Reference C++ code

```cpp
int dummy(int x) {
    int ret = x * 19;
    return ret;
}
```

Reference AT&T ASM code

```
	.file	"cppAssembly3.cpp"
	.text
	.globl	_Z5dummyi
	.type	_Z5dummyi, @function
_Z5dummyi:
.LFB0:
	.cfi_startproc
	endbr64
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movl	%edi, -20(%rbp)
	movl	-20(%rbp), %edx
	movl	%edx, %eax
	sall	$3, %eax
	addl	%edx, %eax
	addl	%eax, %eax
	addl	%edx, %eax
	movl	%eax, -4(%rbp)
	movl	-4(%rbp), %eax
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE0:
	.size	_Z5dummyi, .-_Z5dummyi
	.ident	"GCC: (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0"
	.section	.note.GNU-stack,"",@progbits
	.section	.note.gnu.property,"a"
	.align 8
	.long	1f - 0f
	.long	4f - 1f
	.long	5
0:
	.string	"GNU"
1:
	.align 8
	.long	0xc0000002
	.long	3f - 2f
2:
	.long	0x3
3:
	.align 8
4:
```

1. **(Max) Given an integer x, can x multiply by 19 be implemented by using shifts and adds only? How?**

   First we define "x multiply by 19" as the equation:

$$x \cdot 19 = 19x$$

   Notice that 19 can be broken into a sum of 16 and 3 which allows us to rewrite the above to:

$$19x = (16 + 3)x$$

$$19x = 16x + 3x$$

   Bit shifting manipulates numbers through powers of 2, we want to break down the expression such that we can express it as the sum of powers of 2. By inspection we can see that 16 is a power of 2. Using the same technique above we can also break down 3.

$$19x = 2^4x + (2 + 1)x$$

$$19x = 2^4x + 2^1x + 2^0x$$

   The expression 2^0 is just 1 and x multiplied by 1 is just x, so we end up with the following:

$$19x = 2^4x + 2^1x + x$$

   Now that we're able to represent the product as a sum of powers of 2, we can then substitute the individual products to their corresponding bit shifts.

$$19x = (x << 4) + (x << 1) + x$$

2. **(Max) What does the assembly version do? Does it use the multiply instruction?**

   We start by getting the assembly instructions that correspond with the arithmetic logic of the `dummy` function. When a C++ source file is compiled to assembly with the g++ compiler, the assembly instructions that correspond to the function body is located inside the block labeled with the function's *mangled name*.

   Since the function's name is `dummy`, a 5 letter string, and it takes in 1 integer parameter, instructions pertaining to the logic of the function body can be found inside the block labeled `_Z5dummyi`. For the specifics of the GAS name mangling syntax we consulted this resource.

```
_Z5dummyi:
.LFB0:
    .cfi_startproc
    endbr64
    pushq       %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq        %rsp, %rbp
    .cfi_def_cfa_register 6
    movl        %edi, -20(%rbp)
    movl        -20(%rbp), %edx
    movl        %edx, %eax
    sall        $3, %eax
    addl        %edx, %eax
    addl        %eax, %eax
    addl        %edx, %eax
    movl        %eax, -4(%rbp)
    movl        -4(%rbp), %eax
    popq        %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

Ignoring the instructions allotted for the function prologue and epilogue and for stack management, the arithmetic logic of the `dummy` function is contained within the lines:

```
movl        %edx, %eax
sall        $3, %eax
addl        %edx, %eax
addl        %eax, %eax
addl        %edx, %eax
movl        %eax, -4(%rbp)
popq        %rbp
```

The value inside the `%edx` register (the function parameter) is copied into the register `%eax`, where it is then shifted 3 bits to the left and stored back in the same register. `%eax` is then incremented by the value inside `%edx` which remains unchanged. Let x denote the function parameter x, the value inside the register `%eax` can be obtained by performing the following operations.

$$\texttt{\%eax} = 0 + \texttt{\%edx} = x$$

$$\texttt{\%eax} = x << 3 = 8x$$

$$\texttt{\%eax} = 8x + \texttt{\%edx} = 8x + x = 9x$$

At the fourth line `%eax` is added onto itself and stored back inside `%eax`, after which it is again incremented by the the value inside `%edx` which is still just x. The value inside the `%eax` register is now $19x$, the desired product.

$$\texttt{\%eax} = \texttt{\%eax} + \texttt{\%eax} = 9x + 9x = 18x$$

$$\texttt{\%eax} = 18x + \texttt{\%edx} = 18x + x = 19x$$

The final lines:

```
movl        %eax, -4(%rbp)
popq        %rbp
```

. . . copies the value stored inside `%eax` into the base pointer to which it is then popped back to the main stack—effectively "returning" the value of the function. **The assembly code generated by the g++ compiler did NOT use the multiply instruction. Instead it performed the appropriate bit shifts and additions to reproduce the desired expression.**

3. **(Paco) What happens for the case of $x \cdot 45$?**

Rewriting the C++ code to reflect the new expression $45 \cdot x$

```
int dummy(int x) {
    int ret = x * 45;
    return ret;
}
```

We recompile to assembly again to produce the following instruction sequence, instructions pertaining to the function prologue, epilogue and stack management were ignored:

```
movl        %edi, -20(%rbp)
movl        -20(%rbp), %eax
imull        $45, %eax, %eax
movl        %eax, -4(%rbp)
movl        -4(%rbp), %eax
popq        %rbp
```

In the case of x being multiplied by 45, the compiler does not perform any optimizations and does the multiplication directly through the imull instruction.

4. **(Ian) What happens for the case of $x \cdot -2$?**

   Rewriting the C++ code to reflect the new expression $-2 \cdot x$

   ```
   int dummy(int x) {
       int ret = x * -2;
       return ret;
   }
   ```

   We recompile to assembly again to produce the following instruction sequence, instructions pertaining to the function prologue, epilogue and stack management were ignored:

   ```
   movl        %edi, -20(%rbp)
   movl        -20(%rbp), %edx
   movl        $0, %eax
   subl        %edx, %eax
   addl        %eax, %eax
   movl        %eax, -4(%rbp)
   movl        -4(%rbp), %eax
   popq        %rbp
   ```

   In the case of x being multiplied by -2, 0 is first loaded into the return register `%eax`. x is then subtracted from from the value stored inside `%eax`, of which the difference is added to itself. At this point the value stored inside `%eax` already sufficiently reproduces the expression $-2 \cdot x$.

5. **(Ian) What happens for the case of $x \cdot 0$?**

   Rewriting the C++ code to reflect the new expression $0 \cdot x$

   ```
   int dummy(int x) {
       int ret = x * 0;
       return ret;
   }
   ```

   We recompile to assembly again to produce the following instruction sequence, instructions pertaining to the function prologue, epilogue and stack management were ignored:

   ```
   movl        %edi, -20(%rbp)
   movl        $0, -4(%rbp)
   movl        -4(%rbp), %eax
   popq        %rbp
   ```

   In the case of x being multiplied by 0, the compiler automatically detects that one of the factors of the product is a 0. Thus the return value is automatically a 0.