# **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** |
| --- | --- |
| 1. 0x0000000000000000 <+0>:   push %rbp   1. 0x0000000000000001 <+1>:   mov %rsp,%rbp   1. 0x0000000000000004 <+4>:   sub $0x10,%rsp | 1. This instruction saves the base pointer of the previous stack frame by pushing it onto the stack 2. This sets the base pointer for the current stack frame, it moves the stack pointer into the base pointer register 3. This line subtracts the hexadecimal value 0x10 (16 in base 10 or decimal form) from the stack pointer register, allocating 16 bytes of space on the stack to be used for local variables. |
| 1. 0x0000000000000008 <+8>:   movl $0x1,-0x8(%rbp)   1. 0x000000000000000f <+15>:   cmpl $0x9,-0x8(%rbp)   1. 0x0000000000000013 <+19>:   jg 0xa3 <main+163> | 1. Moves the value %0x1, which is 1 in decimal form, to the location   -0x8(%rbp), an -8 offset or 8 bytes above the base pointer register. This will be the control variable for the outer loop in the main function.   1. This instruction compares the value at   -0x8(%rbp) with the value 9.   1. Based on the previous comparison, this instruction jumps the program’s execution to the address 0xa3 (line 163 of the main function) if the value at -0x8(%rbp) is greater than 9. Otherwise, the execution continues on to the next instruction. |
| 1. 0x0000000000000019 <+25>:   movl $0x1,-0xc(%rbp)   1. 0x0000000000000020 <+32>:   cmpl $0x9,-0xc(%rbp)   1. 0x0000000000000024 <+36>:   jg 0x9a <main+154> | 1. This instruction moves the value 1 into the location -0xc(%rbp), (0xc converted to decimal form is 12), an -12 offset or 12 bytes above the base pointer register. This will be the control variable for the inner for loop, the variable “i”. 2. This instruction compares the value at -0xc(%rbp) with the value of 9. 3. Based on the previous comparison, this instruction jumps the program’s execution to the address 0x9a (line 154 of the main function) if the value at -0xc(%rbp) is greater than 9. Otherwise, the execution continues on to the next instruction. |
| 1. 0x0000000000000026 <+38>:   mov -0x8(%rbp),%eax   1. 0x0000000000000029 <+41>:   imul -0xc(%rbp),%eax   1. 0x000000000000002d <+45>:   mov %eax,-0x4(%rbp) | 1. This instruction moves the value located at -0x8(%rbp) (the variable “a”) into the %eax register to prepare for the following multiplication. 2. This instruction multiplies the value stored at the location -0xc(%rbp), (the variable “i”) with the variable “a” that was stored in the %eax register and takes the product and stores this value back into the %eax register. 3. This instruction moves the product from the previous multiplication step from the %eax register to the location -0x4(%rbp), an -4 offset or 4 bytes above the base pointer register. -0x4(%rbp) is the variable “x” |
| 1. 0x0000000000000030 <+48>:   mov -0x8(%rbp),%eax   1. 0x0000000000000033 <+51>:   mov %eax,%esi   1. 0x0000000000000035 <+53>:   lea 0x0(%rip),%rdi # 0x3c <main+60>   1. 0x000000000000003c <+60>:   callq 0x41 <main+65>   1. 0x0000000000000041 <+65>:   lea 0x0(%rip),%rsi # 0x48 <main+72>   1. 0x0000000000000048 <+72>:   mov %rax,%rdi   1. 0x000000000000004b <+75>:   callq 0x50 <main+80>   1. 0x0000000000000050 <+80>:   mov %rax,%rdx   1. 0x0000000000000053 <+83>:   mov -0xc(%rbp),%eax   1. 0x0000000000000056 <+86>:   mov %eax,%esi   1. 0x0000000000000058 <+88>:   mov %rdx,%rdi   1. 0x000000000000005b <+91>:   callq 0x60 <main+96> | 1. This instruction moves the value stored at -0x8(%rbp) into the %eax register to prepare it for use with the output function call (callq). 2. This line moves the value previously stored in the %eax register (the variable “a”) into the %esi register, a second argument register that is used in the function call to output the variable “a”. 3. This loads the effective address of 0x0(%rip), which is the next instruction, into the %rdi register, the first argument register. 4. This calls the address 0x41, the function to output the variable “a”. 5. This loads the effective address of the next instruction 0x0(%rip) into the %rsi register. 6. The value stored in the %rax register is the return value from the previous function call. This value is moved into the %rdi register, the first argument register. 7. This instruction calls the function at address 0x50 to print the string “ \* “. 8. This moves the return value of the previous function call into the %rdx register. 9. This moves the value stored at -0xc(%rbp), the variable “i”, into the %eax register for use with the output function call. 10. This moves the previously stored value located in the %eax register into the %esi register, the second argument register to be used with the call to output. 11. This moves the value stored in the %rdx register into the %rsi register for the next call to output. 12. This calls the function located at the address 0x60 and outputs the variable “i”. |
| 1. 0x0000000000000060 <+96>:   lea 0x0(%rip),%rsi # 0x67 <main+103>   1. 0x0000000000000067 <+103>:   mov %rax,%rdi   1. 0x000000000000006a <+106>:   callq 0x6f <main+111>   1. 0x000000000000006f <+111>:   mov %rax,%rdx   1. 0x0000000000000072 <+114>:   mov -0x4(%rbp),%eax   1. 0x0000000000000075 <+117>:   mov %eax,%esi   1. 0x0000000000000077 <+119>:   mov %rdx,%rdi   1. 0x000000000000007a <+122>:   callq 0x7f <main+127>   1. 0x000000000000007f <+127>:   mov %rax,%rdx   1. 0x0000000000000082 <+130>:   mov 0x0(%rip),%rax # 0x89 <main+137>   1. 0x0000000000000089 <+137>:   mov %rax,%rsi   1. 0x000000000000008c <+140>:   mov %rdx,%rdi   1. 0x000000000000008f <+143>:   callq 0x94 <main+148> | 1. This loads the effective address of the next instruction located at 0x0(%rip) into the %rsi register to prepare for the next output function call. 2. This moves the return value from the previous function call into the %rsi register for the next output function call. 3. This calls the function at the address 0x6f to print the string “ = “. 4. This moves the return value of the previous function call into the %rdx register. 5. This moves the value stored at -0x4(%rbp), the variable “x”, into the %eax register to prepare it for use in the next output function call. 6. This moves the value previously stored in the %eax register (the variable “x”) into the %esi register for use in the next output call. 7. This moves the value stored in the %rdx register into the %rdi register. 8. This calls the function at the address 0x7f to output the variable “x”. 9. This instruction moves the return value from the previous function call (%rax) register into the %rdi register. 10. This moves the value at the address 0x0(%rip) into the %rax register. 11. The value previously stored in the %rax register is then moved into the %rsi register to prepare for the next output function call. 12. The value stored in the %rdx register is moved to the %rdi register also for the next output function call. 13. Calls the function to output the new line. |
| 1. 0x0000000000000094 <+148>:   addl $0x1,-0xc(%rbp)   1. 0x0000000000000098 <+152>:   jmp 0x20 <main+32> | 1. This instruction increments the control variable, i, for the second for loop or inner for loop by adding the value 1 to the value at -0xc(%rbp) and storing it there. 2. This instruction then jumps to the address 0x20, line 32 of the main function, which is the instruction for the comparison to be made for the inner loop to see whether it should exit the inner loop or not. |
| 1. 0x000000000000009a <+154>:   addl $0x1,-0x8(%rbp)   1. 0x000000000000009e <+158>:   jmpq 0xf <main+15> | 1. This instruction increments the control variable, a, for the first for loop or the outer for loop by adding the value 1 to the value located at -0x8(%rbp) and stores it there. 2. This instruction jumps to the address 0xf, line 15 of the main function, which is the instruction for the comparison to be made for the outer loop to determine whether the outer loop should be exited or not. |
| 1. 0x00000000000000a3 <+163>:   mov $0x0,%eax   1. 0x00000000000000a8 <+168>:   leaveq   1. 0x00000000000000a9 <+169>:   retq | 1. Sets the return value for the main function as 0 by moving the value 0 into the %eax register. 2. Restores the previous stack frame. 3. Returns from the main function. |

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

**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** |
| --- | --- | --- |
| 1. 0x0000000000000000 <+0>:   push %rbp   1. 0x0000000000000001 <+1>:   mov %rsp,%rbp | int main() { | Declares a function named main of type int, the entry point for the program. We see this accomplished in the assembly code by the new stack frame being set up for the main function. |
| 1. 0x0000000000000004 <+4>:   sub $0x10,%rsp | int number, i, a, x; | Declares variables “number”, “i”, “a”, and “x” of data type integer. 0x10 in decimal for is 16 bytes. 4 bytes for each variable is allocated. |
| 1. 0x0000000000000008 <+8>:   movl $0x1,-0x8(%rbp)   1. 0x000000000000000f <+15>:   cmpl $0x9,-0x8(%rbp)   1. 0x0000000000000013 <+19>:   jg 0xa3 <main+163> | for (a=1; a<=9; a++) { | This is the beginning of the outer for loop. Line 4 of the assembly code is initializing the variable “a” to be the value 1. Line 5 makes the comparison to determine whether the outer for loop should be exited. Line 6 is the instruction that says if the control variable for the for loop is greater than 9, the loop should be exited and jump to the next instruction after this loop, which is the return statement. Once the code inside this for loop has been executed, the control variable will be incremented by 1 and the comparison will be made again to determine if the for loop should be exited. |
| 1. 0x0000000000000019 <+25>:   movl $0x1,-0xc(%rbp)   1. 0x0000000000000020 <+32>:   cmpl $0x9,-0xc(%rbp)   1. 0x0000000000000024 <+36>:   jg 0x9a <main+154> | for (i = 1; i <=9; i++) { | This is the beginning of the inner for loop. Line 7 of the assembly code is initializing the variable “i” to be the value 1. Line 8 makes the comparison to determine whether the inner for loop should be exited. Line 9 is the instruction that says if the control variable for the inner for loop is greater than 9, the loop should be exited and jump to the next instruction after this loop, which is line 154 and is the instruction that increments the outer for loops control variable (“a”) by the value of 1. Once the code inside this for loop has been executed, the control variable will be incremented by 1 and the comparison will be made again to determine if the inner for loop should be exited. |
| 1. 0x0000000000000026 <+38>:   mov -0x8(%rbp),%eax   1. 0x0000000000000029 <+41>:   imul -0xc(%rbp),%eax   1. 0x000000000000002d <+45>:   mov %eax,-0x4(%rbp) | x = a \* i; | We see here, to get the product for the multiplication of the variables a \* i, the variable “a” is first moved to the %eax register so a calculation can be performed using its value. Then the multiplication is performed with the variable “i” and temporarily stored in the %eax register where this result is then moved and stored into the variable x.  For each iteration of the inner loop, this calculation will be performed. |
| The above lines 13 - 37 | cout << a << “ \* “ << i << “ = “ << x << endl; | This statement is made up of 6 output statements written as one. The above lines (13 – 37) contain the instructs to output each of the six segments for this one cout statement, effectively printing the variable “a”, the string “ \* “, the variable “i”, the string “ = “, the variable “x”, and lastly a newline to output. This statement is also executed during every iteration of the inner loop. |
| 1. 0x0000000000000094 <+148>:   addl $0x1,-0xc(%rbp)   1. 0x0000000000000098 <+152>:   jmp 0x20 <main+32> | i++; | This is part of the inner for loop where the control variable is being incremented by one.  The code then jumps to check the condition of the inner for loop. |
| 1. 0x000000000000009a <+154>:   addl $0x1,-0x8(%rbp)   1. 0x000000000000009e <+158>:   jmpq 0xf <main+15> | a++; | This is the portion of the outer for loop where the control variable is being incremented by one.  The code then jumps back to check the condition of the outer loop. |
| 1. 0x00000000000000a3 <+163>: mov $0x0,%eax 2. 0x00000000000000a8 <+168>: leaveq 3. 0x00000000000000a9 <+169>: retq | return 0; | Returns 0 because the main function is of type integer and expects an integer value to be returned. Returning a 0 value indicates to the operating system that the program exited without any errors. |

## **File Two**

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

| **Blocks of Assembly Code** | **Explanation of Functionality** |
| --- | --- |
| 1. 0x0000000000000000 <+0>:   push %rbp   1. 0x0000000000000001 <+1>:   mov %rsp,%rbp   1. 0x0000000000000004 <+4>:   sub $0x30,%rsp | 1. This instruction saves the base pointer of the previous stack frame by pushing it onto the stack 2. This sets the base pointer for the current stack frame, it moves the stack pointer into the base pointer register 3. This line subtracts the hexadecimal value 0x30 (48 in base 10 or decimal form) from the stack pointer register, allocating 48 bytes of space on the stack to be used for local variables. These variables include variables pi, radius, and volume. |
| 1. 0x0000000000000008 <+8>:   mov %fs:0x28,%rax   1. 0x0000000000000011 <+17>:   mov %rax,-0x8(%rbp)   1. 0x0000000000000015 <+21>:   xor %eax,%eax   1. 0x0000000000000017 <+23>:   lea 0x0(%rip),%rsi # 0x1e <main+30>   1. 0x000000000000001e <+30>:   lea 0x0(%rip),%rdi # 0x25 <main+37>   1. 0x0000000000000025 <+37>:   callq 0x2a <main+42> | 1. Moves the value stored at the FS segment register offset 0x28 into the %rax register. 2. Stores the value in the %rax register into the location -0x8(%rbp), -8 offset or 8 bytes above the base pointer register. 3. Clears the %eax register by xor-ing the register by itself which results in a false or zero value. 4. This loads the effective address 0x0(%rip), the location of the string “Enter radius:”, into the %rsi register to prepare it for the output function call. 5. This loads the effective address 0x0(%rip), the format specifier for input, into the %rdi register. 6. Calls the function to output the string: “Enter radius: “. |
| 1. 0x000000000000002a <+42>:   mov %rax,%rdx   1. 0x000000000000002d <+45>:   mov 0x0(%rip),%rax # 0x34 <main+52>   1. 0x0000000000000034 <+52>:   mov %rax,%rsi   1. 0x0000000000000037 <+55>:   mov %rdx,%rdi   1. 0x000000000000003a <+58>:   callq 0x3f <main+63>   1. 0x000000000000003f <+63>:   lea -0x14(%rbp),%rax   1. 0x0000000000000043 <+67>:   mov %rax,%rsi   1. 0x0000000000000046 <+70>:   lea 0x0(%rip),%rdi # 0x4d <main+77>   1. 0x000000000000004d <+77>:   callq 0x52 <main+82> | 1. Moves the value stored in the %rax register into the %rdx register. 2. Moves the instruction located at 0x0(%rip) into the %rax register 3. This instruction moves the value stored in the %rax register into the %rsi register, which is the second argument register used for the input function call. 4. This instruction moves the value stored in the %rdx register into the %rdi register, which is the first argument register for the input function call. 5. Calls the function stored at the address 0x3f in preparation for the input call. 6. Loads the effective address -0x14(%rbp), an -14 offset or 14 bytes above the base pointer register, into the %rax register. This is the variable used for radius. 7. This moves the value previously stored into the %rax register into the %rsi register to prepare it for use with the input function call. 8. This loads the effective address 0x0(%rip), the format specifier for reading the input, into the %rdi register. 9. This calls the function to read input for the radius variable. |
| 1. 0x0000000000000052 <+82>:   mov -0x14(%rbp),%edx   1. 0x0000000000000055 <+85>:   mov -0x14(%rbp),%eax   1. 0x0000000000000058 <+88>:   imul %eax,%edx   1. 0x000000000000005b <+91>:   mov -0x14(%rbp),%eax   1. 0x000000000000005e <+94>:   imul %edx,%eax   1. 0x0000000000000061 <+97>:   mov %eax,-0x14(%rbp) | 1. This instruction moves the value stored at -0x14(%rbp) into the %edx register (the radius variable) to prepare it for use. 2. This instruction moves the value stored at -0x14(%rbp) into the %edx register (the radius variable) to prepare it for use. We need two instances of this to be able to multiply these together, which is why we are putting this value into two registers. 3. This instruction multiplies the value stored in the %eax register by the value stored in the %edx register, then stores this value back into the %edx register. 4. This loads the value of -0x14(%rbp) into the %eax register again for another use. 5. This instruction multiplies the previously stored value in the %edx register by the value stored in the %eax register and stores the product back into the %eax register. 6. This instruction moves the previously stored value in the %eax register back into the variable location for radius, -0x14(%rbp). We have now effectively multiplied the initial value of radius by itself twice, giving us radius to the power of 3. |
| 1. 0x0000000000000064 <+100>:   mov -0x14(%rbp),%eax   1. 0x0000000000000067 <+103>:   cvtsi2sd %eax,%xmm0   1. 0x000000000000006b <+107>:   movsd 0x0(%rip),%xmm1 # 0x73 <main+115>   1. 0x0000000000000073 <+115>:   mulsd %xmm1,%xmm0   1. 0x0000000000000077 <+119>:   movsd %xmm0,-0x10(%rbp) | 1. This instruction moves the new value stored at location -0x14(%rbp), the variable radius^3), into the %eax register to prepare for use. 2. This instruction converts the value stored in the %eax register to a double and stores it in xmmo. One purpose for xmmo registers are for scientific calculations, which in this case is going to be double-precision calculations. 3. The movsd instruction is used to move a single (64-bit) double word value from a source operand to destination operand. Here we are loading the value of the variable pi into the xmm1 register to prepare it for the following calulation. 4. This multiplies the values stores in %xmm0 by the value stored in %xmm1 and stores the product back into the %xmm0 register. 5. This moves the previously calculated value stored in %xmm0 into the location -0x10(%rbp), an -10 offset or 10 bytes above the base pointer register. This is the variable volume. |
| 1. 0x000000000000007c <+124>:   lea 0x0(%rip),%rsi # 0x83 <main+131>   1. 0x0000000000000083 <+131>:   lea 0x0(%rip),%rdi # 0x8a <main+138>   1. 0x000000000000008a <+138>:   callq 0x8f <main+143>   1. 0x000000000000008f <+143>:   mov %rax,%rdx   1. 0x0000000000000092 <+146>:   mov -0x10(%rbp),%rax   1. 0x0000000000000096 <+150>:   mov %rax,-0x28(%rbp)   1. 0x000000000000009a <+154>:   movsd -0x28(%rbp),%xmm0   1. 0x000000000000009f <+159>:   mov %rdx,%rdi   1. 0x00000000000000a2 <+162>:   callq 0xa7 <main+167> | 1. This loads the effective address 0x0(%rip), which is the address that points to the string: “The volume is: “, into the %rsi register to prepare for the output function call. 2. This loads the effective address 0x0(%rip), the address of the format specifier for output, into the %rdi register to prepare for the output function call. 3. This calls the function at address 0x8f to output the string: “The volume is: “. The callq instruction saves the return address on the stack and jumps to the specified address, which is the following line of code. 4. This moves the return value from the previous function call, which is stored in the %rax register, and moves it into the %rdx register to save it for future use. 5. This instruction moves the value at -0x10(%rbp) into the %rax register, this is the variable for volume 6. This moves the value previously stored in the %rax register into the location -0x28(%rbp), an -28 offset or 28 bytes above the base pointer register. 7. This converts the value stored at the location -0x28(%rbp) into a double and stores it in the %xmm0 register to prepare it for use in output. 8. This moves the value stored in the %rdx register to the %rdi register to set up the first argument for the next function call. 9. This calls the function at the address 0xa7 to output the value of volume. |
| 1. 0x00000000000000a7 <+167>:   mov $0x0,%eax   1. 0x00000000000000ac <+172>:   mov -0x8(%rbp),%rcx   1. 0x00000000000000b0 <+176>:   xor %fs:0x28,%rcx   1. 0x00000000000000b9 <+185>:   je 0xc0 <main+192>   1. 0x00000000000000bb <+187>:   callq 0xc0 <main+192>   1. 0x00000000000000c0 <+192>:   leaveq   1. 0x00000000000000c1 <+193>:   retq | 1. This instruction moves the value 0 into the %eax register, effectively setting the return value of the function to 0, which indicates the successful execution of the function. 2. This moves the value stored at location -0x8(%rbp) into the %rcx register. 3. This performs an XOR operation between the value stored at the location %fs:0x28 and the value stored in the %rcx register, which is used to check for errors. 4. If the result from the previous XOR operation is zero, this instruction jumps to the address 0xc0, which is line 192 of the main function. 5. This calls the function at address 0xc0 to handle any errors 6. Restores the previous stack frame. 7. Returns from the main function. |
|  |  |

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

**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** |
| --- | --- | --- |
| 1. 0x0000000000000000 <+0>:   push %rbp   1. 0x0000000000000001 <+1>:   mov %rsp,%rbp   1. 0x0000000000000004 <+4>:   sub $0x30,%rsp | int main() {  double pi = 3.14;  double radius; | Declares a function named main of type int, the entry point for the program. We see this accomplished in the assembly code by the new stack frame being set up for the main function. Additionally, 48 bytes are allocated on the stack for local variables. |
| 1. 0x0000000000000008 <+8>:   mov %fs:0x28,%rax   1. 0x0000000000000011 <+17>:   mov %rax,-0x8(%rbp)   1. 0x0000000000000015 <+21>:   xor %eax,%eax   1. 0x0000000000000017 <+23>:   lea 0x0(%rip),%rsi # 0x1e <main+30>   1. 0x000000000000001e <+30>:   lea 0x0(%rip),%rdi # 0x25 <main+37>   1. 0x0000000000000025 <+37>:   callq 0x2a <main+42> | cout << “Enter radius: “; | This retrieves the string “Enter radius: “ and prepares it for use in the appropriate registers for the cout function call. This function call displays “Enter radius: “ to the user. |
| 1. 0x000000000000002a <+42>:   mov %rax,%rdx   1. 0x000000000000002d <+45>:   mov 0x0(%rip),%rax # 0x34 <main+52>   1. 0x0000000000000034 <+52>:   mov %rax,%rsi   1. 0x0000000000000037 <+55>:   mov %rdx,%rdi   1. 0x000000000000003a <+58>:   callq 0x3f <main+63>   1. 0x000000000000003f <+63>:   lea -0x14(%rbp),%rax   1. 0x0000000000000043 <+67>:   mov %rax,%rsi   1. 0x0000000000000046 <+70>:   lea 0x0(%rip),%rdi # 0x4d <main+77>   1. 0x000000000000004d <+77>:   callq 0x52 <main+82> | cin >> radius; | These assembly instructions are prepared for taking input for the variable radius by moving the arguments for the input function call to the appropriate registers, then the cin function is called to receive input from the user for the radius variable. |
| 1. 0x0000000000000052 <+82>:   mov -0x14(%rbp),%edx   1. 0x0000000000000055 <+85>:   mov -0x14(%rbp),%eax   1. 0x0000000000000058 <+88>:   imul %eax,%edx   1. 0x000000000000005b <+91>:   mov -0x14(%rbp),%eax   1. 0x000000000000005e <+94>:   imul %edx,%eax   1. 0x0000000000000061 <+97>:   mov %eax,-0x14(%rbp)   1. 0x0000000000000064 <+100>:   mov -0x14(%rbp),%eax   1. 0x0000000000000067 <+103>:   cvtsi2sd %eax,%xmm0   1. 0x000000000000006b <+107>:   movsd 0x0(%rip),%xmm1 # 0x73 <main+115>   1. 0x0000000000000073 <+115>:   mulsd %xmm1,%xmm0   1. 0x0000000000000077 <+119>:   movsd %xmm0,-0x10(%rbp) | double volume = (4.0 / 3.0) \* pi \* (radius \* radius \* radius); | Lines 19-24 of the assembly code are for calculating the radius to the power of three.  Note: I included the calculation for (4.0 / 3.0) because the only formula for volume I could find that included a radius and the radius used to the power of three was the volume of a sphere, which includes the calculation (4.0 / 3.0). No other volume formula made sense for the calculations we can see in the assembly code.  Lines 25 – 29 take the previously calculated value of radius to the power of three and multiply this value by the variable pi to get the value of volume. |
| 1. 0x000000000000007c <+124>:   lea 0x0(%rip),%rsi # 0x83 <main+131>   1. 0x0000000000000083 <+131>:   lea 0x0(%rip),%rdi # 0x8a <main+138>   1. 0x000000000000008a <+138>:   callq 0x8f <main+143>   1. 0x000000000000008f <+143>:   mov %rax,%rdx   1. 0x0000000000000092 <+146>:   mov -0x10(%rbp),%rax   1. 0x0000000000000096 <+150>:   mov %rax,-0x28(%rbp)   1. 0x000000000000009a <+154>:   movsd -0x28(%rbp),%xmm0   1. 0x000000000000009f <+159>:   mov %rdx,%rdi   1. 0x00000000000000a2 <+162>:   callq 0xa7 <main+167> | cout << “The volume is: “ << volume; | This portion of code consists of two output function calls combined into one cout statement. First a call to the output function displays “the volume is: “, and a second function call to output displays the value stored into the volume variable. |
| 1. 0x00000000000000a7 <+167>:   mov $0x0,%eax   1. 0x00000000000000ac <+172>:   mov -0x8(%rbp),%rcx   1. 0x00000000000000b0 <+176>:   xor %fs:0x28,%rcx   1. 0x00000000000000b9 <+185>:   je 0xc0 <main+192>   1. 0x00000000000000bb <+187>:   callq 0xc0 <main+192>   1. 0x00000000000000c0 <+192>:   leaveq   1. 0x00000000000000c1 <+193>:   retq | return 0; | Returns 0 because the main function is of type integer and expects an integer value to be returned. Returning a 0 value indicates to the operating system that the program exited without any errors. |

## **File Three**

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

| **Blocks of Assembly Code** | **Explanation of Functionality** |
| --- | --- |
| 1. 0x0000000000000000 <+0>:   push %rbp   1. 0x0000000000000001 <+1>:   mov %rsp,%rbp   1. 0x0000000000000004 <+4>:   sub $0x20,%rsp | 1. This instruction saves the base pointer of the previous stack frame by pushing it onto the stack. 2. This sets the base pointer for the current stack frame, it moves the stack pointer into the base pointer register. 3. This line subtracts the hexadecimal value 0x20 (32 in base 10 or decimal form) from the stack pointer register, allocating 32 bytes of space on the stack to be used for local variables. These variables include i, j, and l. |
| 1. 0x0000000000000008 <+8>:   mov %fs:0x28,%rax   1. 0x0000000000000011 <+17>:   mov %rax,-0x8(%rbp)   1. 0x0000000000000015 <+21>:   xor %eax,%eax   1. 0x0000000000000017 <+23>:   movl $0x1,-0xc(%rbp)   1. 0x000000000000001e <+30>:   lea 0x0(%rip),%rsi # 0x25 <main+37>   1. 0x0000000000000025 <+37>:   lea 0x0(%rip),%rdi # 0x2c <main+44>   1. 0x000000000000002c <+44>:   callq 0x31 <main+49> | 1. This instruction moves the value stored in the fragment segment offset 0x28 into the %rax register. 2. This moves the previously stored value in the %rax register into the location -0x8(%rbp), an -8 offset or 8 bytes above the base pointer register. 3. This instruction performs an XOR operation between the %eax register and itself, effectively clearing out the %eax register and setting its value to zero. 4. This moves the value 1 into the location -0xc(%rbp), -0xc in base 10 is 12, therefore -0xc(%rbp) is an -12 offset or 12 bytes above the base pointer register. 5. This loads the effective address 0x0(%rip), which is the next instruction, into the %rsi register (the second argument register) to prepare it for use in the following function call. 6. This loads the effective address 0x0(%rip), which is the next instruction, into the %rdi register, the first argument register, in preparation for the following function call. 7. This line calls the function located at address 0x31. |
| 1. 0x0000000000000034 <+52>:   mov 0x0(%rip),%rax # 0x3b <main+59>   1. 0x000000000000003b <+59>:   mov %rax,%rsi   1. 0x000000000000003e <+62>:   mov %rdx,%rdi   1. 0x0000000000000041 <+65>:   callq 0x46 <main+70> | 1. This instruction moves the value stored at 0x0(%rip), which is the next instruction, into the %rax register in preparation for the output function call. 2. This moves the value previously stored in the %rax register into the %rsi register (the second argument register) to be used in the output function call. 3. This instruction moves the value stored in the %rdx register (the return value from the previous function call) into the %rdi call to be used in the following function call. 4. Calls the function to output the string: “Enter number of rows “ |
| 1. 0x0000000000000046 <+70>:   lea -0x18(%rbp),%rax   1. 0x000000000000004a <+74>:   mov %rax,%rsi   1. 0x000000000000004d <+77>:   lea 0x0(%rip),%rdi # 0x54 <main+84>   1. 0x0000000000000054 <+84>:   callq 0x59 <main+89> | 1. This loads the effective address -0x18(%rbp), an -18 offset or 18 bytes above the base pointer register, into the %rax register. This is the variable that will be used for the number of rows “i”. 2. This moves the previously stored value in the %rax register into the %rsi register (the second argument register) to be used in the input function call. 3. This loads the effective address 0x0(%rip), the next instuction into the %rdi register (the first argument register) for the following input function call. 4. This calls the input function to read input from the user. |
| 1. 0x0000000000000059 <+89>:   mov -0x18(%rbp),%eax   1. 0x000000000000005c <+92>:   sub $0x1,%eax   1. 0x000000000000005f <+95>:   mov %eax,-0xc(%rbp)   1. 0x0000000000000062 <+98>:   movl $0x1,-0x10(%rbp)   1. 0x0000000000000069 <+105>:   mov -0x18(%rbp),%eax   1. 0x000000000000006c <+108>:   cmp %eax,-0x10(%rbp)   1. 0x000000000000006f <+111>:   jg 0xe3 <main+227> | 1. This moves the value stored at -0x18(%rbp) (the variable “i”, which is the number of rows, into the %eax register. 2. This subtracts the value 1 from the value in the %eax register and stores it back into the %eax register. 3. This moves the value stored in the %eax register into the location -0xc(%rbp), 0xc in decimal is 12, therefore, this is an -12 offset or 12 bytes above the base pointer register. This will be the variable “k”. 4. This moves the value 1 into the location -0x10(%rbp) (0x10 in decimal form is 16), therefore this is an -16 offset or 16 bytes above the base pointer register. This is setting the control variable for the first outer loop, variable “j”, to be equal to 1. 5. This moves the orginial value of -0x18(%rbp), the number of rows, back into the %eax register. 6. This makes a comparison between the value stored in the %eax register (number of rows) with the value stored at -0x10(%rbp). 7. If from the previous comparison, the value in the %eax register is greater than the value stored at -0x10(%rbp), this instruction jumps the execution to address 0xe3 or line 227 of the main function. |
| 1. 0x0000000000000071 <+113>:   movl $0x1,-0x14(%rbp)   1. 0x0000000000000078 <+120>:   mov -0x14(%rbp),%eax   1. 0x000000000000007b <+123>:   cmp -0xc(%rbp),%eax   1. 0x000000000000007e <+126>:   jg 0x99 <main+153> | 1. This moves the value 1 into the location -0x14(%rbp), 0x14 in base 10 is 20, therefore, this is an -20 offset or 20 bytes above the base pointer register. This will be variable “l”. this is the control variable for the inner loop. 2. This moves the value stored at -0x14(%rbp) into the %eax register. 3. This compares the value stored at -0xc(%rbp) to the value stored in the %eax register. 4. If from the previous comparison the value stored in the -0xc(%rbp) is greater than the value stored in the %eax register the execution jumps to address 0x99 which is line 153 of the main function. |
| 1. 0x0000000000000080 <+128>:   lea 0x0(%rip),%rsi # 0x87 <main+135>   1. 0x0000000000000087 <+135>:   lea 0x0(%rip),%rdi # 0x8e <main+142>   1. 0x000000000000008e <+142>:   callq 0x93 <main+147> | 1. This instruction loads the effective address 0x0(%rip), which is the next instruction, and stores it into the %rsi register, the second argument register to be used in the output function call. 2. This loads the effective address of 0x0(%rip), whioch is the next instruction, into the %rdi register, which is the first argument register to be used in the output function call. 3. Calls the function to output the string “ “. |
| 1. 0x0000000000000093 <+147>:   addl $0x1,-0x14(%rbp)   1. 0x0000000000000097 <+151>:   jmp 0x78 <main+120> | 1. This adds the value 1 to the value stored at -0x14(%rbp), this is the control variable for the inner loop “l”, and stored it back into -0x14(%rbp). 2. This jumps the execution back to the address 0x78 or line 120 of the main function, which is the beginning of the inner loop. |
| 1. 0x0000000000000099 <+153>:   subl $0x1,-0xc(%rbp)   1. 0x000000000000009d <+157>:   movl $0x1,-0x14(%rbp)   1. 0x00000000000000a4 <+164>:   mov -0x10(%rbp),%eax   1. 0x00000000000000a7 <+167>:   add %eax,%eax   1. 0x00000000000000a9 <+169>:   sub $0x1,%eax   1. 0x00000000000000ac <+172>:   cmp %eax,-0x14(%rbp)   1. 0x00000000000000af <+175>:   jg 0xca <main+202> | 1. This instruction subtracts the value 1 from the value stored at -0xc(%rbp), which is the variable “k”. This value is then stored back at location -0xc(%rbp). 2. This moves the value 1 into the location -0x14(%rbp), which is the control variable “l” for the next inner loop. 3. This instruction moves the value stored at -0x10(%rbp) into the %eax register. (This is the value of the variable “j”. 4. This instruction adds the value stored in the %eax register to itself and stores it back into the %eax register. This is essentially multiplying the value by 2. 5. This instruction subtracts the value 1 from the current value stored in the %eax register , then stores the new value back into the %eax register. 6. This instruction compares the value in the %eax register with the value stored at the location -0x14(%rbp), which is the control variable for the second inner loop. 7. If from the previous comparison, the value in the %eax register is greater than the value stored at location -0x14(%rbp), the execution jumps to address 0xca, which is line 202 of the main function. |
| 1. 0x00000000000000b1 <+177>:   lea 0x0(%rip),%rsi # 0xb8 <main+184>   1. 0x00000000000000b8 <+184>:   lea 0x0(%rip),%rdi # 0xbf <main+191>   1. 0x00000000000000bf <+191>:   callq 0xc4 <main+196> | 1. This instruction loads the effective address 0x0(%rip), which is the next instruction, into the %rsi register, the second argument register used for the output function call 2. This loads the effective address 0x0(%rsi), the next instruction, into the %rdi register, which is the first argument register for the output function call. 3. This calls the function to output the string “\*”. |
| 1. 0x00000000000000c4 <+196>:   addl $0x1,-0x14(%rbp)   1. 0x00000000000000c8 <+200>:   jmp 0xa4 <main+164> | 1. This instruction adds the value 1 to the value stored at location -0x14(%rbp), which is the control variable for the second inner loop. 2. This instruction jumps the execution to address 0xa4, which is line 164 of the main function, or the beginning of the second inner loop. |
| 1. 0x00000000000000ca <+202>:   lea 0x0(%rip),%rsi # 0xd1 <main+209>   1. 0x00000000000000d1 <+209>:   lea 0x0(%rip),%rdi # 0xd8 <main+216>   1. 0x00000000000000d8 <+216>:   callq 0xdd <main+221> | 1. This instruction loads the effective address 0x0(%rip), which is the next instruction, into the %rsi register, the second argument register used for the output function call 2. This loads the effective address 0x0(%rsi), the next instruction, into the %rdi register, which is the first argument register for the output function call. 3. This instruction calls the output function to return a new line. |
| 1. 0x00000000000000dd <+221>:   addl $0x1,-0x10(%rbp)   1. 0x00000000000000e1 <+225>:   jmp 0x69 <main+105> | 1. This instruction adds the value 1 to the value stored at location -0x10(%rbp), which is the variable “j”, and also the control variable for the first outer loop. 2. This instruction jumps the execution to address 0x69, which is the beginning of the first outer loop. |
| 1. 0x00000000000000e3 <+227>:   movl $0x1,-0xc(%rbp)   1. 0x00000000000000ea <+234>:   movl $0x1,-0x10(%rbp)   1. 0x00000000000000f1 <+241>:   mov -0x18(%rbp),%eax   1. 0x00000000000000f4 <+244>:   sub $0x1,%eax   1. 0x00000000000000f7 <+247>:   cmp %eax,-0x10(%rbp)   1. 0x00000000000000fa <+250>:   jg 0x171 <main+369> | 1. This instruction moves the value 1 into the location -0xc(%rbp), which is the variable “k”. 2. This instruction moves the value 1 into the location -0x10(%rbp), which is the variable “j”, the control variable for the second outer loop. 3. This moves the value stored at -0x18(%rbp), the variable “i” into the %eax register 4. This subtracts the value 1 from the value in %eax and stores this new value back into the %eax register. 5. This compare the value stored in the %eax register with the value stored at the location -0x10(%rbp). 6. If from the previous comparison, the value stored in the %eax register is greater than the value stored in -0x10(%rbp), this line jumps the execution to address 0x171, which is line 369 of the main function. |
| 1. 0x00000000000000fc <+252>:   movl $0x1,-0x14(%rbp)   1. 0x0000000000000103 <+259>:   mov -0x14(%rbp),%eax   1. 0x0000000000000106 <+262>:   cmp -0xc(%rbp),%eax   1. 0x0000000000000109 <+265>:   jg 0x124 <main+292> | 1. This instruction moves the value 1 into the location -0x14(%rbp), which is the variable “l”. 2. This instruction moves the value stored at -0x14(%rbp) into the %eax register to prepare for the comparison, 3. This compares the value stored at -0xc(%rbp), the variable “k” with the value stored in the %eax register. 4. If from the previous comparison, the value stored at -0xc(%rbp) is greater than the value stored in the %eax register, this instruction jumps the execution to address 0x124, which is line 292 of the main function. |
| 1. 0x000000000000010b <+267>:   lea 0x0(%rip),%rsi # 0x112 <main+274>   1. 0x0000000000000112 <+274>:   lea 0x0(%rip),%rdi # 0x119 <main+281>   1. 0x0000000000000119 <+281>:   callq 0x11e <main+286> | 1. This instruction loads the effective address 0x0(%rip), which is the next instruction, into the %rsi register (the second argument register) used for the output function call. 2. This instruction loads the effective address 0x0(%rip), the next instruction, into the %rdi register (the first argument register) used for the output function call. 3. This calls the output function to display the string “ “. |
| 1. 0x000000000000011e <+286>:   addl $0x1,-0x14(%rbp)   1. 0x0000000000000122 <+290>:   jmp 0x103 <main+259> | 1. This adds the value one to the value stored at location -0x14(%rbp) which is the control variable for the second inner loop. 2. This instruction jumps the execution to address 0x103 which is line 259 of the main function (the beginning of the first inner loop for the second outer loop. |
| 1. 0x0000000000000124 <+292>:   addl $0x1,-0xc(%rbp)   1. 0x0000000000000128 <+296>:   movl $0x1,-0x14(%rbp)   1. 0x000000000000012f <+303>:   mov -0x18(%rbp),%eax   1. 0x0000000000000132 <+306>:   sub -0x10(%rbp),%eax   1. 0x0000000000000135 <+309>:   add %eax,%eax   1. 0x0000000000000137 <+311>:   sub $0x1,%eax   1. 0x000000000000013a <+314>:   cmp %eax,-0x14(%rbp)   1. 0x000000000000013d <+317>:   jg 0x158 <main+344> | 1. This instruction adds the value 1 to the value stored at location -0xc(%rbp), which is the variable “k”. 2. This moves the value 1 and stores it in the location at -0x14(%rbp), which is the variable “l”. 3. This instruction moves the value stored at location -0x18(%rbp), the variable for the number of rows, “i”, into the %eax register. 4. This subtracts the value stored at -0x10(%rbp), which is the variable “j”, from the value stored in the %eax register, then stores this result back into the %eax register. 5. This adds the value stored in the %eax register with itself, then stores this result back into the %eax register. This effectively multiplies the value stored in %eax by 2 times. 6. This subtracts the value 1 from the value stored in the %eax register, then stores this resulting value back into the %eax register. 7. This line makes a comparison between the value stored in the %eax register and the value stored at location -0x14(%rbp), which is the variable “l’. 8. If from the previous comparison the value in %eax is greater than the value stored at -0x14(%rbp) the execution jumps to address 0x158, which is line 344 of the main function |
| 1. 0x000000000000013f <+319>:   lea 0x0(%rip),%rsi # 0x146 <main+326>   1. 0x0000000000000146 <+326>:   lea 0x0(%rip),%rdi # 0x14d <main+333>   1. 0x000000000000014d <+333>:   callq 0x152 <main+338> | 1. This loads the effective address 0x0(%rip), which is the next instruction into the %rsi register (the second argument register) used for the output function call. 2. This loads the effective address 0x0(%rip), which is the next instruction, into the %rdi register (the first argument register) used for the output function call. 3. The calls the function to output the string “\*” |
| 1. 0x0000000000000152 <+338>:   addl $0x1,-0x14(%rbp)   1. 0x0000000000000156 <+342>:   jmp 0x12f <main+303> | 1. This adds the value 1 to the value stored at -0x14(%rbp), then stores it back into the same location. 2. This instruction jumps the execution to address 0x12f, which is line 303 of the main function (the beginning of the second inner loop of the second outer loop. |
| 1. 0x0000000000000158 <+344>:   lea 0x0(%rip),%rsi # 0x15f <main+351>   1. 0x000000000000015f <+351>:   lea 0x0(%rip),%rdi # 0x166 <main+358>   1. 0x0000000000000166 <+358>:   callq 0x16b <main+363> | 1. This loads the effective address 0x0(%rip), which is the next instruction into the %rsi register (the second argument register) used for the output function call. 2. This loads the effective address 0x0(%rip), which is the next instruction, into the %rdi register (the first argument register) used for the output function call. 3. The calls the function to output a new line. |
| 1. 0x000000000000016b <+363>:   addl $0x1,-0x10(%rbp)   1. 0x000000000000016f <+367>:   jmp 0xf1 <main+241> | 1. This adds the value 1 to the value stored at location -0x10(%rbp), which is the variable ‘j’, the control variable for the second outer loop, storing this value back into the location -0x10(%rbp). 2. This instructs the execution to jump to address 0xf1, which is line 241 of the main function, and also the beginning of the second outer loop. |
| 1. 0x0000000000000171 <+369>:   mov $0x1,%eax   1. 0x0000000000000176 <+374>:   mov -0x8(%rbp),%rcx   1. 0x000000000000017a <+378>:   xor %fs:0x28,%rcx   1. 0x0000000000000183 <+387>:   je 0x18a <main+394>   1. 0x0000000000000185 <+389>:   callq 0x18a <main+394>   1. 0x000000000000018a <+394>:   leaveq   1. 0x000000000000018b <+395>:   retq | 1. This instruction moves the value 1 into the %eax register 2. This moves the value stored at -0x8(%rbp) into the %rcx register. 3. This instruction performs an XOR operation on the value stored at %fs:0x28, a 28 offset from the fragment segment register. 4. This instruction jumps to the address 0x18a, which is line 394 of the main function if from the previous XOR comparison if the zero flag is set. 5. This instruction calls the function at address 0x18a, likely to clean up any errors that may have been encountered. 6. Restores the previous stack frame. 7. Returns from the main function. |

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

**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** |
| --- | --- | --- |
| 1. 0x0000000000000000 <+0>:   push %rbp   1. 0x0000000000000001 <+1>:   mov %rsp,%rbp   1. 0x0000000000000004 <+4>:   sub $0x20,%rsp | int main() {  int i, j, l; | Declares a function named main of type int, the entry point for the program. We see this accomplished in the assembly code by the new stack frame being set up for the main function. Additionally, 32 bytes are allocated on the stack for local variables. |
| 1. 0x0000000000000008 <+8>:   mov %fs:0x28,%rax   1. 0x0000000000000011 <+17>:   mov %rax,-0x8(%rbp)   1. 0x0000000000000015 <+21>:   xor %eax,%eax   1. 0x0000000000000017 <+23>:   movl $0x1,-0xc(%rbp)   1. 0x000000000000001e <+30>:   lea 0x0(%rip),%rsi # 0x25 <main+37>   1. 0x0000000000000025 <+37>:   lea 0x0(%rip),%rdi # 0x2c <main+44>   1. 0x000000000000002c <+44>:   callq 0x31 <main+49>   1. 0x0000000000000034 <+52>:   mov 0x0(%rip),%rax # 0x3b <main+59>   1. 0x000000000000003b <+59>:   mov %rax,%rsi   1. 0x000000000000003e <+62>:   mov %rdx,%rdi   1. 0x0000000000000041 <+65>:   callq 0x46 <main+70> | cout << “Enter a number of rows\n”; | These instructions are used to prepare the function call to output, which outputs the string “Enter a number of rows: followed by a new line, which is seen here by the c++ statement for cout which outputs to the console the string. |
| 1. 0x0000000000000046 <+70>:   lea -0x18(%rbp),%rax   1. 0x000000000000004a <+74>:   mov %rax,%rsi   1. 0x000000000000004d <+77>:   lea 0x0(%rip),%rdi # 0x54 <main+84>   1. 0x0000000000000054 <+84>:   callq 0x59 <main+89> | cin >> i; | This reads in from input a number entered by the user and stores the value into the local variable i. |
| 1. 0x0000000000000059 <+89>:   mov -0x18(%rbp),%eax   1. 0x000000000000005c <+92>:   sub $0x1,%eax   1. 0x000000000000005f <+95>:   mov %eax,-0xc(%rbp)   1. 0x0000000000000062 <+98>:   movl $0x1,-0x10(%rbp)   1. 0x0000000000000069 <+105>:   mov -0x18(%rbp),%eax   1. 0x000000000000006c <+108>:   cmp %eax,-0x10(%rbp)   1. 0x000000000000006f <+111>:   jg 0xe3 <main+227>   1. 0x00000000000000dd <+221>:   addl $0x1,-0x10(%rbp)   1. 0x00000000000000e1 <+225>:   jmp 0x69 <main+105> | int k = i – 1;  for (j = 1; j <= i; j++) { | This declares the local variable “k” and initializes its value equal to the value of variable i minus 1.  Then the first outer loop of the program is set up by setting the value of j = 1, which is the control variable for this loop.  The comparison instruction tells us how many iterations the for loop will execute and in this case, it is as long as j is less than or equal to the value of I (the number of rows).  The last two assembly instructions here are the final piece of the for loop where the variable j is incremented by 1 and the loop continues until j is greater than i. |
| 1. 0x0000000000000071 <+113>:   movl $0x1,-0x14(%rbp)   1. 0x0000000000000078 <+120>:   mov -0x14(%rbp),%eax   1. 0x000000000000007b <+123>:   cmp -0xc(%rbp),%eax   1. 0x000000000000007e <+126>:   jg 0x99 <main+153>   1. 0x0000000000000093 <+147>:   addl $0x1,-0x14(%rbp)   1. 0x0000000000000097 <+151>:   jmp 0x78 <main+120> | for (l = 1; l <= k; l++) { | This sets up the first inner for loop nested inside the first outer loop. It initializes the variable l to 1, which is the for loops control variable. The comparison tells the for loop how many iterations the for loop will execute, and in this case it will execute as long as the variable l is less than or equal to the variable k.  The last two instructions here are the final piece of the for loop where the variable l is incremented by 1 each iteration of the loop. |
| 1. 0x0000000000000080 <+128>:   lea 0x0(%rip),%rsi # 0x87 <main+135>   1. 0x0000000000000087 <+135>:   lea 0x0(%rip),%rdi # 0x8e <main+142>   1. 0x000000000000008e <+142>:   callq 0x93 <main+147> | cout << “ “; | For every iteration of the inside for loop, a function call is made to output the string “ “. |
| 1. 0x0000000000000099 <+153>:   subl $0x1,-0xc(%rbp)   1. 0x000000000000009d <+157>:   movl $0x1,-0x14(%rbp)   1. 0x00000000000000a4 <+164>:   mov -0x10(%rbp),%eax   1. 0x00000000000000a7 <+167>:   add %eax,%eax   1. 0x00000000000000a9 <+169>:   sub $0x1,%eax   1. 0x00000000000000ac <+172>:   cmp %eax,-0x14(%rbp)   1. 0x00000000000000af <+175>:   jg 0xca <main+202>   1. 0x00000000000000c4 <+196>:   addl $0x1,-0x14(%rbp)   1. 0x00000000000000c8 <+200>:   jmp 0xa4 <main+164> | for (l = 1; l <= ((j \* 2) – 1); l++) { | This sets up the second inner loop that is nested inside the first outer loop. It initializes the variable l to 1, which is the for loops control variable. The comparison tells the for loop how many iterations it is going to be executed, and in this case, it will execute as long as the value of l is less than or equal to the value of (j \*2) -1.  The last two instructions here are the final pieces of the for loop where the variable l is incremented by 1 each iteration of the for loop. |
| 1. 0x00000000000000b1 <+177>:   lea 0x0(%rip),%rsi # 0xb8 <main+184>   1. 0x00000000000000b8 <+184>:   lea 0x0(%rip),%rdi # 0xbf <main+191>   1. 0x00000000000000bf <+191>:   callq 0xc4 <main+196> | cout << “\*”; | For every iteration of this inner for loop, a function call is made to output the string “\*”. |
| 1. 0x00000000000000ca <+202>:   lea 0x0(%rip),%rsi # 0xd1 <main+209>   1. 0x00000000000000d1 <+209>:   lea 0x0(%rip),%rdi # 0xd8 <main+216>   1. 0x00000000000000d8 <+216>:   callq 0xdd <main+221> | cout << endl; | This calls the function to output a new line to the console. |
| 1. 0x00000000000000e3 <+227>:   movl $0x1,-0xc(%rbp)   1. 0x00000000000000ea <+234>:   movl $0x1,-0x10(%rbp)   1. 0x00000000000000f1 <+241>:   mov -0x18(%rbp),%eax   1. 0x00000000000000f4 <+244>:   sub $0x1,%eax   1. 0x00000000000000f7 <+247>:   cmp %eax,-0x10(%rbp)   1. 0x00000000000000fa <+250>:   jg 0x171 <main+369>   1. 0x000000000000016b <+363>:   addl $0x1,-0x10(%rbp)   1. 0x000000000000016f <+367>:   jmp 0xf1 <main+241> | for (j = 1; j <= I; j++) { | Then second outer loop of the program is set up by setting the value of j = 1, which is the control variable for this loop.  The comparison instruction tells us how many iterations the for loop will execute and in this case, it is as long as j is less than or equal to the value of I (the number of rows).  The last two assembly instructions here are the final piece of the for loop where the variable j is incremented by 1 and the loop continues until j is greater than i. |
| 1. 0x00000000000000fc <+252>:   movl $0x1,-0x14(%rbp)   1. 0x0000000000000103 <+259>:   mov -0x14(%rbp),%eax   1. 0x0000000000000106 <+262>:   cmp -0xc(%rbp),%eax   1. 0x0000000000000109 <+265>:   jg 0x124 <main+292>   1. 0x000000000000011e <+286>:   addl $0x1,-0x14(%rbp)   1. 0x0000000000000122 <+290>:   jmp 0x103 <main+259> | for (l = 1; l <= k; l++) { | This sets up the first inner for loop nested inside the second outer loop. It initializes the variable l to 1, which is the for loops control variable. The comparison tells the for loop how many iterations the for loop will execute, and in this case it will execute as long as the variable l is less than or equal to the variable k.  The last two instructions here are the final piece of the for loop where the variable l is incremented by 1 each iteration of the loop. |
| 1. 0x000000000000010b <+267>:   lea 0x0(%rip),%rsi # 0x112 <main+274>   1. 0x0000000000000112 <+274>:   lea 0x0(%rip),%rdi # 0x119 <main+281>   1. 0x0000000000000119 <+281>:   callq 0x11e <main+286> | cout << “ “; | For every iteration of the inside for loop, a function call is made to output the string “ “. |
| 1. 0x0000000000000124 <+292>:   addl $0x1,-0xc(%rbp)   1. 0x0000000000000128 <+296>:   movl $0x1,-0x14(%rbp)   1. 0x000000000000012f <+303>:   mov -0x18(%rbp),%eax   1. 0x0000000000000132 <+306>:   sub -0x10(%rbp),%eax   1. 0x0000000000000135 <+309>:   add %eax,%eax   1. 0x0000000000000137 <+311>:   sub $0x1,%eax   1. 0x000000000000013a <+314>:   cmp %eax,-0x14(%rbp)   1. 0x000000000000013d <+317>:   jg 0x158 <main+344>   1. 0x0000000000000152 <+338>:   addl $0x1,-0x14(%rbp)   1. 0x0000000000000156 <+342>:   jmp 0x12f <main+303> | for (l = 1; l <= ((( i - j) \*2 )-1); l++) { | This sets up the second inner loop that is nested inside the second outer loop. It initializes the variable l to 1, which is the for loops control variable. The comparison tells the for loop how many iterations it is going to be executed, and in this case, it will execute as long as the value of l is less than or equal to the value of ((i – j) \* 2) - 1.  The last two instructions here are the final pieces of the for loop where the variable l is incremented by 1 each iteration of the for loop. |
| 1. 0x000000000000013f <+319>:   lea 0x0(%rip),%rsi # 0x146 <main+326>   1. 0x0000000000000146 <+326>:   lea 0x0(%rip),%rdi # 0x14d <main+333>   1. 0x000000000000014d <+333>:   callq 0x152 <main+338> | cout << “\*”; | For every iteration of this inner for loop, a function call is made to output the string “\*”. |
| 1. 0x0000000000000158 <+344>:   lea 0x0(%rip),%rsi # 0x15f <main+351>   1. 0x000000000000015f <+351>:   lea 0x0(%rip),%rdi # 0x166 <main+358>   1. 0x0000000000000166 <+358>:   callq 0x16b <main+363> | cout << endl; | This calls the function to output a new line to the console. |
| 1. 0x0000000000000171 <+369>:   mov $0x1,%eax   1. 0x0000000000000176 <+374>:   mov -0x8(%rbp),%rcx   1. 0x000000000000017a <+378>:   xor %fs:0x28,%rcx   1. 0x0000000000000183 <+387>:   je 0x18a <main+394>   1. 0x0000000000000185 <+389>:   callq 0x18a <main+394>   1. 0x000000000000018a <+394>:   leaveq   1. 0x000000000000018b <+395>:   retq | return 0; | Returns 0 because the main function is of type integer and expects an integer value to be returned. Returning a 0 value indicates to the operating system that the program exited without any errors. |

## **File Four**

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

| **Blocks of Assembly Code** | **Explanation of Functionality** |
| --- | --- |
| 1. 0x0000000000000000 <+0>:   push %rbp   1. 0x0000000000000001 <+1>:   mov %rsp,%rbp   1. 0x0000000000000004 <+4>:   sub $0x30,%rsp | 1. This instruction saves the base pointer of the previous stack frame by pushing it onto the stack. 2. This sets the base pointer for the current stack frame, it moves the stack pointer into the base pointer register. 3. This line subtracts the hexadecimal value 0x30 (48 in base 10 or decimal form) from the stack pointer register, allocating 82 bytes of space on the stack to be used for local variables. These variables include the variables binaryNumber, decimalNumber, I, and remainder. |
| 1. 0x0000000000000008 <+8>:   mov %fs:0x28,%rax   1. 0x0000000000000011 <+17>:   mov %rax,-0x8(%rbp)   1. 0x0000000000000015 <+21>:   xor %eax,%eax   1. 0x0000000000000017 <+23>:   movq $0x0,-0x20(%rbp)   1. 0x000000000000001f <+31>:   movq $0x1,-0x18(%rbp)   1. 0x0000000000000027 <+39>:   lea 0x0(%rip),%rsi # 0x2e <main+46>   1. 0x000000000000002e <+46>:   lea 0x0(%rip),%rdi # 0x35 <main+53>   1. 0x0000000000000035 <+53>:   callq 0x3a <main+58> | 1. This instruction moves the value stored in the fragment segment offset 0x28 into the %rax register. 2. The moves the previously stored value from the %rax register into the location -0x8(%rbp), an -8 offset or 8 bytes above the base pointer register. 3. This makes an XOR comparison of the value in the %eax register against itself, effectively setting the value of the %eax register to zero. 4. This instruction moves the value 0 into the location -0x20(%rbp), and stores it. 0x20 in base 10 is 32, therefore this is a -32 offset or 32 bytes above the base pointer register. This is the variable for decimalNumber. 5. This moves the value 1 into the location -0x18(%rbp) and stores it. 0x18 in base 10 is 24, therefore this is sored at a -24 offset or 24 bytes above the base pointer register. This is the variable for i. 6. This loads the effective address 0x0(%rip), which is the next instruction, into the %rsi register, the second argument register for the output function call. 7. This loads the effective address 0x0(%rip), which is the next instruction, into the %rdi register, which is the first argument register used for the following output function call. 8. This calls the function to output the string: “Enter the binary number:\n” |
| 1. 0x000000000000003a <+58>:   mov %rax,%rdx   1. 0x000000000000003d <+61>:   mov 0x0(%rip),%rax # 0x44 <main+68>   1. 0x0000000000000044 <+68>:   mov %rax,%rsi   1. 0x0000000000000047 <+71>:   mov %rdx,%rdi   1. 0x000000000000004a <+74>:   callq 0x4f <main+79> | 1. This instruction moves the value stored in the %rax register into the %rdx register. 2. This moves the address 0x0(%rip), which is the next instruction, into the %rax register. 3. This moves the value previously stored in the %rax register into the %rsi register, the second argument register to be used in the function call. 4. This moves the value previously stored in the %rdx register into the %rdi register, the first argument register to be used in the following function call. 5. This calls the function to output a new line. |
| 1. 0x000000000000004f <+79>:   lea -0x28(%rbp),%rax   1. 0x0000000000000053 <+83>:   mov %rax,%rsi   1. 0x0000000000000056 <+86>:   lea 0x0(%rip),%rdi # 0x5d <main+93>   1. 0x000000000000005d <+93>:   callq 0x62 <main+98> | 1. This loads the effective address -0x28(%rbp), an -28 offset or 28 bytes above the base pointer register, and stores it into the %rax register. This is the binaryNuber variable. 2. This instruction moves the previously stored value in the %rax register into the %rsi register (the second argument register), to be used with the input function call. 3. This loads the effective address 0x0(%rip), the next instruction into the %rdi register, the first argument register, which is used in the following input function call. 4. Calls the function to read input and store it in the variable binaryNumber. |
| 1. 0x0000000000000062 <+98>:   mov -0x28(%rbp),%rax   1. 0x0000000000000066 <+102>:   test %rax,%rax   1. 0x0000000000000069 <+105>:   je 0xf2 <main+242> | 1. This instruction moves the value store at location -0x28(%rbp) into the %rax register to be used for the while loop comparison. 2. This is the while loop instruction that tests whether the value stored in the %rax register is equal to itself. If it is, a Zero Flag is set. 3. This instruction jumps the execution to the address 0xf2, which is line 242 of the main function, if the previous Zero Flag is set, meaning that the binaryNumber is zero. |
| 1. 0x000000000000006f <+111>:   mov -0x28(%rbp),%rcx   1. 0x0000000000000073 <+115>:   movabs $0x6666666666666667,%rdx   1. 0x000000000000007d <+125>:   mov %rcx,%rax   1. 0x0000000000000080 <+128>:   imul %rdx   1. 0x0000000000000083 <+131>:   sar $0x2,%rdx   1. 0x0000000000000087 <+135>:   mov %rcx,%rax   1. 0x000000000000008a <+138>:   sar $0x3f,%rax   1. 0x000000000000008e <+142>:   sub %rax,%rdx   1. 0x0000000000000091 <+145>:   mov %rdx,%rax   1. 0x0000000000000094 <+148>:   mov %rax,-0x10(%rbp)   1. 0x0000000000000098 <+152>:   mov -0x10(%rbp),%rdx   1. 0x000000000000009c <+156>:   mov %rdx,%rax | 1. This instruction moves the value stored at -0x28(%rbp), the variable binaryNumber, into the %rcs register. 2. This instruction moves the value 0x6666666666666667 into the %rdx register. This value is used as part of the binary to decimal conversion. 3. This instruction moves the value stored in the %rcx register into the %rax register (the binaryNumber variable) 4. This instruction multiplies the value previously stored in the %rax register with the value stored in the %rdx register then the result is a paired result, the higher-order 64 bits go to the %rdx register, and the lower-order 64 bits go to the %rax register. 5. The sar instruction is a shift arithmetic right instruction which shifts the bits of the operands to the right by 2 positions, effectively dividing the value stored in the %rdx register by 4. 6. The value stored in the %rcx register is moved to the %rax register. 7. The sar instruction is a shift arithmetic right instruction, which shifts the bits of the operands to the right by 3 positions, effectively dividing the value is stored in the %rax register by 2^3=8, then storing it back into the %rax register. 8. This subtracts the value stored in the 5rax register with the value stored in the %rdx register, then stores this result back into the %rdx register. 9. This moves the value in the %rdx register into the %rax register. 10. This moves the value in the %rax register to the location -0x10(%rbp), 10 in base 10 is 16, therefore, this is an -16 offset or 16 bytes above the base pointer register. This is the variable remainder. 11. This moves the value located at -0x10(%rbp) into the %rdx register. 12. This moves the value store in the %rdx register into the %rax register. |
| 1. 0x000000000000009f <+159>:   shl $0x2,%rax   1. 0x00000000000000a3 <+163>:   add %rdx,%rax   1. 0x00000000000000a6 <+166>:   add %rax,%rax   1. 0x00000000000000a9 <+169>:   sub %rax,%rcx   1. 0x00000000000000ac <+172>:   mov %rcx,%rax   1. 0x00000000000000af <+175>:   mov %rax,-0x10(%rbp)   1. 0x00000000000000b3 <+179>:   mov -0x10(%rbp),%rax   1. 0x00000000000000b7 <+183>:   imul -0x18(%rbp),%rax   1. 0x00000000000000bc <+188>:   add %rax,-0x20(%rbp) | 1. This performs a shirt arithmetic left instruction, shifting the %rax bits left two positions, effectively multiplying the vale stored in the %rax register by 2^2=4 2. This instruction adds together the value stored in the %rdx register with the value stored in the %rax register and stores this back into the %rax register. 3. This adds the value stored in the %rax register with itself and stores the result back into the %rax register. 4. This subtracts the value stored in the %rax register with the value stored in the %rcx register, then stores the result back into the %rcx register. 5. This moves the value previously stored in the %rcx register into the %rax register. 6. This moves the the value stored in the %rax register into the location -0x10(%rbp), which is the lcoation for the variable remainder. 7. This moves the value stored in location -0x10(%rbp), the variable remainder, into the %rax register. 8. This instruction multiplies the value stored at location -0x18(%rbp) with the value stored in the %rax register, then stores the result back into the %rax register. 9. This instruction adds the value stored in the %rax register with the value stored at the location -0x20(%rbp) and stores the result back at location -0x20(%rbp). this is the location of the variable decimalNumber. |
| 1. 0x00000000000000c0 <+192>:   shlq -0x18(%rbp)   1. 0x00000000000000c4 <+196>:   mov -0x28(%rbp),%rcx   1. 0x00000000000000c8 <+200>:   movabs $0x6666666666666667,%rdx   1. 0x00000000000000d2 <+210>:   mov %rcx,%rax   1. 0x00000000000000d5 <+213>:   imul %rdx   1. 0x00000000000000d8 <+216>:   sar $0x2,%rdx   1. 0x00000000000000dc <+220>:   mov %rcx,%rax   1. 0x00000000000000df <+223>:   sar $0x3f,%rax   1. 0x00000000000000e3 <+227>:   sub %rax,%rdx   1. 0x00000000000000e6 <+230>:   mov %rdx,%rax   1. 0x00000000000000e9 <+233>:   mov %rax,-0x28(%rbp)   1. 0x00000000000000ed <+237>:   jmpq 0x62 <main+98>  Lines 46 through 54 are basically repeating the process in lines 25 through 32. This is how we are getting the power raised to the exponent i \* the remainder and storing the final value into the variable decimalNumber. | 1. This instruction performs a shift left on the value stored at -0x18(%rbp), this effectively multiplies the value by 2^2=4 and stores the result back into the location -0x18(%rbp). 2. This moves the value stored at -0x28(%rbp) into the %rcs register. 3. This instruction moves the value 0x6666666666666667 into the %rdx register. This value is used as part of the binary to decimal conversion. 4. This instruction moves the value stored in the %rcx register into the %rax register (the binaryNumber variable) 5. This instruction multiplies the value previously stored in the %rax register with the value stored in the %rdx register then the result is a paired result, the higher-order 64 bits go to the %rdx register, and the lower-order 64 bits go to the %rax register. 6. The sar instruction is a shift arithmetic right instruction which shifts the bits of the operands to the right by 2 positions, effectively dividing the value stored in the %rdx register by 4. 7. The value stored in the %rcx register is moved to the %rax register. 8. The sar instruction is a shift arithmetic right instruction, which shifts the bits of the operands to the right by 3 positions, effectively dividing the value is stored in the %rax register by 2^3=8, then storing it back into the %rax register. 9. This subtracts the value stored in the 5rax register with the value stored in the %rdx register, then stores this result back into the %rdx register. 10. This moves the value in the %rdx register into the %rax register. 11. This moves the previously stored value in the %rax register and stores it into the location -0x28(%rbp). This is the variable for binaryNumebr. 12. This instruction tells the execution to jump to the address located at 0x62, which is line 98 of the main function and the beginning of the while loop where the comparison will be made to determine if the while loop should execute again or not. |
| 1. 0x00000000000000f2 <+242>:   lea 0x0(%rip),%rsi # 0xf9 <main+249>   1. 0x00000000000000f9 <+249>:   lea 0x0(%rip),%rdi # 0x100 <main+256>   1. 0x0000000000000100 <+256>:   callq 0x105 <main+261> | 1. This instruction loads the effective address 0x0(%rip) into the %rsi register, the second argument for the function call. 2. This instruction loads the effective address 0x0(%rip), the next instruction into the %rdi register, the first srgument for the following function call. 3. This function call is used in preparation for the decimalNumber conversion to hexadecimal. |
| 1. 0x0000000000000105 <+261>:   mov %rax,%rdx   1. 0x0000000000000108 <+264>:   mov -0x20(%rbp),%rax   1. 0x000000000000010c <+268>:   mov %rax,%rsi   1. 0x000000000000010f <+271>:   mov %rdx,%rdi   1. 0x0000000000000112 <+274>:   callq 0x117 <main+279> | 1. This moves the value previously stored in the %rax register into the %rdx register. 2. This moves the value stored at location -0x20(%rbp), the decimalNumber variable, into the %rax register. 3. This moves the vale stored in the %rax register into the %rsi register, the second argument register for the function call. 4. This moves the value stored in the %rdx register into the %rdi register, the first argument register for the following function call. 5. This calls the function to convert the decimalNumber to a hex value and turn it into a string. |
| 1. 0x0000000000000117 <+279>:   mov %rax,%rdx   1. 0x000000000000011a <+282>:   mov 0x0(%rip),%rax # 0x121 <main+289>   1. 0x0000000000000121 <+289>:   mov %rax,%rsi   1. 0x0000000000000124 <+292>:   mov %rdx,%rdi   1. 0x0000000000000127 <+295>:   callq 0x12c <main+300> | 1. This moves the value stored in the %rax register into the %rdx register. 2. This moves the value stored at the address 0x0(%rip) which is the next instruction, into the %rax register. 3. This moves the value stored in the %rax register into the %rsi register, the second argument register used for the function call. 4. This moves the value stored in the %rdx register into the %rdi register, the first argument register for use with the folowing function call. 5. This calls the function to output the string “Quivalent hexadecimal value: “ followed by the resulting hexOutput string. |
| 1. 0x000000000000012c <+300>:   mov $0x0,%eax   1. 0x0000000000000131 <+305>:   mov -0x8(%rbp),%rsi   1. 0x0000000000000135 <+309>:   xor %fs:0x28,%rsi   1. 0x000000000000013e <+318>:   je 0x145 <main+325>   1. 0x0000000000000140 <+320>:   callq 0x145 <main+325>   1. 0x0000000000000145 <+325>:   leaveq   1. 0x0000000000000146 <+326>:   retq | 1. This instruction moves the value 0 into the %eax register. This is to be used to indicate that the program executed without error. 2. The value stores at location -0x8(%rbp) is moved into the %rsi register. 3. This makes an XOR comparison on the value stored in the %fs:0x28 fragment segment register with an 28 offset, with the value stored in the %rsi register. This is typically used for error checking. If the result is zero, the zero flag is set for the following instruction. 4. If the zero flag is set from the previous instruction, the execution jumps to address 0x145, which is line 325 of the main function. 5. This line is reached after error checking from the XOR operation and calls theaddress at 0x145, which is line 325 of the main function. 6. Restores the previous stack frame. 7. Returns from the main function. |
|  |  |

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

**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** |
| --- | --- | --- |
| 1. 0x0000000000000000 <+0>:   push %rbp   1. 0x0000000000000001 <+1>:   mov %rsp,%rbp   1. 0x0000000000000004 <+4>:   sub $0x30,%rsp | int main() {  long long binary;  int remainder; | Declares a function named main of type int, the entry point for the program. We see this accomplished in the assembly code by the new stack frame being set up for the main function. Additionally, 42 bytes are allocated on the stack for local variables. |
| 1. 0x0000000000000008 <+8>:   mov %fs:0x28,%rax   1. 0x0000000000000011 <+17>:   mov %rax,-0x8(%rbp)   1. 0x0000000000000015 <+21>:   xor %eax,%eax   1. 0x0000000000000017 <+23>:   movq $0x0,-0x20(%rbp)   1. 0x000000000000001f <+31>:   movq $0x1,-0x18(%rbp)   1. 0x0000000000000027 <+39>:   lea 0x0(%rip),%rsi # 0x2e <main+46>   1. 0x000000000000002e <+46>:   lea 0x0(%rip),%rdi # 0x35 <main+53>   1. 0x0000000000000035 <+53>:   callq 0x3a <main+58>   1. 0x000000000000003a <+58>:   mov %rax,%rdx   1. 0x000000000000003d <+61>:   mov 0x0(%rip),%rax # 0x44 <main+68>   1. 0x0000000000000044 <+68>:   mov %rax,%rsi   1. 0x0000000000000047 <+71>:   mov %rdx,%rdi   1. 0x000000000000004a <+74>:   callq 0x4f <main+79> | int decimalNumber = 0;  int i = 0;   cout << “Enter a binary number: \n”; | With this block of code, we are seeing variables decimalNumber and I declared as type integers and initialized.  Then the function is called to output the string: “Enter a binary number: “  Ines 12-16 of the assembly call the function to output the newline. |
| 1. 0x000000000000004f <+79>:   lea -0x28(%rbp),%rax   1. 0x0000000000000053 <+83>:   mov %rax,%rsi   1. 0x0000000000000056 <+86>:   lea 0x0(%rip),%rdi # 0x5d <main+93>   1. 0x000000000000005d <+93>:   callq 0x62 <main+98> | cin >> binaryNumber; | This calls the function to read in input from the user and store the value in the variable bbinaryNumber. |
| 1. 0x0000000000000062 <+98>:   mov -0x28(%rbp),%rax   1. 0x0000000000000066 <+102>:   test %rax,%rax   1. 0x0000000000000069 <+105>:   je 0xf2 <main+242> | while (binaryNumber!=0) { | This tests to see if the value stored in the binaryNumber variable is equal to zero, and if it is not the while loop will execute, otherwise it will jump to the portion of code after the while loop. |
| 1. 0x000000000000006f <+111>:   mov -0x28(%rbp),%rcx   1. 0x0000000000000073 <+115>:   movabs $0x6666666666666667,%rdx   1. 0x000000000000007d <+125>:   mov %rcx,%rax   1. 0x0000000000000080 <+128>:   imul %rdx   1. 0x0000000000000083 <+131>:   sar $0x2,%rdx   1. 0x0000000000000087 <+135>:   mov %rcx,%rax   1. 0x000000000000008a <+138>:   sar $0x3f,%rax   1. 0x000000000000008e <+142>:   sub %rax,%rdx   1. 0x0000000000000091 <+145>:   mov %rdx,%rax   1. 0x0000000000000094 <+148>:   mov %rax,-0x10(%rbp) | remainder = binarynumber % 10; | This line of code calculates the value of remainder by performing the operation binaryNumber % (modulo) 10. This effectively calculates the remainder of a number after dividing that number by 10. |
| 1. 0x0000000000000098 <+152>:   mov -0x10(%rbp),%rdx   1. 0x000000000000009c <+156>:   mov %rdx,%rax   1. 0x000000000000009f <+159>:   shl $0x2,%rax   1. 0x00000000000000a3 <+163>:   add %rdx,%rax   1. 0x00000000000000a6 <+166>:   add %rax,%rax   1. 0x00000000000000a9 <+169>:   sub %rax,%rcx   1. 0x00000000000000ac <+172>:   mov %rcx,%rax   1. 0x00000000000000af <+175>:   mov %rax,-0x10(%rbp)   1. 0x00000000000000b3 <+179>:   mov -0x10(%rbp),%rax   1. 0x00000000000000b7 <+183>:   imul -0x18(%rbp),%rax   1. 0x00000000000000bc <+188>:   add %rax,-0x20(%rbp)   1. 0x00000000000000c0 <+192>:   shlq -0x18(%rbp)   1. 0x00000000000000c4 <+196>:   mov -0x28(%rbp),%rcx   1. 0x00000000000000c8 <+200>:   movabs $0x6666666666666667,%rdx   1. 0x00000000000000d2 <+210>:   mov %rcx,%rax   1. 0x00000000000000d5 <+213>:   imul %rdx   1. 0x00000000000000d8 <+216>:   sar $0x2,%rdx   1. 0x00000000000000dc <+220>:   mov %rcx,%rax   1. 0x00000000000000df <+223>:   sar $0x3f,%rax   1. 0x00000000000000e3 <+227>:   sub %rax,%rdx   1. 0x00000000000000e6 <+230>:   mov %rdx,%rax   1. 0x00000000000000e9 <+233>:   mov %rax,-0x28(%rbp)   1. 0x00000000000000ed <+237>:   jmpq 0x62 <main+98> | binaryNumber /= 10;  decimalNumber += remainder \* (2 ^ i);  i++; | This calculates the new binary number by dividing its current value by 10 and then storing the result back into the binaryNumber variable.  The new value of decimalNumber is calculated by adding its current value to the value of the operation: remainder \* (2 ^ i), which is esentially saying remainder \* 2 to the power of i. The result of this calculation is then stored back into the decimalNumber variable.  The variable i is incremented by one for each iteration of the while loop, which is used in the previous operation when calculating powers. |
| 1. 0x00000000000000f2 <+242>:   lea 0x0(%rip),%rsi # 0xf9 <main+249>   1. 0x00000000000000f9 <+249>:   lea 0x0(%rip),%rdi # 0x100 <main+256>   1. 0x0000000000000100 <+256>:   callq 0x105 <main+261> | stringstream hexStream; | Declares a stringstream variable called hexStream. This is used for the following conversion od the decimalNumber to hexadecimal value. |
| 1. 0x0000000000000105 <+261>:   mov %rax,%rdx   1. 0x0000000000000108 <+264>:   mov -0x20(%rbp),%rax   1. 0x000000000000010c <+268>:   mov %rax,%rsi   1. 0x000000000000010f <+271>:   mov %rdx,%rdi   1. 0x0000000000000112 <+274>:   callq 0x117 <main+279> | hexStream << hex << decimalNumber;  string hexOutput = hexStream.str(); | This converts the decimalNumber variable to a hex value.  This takes the resulting hex value and turns it into a readble string placing the resulting string in the variable hexOutput. |
| 1. 0x0000000000000117 <+279>:   mov %rax,%rdx   1. 0x000000000000011a <+282>:   mov 0x0(%rip),%rax # 0x121 <main+289>   1. 0x0000000000000121 <+289>:   mov %rax,%rsi   1. 0x0000000000000124 <+292>:   mov %rdx,%rdi   1. 0x0000000000000127 <+295>:   callq 0x12c <main+300> | cout << “Equivalent hexadecimal value: “ << hexOutput; | Calls the output function to display the string: “Equivalent hexadecimal vale: “ followed by an additional output call printing the string hexOutput. |
| 1. 0x000000000000012c <+300>:   mov $0x0,%eax   1. 0x0000000000000131 <+305>:   mov -0x8(%rbp),%rsi   1. 0x0000000000000135 <+309>:   xor %fs:0x28,%rsi   1. 0x000000000000013e <+318>:   je 0x145 <main+325>   1. 0x0000000000000140 <+320>:   callq 0x145 <main+325>   1. 0x0000000000000145 <+325>:   leaveq   1. 0x0000000000000146 <+326>:   retq | Return 0; | Returns 0 because the main function is of type integer and expects an integer value to be returned. Returning a 0 value indicates to the operating system that the program exited without any errors. |