_0 is the least. (Assuming the input is a binary number).
- Data output lines are written using z. For example, a 3 bit output lines are written $z_2z_1z_0$. z_2 is the most significant bit and z_0 is the least. (Assuming the output is binary).
- Full adders will use A, B, C_i, C_o, and S inside the chip to represent the two bits being added, the carry in, the carry out, and the sum, respectively.
- Inside the chip, use capital letters. Outside the chip use lowercase letters to represent inputs coming from outside world, and outputs going to outside world.

Useful facts:

- $1000\ 0000_2$ = 128 and $1111\ 1111_2$ = 255
- The following are gates: AND, OR, NOT, XOR, XNOR, NAND, and NOR. Unless otherwise specified, you may assume that each gate (except NOT) can have infinite fan-in.
- The following are parts: any gates, multiplexer (MUX), demultiplexer (DEMUX), encoder, decoder, half adder (HA), full adder (FA), k-bit ripple carry adder, priority encoder.
  - HA uses inputs: A, B and outputs: C (carry out), S (sum).
  - FA uses inputs: A, B, C_i (carry in) and outputs: C_o (carry out), S (sum).

Grading

Part	Points
Quiz 1	
Quiz 2	
Quiz 3	
Challenge	
Total	

CMSC 311 Computer Organization
Fall 2002
RTfP

Name	
Student ID	

Quiz 1 (Data Representation): ".. and the bases are loaded" (40 points)

1.1 (2 pts) Here's a declaration in a C program:

```
int x ;
```

What's the internal data representation for `x`?

Answer: 32 bit two's complement.

1.2 (2 pts) Here's a declaration in a C program:

```
float y ;
```

What's the internal data representation for `y`? (Hint: there's a `double` type that stores a longer version of a `float`)

Answer: IEEE 754 single precision floating point.

1.3 (2 pts) Here's a declaration in a C program:

```
char ch ;
```

What's the internal data representation for `ch` on most computers (e.g., DC class cluster)?

Answer: ASCII.

1.4 (3 pts) Given a k-bit bitstring (where $k \geq 1$), how many different bit strings are there? Write a formula for this (hint: use k).

Answer: 2^k different bitstring patterns.

1.5 (3 pts) Suppose you are given a bitstring with *exactly* k 1's, and you're told the number is an unsigned integer. Write a formula for the base 10 (decimal) value of this bitstring. You can assume $k \geq 2$. (Don't use summations, unless you can't think of another way).

Answer: $2^k - 1$

Recall 2^k is 1, followed by k zeroes. If you subtract 1, you get k 1's. This is similar to having an odometer that's 1 followed by k zeroes. If you subtract 1, you get k 9's.

1.6 (3 pts) Suppose you are given a bitstring with *exactly* k 1's, and you're told the number is a signed integer. Write a formula for the base 10 (decimal) value of this bitstring. You can assume $k \geq 2$. State ANY ASSUMPTIONS needed for the formula to be true.

Answer: -1, assuming the representation is two's complement.

(Could also say, 0, if it's one's complement, or $-(2^{k-1} - 1)$ if it's signed magnitude).

Make sure to state which signed number representation you are using.

1.7 (4 pts) Write $+5_{10}$ in the following representations using 4 bits. If it is not possible to write it in the representation, write N/A.

Answer: To convert a value from base 10 to excess representation, add the excess (i.e., the bias) to the value, and convert it to an unsigned binary number. For example, to convert 5 to excess 8, add 8 to 5 to get 13, and convert 13 to unsigned binary (which is 1101).

The most common error is to flip bits for one's complement and flip bits and add 1 for two's complement. You only flip bits, etc. if the number is negative. 5 is not negative, so you just write it as if it were unsigned.

Signed Magnitude	0101
1's Complement	0101
2's Complement	0101
Excess 8	1101
Excess -3	0010

1.8 (4 pts) Write -8_{10} in the following representations using 4 bits. If it is not possible to write it in the representation, write N/A.

Answer: Notice it's not possible to represent -8 in all the representations (using 4 bits, that is) except in 2's complement and excess 8. For one's complement and signed magnitude, the range of values is between -7 and 7, inclusive.

For excess 8, adding 8 to -8 results in 0. Convert 0, as if it were unsigned binary results in 0000.

For excess -3, add -3 to -8 to get -11. Since the number is negative, you are outside the range of valid values for excess -3 (after adding, the excess, the value should lie between 0000 and 1111, or 0 and 15, and -11 is outside that range).

Signed Magnitude	N/A
1's Complement	N/A
2's Complement	1000
Excess 8	0000
Excess -3	N/A

1.9 (3 pts) Define overflow. That is, explain what overflow MEANS, not how to detect it.

Answer: Overflow occurs when the result of an operation produces a value that is outside the range of the representation.

For example, a carry out of 1 does not define overflow. It is used to detect overflow *when adding two k-bit integer values in unsigned representation.* This is a way to *detect* overflow (for the representation), but is not a definition what overflow is.

1.10 (4 pts) Someone provides you the following function which does arithmetic with `float` values (32 bits). This function performs add, subtract, and multiply operations in some sequence with the arguments, though you don't know in which order. It returns a `float` value.

```
float doStuff( float x, float y ) ;
```

We want to use this function with `int` values, and to do so by casting from `int` to `float`. For example,

```
int j = 3, k = 4 ;
int result
  = static_cast<int>( doStuff( static_cast<float>( j ),
                               static_cast<float>( k ) ) ) ;
```

Does casting to float and back to int cause any problems for any arbitrary, but valid, values of j and k (answer YES, if it does, and NO if it does not). Assume a working compiler that does casting correctly. EXPLAIN for full credit.

Answer: No. 32 bits are used to represent two's complement signed int and 32 bits are also used to represent a float (using IEEE 754 single precision). Float values can be much larger than int values. However, since there are exactly as many float representations as int representations (2^{32} values), there are int values that can't be represented as floats. (Certainly, float values with fractions can't be represented as ints, but since we are considering int values converted to floats and using add, subtract and multiply operations, we don't have deal with that case).

In particular, if an int has more than 24 significant bits (23 bits for the fraction, 1 bit for the hidden bit), it can't be represented in IEEE 754 single precision, since not enough bits are available in the mantissa, thus some error is introduced when converting such an int to a float, and that error would continue as you convert the float back to an int.

1.11 (4 pts) Which CATEGORY (hint: there are 5 of them) do the following 32 bit floating point values belong?

(a) 0 1111 1111 0000 0000 0000 0000 0000 000

Answer: (Positive) infinity. (Exponent is all 1's, sign bit is 0, and fraction is all 0's).

(b) 0 1111 1111 0011 0000 0000 0000 0000 000

Answer: NaN (not a number). (Exponent is all 1's, sign bit is 0, and fraction is NOT all 0's).

(c) 0 0000 0001 0011 0000 0000 0000 0000 000

Answer: Normalized. (Exponent is not all 1's, nor all 0's)

(d) 0 1000 0000 0000 0000 0000 0000 0000 000

Answer: Normalized. (Exponent is not all 1's, nor all 0's)

(e) 0 0000 0000 1111 0000 0000 0000 0000 000

Answer: Denormalized. (Exponent is all 0's, but fraction is NOT all 0's)

1.12 (3 pts) Convert -17.5 to IEEE single precision. Label bits clearly (sign bit, etc).

Answer: To convert, you convert the integer part first (to 10001), then the fraction part (to 0.1), and add to get 10001.1.

Then, write this in scientific notation (which means you need a single bit left of the radix point): $1.00011 \times 2^{4_{10}}$.

The exponent needs to be converted to excess 127, so you add 127 to 4 to get 131. To convert 131, it's easiest to see this as $131 = 128 + 3$. 128 in binary is 10000000. 3 in binary is 00000011. Add the two to get 10000011.

So, the sign bit is 0, the exponent is 1000 0011 and the fraction is 00011000000000000000000.

The final answer (written as sign bit, exponent, fraction) is written as:

0 1000 0011 0001 1000 0000 0000 0000 000

Notice that the hidden 1 is not represented.

1.13 (3 pts) Assume the following representation for floating point:

- No sign bit (numbers are non-negative)
- 3 bits used to represent the mantissa/significand
- No hidden 1
- 3 bits used to represent the exponent (assume representation is unsigned).

You want to multiply $1.01_2 \times 2^{2_{10}}$ by $1.11_2 \times 2^{3_{10}}$. Both numbers are written in the desired representation (i.e., use 3 bits for number, etc). What is the answer in scientific notation using the above constraints? (Note: the values shown earlier are in scientific notation). Assume truncation is used.

Answer: This is merely multiplication. All the information provided is something of a smoke screen.

In order to multiply floating point numbers, multiply the mantissa first (i.e., multiply 1.01_2 to 1.11). The result is: 10.0011.

Then, add the exponents (recall that multiplying a^b to a^c results in a^{b+c}. The result is $2^{5_{10}}$.

So far, the number is: $10.0011 \times 2^{5_{10}}$.

This is not yet normalized. You need to move the radix point so that only a single 1 appears left of the radix point. This requires moving the radix point left by 1 bit. Since the mantissa (or significand) has decreased, you must increase the exponent by 1, to get: $1.00011 \times 2^{6_{10}}$.

Finally, since only 3 bits are used in the representation, we truncate the result to 3 bits.

The final answer is: $1.00 \times 2^{6_{10}}$.

1.14 (2 pts) Other than ASCII, name another (common) representation for characters. Describe any facts you know about this representation (one fact is fine).

Answer: Unicode. It uses 16 bits to represent a character. (Also used to represent characters in Java).

Other answers include EBCDIC and various UTF representations. Unicode is the most common.

Quiz 2 (Combinational Logic): "Conjunction Junction" (40 points)

Name	
Student ID	

2.1 (3 pts) Combinational logic circuits implement what mathematical abstraction?

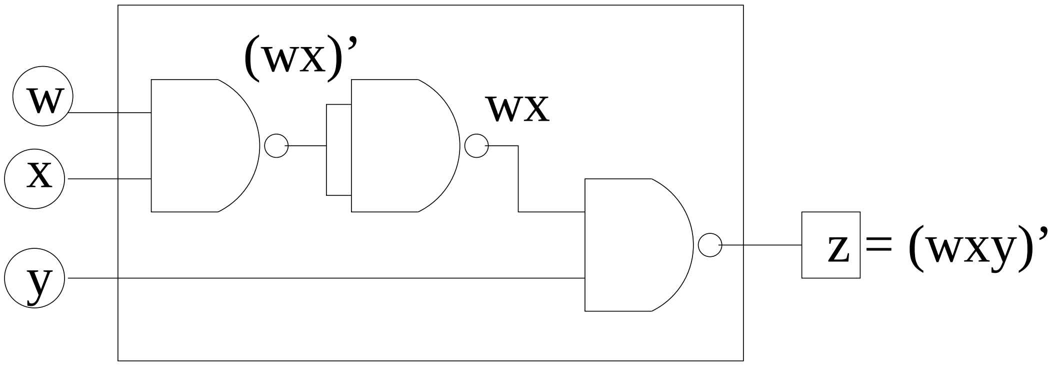
Answer: Boolean functions (see next question, where it tells you the answer). Truth tables also acceptable.

2.2 (5 pts) A NAND$_3$-gate implements the Boolean function $f(w, x, y) = (wxy)'$

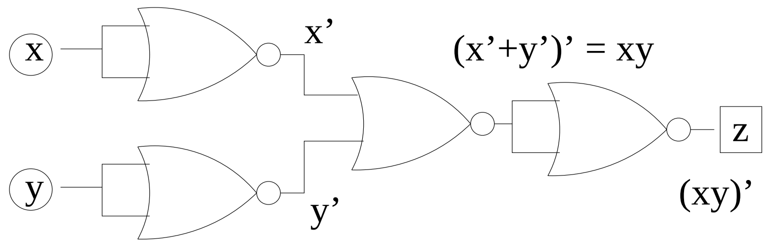
Implement this circuit using as FEW NAND$_2$ (2-inputs) gates as needed. Label inputs as w, x, y (draw circle around each input variables) and label output as z (draw a box around the z).

Answer: The key to this is to realize that NAND is not associative. You can't put 2 NAND$_2$ gates together (like you can with two AND$_2$ gates) to get a NAND$_3$ gate.

Here's the circuit:



2.3 (5 pts) Implement a $NAND_2$-gate (2-input) using as few NOR_2 gates as necessary.



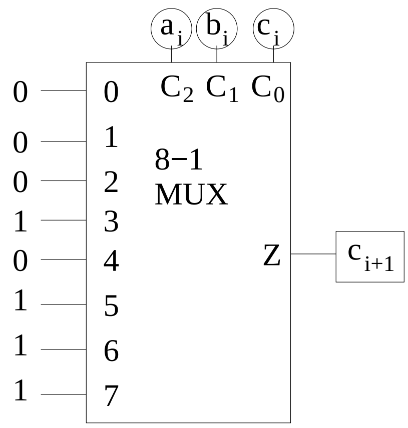
Converting NAND to NOR (or OR) usually involves negating the input variables so that you can take advantage of DeMorgan's Law.

2.4 (5 pts) Implement a circuit which is equivalent to the carry out of a full adder using any part except a full adder or half adder. (See back of page 1 for definition of parts).

Answer: Any circuit will do, but the quick and dirty way is to use the MUX method. The truth table for the carry out looks like:

a_i	b_i	c_i	c_{i+1}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

The actual circuit is



Although you could have written a sum-of-products expression for c_{i+1}, in an exam such as this, quick answers are usually what you want. There were no restrictions such as "make it with few gates" or "make it elegant". So this is an ideal question for a MUX trick.

2.5 (5 pts) You wish to implement an AND$_k$ (i.e., a k-input AND gate), but all you have are AND$_2$ gates (as many as you want). Assume the time delay of a gate is 1 unit. Write a formula that expresses the MINIMUM delay for an implementation of AND$_k$ using AND$_2$ gates. EXPLAIN how you came up with this formula.

Answer: $\lceil O(\lg n) \rceil$ units.

Essentially, create a complete binary tree of AND_2 gates. A complete tree is made by starting at the root and filling out the AND gates row by row, until you have enough inputs. Each additional AND$_2$ gate gives you an additional input, so you need $n - 1$ AND$_2$ gates for n inputs (where $n \geq 2$).

This is the best you can do since you are maximally using all inputs near the root.

2.6 (10 pts) Fill out the following table, indicating how many control, data inputs, outputs each has (in general).

Use "k" in your expression, as needed.

Answer: This is a useful exercise to help you learn how many inputs and outputs various devices have.

	Control Inputs	Data inputs	Outputs
MUX	k	2^k	1
DeMUX	k	1	2^k
Encoder	0	2^k	k
Decoder	0	k	2^k

This table was presented in class. This assumes a few things. First, it assumes that inputs or outputs are powers of 2. This assumption can be relaxed somewhat, . In this case, you might prefer to replace quantities like 2^k with n and quantities like k with $\lceil \lg n \rceil$. That's a better answer (of course, you'd write it with k since you were instructed to do so, but I picked n so you could see the relationship between n and k).

2.7 (6 pts) Suppose you have a priority encoder with four inputs: x_3, x_2, x_1, x_0. In lecture, we said that higher indexes represent higher priorities. Suppose lower indexes represent higher priorities (index 0 has highest priority).

Write the Boolean expression for the outputs of z_1 and z_0, where $z_1 z_0$ represent the 2 bit subscript of the input with the highest priority.

Answer: This is perhaps the toughest question in this part of the exam. However, we did do an example in class of this. Basically, this is what you do. Write the equations for a regular encoder. Then, modify it so that you negate literals that have higher priority, unless that literal shows up in the expression.

For example, suppose you have x_3. That has the lowest priority. So, you modify it to $x_3 x_2' x_1' x_0'$ which is an expression that is true if x_3 has the highest priority (since x_3 would have to be 1, and x_2, x_1, x_0 would have to be 0). However, suppose x_1 appears in the expression. Then, there's no need to have x_1'. So, you can simplify to $x_3 x_2' x_0'$. Conceptually, removing the terms is the hardest part of writing an efficient priority encoder formula. If you understand this, you are in prety good shape.

Here's a table for a regular encoder.

x_3	x_2	x_1	x_0	z_1	z_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Notice that this truth table only has 4 rows. The assumption of a regular encoder is that you can have, at most, one input that is 1. Therefore, the Boolean equations/expressions/formula for this table is generated by creating a simplified sum-of-products expression. Again, the sum-of-products can be simplified because the exactly one input can be 1 (otherwise, you would need minterms).

$$
\begin{aligned}
z_1 &= x_2 + x_3 \\
z_0 &= x_1 + x_3
\end{aligned}
$$

From this, you can create the priority encoder equations by adding negated literals of higher priority to each.

$$
\begin{aligned}
z_1 &= x_2 x_1' x_0' + x_3 x_1' x_0' \\
z_0 &= x_1 x_0' + x_3 x_2' x_0'
\end{aligned}
$$

Notice that in z_1, there is a term $x_3 x_1' x_0'$. The negated literal x_2' does not appear. It doesn't have to since x_2 appears as $x_2 x_1' x_0'$. Thus, if x_2 were 1, then that term would be 1, and z_1 would be 1, so that's fine.

Again, if you follow this, you're in great shape. This is the kind of circuit that separates those who can follow equations from those who can't. With enough time (and TA assistance), everyone can do this.

2.8 (4 pts) Answer the following questions about encoders.

(a) What assumptions, if any, are made about the inputs of a regular encoder?
Answer: Exactly one input has value 1 (the rest have 0).

(b) What assumptions, if any, are made about the inputs of a priority encoder?
Answer: At least one input has value 1 (the rest have 0). Note: the reason you must have at least one input with value 1 is because if all inputs were 0, what would the encoder produce as output? For each possible input, there is an encoding (including 0). Thus, having all 0's would have the same priority as have $x_0 = 1$, and there would be conflict as to whether x_0 was 1 or all inputs were 0.

Quiz 3 (Low-level C functions/Odds and Ends): "10 Little Endians" (20 points)

Name	
Student ID	

3.1 (4 pts) The following are the 5 steps to execute an instruction. Explain each step.

Answer: Notice that these steps are NOT the same as the ones in the previous exam. However, I did mention this in class (which is why reviewing class notes is important). The order of the execution comes from the textbook.

IF Instruction fetch. Fetch instruction from memory.

ID Instruction Decode. Decode instruction and fetch register operands (since this is a RISC architecuture, both can be done simulateneously).

ALU ALU operation. Perform the operation from the instruction.

MEM Memory operation. Read data from memory, or write data to memory. Used in load/store instructions (any mention of memory was fine).

WB Write back. Write back the result of the ALU operation to the register file.

3.2 (5 pts) Here's a prototype for a C++ function:

```
bool checkRange( unsigned int x, int low, int high ) ;
```

Using bitwise operations and shifting, determine if AT LEAST ONE bit between index `low` and index `high`, inclusive, has value 1. Assume low $\leq$ high and that the indexes lie between 0 and 31, inclusive. Assume index 0 is the LSB and index 31 is the MSB.

```
bool checkRange( unsigned int x, int low, int high )
{
  int mask = ~0 ; // create all 1's
  mask <<= ( high - low ) + 1 ; // 1's followed by (high - low) + 1 zeroes
  mask = ~mask ; // 0's followed by (high - low) + 1 ones
  mask <<= low ;  // shift left low bits (creates 1's between index low and
                  // high and 0's elsewhere)
  return x & mask ;
}
```

Some students used loops and had a single 1 starting at index low, and moving left. They ANDed the 1 with x and if this were ever true, they returned true.

A few students made the mistake of trying to index individual bits. You can't index individual bits in C/C++. Thus, x[1] does not represent bit 1 of x. It is a compiler error in C to do index in this manner.

Another common error was not realizing that shift operators do NOT modify the object. Thus, $x << 3$ produces a VALUE that is x shifted left by 3, but does not modify x itself. This is why you see assignments like $x <<= 3$ which is the same as $x = (x << 3)$.

Another "error" was not using bitshift operations. I had several answers that used the power function to do what could be done with bitshifts. It's not so much that the answers are "wrong", but that they could be done without using a power function.

3.3 (3 pts) Given the following 32 bit value: $ABCDEF01_{16}$

Write these values in a byte-addressable memory starting at address 1000 in LITTLE-ENDIAN order. Make sure you write ALL addresses properly.

Answer: For some reason this question generated a LOT of errors. In particular, "byte addressable" means that each address stores one byte. Each hex digit is only 4 bits. So, you need two hex digits to represent a byte. (In an older exam, I did do "nybble addressable" in which each memory address holds one nybble, which is 4 bits. That was the most common answer I got for this question).

Address	Data
1000	01
1001	EF
1002	CD
1003	AB

3.4 (5 pts) Here's a prototype for a C++ function:

```
void clearBits( unsigned int & x, int low, int high ) ;
```

Clear bits (i.e, set it to 0) from index `low` to index `high`, inclusive, but leave remaining bits unchanged. Use same assumptions as `checkRange`.

Answer: This function is very similar to `checkRange` except the mask is flipped.

```
void clearBits( unsigned int & x, int low, int high )
{
  int mask = ˜0 ; // create all 1's
  mask <<= ( high - low ) + 1 ; // 1's followed by (high - low) + 1 zeroes
  mask = ˜mask ; // 0's followed by (high - low) + 1 ones
  mask <<= low ;  // shift left low bits (creates 1's between index low and
                  // high and 0's elsewhere)
  mask = ˜mask ;  // create 0's between low and high, and 1's elsewhere
  x &= mask ;     // Zero out the masked area
}
```

3.5 (3 pts) Without using a temp variable (i.e., using logical bit operators), write a swap function

```
void swap( unsigned int & x, unsigned int & y ) ;
```

Answer: This was done very briefly in class. Again, it helps to review notes thoroughly.

```
void swap( unsigned int \& x, unsigned int \& y )
{
  // Let x' and y' be the ORIGINAL values of x and y, when passed
  // in

  x = xˆy ; // x now holds x' XOR y'
  y = xˆy ; // y now holds (x' XOR y') XOR y' = x'. Thus, y holds x's old value
  x = xˆy ; // x now holds (x' XOR y') XOR x' = y'. Thus, x holds y's old value
}
```

Challenge ("Drool Kitty") Section: "Meow!" (40 points)

C1 (10 pts) Consider the following 6-1 MUX built from 3-1 MUXes and 2-1 MUXes. The 3-1 MUXes have the following operation

c_1	c_0	z
0	0	x_0
0	1	x_1
1	0	x_2
1	1	random (don't care)

(a) Write a table for a 6-1 MUX and state assumptions about its operations.

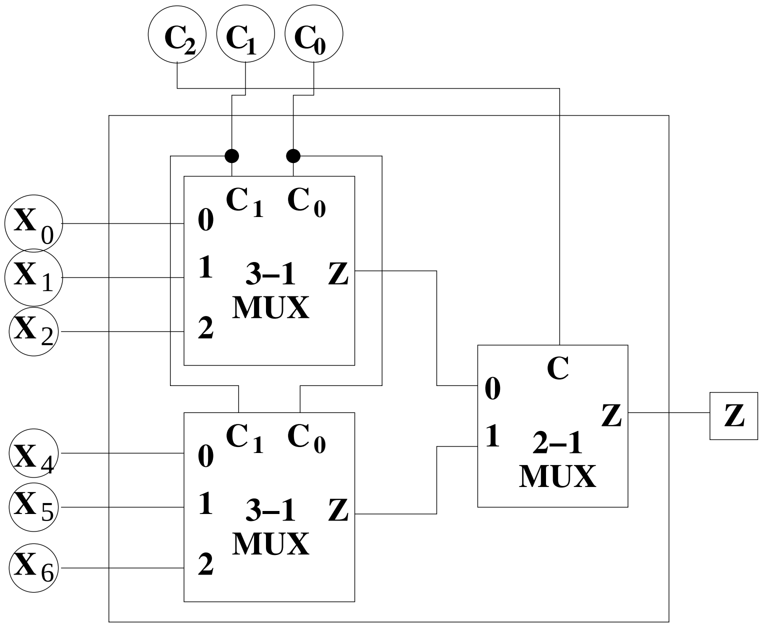
Answer: A table refers to a truth table, similar to the one shown above. A few students didn't understand what was meant by this.

c_2	c_1	c_0	z
0	0	0	x_0
0	0	1	x_1
0	1	0	x_2
0	1	1	random (don't care)
1	0	0	x_4
1	0	1	x_5
1	1	0	x_6
1	1	1	random (don't care)

It's important that the don't cares appear where they do. We want to encode 000, 001, 010 and 100, 101, 110, since the difference between the first 3 values and the last 3 values is the most significant bit. The most common mistake was to have the values contiguous (i.e. define it for 000 up to 101, and have 110 and 111 as don't cares). You will find this doesn't work if you trace the inputs of the circuits.

One student had a solution where even inputs were placed in one MUX, and odds in another MUX. Smoe students wanted the inputs to be contiguous (notice that x_3 is skipped over in the table above), but had the don't care's in the correct places. The reason x_3 is skipped is because we assume the user inputs the subscript of the input used. However, no credit was taken off for those who wrote contiguous subscripts (from x_0 to x_5) provided the don't cares appeared in the fourth and eighth row.

(b) Clearly label inputs and outputs. Use conventions for data inputs using x with subscripts and control inputs using c with subscripts.



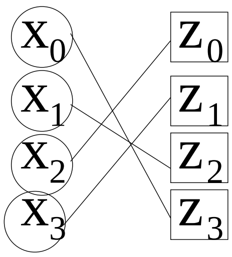
C2 (10 pts) Design a circuit which implements the following. Elegant answers (i.e. simple ones) receive more credit. However, any valid answers will get some credit.

x_3	x_2	x_1	x_0	z_3	z_2	z_1	z_0
0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0
0	0	1	0	0	1	0	0
0	0	1	1	1	1	0	0
0	1	0	0	0	0	0	1
0	1	0	1	1	0	0	1
0	1	1	0	0	1	0	1
0	1	1	1	1	1	0	1
1	0	0	0	0	0	1	0
1	0	0	1	1	0	1	0
1	0	1	0	0	1	1	0
1	0	1	1	1	1	1	0
1	1	0	0	0	0	1	1
1	1	0	1	1	0	1	1
1	1	1	0	0	1	1	1
1	1	1	1	1	1	1	1

Clearly label inputs (by circling them) and outputs (by boxing). You can use any part/gate you wish.

Answer: Some people tried to write sum-of-products. It works, but it's slow. Some tried encoder-decoder. Also works. Also slow (but more creative). This one only requires 4 wires. Many, many students got this correct, so I was rather happy.

The circuit looks like:



C3 (10 pts) For a carry lookahead adder, what are the general formulas for:

(a) g_i and p_i

Answer: The formulas are:

$$\begin{aligned} g_i &= a_i \cdot b_i \\ p_i &= a_i + b_i \end{aligned}$$

(b) Briefly explain what g_i and p_i refer to (not just the names, but the purpose).

Answer: g_i is the "generate" part. If this value is 1, this means the sum of a_i and b_i generates a 1 regardless of the value of the carry in c_i.

p_i is the "propagate" part. If this value is 1, this means that the carry-in c_i, if 1, gets propagated to the carry-out c_{i+1}. If the carry-in is 0, this gets propagated too, but only if g_i is not 1.

(c) Assume you are adding two 4-bit numbers

$$a_3a_2a_1a_0 + b_3b_2b_1b_0$$

and c_0 is the carry-in to the least significant bit (normally, a ripple-carry adder has no carry-in to the LSB, but pretend you have a full adder adding the LSB).

Write a formula for c_2 for a carry-lookahead adder, using only the following variables: $c_0, a_3, a_2, a_1, a_0, b_3, b_2, b_1, b_0$. Not all variables need to appear in the formula.

Answer: The general carry formula is $c_{i+1} = g_i + p_i c_i$.

Using this, we can see that $c_2 = g_1 + p_1 c_1$.

Using the same formula, $c_1 = g_0 + p_0 c_0$.

Plug in the expression for c_1 into the c_2 formula to get: $c_2 = g_1 + p_1(g_0 + p_0 c_0)$.

Multiply and rewrite as: $c_2 = g_1 + g_0 p_1 + p_1 p_0 c_0$.

Plug in for g_1, g_0, p_1, p_0.

$c_2 = (a_1 b_1) + (a_0 b_0)(a_1 + b_1) + (a_1 + b_1)(a_0 + b_0)c_0$.

I like this kind of problem because it has enough formulas that you need to spend time figuring it out, and I didn't cover it in class, so you had to read the book and work it out yourself or with friends. This kind of question is more common in engineering courses (where you must learn to derive formulas and figure things out from books). Reading and learning from books is an invaluable skill to learn.

Learning from class is great, but you will find many more occasions where all you have is book, and your ability to sit and think about what it says, rather than hearing an explanation and asking questions when you are stuck.

C4 (10 pts) Drool Kitty is a kitty that drools so much, that she needs a towel to sop it up. She also loves circuit problems. She argues that another way of performing two's complement operation is to subtract 1, and flip bits (the usual way is to flip bits and add 1) where subtract 1 may wrap around (thus 0000 - 1 results in 1111, for 4 bits). Is Drool Kitty correct?

If so, provide a mathematical argument. To do so, use facts about function composition and function inverses, and also check boundary cases (what happens near all 0's or all 1's).

If not, derive a counterexample where this fails. You should assume that the operation applies to k bit numbers where $k \geq 2$.

Answer: This question is "easy" provided you remember your 250 very well. The question was "inspired" by a student, and the answer was provided by Dr. Kruskal. Its inclusion in the exam is mostly to illustrate the following point: you sometimes need math to verify an idea you believe is intuitively true.

A lot of people feel discrete math or any math is useless. Part of this feeling comes from finding the material difficult. If it were easy, then you wouldn't care whether it's useless or not. However, a good grounding in math allows you to see a problem like this and convince yourself (in a mathematical way) that the answer is true.

In CMSC 250, you studied functions. Functions can have inverses provided that the function is 1-1 and onto. The two operations used in two's complement is flipping bits (call this function flip(x)) and adding 1 (call this function add1(x)).

Both these functions are 1-1 and onto. That is, given the value $f(x)$, there is a unique x that it is mapped from (that's 1-1). Also, every value is mapped to (that's onto). Flipping bits is more obvious. Adding 1 is adding 1 modulo 2^k for some value of k.

Thus, if a function is 1-1 and onto (i.e., it's a bijectin) then the functions both have inverses. The inverse of flip(x) is flip(x). That is, flipping bits is its own inverse. The inverse of add1(x) is sub1(x), i.e., subtracting 1 modulo 2^k.

If two functions are 1-1 and onto, then their composition is also 1-1 and onto (provided the domains and codomains are the same in both functions). Thus, add1 $\circ$ flip is 1-1 and onto.

This means the function has an inverse. The inverse of a composition is the inverses of each function in reverse order. Thus, (add1$\circ$flip)$^{-1}$ is flip$^{-1}\circ$add1^{-1}, which is flip$\circ$sub1, which is just subtracting 1 and flipping bits.

Finally, since $\text{add1} \circ \text{flip}$ is its own inverse (that is, flipping bits and adding 1, then doing it again, produces back the same number), then $\text{add1} \circ \text{flip} = (\text{add1} \circ \text{flip})^{-1} = \text{flip} \circ \text{sub1}$

The only part of the “proof” that's somewhat weak is that two's complement is its own inverse (which was assume, rather than proven). However, anything even partly resembling this got full credit.

Some people tried “induction”, but only showed a base case, and no indication of how they might try to show an inductive hypothesis or prove an inductive case. Such answers received minimal credit.