Lab 03 Binary Number System

General Information

What you should do

• You should first review Lecture 2 and be familiar with the concepts.

• You should try to come up with the code yourself as much as possible. Do not be afraid of making mistakes, since debugging (finding out where your code goes wrong and fixing it) is part of the learning process.

• We do not give out model programs to the exercises. There can be multiple ways to write the code that solves the same problem. It is important that you build up the program logic yourself instead of merely looking at some code that you do not understand. At any time if you are lost or if you have any questions, feel free to ask the instructor, tutor, or teaching assistant and we will be very happy to help you.

Self-Discovery

• Most lab tasks are designed to be relatively simple such that you can take the time to think about the related underlying concepts. Besides, we also encourage you to discover things on your own which may not be specified in the tasks.

Task 1.1 Decimal to 4-bit Binary Number Conversion for Unsigned Integers

In this task we would like to convert an unsigned integer from decimal to binary. We assume that a 4-bit word is used to represent the unsigned integer in binary. As a result, the possible unsigned integers that can be represented by a 4-bit word are in the range from 0 to 15.

From Lecture 2, you have learnt that converting a decimal number to binary can be achieved through **repeated division by 2**. Since we are dealing with 4-bit word representation, the division by 2 is repeated 4 times. The remainders from the divisions correspond to the resulting bits. For example, consider an unsigned integer M = 8 and carry out the divisions 4 times to get the remainders:

8/2 = 4 … 0

4/2 = 2 … 0

2/2 = 1 … 0

1/2 = 0 … 1

The 4-bit binary representation of 8 is 1000.

Consider another example for M = 5 and carry out the divisions 4 times to get the remainders:

5/2 = 2 … 1

2/2 = 1 … 0

1/2 = 0 … 1

0/2 = 0 … 0

The 4-bit binary representation of 5 is 0101.

Now we need to find a way to compute these remainders from any given value of the variable M in a general way. Let us denote the quotients and remainders of these divisions by *q0*, *r0*, *q1*, *r1*, *q2*, *r2*, *q3*, *r3* respectively. To extract the binary bits of the value from the variable M, we can carry out the repeated divisions 4 times:

M / 2 = *q0* … *r0*

q0 / 2 = *q1* … *r1*

q1 / 2 = *q2* … *r2*

q2 / 2 = *q3* … *r3*

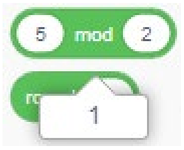
In mathematics, the remainder r of y divided by x is often expressed as $r = y$ *mod* x. For example,

8/2 = 4 … 0, the remainder can be expressed as 0 = 8 mod 2

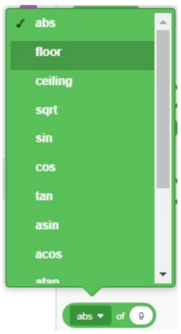
5/2 = 2 … 1, the remainder can be expressed as 1 = 5 mod 2

In Scratch, there is a block

in the Operators block category which performs this *modulus* function to find the remainder when the first number is divided by the second number. Now enter 5 in the first circle and 2 in the second circle and click on this block. You should see that the number 1 is displayed as the result.



To get the quotient, let us look at the previous examples again. For the case 8/2 = 4 … 0, It is easy to see that the quotient is 4 as 8 is evenly divisible by 2. For another case 5/2 = 2 … 1, if you type 5/2 in your calculator, you would get 2.5 as the result. We have already determined that the quotient of this division is 2, which is the integer part of 2.5, i.e., after throwing out all the decimal values. In mathematics, there is an operation called *floor* which only retains the integer part of a number, such that floor(2.4)=2, floor(4.9)=4, floor(0.5)=0. In Scratch, there is a block in the Operators block category which performs this floor function.



Now select floor from the list in the block and enter some numbers such as 2.4, 4.9, 0.5 in the white circle and click the block to verify that the results are matched with the numbers described previously.

To summarize, the remainder can expressed by using the mod function and the quotient can be computed by using the floor function. So $q0$, $r0$, $q1$, $r1$, $q2$, $r2$, $q3$, $r3$ can be obtained from the following equations:

$r0 = M$ mod 2
$q0$ = floor(M / 2)
$r1 = q0$ mod 2
$q1$ = floor($q0$ / 2)
$r2 = q1$ mod 2
$q2$ = floor($q1$ / 2)
$r3 = q2$ mod 2
$q3$ = floor($q2$ / 2)

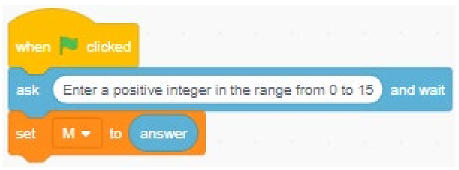
The resulting 4 binary bits $r3$, $r2$, $r1$, $r0$ are for representing the decimal number M.

Follow the steps below to obtain the 4 binary bits from the value of the variable M in Scratch:

1) Create a new project and put down the project name as Lab 03 Task 1.1.

2) Create the variables M, $q0$, $r0$, $q1$, $r1$, $q2$, $r2$, $q3$, $r3$.

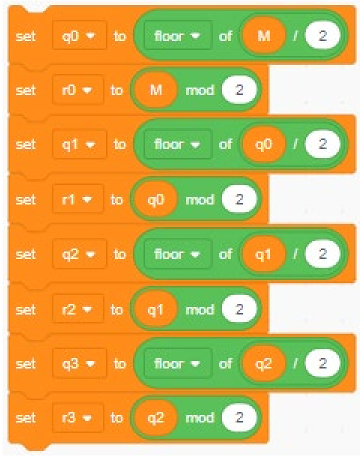
3) When the program starts, we would like the user to enter an unsigned integer and store it to M. Construct the following block:



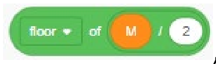
```
when [flag] clicked
ask [Enter a positive integer in the range from 0 to 15] and wait
set M to (answer)
```

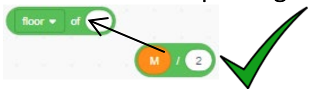
Note that a good programmer would then validate the user input to check if it is an unsigned integer in the range from 0 to 15. Here we will not do this check and just assume that the user indeed has entered the proper input.

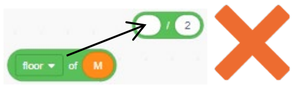
4) Construct the following blocks and attach them to the end of the above block created from 3):



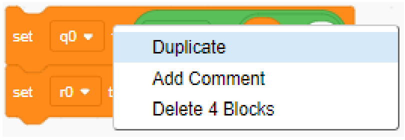
```
set q0 to (floor of ((M) / (2)))
set r0 to ((M) mod (2))
set q1 to (floor of ((q0) / (2)))
set r1 to ((q0) mod (2))
set q2 to (floor of ((q1) / (2)))
set r2 to ((q1) mod (2))
set q3 to (floor of ((q2) / (2)))
set r3 to ((q2) mod (2))
```

Hint 1: In constructing the code block  , pay attention to the order with which these blocks should be put together:

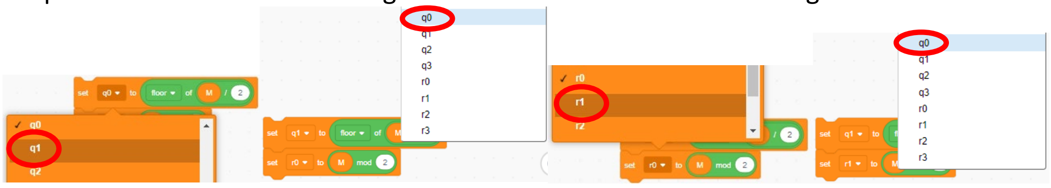




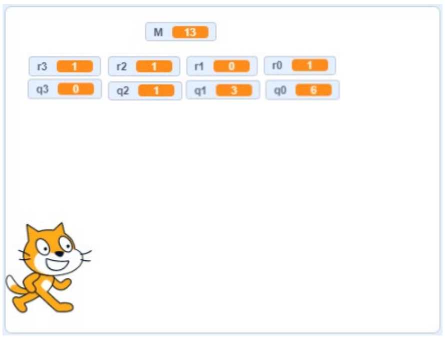
Hint 2: First construct the first 2 blocks, and then use the duplicate tool (right click on the first block and select duplicate) to create similar blocks and then change the parameters



Hint 3: Change the parameters of the duplicated blocks by selecting the correct variable from the dropdown list on the left AND right click on a variable block to change to another one

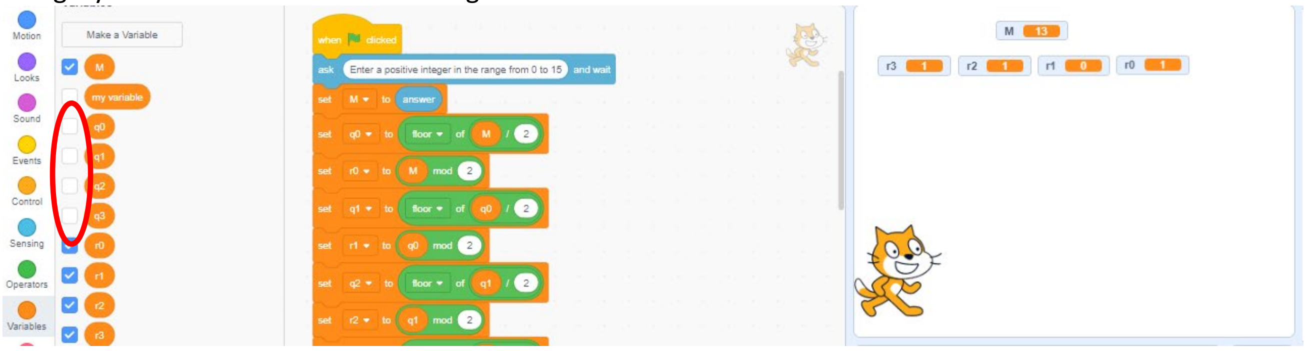


5) Move the variables around on the Stage area and rearrange them in a way that is easier for you to visualize the result.



6) Run your program a few times by entering different unsigned integers in the range from 0 to 15 to be stored with M and check to make sure that each time the variables $r3$, $r2$, $r1$, $r0$ yield the correct binary bits that correspond to the binary representation of the number M.

7) Hide all the quotient variables by unchecking the corresponding boxes in the Variables block category as we would not need them again.



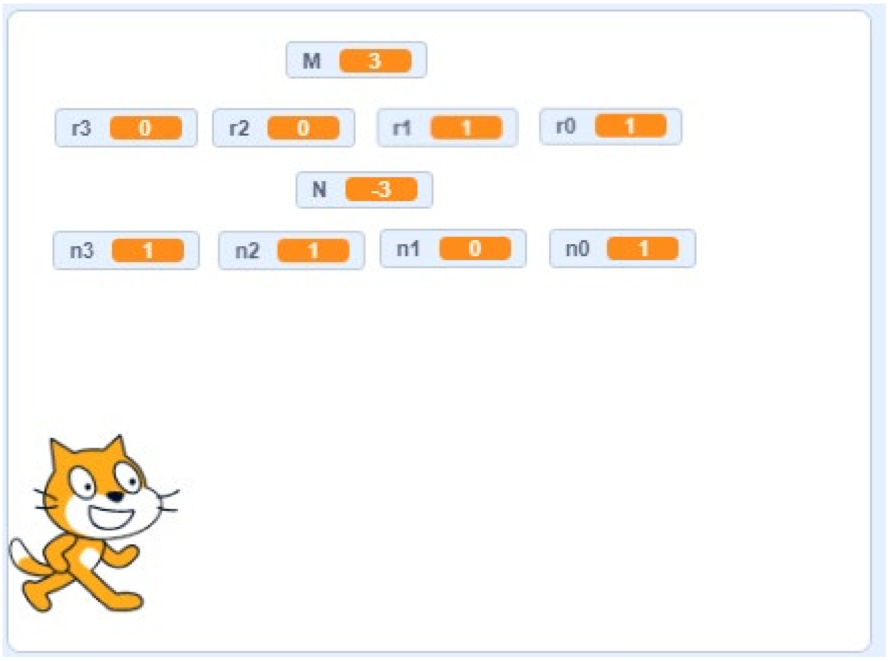
```
when [flag] clicked
ask [Enter a positive integer in the range from 0 to 15] and wait
set M to (answer)
set q0 to (floor of (M / 2))
set r0 to (M mod 2)
set q1 to (floor of (q0 / 2))
set r1 to (q0 mod 2)
set q2 to (floor of (q1 / 2))
set r2 to (q1 mod 2)
```

```
M 13
r3 1   r2 1   r1 0   r0 1
```

Now think of an unsigned integer from 0 to 15. Then convert it to binary by hand using the method introduced in Lecture 2. Then verify your answer by running the program from this task. Repeat this a few times so that you can master the technique of decimal to binary number conversion.

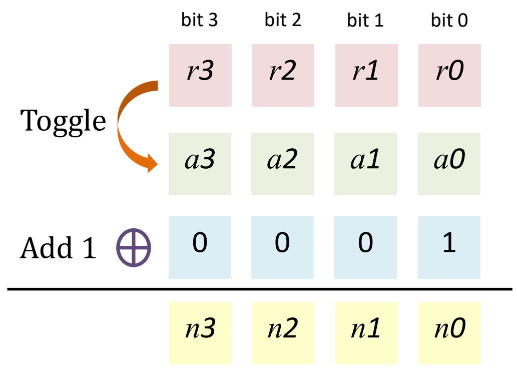
Task 1.2 Two's Complement Representation with 4-bit Word

In this task we would like to represent a signed integer using a 4-bit word. First we will ask the user to input an integer in the range from 0 to 7. Then we will show its equivalent 4-bit binary number. Afterwards, the two's complement will be applied to this 4-bit binary number to obtain another 4-bit binary number representing the negative of the input integer. The following screenshot shows an example of the completed program:



```
M 3
r3 0   r2 0   r1 1   r0 1
N -3
n3 1   n2 1   n1 0   n0 1
```

Recall from Lecture 2 that the two's complement operation has 2 steps: 1) toggle all bits (from 0 to 1 or from 1 to 0); and 2) add 1 to the result. In this problem, we would like to apply two's complement to the 4-bit number *r3 r2 r1 r0*. Let us denote the result after toggling all bits of *r3 r2 r1 r0* as *a3 a2 a1 a0*. Then we add 0001 to *a3 a2 a1 a0* and obtain the desired result as *n3 n2 n1 n0* which is the negative of the binary number *r3 r2 r1 r0.* The following figure illustrates this process:



Task 1.2A Toggling

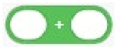
Let's focus on the first step on toggling the bits. Each bit from *r3 r2 r1 r0* is processed individually. The toggled results are represented by *a3 a2 a1 a0.* Without loss of generality, let us consider *r0*. The value of *r0* is either 0 or 1. When *r0* = 0, then *a0* = 1. When *r0* = 1, then *a0* = 0. The following table shows these possible values:

r0	*a0*
0	1
1	0

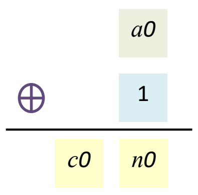
How do you write the code to get the correct value of *a0* given *r0*? If you look at the sum of these variables, i.e., *r0* + *a0*, you can observe that it is always equal to 1 in both cases. As a result, *r0* + *a0* = 1, which is equivalent to *a0* = 1 - *r0*, and you can use this equation to obtain *a0* from *r0*. *a3*, *a2*, *a1* can be obtained in a similar way from *r3*, *r2*, *r1*, i.e.,

a0 = 1 – *r0* *a1* = 1 – *r1* *a2* = 1 – *r2* *a3* = 1 – *r3*

Task 1.2B Adding 1

After obtaining *a3 a2 a1 a0* from the previous step, we need to add 0001 to it in order to obtain the result *n3 n2 n1 n0*. Since we are dealing with binary addition here, we could not directly use the operator block  to add them as if they are decimal numbers. We need to think about the program logic to obtain *n3 n2 n1 n0* by adding 0001 to *a3 a2 a1 a0.* It may seem to be quite complicated when you first think about this problem, so let us divide this task into simpler ones by considering one bit at a time from right to left.

1) At bit 0, in adding *a0* and 1, there are 2 possible results since *a0* is either 0 or 1.
   If *a0* = 0, then *a0* + 1 = 1, thus *n0* = 1
   If *a0* = 1, then *a0* + 1 = 10, thus *n0* = 0, and there is a carry bit (let's denote it by *c0*) that needs to be added together with the other bit 1. These 2 cases can be illustrated with the following figure and table:

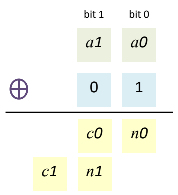


a0	*a0*+1_{10}	*c0*	*n0*
0	1	0	1
1	2	1	0

   Now we need to write the code to get the correct value of *c0* and *n0* given *a0*. After going through Task 1.2A, you should immediately see that *n0* = 1 - *a0*. It is also obvious from the above table that *c0* = *a0.*

   n0 = 1 - *a0* *c0* = *a0*

2) At bit 1, we have *a1* from the first number to be added, and 0 from the second number (1) to be added. However, we also need to add the carry bit *c0* that comes from the addition of bit 0. The addition of the last 2 bits is illustrated by the following figure and table:

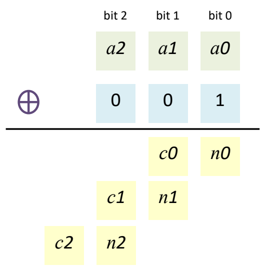


a1	*c0*	$(a1+c0)_{10}$	*c1*	*n1*
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	0

In short, when handling addition at bit 1, we need to include the bit 1 from the 2 numbers to be added, as well as the carry bit from the addition at bit 0, to derive the sum $(a1+0+c0)_{10}=(a1+c0)_{10}$, which can be represented as a 2-bit binary number *c1 n1.* You should observe from the above table that we list all the 4 possible combinations of *a1* and *c0* with the corresponding results of *c1* and *n1.* Now we need to write the code to get the correct value of *c1* and *n1* given *a0* and *c0*. The relationship may not be obvious at first sight. If you compare the 3rd column $(a1+c0)_{10}$ and the last column *n1* from the above table, you would see that the values of the first 3 rows are exactly the same. A more important discovery is that when $(a1+c0)_{10}$ is odd (i.e., equal to 1 in this case), *n1*=1. On the other hand, when $(a1+c0)_{10}$ is even (i.e., equal to 0 or 2 in this case), *n1*=0. As a result, you can use the modulus operator that we learnt in Task 1.1 to derive such relationship, *n1* = (*a1*+*c0*) mod 2. Now for *c1*, you observe that the first 3 rows are 0 and the last row is 1. Compared with $(a1+c0)_{10}$, the last row increases from the first row by 2. One may thus try dividing (*a1*+*c0*) by 2 and obtain the values in the 4 rows as 0, 0.5, 0.5, 1, where the first value and the last value are equal to *c1*. The middle 2 values are not exactly the same (0.5 vs 0), but their integer part is the same (0). Recall from Task 1.1 that you can apply the floor operator to extract the integer part of a floating-point number. So if you evaluate the expression floor((*a1*+*c0*)/2), you will find that the resulting values would be exactly the same as *c1*. The following equations summarize these relationships:

$n1 = (a1+c0) \bmod 2 \qquad c1 = \text{floor}(\ (a1+c0)/2\ )$

3) At bit 2, we have *a2* from the first number to be added, and 0 from the second number (1) to be added. However, we also need to add the carry bit *c1* that comes from the addition of bit 1. The addition of the last 3 bits is illustrated by the following figure and table:

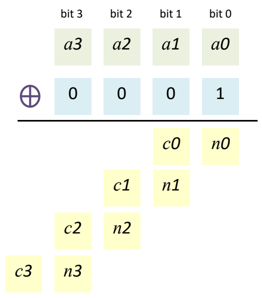


a2	*c1*	$(a2+c1)_{10}$	*c2*	*n2*
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	0

You can see from the above table that the values in each row are exactly the same as the table in 2). As a result, you can quickly write the code to get the correct value of *c2* and *n2* given *a2* and *c1* by deriving the following equations:

$n2 = (a2+c1) \bmod 2 \qquad c2 = \text{floor}(\ (a2+c1)/2\ )$

4) At bit 3, we have *a3* from the first number to be added, and 0 from the second number (1) to be added. However, we also need to add the carry bit *c2* that comes from the addition of bit 2. The addition of all these 4 bits is illustrated by the following figure and table:



a3	*c2*	$(a3+c2)_{10}$	*c3*	*n3*
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	0

You can see from the above table that the values in each row are also exactly the same as the table in 2). Besides, since our word size is 4 bits, we do not need to handle the carry bit $c3$. As a result, you can quickly write the code to get the correct value of $n3$ given $a3$ and $c2$ by deriving the following equation:

$n3 = (a3+c2) \bmod 2$

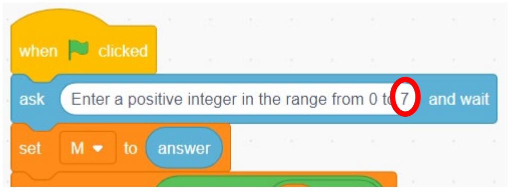
Task 1.2C Writing the Code

Follow the steps below to modify your code from Task 1.1 such that after obtaining the 4 binary bits from the value of the variable M, it will obtain the two's complement representation for the variable N=-M:

1) In the menu, choose File > Save as a Copy and update the project name as Lab 03 Task 1.2.

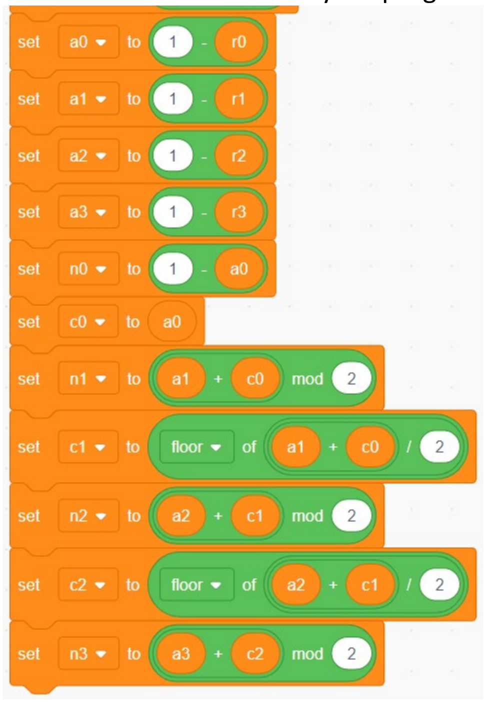
2) Create the variables N, $a0$, $a1$, $a2$, $a3$, $n0$, $n1$, $n2$, $n3$, $c0$, $c1$, $c2$.

3) Modify the ask block so that now it asks the user to enter an unsigned integer in the range from 0 to 7 (instead of from 0 to 15).



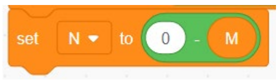
```
when [flag] clicked
ask [Enter a positive integer in the range from 0 to 7] and wait
set M to (answer)
```

4) Construct the following block that captures the relationships derived from Task 1.2A and Task 1.2B and attach it to the end of your program:



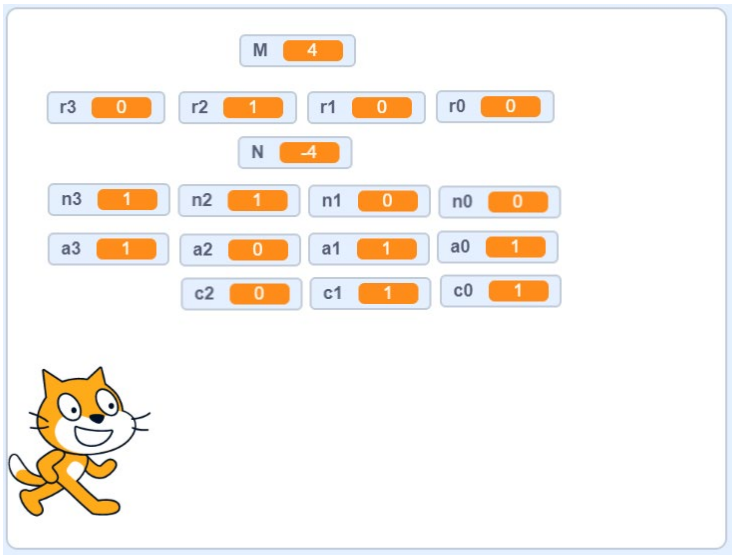
```
set a0 to (1 - r0)
set a1 to (1 - r1)
set a2 to (1 - r2)
set a3 to (1 - r3)
set n0 to (1 - a0)
set c0 to a0
set n1 to ((a1 + c0) mod 2)
set c1 to (floor of ((a1 + c0) / 2))
set n2 to ((a2 + c1) mod 2)
set c2 to (floor of ((a2 + c1) / 2))
set n3 to ((a3 + c2) mod 2)
```

5) Finally attach the following block at the end of your program to represent the negative of the variable M:



```
set N to (0 - M)
```

6) Rearrange the variable displays so that they are aligned making it easier to verify their values:



M 4

r3 0 r2 1 r1 0 r0 0

N -4

n3 1 n2 1 n1 0 n0 0

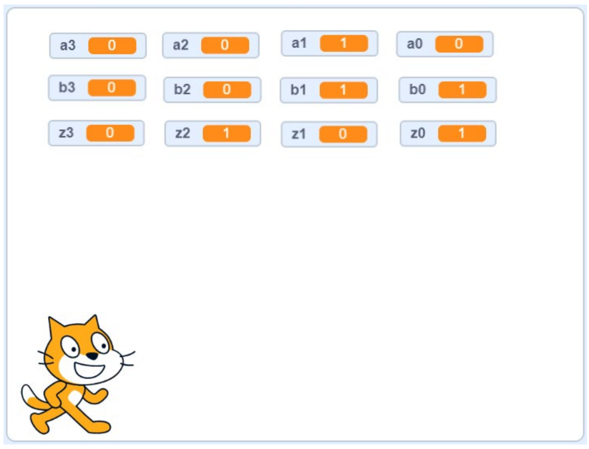
a3 1 a2 0 a1 1 a0 1

c2 0 c1 1 c0 1

Now try to do some practice on two's complement operation. On a piece of paper, write down the 4-bit binary number for each of the integers 0,1,2,3,4,5,6,7. In each case, apply the two's complement operation to determine the resulting 4-bit binary number for each of the integers -0,-1,-2,-3,-4,-5,-6,-7. Then run your program and verify whether the program returns the same result as you. If not, then you should determine whether you have a mistake on the paper or on your program and correct it.

Task 1.3 Binary Addition with 4-bit Numbers

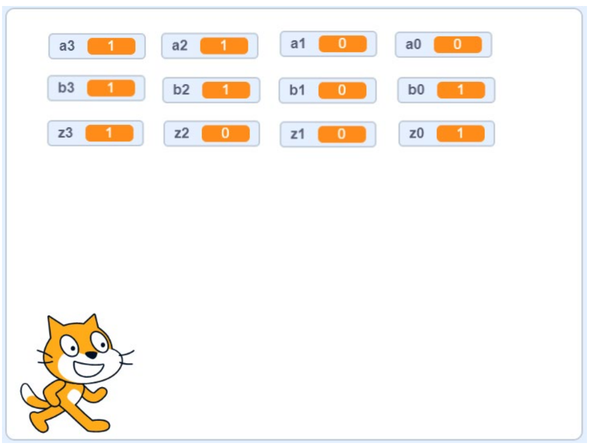
In this task we would like to first ask the user to input two 4-bit numbers and then show the resulting 4-bit sum by adding these 2 numbers. The following screenshot shows examples of the completed program:



a3 0 a2 0 a1 1 a0 0

b3 0 b2 0 b1 1 b0 1

z3 0 z2 1 z1 0 z0 1



a3 1 a2 1 a1 0 a0 0

b3 1 b2 1 b1 0 b0 1

z3 1 z2 0 z1 0 z0 1

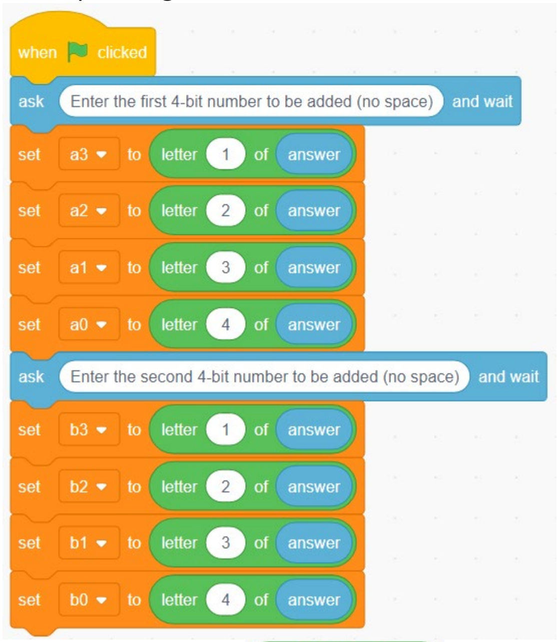
Task 1.3A Getting Input from User

We will ask the user to input a 4-bit number first, and store the resulting 4 bits to the variables $a3$, $a2$, $a1$, $a0$. Then we will ask the user to input another 4-bit number, and store the resulting 4 bits to the variables $b3$, $b2$, $b1$, $b0$.

1) Create a new project and put down the project name as Lab 03 Task 1.3.

2) Create the variables $a0$, $a1$, $a2$, $a3$, $b0$, $b1$, $b2$, $b3$. Rearrange these variable displays on the stage as shown in the above screenshot.

3) Compose the following block that will ask for the user input twice and store the bits to the corresponding variables:



```
when [flag] clicked
ask (Enter the first 4-bit number to be added (no space)) and wait
set a3 to (letter (1) of (answer))
set a2 to (letter (2) of (answer))
set a1 to (letter (3) of (answer))
set a0 to (letter (4) of (answer))
ask (Enter the second 4-bit number to be added (no space)) and wait
set b3 to (letter (1) of (answer))
set b2 to (letter (2) of (answer))
set b1 to (letter (3) of (answer))
set b0 to (letter (4) of (answer))
```

Note that the block `letter (1) of (apple)` would take one character from the second box at the position specified in the first box. The position starts at 1 for the leftmost character and increases from left to right. For example, the block `letter (1) of (apple)` would return the letter ‘a’ since it is the first character of the string “apple”. Also note that we need to explicitly remind the user not to add any space in the input otherwise the above code could not extract the correct bits. If you follow the left example screenshot shown on the previous page, then you would first input 0010 as the answer for the first question and then input 0011 as the answer for the second question.

Run your program to make sure that the bits you entered show up correctly in the variable displays *a3*, *a2*, *a1*, *a0*, *b3*, *b2*, *b1*, *b0*.

Task 1.3B Adding

Let us denote by *z3 z2 z1 z0* as the resulting 4 bits after adding *a3 a2 a1 a0* and *b3 b2 b1 b0*. Now let us carry out similar analysis as in Task 1.2B by considering the addition of one bit at a time from right to left.

1) At bit 0, in adding *a0* and *b0*, there are 4 possible combinations of these values as shown in the following table. Complete the following table and determine the resulting values of the sum $(a0+b0)_{10}$ in decimal as well as the corresponding binary representation with the bits *c0 z0*:

$$\begin{array}{cc} & a0 \\ \oplus & b0 \\ \hline c0 & z0 \end{array}$$

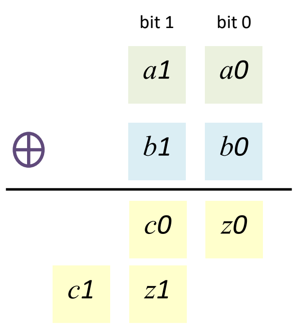
$a0$	$b0$	$(a0+b0)_{10}$	$c0$	$z0$
0	0			
0	1			
1	0			
1	1			

Now you need to write the code to get the correct value of *c0* and *z0* given *a0* and *b0*. Refer to Task 1.2A and apply similar logic to derive the equations for *c0* and *z0*.

c0 =

z0 =

2) At bit 1, we have $a1$ from the first number to be added, and $b1$ from the second number to be added. However, we also need to add the carry bit $c0$ that comes from the addition of bit 0. There are 8 possible combinations of $a1$, $b1$, $c0$. Complete the following table and determine the resulting values of the sum $(a1+b1+c0)_{10}$ in decimal as well as the corresponding binary representation with the bits $c1$ $z1$:



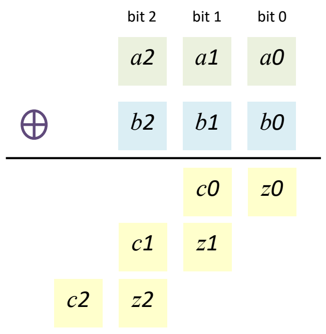
$a1$	$b1$	$c0$	$(a1+ b1+c0)_{10}$	$c1$	$z1$
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

Now you need to write the code to get the correct value of $c1$ and $z1$ given $a1$, $b1$ and $c0$. Refer to Task 1.2A and apply similar logic to derive the equations for $c1$ and $z1$.

$c1$ =

$z1$ =

3) At bit 2, we have $a2$ from the first number to be added, and $b2$ from the second number to be added. However, we also need to add the carry bit $c1$ that comes from the addition of bit 1. Complete the following table by listing all possible combinations of $a2$, $b2$, $c1$ and determining the resulting values of the sum $(a2+b2+c1)_{10}$ in decimal as well as the corresponding binary representation with the bits $c2$ $z2$:



$a2$	$b2$	$c1$	$(a2+ b2+c1)_{10}$	$c2$	$z2$

Now you need to write the code to get the correct value of $c2$ and $z2$ given $a2$, $b2$ and $c1$. Refer to Task 1.2A and apply similar logic to derive the equations for $c2$ and $z2$.

$c2$ =

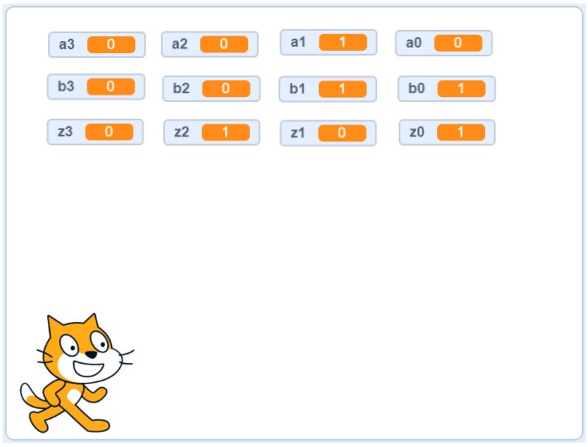
$z2$ =

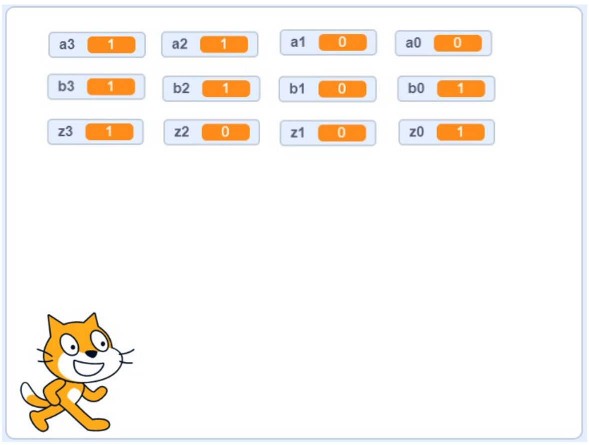
4) Can you determine the result of the addition at bit 3 and derive the equation for $z3$?

$z3$ =

Task 1.3C Completing the Code

Create any additional variables that you need and construct the blocks that implement the equations derived in Task 1.3B. Rearrange the variable display so that it is as shown in the following screenshots:





Now try to do some practice on binary addition. On a piece of paper, randomly choose two 4-bit numbers and add them by hand. Then run your program and verify whether the program returns the same result as you. If not, then you should determine whether you have a mistake on the paper or on the program and correct it. Repeat this several times until you are sure that your program runs correctly and that you have mastered the skill on binary addition.

For each case that you randomly generate, also determine if overflow occurs after the addition. If overflow does not occur in all your cases, then try to generate new cases in which overflow occurs.

Task 2 Complete the assessment from the Canvas course page

You should complete the Lab 03 Assessment from the Canvas course page before the posted deadline.

Task 3 Challenge your classmates

You can first reflect on what you have learnt in this lab, and then come up with problems to challenge your classmates. You can post your problem on the Canvas course page, under this Discussion page. One should be able to solve your problem by using what he/she learns in Lab 03. You will not get extra marks by posting a challenging problem or solving a challenging problem posted by another student, but you will earn your fame so that you can impress the course leader, the lab tutors, and your classmates.