We Can Compute Many Boolean Values Simultaneously

Let’s do an example with bitwise operators.
In this example,
◦ we calculate all rows
◦ of a truth table for function $F(A,B,C)$
◦ using a single assignment.
To accomplish this task,
◦ we use one bit of variables $A$, $B$, $C$, and $F$
◦ for each row of the truth table.

Use Unsigned Data Types for Bit Operations

When working with bits, we **use unsigned data types**.
This choice is particularly important
◦ when shifts are necessary,
◦ but it’s a good habit for all bit operations.
Our function has three input variables,
◦ so we need eight bits
◦ to represent the eight rows
of the truth table.

Assign Truth Table Row Values to Input Bits

We first declare the three input variables:
```c
uint8_t A = 0xF0; /* A is 11110000 */
uint8_t B = 0xCC; /* B is 11001100 */
uint8_t C = 0xAA; /* C is 10101010 */
```
Notice that each triple of bits (vertically)
◦ forms a unique combination of input values
◦ from **111 for bit 7** down to **000 for bit 0**.

*A uint8_t is the same as an unsigned char (8-bit unsigned), but see the notes for details.*
Compute All Values of $F(A,B,C)$ Using One Assignment

We also declare two other variables:

```c
uint8_t F; /* the function F */
int32_t i; /* truth table row iteration variable */
```

Let's say that $F(A,B,C) = (A+B)(A'+C')$

We can calculate

- all possible values of $F$
- with one statement using bitwise operators:

```c
F = ((A | B) & ((~A) | (~C)));
```

Pitfall: Using Larger Data Types

What happens if we instead use, for example, `uint32_t` for $A$, $B$, $C$, and $F$?

If our initializations are the same, all higher bits of $A$, $B$, and $C$ are 0.
So those bits of $F$ correspond to $F(0,0,0)$.

*Do not assume that those bits are 0!*

Instead, use another bitwise AND to mask the higher bits out (force them to 0) before using $F$.

We Print One Row at a Time

Now we can print the header for our truth table.

```c
printf ("A B C | F\n");
printf ("-------+-\n");
```

To print the rows, we use

- a single `printf` per row:

```c
for (i = 0; 8 > i; i = i + 1) {
    /* printf goes here */
}
```

The `printf` Merely Extracts Bits of $F$

The input variables

- are extracted from the loop index $i$, and
- $F(A,B,C)$ is the $i$th bit of $F$.

```c
printf ("%c %c %c | %c\n",
    '0' + (0 != (i & 4)), /* A */
    '0' + (0 != (i & 2)), /* B */
    '0' + (0 != (i & 1)), /* C */
    '0' + (0 != (F & (1 << i))));
```