While enjoying the entries from *The International Obfuscated C Code Contest*, I’ve decided to examine one of them, Dave Burton’s the best one-liner from the 27th IOCCC Winner.

The program

bin.c

```
int main(int b,char**i){long long n=B,a=I^n,r=(a/b&a)>>4,y=a-
toi(*++i),_=(((a^n/b)*(y>>T)|y>>S)&r)|(a^r);print-
f("%.8s\n",(char*)&_);}
```

The program takes a decimal value from 0 to 255 (for little endian systems) or 0 to 511 (for big endian systems) and prints the binary version of it. Here is an example of outputs.

```
>prog.exe 4
>00000100
```

```
>prog.exe 255
>11111111
```

How does it work?

- B, I, T, and S are macros that should be given while compiling with the help of compile flags. Here are the values for little-endian systems.

```
-DB=6945503773712347754LL -DI=5859838231191962459LL -DT=0 -DS=7
```

- '_' is a variable name. I changed it to 'foo'.
- int b, char**i are often written as int argc, char *argv[] respectively.
- The program takes one argument so b is 2.
  (because C standards say that the first arg should be the name of the program)

Under these conditions and after some calculations with given hardcoded values, the program can be rewritten as follows.

```
int main(int b, char**i)
{
  long long y=atoi(*++i);
  long long foo=(((72198606942111748LL*y)|(y>>7)) &
72340172838076673LL) | 3472328296227680304LL;
  printf("%.8s\n",(char*)&foo);
}
```

Only four bitwise operations and one truncation (by printf) are enough to give this functionality to this program. Let’s continue examining.

I gave names to these numbers alpha, beta and gamma.

```
#define alpha 72198606942111748LL
//0000000100000000100000000100000000100000000100000000100000000100

#define beta 72340172838076673LL
//0000000100000001000000010000000100000001000000010000000100000001

#define gamma 3472328296227680304LL
//0011000000110000001100000011000000110000001100000011000000110000

int main(int b, char**i)
{
  long long y = atoi(*++i);
  long long foo = (((alpha*y) | (y>>7)) & beta) | gamma;
  printf("%.8s\n", (char*)&foo);
}
```

Now, we will multiply numbers less than 256 with alpha.

```
// 0 * alpha
0000000000000000000000000000000000000000000000000000000000000000
```

```
// 1 * alpha
00000001000000001000000001000000001000000001000000000100000000100

// 2 * alpha
00000010000000010000000010000000010000000010000000001000000001000

// 127 * alpha
01111111001111111001111111001111111001111111001111111100111111100

// 255 * alpha
11111111011111111011111111011111111011111111011111111101111111100

// define abcdefgh_binary = y_decimal

// result = y * alpha
abcdefgh0abcdefgh0abcdefgh0abcdefgh0abcdefgh0abcdefgh0abcdefgh00
```

doing “| y >> 7” operation

```
// User input y is bounded to 0-255 (8-bit).
y>>7 = 1 (if y is 128-255)
y>>7 = 0 (if y is 0-127)

// we called y_decimal = abcdefgh_binary
// under this condition
y>>7 = a

// result2 = result | y>>7
abcdefgh0abcdefgh0abcdefgh0abcdefgh0abcdefgh0abcdefgh0abcdefgh00 (re-
sult)
00000000000000000000000000000000000000000000000000000000000000a
(y>>7)
abcdefgh0abcdefgh0abcdefgh0abcdefgh0abcdefgh0abcdefgh0abcdefgh0a (re-
sult2)
```

then we are going to do bitwise and operation with beta after then, or

operation with gamma.

```
// & operation
abcdefgh0abcdefgh0abcdefgh0abcdefgh0abcdefgh0abcdefgh0abcdefgh0a (re-
sult2)
0000000100000001000000010000000100000001000000010000000100000001 (be-
ta)
0000000h0000000g0000000f0000000e0000000d0000000c0000000b0000000a (re-
sult3)

// | operation
0000000h0000000g0000000f0000000e0000000d0000000c0000000b0000000a (re-
sult3)
0011000000110000001100000011000000110000001100000011000000110000
(gamma)
0011000h0011000g0011000f0011000e0011000d0011000c0011000b0011000a
(foo)
```

Finally printing the value we got. printf() prints the ascii chars up to null char if a char array is given.

```
char* example = {'A', 'B', '\x0'};
printf("%s", example);
// output: AB

// printf takes a char pointer and print the value that pointer
points.
// then, it increases pointer varible by doing pointer aritmetic and
prints the value of (pointer+1)
// brifly what printf is doing is:
// print *(pointer)
// print *(pointer+1)
// print *(pointer+2), printf encounters null char so stops.

// %.8 is just limiting the output size. printf will print up to 8
```

```
char then ignore the rest.
```

Endianness plays an important role here. Little-endian machines like windows store the least significant bytes first.

```
// final value 'foo' to print
0011000h0011000g0011000f0011000e0011000d0011000c0011000b0011000a
0011000h 0011000g 0011000f 0011000e 0011000d 0011000c 0011000b
0011000a

// this value stored in little endian machines like that
0011000a 0011000b 0011000c 0011000d 0011000e 0011000f 0011000g
0011000h

// printf sees it as char array. every bytes will be printed as
ascii char.
00110000 = 0x30 = '0'
00110001 = 0x31 = '1'

// final char array, remember y_decimal = abcdefgh_binary
a b c d e f g h

// final char array doesn't have the null terminator but %.8 limits
already.
```

Final Thought

Briefly, this program takes a decimal input and creates a char array that contains either ‘0’ or ‘1’ by manipulating bits.

I’m a bit lazy to show the calculations for the big-endian machines. But what is amazing about this program is it is actually working on big-endian machines as well with some different hardcoded input values.