# **CS 410 Binary to C++ Activity Template**

## **File One**

**Step 2:** Explain the functionality of the blocks of assembly code.

| **Blocks of Assembly Code** | **Explanation of Functionality** |
| --- | --- |
| push %rbp  mov %rsp,%rbp  sub $0x10,%rsp  movl $0x1,-0x8(%rbp)  cmpl $0x9,-0x8(%rbp)  jg a3 <main+0xa3>  movl $0x1,-0xc(%rbp)  cmpl $0x9,-0xc(%rbp)  jg 9a <main+0x9a>  mov -0x8(%rbp),%eax  imul -0xc(%rbp),%eax  mov %eax,-0x4(%rbp)  mov -0x8(%rbp),%eax  mov %eax,%esi  lea 0x0(%rip),%rdi # 3c <main+0x3c>  callq 41 <main+0x41>  lea 0x0(%rip),%rsi # 48 <main+0x48>  mov %rax,%rdi  callq 50 <main+0x50>  mov %rax,%rdx  mov -0xc(%rbp),%eax  mov %eax,%esi  mov %rdx,%rdi  callq 60 <main+0x60>  lea 0x0(%rip),%rsi # 67 <main+0x67>  mov %rax,%rdi  callq 6f <main+0x6f>  mov %rax,%rdx  mov -0x4(%rbp),%eax  mov %eax,%esi  mov %rdx,%rdi  callq 7f <main+0x7f>  mov %rax,%rdx  mov 0x0(%rip),%rax # 89 <main+0x89>  mov %rax,%rsi  mov %rdx,%rdi  callq 94 <main+0x94>  addl $0x1,-0xc(%rbp)  jmp 20 <main+0x20>  addl $0x1,-0x8(%rbp)  jmpq f <main+0xf>  mov $0x0,%eax  leaveq | Uses a nested for loop to iterate through and print 1\*1 through 1\*9. Continues with 2\*1 through 2\*9, this process continues until 9\*9 and then the program terminates. |

**Step 4:** Convert the assembly code to C++ code.

#include<iostream>

using namespace std;

int main()

{

  int number, i, a, x;

  for (a = 1; a <= 9; a++)

    {

        for (i = 1; i <= 9; i++){

          x = a \* i;

          cout << a << " \* " << i << " = " << x << endl;

        }

     }

  return 0;

}

**Step 5:** Explain how the C++ code performs the same tasks as the blocks of assembly code.

| **Blocks of Assembly Code** | **C++ Code** | **Explanation of Functionality** |
| --- | --- | --- |
| push %rbp  mov %rsp,%rbp  sub $0x10,%rsp  movl $0x1,-0x8(%rbp)  cmpl $0x9,-0x8(%rbp)  jg a3 <main+0xa3>  movl $0x1,-0xc(%rbp)  cmpl $0x9,-0xc(%rbp)  jg 9a <main+0x9a>  mov -0x8(%rbp),%eax  imul -0xc(%rbp),%eax  mov %eax,-0x4(%rbp)  mov -0x8(%rbp),%eax  mov %eax,%esi  lea 0x0(%rip),%rdi # 3c <main+0x3c>  callq 41 <main+0x41>    lea 0x0(%rip),%rsi # 48 <main+0x48>  mov %rax,%rdi  callq 50 <main+0x50>  mov %rax,%rdx  mov -0xc(%rbp),%eax  mov %eax,%esi  mov %rdx,%rdi  callq 60 <main+0x60>  lea 0x0(%rip),%rsi # 67 <main+0x67>  mov %rax,%rdi  callq 6f <main+0x6f>  mov %rax,%rdx  mov -0x4(%rbp),%eax  mov %eax,%esi  mov %rdx,%rdi  callq 7f <main+0x7f>   mov %rax,%rdx  mov 0x0(%rip),%rax # 89 <main+0x89>  mov %rax,%rsi  mov %rdx,%rdi  callq 94 <main+0x94>  addl $0x1,-0xc(%rbp)  jmp 20 <main+0x20>  addl $0x1,-0x8(%rbp)  jmpq f <main+0xf>  mov $0x0,%eax  leaveq  retq | int main()  {  int number, i, a, x;    for (a = 1; a <= 9; a++)  {  for (i = 1; i <= 9; i++){  x = a \* i;  cout << a << " \* " << i << " = " << x << endl;  }  }  return 0;  } | Initialize registers for variables  jg after cmpl means jump if greater (the for loops termination statements)  imul is our x = a \* I statement.  callq and load effective address lines refer to our cout << a << " \* " << i << " = " << x << endl;  and the variables referenced.  Each of them generally referencing the memory addresses for variables that were used in the calculations.  Finally when the for loops terminate, end the program. |

## **File Two**

**Step 2:** Explain the functionality of the blocks of assembly code.

| **Blocks of Assembly Code** | **Explanation of Functionality** |
| --- | --- |
| push %rbp  mov %rsp,%rbp  sub $0x30,%rsp  mov %fs:0x28,%rax  mov %rax,-0x8(%rbp)  xor %eax,%eax  lea 0x0(%rip),%rsi # 1e <main+0x1e>  lea 0x0(%rip),%rdi # 25 <main+0x25>  callq 2a <main+0x2a>  mov %rax,%rdx  mov 0x0(%rip),%rax # 34 <main+0x34>  mov %rax,%rsi  mov %rdx,%rdi  callq 3f <main+0x3f>  lea -0x14(%rbp),%rax  mov %rax,%rsi  lea 0x0(%rip),%rdi # 4d <main+0x4d>  callq 52 <main+0x52>  mov -0x14(%rbp),%edx  mov -0x14(%rbp),%eax | Initializing registers  xor statement initializes register eax  lea statements load effective address for variables  callq statements provide us with some cout and cin statements  mov the user provided value into register |
| imul %eax,%edx  mov -0x14(%rbp),%eax  imul %edx,%eax  mov %eax,-0x14(%rbp)  mov -0x14(%rbp),%eax  cvtsi2sd %eax,%xmm0  movsd 0x0(%rip),%xmm1 # 73 <main+0x73>  mulsd %xmm1,%xmm0  movsd %xmm0,-0x10(%rbp)  lea 0x0(%rip),%rsi # 83 <main+0x83>  lea 0x0(%rip),%rdi # 8a <main+0x8a>  callq 8f <main+0x8f>  mov %rax,%rdx  mov -0x10(%rbp),%rax  mov %rax,-0x28(%rbp)  movsd -0x28(%rbp),%xmm0  mov %rdx,%rdi  callq a7 <main+0xa7>  mov $0x0,%eax  mov -0x8(%rbp),%rcx  xor %fs:0x28,%rcx  je c0 <main+0xc0>  callq c0 <main+0xc0>  leaveq  retq | Several multiplication statements and moving to registers before cvtsi2sd line. This is our first hint at pi as cvtsi2sd is “Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value”  movsd moves a scalar double-precision floating-point value from the source operand (second operand) to the destination operand (first operand)  mulsd multiplies our double and moves the product into ,-0x10(%rbp)  the rest of these calls are for the cout statement which prints the value of our double multiplication, this is revealed as volume in the strings.  Finally statements to terminate the program |

**Step 4:** Convert the assembly code to C++ code.

**#include<iostream>**

**using namespace std;**

**int main(){**

**int number;**

**double volume;**

**cout << "Enter the Radius: " << endl;**

**cin >> number;**

**volume = 3.14\*number\*(number\*number);**

**cout << "\nThe volume is: " << volume;**

**return 1;**

**}**

**Step 5:** Explain how the C++ code performs the same tasks as the blocks of assembly code.

| **Blocks of Assembly Code** | **C++ Code** | **Explanation of Functionality** |
| --- | --- | --- |
| push %rbp  mov %rsp,%rbp  sub $0x30,%rsp  mov %fs:0x28,%rax  mov %rax,-0x8(%rbp)  xor %eax,%eax  lea 0x0(%rip),%rsi # 1e <main+0x1e>  lea 0x0(%rip),%rdi # 25 <main+0x25>  callq 2a <main+0x2a>  mov %rax,%rdx  mov 0x0(%rip),%rax # 34 <main+0x34>  mov %rax,%rsi  mov %rdx,%rdi  callq 3f <main+0x3f>  lea -0x14(%rbp),%rax  mov %rax,%rsi  lea 0x0(%rip),%rdi # 4d <main+0x4d>  callq 52 <main+0x52>  mov -0x14(%rbp),%edx  mov -0x14(%rbp),%eax | **int number;**  **double volume;**  **cout << "Enter the Radius: " << endl;**  **cin >> number;** | As stated above most of this is just initializing our variables, callq lines are cout and string calls and our final mov statements are our variable ‘number’ being moved into a register for our multiplication below |
| imul %eax,%edx  mov -0x14(%rbp),%eax  imul %edx,%eax  mov %eax,-0x14(%rbp)  mov -0x14(%rbp),%eax  cvtsi2sd %eax,%xmm0  movsd 0x0(%rip),%xmm1 # 73 <main+0x73>  mulsd %xmm1,%xmm0  movsd %xmm0,-0x10(%rbp)  lea 0x0(%rip),%rsi # 83 <main+0x83>  lea 0x0(%rip),%rdi # 8a <main+0x8a>  callq 8f <main+0x8f>  mov %rax,%rdx  mov -0x10(%rbp),%rax  mov %rax,-0x28(%rbp)  movsd -0x28(%rbp),%xmm0  mov %rdx,%rdi  callq a7 <main+0xa7>  mov $0x0,%eax  mov -0x8(%rbp),%rcx  xor %fs:0x28,%rcx  je c0 <main+0xc0>  callq c0 <main+0xc0>  leaveq  retq | **volume = 3.14\*number\*(number\*number);**  **cout << "\nThe volume is: " << volume;**  **return 1;** | (number\*number)\*number is what these imul statements are doing while cvtsi2sd makes sure the data types behave  movsd and mulsd multiply the product from the previous step and then commit our new product to memory, our variable ‘volume’  callq statements form our cout line and print the variable for volume |

## **File Three**

**Step 2:** Explain the functionality of the blocks of assembly code.

| **Blocks of Assembly Code** | **Explanation of Functionality** |
| --- | --- |
| push %rbp  mov %rsp,%rbp  sub $0x20,%rsp  mov %fs:0x28,%rax  mov %rax,-0x8(%rbp)  xor %eax,%eax  movl $0x1,-0xc(%rbp)  lea 0x0(%rip),%rsi # 25 <main+0x25>  lea 0x0(%rip),%rdi # 2c <main+0x2c>  callq 31 <main+0x31>  mov %rax,%rdx  mov 0x0(%rip),%rax # 3b <main+0x3b>  mov %rax,%rsi  mov %rdx,%rdi  callq 46 <main+0x46>  lea -0x18(%rbp),%rax  mov %rax,%rsi  lea 0x0(%rip),%rdi # 54 <main+0x54>  callq 59 <main+0x59>  mov -0x18(%rbp),%eax  sub $0x1,%eax  mov %eax,-0xc(%rbp)  movl $0x1,-0x10(%rbp)  mov -0x18(%rbp),%eax | Initialize some variables and prompt the user to enter a number of rows. Value of ‘rows’ is then saved. |
| cmp %eax,-0x10(%rbp)  jg e3 <main+0xe3>  movl $0x1,-0x14(%rbp)  mov -0x14(%rbp),%eax  cmp -0xc(%rbp),%eax  jg 99 <main+0x99>  lea 0x0(%rip),%rsi # 87 <main+0x87>  lea 0x0(%rip),%rdi # 8e <main+0x8e>  callq 93 <main+0x93>  addl $0x1,-0x14(%rbp)  jmp 78 <main+0x78>  subl $0x1,-0xc(%rbp)  movl $0x1,-0x14(%rbp)  mov -0x10(%rbp),%eax  add %eax,%eax  sub $0x1,%eax  cmp %eax,-0x14(%rbp)  jg ca <main+0xca>  lea 0x0(%rip),%rsi # b8 <main+0xb8>  lea 0x0(%rip),%rdi # bf <main+0xbf>  callq c4 <main+0xc4>  addl $0x1,-0x14(%rbp)  jmp a4 <main+0xa4>  lea 0x0(%rip),%rsi # d1 <main+0xd1>  lea 0x0(%rip),%rdi # d8 <main+0xd8>  callq dd <main+0xdd>  addl $0x1,-0x10(%rbp)  jmp 69 <main+0x69>  movl $0x1,-0xc(%rbp)  movl $0x1,-0x10(%rbp)  mov -0x18(%rbp),%eax  sub $0x1,%eax  cmp %eax,-0x10(%rbp)  jg 171 <main+0x171>  movl $0x1,-0x14(%rbp)  mov -0x14(%rbp),%eax  cmp -0xc(%rbp),%eax  jg 124 <main+0x124>  lea 0x0(%rip),%rsi # 112 <main+0x112>  lea 0x0(%rip),%rdi # 119 <main+0x119>  callq 11e <main+0x11e>  addl $0x1,-0x14(%rbp)  jmp 103 <main+0x103>  addl $0x1,-0xc(%rbp)  movl $0x1,-0x14(%rbp)  mov -0x18(%rbp),%eax  sub -0x10(%rbp),%eax  add %eax,%eax  sub $0x1,%eax  cmp %eax,-0x14(%rbp)  jg 158 <main+0x158>  lea 0x0(%rip),%rsi # 146 <main+0x146>  lea 0x0(%rip),%rdi # 14d <main+0x14d>  callq 152 <main+0x152>  addl $0x1,-0x14(%rbp)  jmp 12f <main+0x12f>  lea 0x0(%rip),%rsi # 15f <main+0x15f>  lea 0x0(%rip),%rdi # 166 <main+0x166>  callq 16b <main+0x16b>  addl $0x1,-0x10(%rbp)  jmp f1 <main+0xf1>  mov $0x1,%eax  mov -0x8(%rbp),%rcx  xor %fs:0x28,%rcx  je 18a <main+0x18a>  callq 18a <main+0x18a>  leaveq  retq | These are a series of nested for loops.  jg lines signify a ‘jump if greater’ and all of these calls are cout statements printing the design based on the specified value for rows.  Each of these add/subtract is related to the for loops, controlling the iterations to produce a design based on rows, first the top half of the triangle and then the bottom. |

**Step 4:** Convert the assembly code to C++ code.

**#include <iostream>;**

**using namespace std;**

**int main() {**

**int i, j, rows;**

**cout << "Enter number of rows" << endl;**

**cin >> rows;**

**for (i = 0; i <= rows; i++) {**

**for (j = 1; j <= rows - i; j++)**

**cout << " ";**

**for (j = 1; j <= 2 \* i - 1; j++)**

**cout << "\*";**

**cout << endl;**

**}**

**for (i = rows - 1; i >= 1; i--) {**

**for (j = 1; j <= rows - i; j++)**

**cout << " ";**

**for (j = 1; j <= 2 \* i - 1; j++)**

**cout << "\*";**

**cout << endl;**

**}**

**}**

**Step 5:** Explain how the C++ code performs the same tasks as the blocks of assembly code.

| **Blocks of Assembly Code** | **C++ Code** | **Explanation of Functionality** |
| --- | --- | --- |
| push %rbp  mov %rsp,%rbp  sub $0x20,%rsp  mov %fs:0x28,%rax  mov %rax,-0x8(%rbp)  xor %eax,%eax  movl $0x1,-0xc(%rbp)  lea 0x0(%rip),%rsi # 25 <main+0x25>  lea 0x0(%rip),%rdi # 2c <main+0x2c>  callq 31 <main+0x31>  mov %rax,%rdx  mov 0x0(%rip),%rax # 3b <main+0x3b>  mov %rax,%rsi  mov %rdx,%rdi  callq 46 <main+0x46>  lea -0x18(%rbp),%rax  mov %rax,%rsi  lea 0x0(%rip),%rdi # 54 <main+0x54>  callq 59 <main+0x59>  mov -0x18(%rbp),%eax  sub $0x1,%eax  mov %eax,-0xc(%rbp)  movl $0x1,-0x10(%rbp)  mov -0x18(%rbp),%eax | **int i, j, rows;**  **cout << "Enter number of rows" << endl;**  **cin >> rows;** | Most of this is variable initialization and prompting the user. Most importantly obtaining a value for ‘rows’ |
| cmp %eax,-0x10(%rbp)  jg e3 <main+0xe3>  movl $0x1,-0x14(%rbp)  mov -0x14(%rbp),%eax  cmp -0xc(%rbp),%eax  jg 99 <main+0x99>  lea 0x0(%rip),%rsi # 87 <main+0x87>  lea 0x0(%rip),%rdi # 8e <main+0x8e>  callq 93 <main+0x93>  addl $0x1,-0x14(%rbp)  jmp 78 <main+0x78>  subl $0x1,-0xc(%rbp)  movl $0x1,-0x14(%rbp)  mov -0x10(%rbp),%eax  add %eax,%eax  sub $0x1,%eax  cmp %eax,-0x14(%rbp)  jg ca <main+0xca>  lea 0x0(%rip),%rsi # b8 <main+0xb8>  lea 0x0(%rip),%rdi # bf <main+0xbf>  callq c4 <main+0xc4>  addl $0x1,-0x14(%rbp)  jmp a4 <main+0xa4>  lea 0x0(%rip),%rsi # d1 <main+0xd1>  lea 0x0(%rip),%rdi # d8 <main+0xd8>  callq dd <main+0xdd>  addl $0x1,-0x10(%rbp)  jmp 69 <main+0x69>  movl $0x1,-0xc(%rbp)  movl $0x1,-0x10(%rbp)  mov -0x18(%rbp),%eax  sub $0x1,%eax  cmp %eax,-0x10(%rbp)  jg 171 <main+0x171>  movl $0x1,-0x14(%rbp)  mov -0x14(%rbp),%eax  cmp -0xc(%rbp),%eax  jg 124 <main+0x124>  lea 0x0(%rip),%rsi # 112 <main+0x112>  lea 0x0(%rip),%rdi # 119 <main+0x119>  callq 11e <main+0x11e>  addl $0x1,-0x14(%rbp)  jmp 103 <main+0x103>  addl $0x1,-0xc(%rbp)  movl $0x1,-0x14(%rbp)  mov -0x18(%rbp),%eax  sub -0x10(%rbp),%eax  add %eax,%eax  sub $0x1,%eax  cmp %eax,-0x14(%rbp)  jg 158 <main+0x158>  lea 0x0(%rip),%rsi # 146 <main+0x146>  lea 0x0(%rip),%rdi # 14d <main+0x14d>  callq 152 <main+0x152>  addl $0x1,-0x14(%rbp)  jmp 12f <main+0x12f>  lea 0x0(%rip),%rsi # 15f <main+0x15f>  lea 0x0(%rip),%rdi # 166 <main+0x166>  callq 16b <main+0x16b>  addl $0x1,-0x10(%rbp)  jmp f1 <main+0xf1>  mov $0x1,%eax  mov -0x8(%rbp),%rcx  xor %fs:0x28,%rcx  je 18a <main+0x18a>  callq 18a <main+0x18a>  leaveq  retq | for (i = 0; i <= rows; i++) {  for (j = 1; j <= rows - i; j++)  cout << " ";  for (j = 1; j <= 2 \* i - 1; j++)  cout << "\*";  cout << endl;  }  for (i = rows - 1; i >= 1; i--) {  for (j = 1; j <= rows - i; j++)  cout << " ";  for (j = 1; j <= 2 \* i - 1; j++)  cout << "\*";  cout << endl;  } | We can link each cmp line to our for loop and each math function is doing something with an iterator value  After the top triangle is printed, the program continues to printing the smaller, lower triangle.  Once all the for loops have run, a final compare brings us to program termination. |

## **File Four**

**Step 2:** Explain the functionality of the blocks of assembly code.

| **Blocks of Assembly Code** | **Explanation of Functionality** |
| --- | --- |
| push %rbp  mov %rsp,%rbp  sub $0x30,%rsp  mov %fs:0x28,%rax  mov %rax,-0x8(%rbp)  xor %eax,%eax  movq $0x0,-0x20(%rbp)  movq $0x1,-0x18(%rbp)  lea 0x0(%rip),%rsi # 2e <main+0x2e>  lea 0x0(%rip),%rdi # 35 <main+0x35>  callq 3a <main+0x3a>  mov %rax,%rdx  mov 0x0(%rip),%rax # 44 <main+0x44>  mov %rax,%rsi  mov %rdx,%rdi  callq 4f <main+0x4f>  lea -0x28(%rbp),%rax  mov %rax,%rsi  lea 0x0(%rip),%rdi # 5d <main+0x5d>  callq 62 <main+0x62>  mov -0x28(%rbp),%rax | Initializing several variables and we see a cout statement prompting a user for a variable, which it saves |
| test %rax,%rax  je f2 <main+0xf2>  mov -0x28(%rbp),%rcx  movabs $0x6666666666666667,%rdx  mov %rcx,%rax  imul %rdx  sar $0x2,%rdx  mov %rcx,%rax  sar $0x3f,%rax  sub %rax,%rdx  mov %rdx,%rax  mov %rax,-0x10(%rbp)  mov -0x10(%rbp),%rdx  mov %rdx,%rax  shl $0x2,%rax  add %rdx,%rax  add %rax,%rax  sub %rax,%rcx  mov %rcx,%rax  mov %rax,-0x10(%rbp)  mov -0x10(%rbp),%rax  imul -0x18(%rbp),%rax  add %rax,-0x20(%rbp)  shlq -0x18(%rbp)  mov -0x28(%rbp),%rcx  movabs $0x6666666666666667,%rdx  mov %rcx,%rax  imul %rdx  sar $0x2,%rdx  mov %rcx,%rax  sar $0x3f,%rax  sub %rax,%rdx  mov %rdx,%rax  mov %rax,-0x28(%rbp)  jmpq 62 <main+0x62>  lea 0x0(%rip),%rsi # f9 <main+0xf9>  lea 0x0(%rip),%rdi # 100 <main+0x100>  callq 105 <main+0x105>  mov %rax,%rdx  mov -0x20(%rbp),%rax  mov %rax,%rsi  mov %rdx,%rdi  callq 117 <main+0x117>  mov %rax,%rdx  mov 0x0(%rip),%rax # 121 <main+0x121>  mov %rax,%rsi  mov %rdx,%rdi  callq 12c <main+0x12c>  mov $0x0,%eax  mov -0x8(%rbp),%rsi  xor %fs:0x28,%rsi  je 145 <main+0x145>  callq 145 <main+0x145>  leaveq  retq | test and je lines test a bool and determine whether to terminate the while loop, one of our variables acts as a base for converting binary. Sar, shl and shlq lines are bitshift operators which modify the last digit read and user prompted binary value.  imul modifies the base and decimalValue  when the while loop terminates (because all of the user binary number has been converted) We jump to the final cout line, giving the user our decimal value(the string in the code incorrectly states hexadecimal value) |

**Step 4:** Convert the assembly code to C++ code.

**#include <iostream>**

**using namespace std;**

**int main() {**

**int binNum;**

**int decValue = 0;**

**//This will handle base value, like 2^0**

**int base = 1;**

**cout << "Enter the binary number: " << endl;**

**cin >> binNum;**

**while (binNum) {**

**int lastDigit = binNum % 10;**

**binNum = binNum / 10;**

**decValue += lastDigit \* base;**

**base = base \* 2;**

**}**

**cout << "Equivalent decimal value: " << decValue;**

**}**

**Step 5:** Explain how the C++ code performs the same tasks as the blocks of assembly code.

| **Blocks of Assembly Code** | **C++ Code** | **Explanation of Functionality** |
| --- | --- | --- |
| push %rbp  mov %rsp,%rbp  sub $0x30,%rsp  mov %fs:0x28,%rax  mov %rax,-0x8(%rbp)  xor %eax,%eax  movq $0x0,-0x20(%rbp)  movq $0x1,-0x18(%rbp)  lea 0x0(%rip),%rsi # 2e <main+0x2e>  lea 0x0(%rip),%rdi # 35 <main+0x35>  callq 3a <main+0x3a>  mov %rax,%rdx  mov 0x0(%rip),%rax # 44 <main+0x44>  mov %rax,%rsi  mov %rdx,%rdi  callq 4f <main+0x4f>  lea -0x28(%rbp),%rax  mov %rax,%rsi  lea 0x0(%rip),%rdi # 5d <main+0x5d>  callq 62 <main+0x62>  mov -0x28(%rbp),%rax | **int binNum;**  **int decValue = 0;**  **int base = 1;**  **cout << "Enter the binary number: " << endl;**  **cin >> binNum;** | Variable initialization and call lines are cout prompt and user input via cin statement |
| test %rax,%rax  je f2 <main+0xf2>  mov -0x28(%rbp),%rcx  movabs $0x6666666666666667,%rdx  mov %rcx,%rax  imul %rdx  sar $0x2,%rdx  mov %rcx,%rax  sar $0x3f,%rax  sub %rax,%rdx  mov %rdx,%rax  mov %rax,-0x10(%rbp)  mov -0x10(%rbp),%rdx  mov %rdx,%rax  shl $0x2,%rax  add %rdx,%rax  add %rax,%rax  sub %rax,%rcx  mov %rcx,%rax  mov %rax,-0x10(%rbp)  mov -0x10(%rbp),%rax  imul -0x18(%rbp),%rax  add %rax,-0x20(%rbp)  shlq -0x18(%rbp)  mov -0x28(%rbp),%rcx  movabs $0x6666666666666667,%rdx  mov %rcx,%rax  imul %rdx  sar $0x2,%rdx  mov %rcx,%rax  sar $0x3f,%rax  sub %rax,%rdx  mov %rdx,%rax  mov %rax,-0x28(%rbp)  jmpq 62 <main+0x62>  lea 0x0(%rip),%rsi # f9 <main+0xf9>  lea 0x0(%rip),%rdi # 100 <main+0x100>  callq 105 <main+0x105>  mov %rax,%rdx  mov -0x20(%rbp),%rax  mov %rax,%rsi  mov %rdx,%rdi  callq 117 <main+0x117>  mov %rax,%rdx  mov 0x0(%rip),%rax # 121 <main+0x121>  mov %rax,%rsi  mov %rdx,%rdi  callq 12c <main+0x12c>  mov $0x0,%eax  mov -0x8(%rbp),%rsi  xor %fs:0x28,%rsi  je 145 <main+0x145>  callq 145 <main+0x145>  leaveq  retq | **while (binNum) {**  **int lastDigit = binNum % 10;**  **binNum = binNum / 10;**  **decValue += lastDigit \* base;**  **base = base \* 2;** | We start with our while loop testing for a Boolean.  lastDigit tracks the last digit which was handled in the binary string  The bit shift lines are shifting the user binary number to divide and (I’m pretty sure modulus)  Add and mul statements are modifying our decValue and base.  When the loop terminates (binNum = binNum/10) we give the user the converted value and terminate the program. |