ENEL260 CA2: Sequences and Codes

Ciaran Moore

"It's turtles all the way down!"

As we saw in the previous lecture, a single logical variable answers one *yes/no* question. Assembling groups of logical variables to answer a sequence of *yes/no* questions lets us represent complex information. Once we have groups of logical variables we need ways to talk about some or all of the variables in a group, and to move groups of values around. That leads us to concepts like bytes and words.

As we saw in the last lecture, digital computers represent information using *logical variables*. A logical variable answers a single *yes/no* question. Where we use 0 and 1 to represent our logic variable values, we call these *bits* (binary digits). We can use *logic gates* to combine logical values from various sources to generate other useful information.

We assign a "meaning" to a logical variable by deciding what question it answers.

Example: You might design a simple seatbelt warning light system using the circuit in Fig. 1, which illuminates the warning light if the speed is greater than 0 and the seatbelt is not buckled.

Example derived from F. Vahid, *Digital Design*, Wiley, 2007.

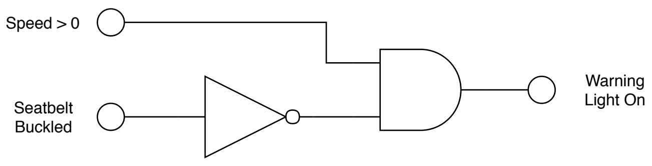

Figure 1: Simple seatbelt warning system

This system works. But it's not particularly flexible. For example, if we want to add the ability to control the warning based on whether a passenger is present, we need to change the hardware. Or if we want to change the speed threshold (perhaps to let very low speeds be acceptable), then we need to change the hardware. What we really want is a more flexible system that allows us to change the logic without having to redesign the hardware. But that requires being able to, for example, take a speed value as an input and do computations on it.

Looking Deeper: Warning System Models You can find some Python-based combinational logic models of variations on the seatbelt warning system in the Computer Architecture Code Examples folder on Learn. Try generating a truth table for one of the designs, to make sure it works the way you expect:

```
>>> for buckled in (0,1):
...   for speed in (0,1):
...     w = seatbelt_warning(speed, buckled)
...     print("{} {} | {}".format(buckled, speed, w))
...
0 0 | 0
0 1 | 1
1 0 | 0
1 1 | 0
```

You can also try modifying the existing designs, or creating new ones by additional further logic gates.

We're going to build up to creating flexible digital logic that can change its behaviour without requiring a hardware redesign. The first step is to look at what we can do if we want to represent information that's more complex than just a single *yes/no* question.

Example: One of the last things we looked at in the previous lecture was the *full adder* circuit, which does binary arithmetic. But, just like adding 5 and 5 in normal base-10 arithmetic takes us outside the $0-9$ range and makes us "carry" a 1 into the 10s place, adding 1 and 1 in binary (base-2) arithmetic results in a number bigger than 1. So a full-adder has two outputs: a sum, and a "carry out" that we use to represent the situation where the sum overflows the $0-1$ range. In other words, the adder output is really *two bits* that together represent information about the sum of the inputs.

What does it mean to put two bits together, like the full-adder example above? Let's take a look.

Sequences

We'll start by ignoring "logical variables" and digital logic theory for a moment, and just think about the practicalities of programming an embedded system.

Example:

Imagine that you're programming a simple little embedded system that has a two clusters of LEDs you can use to show a pattern (or sequence) of colours that will communicate some kind of information (See Fig. 2). Each cluster of LEDs can be either red, green, or blue.

How many different things can your system say? Obviously it's more than just "yes" or "no". The actual answer depends on how many different colour sequences[1] there are. If you list all the possibilities you'll find that there are *nine* possible sequences:

RR RB RG BR BB BG GR GB GG

So your system can say nine different things.

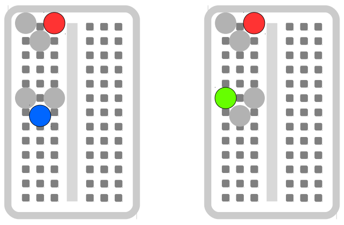

Figure 2: A transition from displaying Red-Blue to displaying Red-Green

[1] Here we'll ignore the situation when an LED might not be on at all–that would just count as another "colour".

Example: If instead you can display a pattern of three clusters of coloured LEDs, where each LED cluster can be either red or blue, then how many sequences are there? If you list all the possibilities you'll find there are *eight* possible sequences:

RRR RRB RBR RBB BRR BRB BBR BBB

In general, if you have M colours and N clusters of LEDs then the number of possible patterns is M^N. In the second case, which is *binary* since $M = 2$, the number of patterns $= 2^3 = 8$.

In the same way, we can take a series of logical variables and form them into a sequence, $a, b, c, d, \ldots$. As with the LEDs, the order of the variables in the sequence is important. If we're using 0 and 1 bits as logical values, the value of the sequence of logical variables will be a sequence of bits called a *binary represented number*, or a *binary sequence*.

Codes

You've already encountered integers, character (ASCII or Unicode) codes, and floating point numbers. These are just arbitrary, but predefined, formats for binary sequences.
Note that most codes are not secret (secret codes are called ciphers).

We can assign a *meaning* to binary sequences, and these are then called *codes*. It's important to note that the same sequence can have an infinity of different meanings[2]. The same sequence of bits may represent the number 65, the letter `A`, a speed in kilometres per hour, or a shade of blue. The meaning which applies depends on *where* the sequence is located and *what format* it is known to have.

[2] Of course, the same thing applies to sequences of letters, which are just another kind of code. The same sequence of letters may have different meanings (or no meaning) in different languages.

Example: With the 8 LED colour patterns we can construct a *code* book. For example, we could assign a vegetable as the meaning for each code:

RRR	Tomato
RRB	Onion
RBR	Beans
RBB	Courgette
BRR	Potato
BRB	Corn
BBR	Pepper
BBB	Spinach

Now if the vegetable shopkeeper has a copy of this code book then you could purchase a potato by setting the LEDs to show BRR (blue-red-red).

The order of the colours matters. If we set the LEDs to show blue-red-red and showed it to the shopkeeper upside down, then we would get an onion instead of a potato.

Example: Maybe you don't find vegetables interesting. Fortunately, we can have many different code books. For example, here's one for types of Emerson's beers:

RRR	Weiss
RRB	Stout
RBR	Pilsner
RBB	IPA
BRR	Porter
BRB	Lager
BBR	Munchel
BBB	Taieri George

With this code-book, if you went to the Emerson's beer shop and asked for a blue-red-red you would get a porter (yum!). However, if you went to the vegetable shop by mistake you would get a potato.

If you had a shopping list of beers and a shopping list of vegetables there is no way of telling which is which without some additional context.

Representing Integers

Codes aren't just for representing vegetables or beer choices. They can also be used for representing numbers, which we can then compute with.

Example: Representing numbers is exactly what we're doing with the full-adder. If we think of the sum and carry-out outputs as being a two-bit sequence, we can assign base-10 values to each possible output sequence that match what we'd expect to see for an arithmetic sum of two input bits and a carry-in bit.

C_{out}	S	Base-10
0	0	0
0	1	1
1	0	2
1	1	3

Example: If we have a 3-bit binary sequence, we can assign 8 different numbers to the 8 possible codes. So we could represent the *unsigned* (non-negative) values 0 to 7 using the following set of codes:

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Returning to the idea of logical variables as yes/no questions, we can see each code as representing an answer to a sequence of questions, e.g. for code 101:

1. Is the number > 3? 1
2. Is the number > 5? 0
3. Is the number $= 5$? 1

When we're representing numbers we can treat each column of the code as having a different "weight"[3]. Here the left-hand column has a greater weight (changing it from 0 to 1 increases the value of the encoded number by 4), and so is called the *most significant bit* (MSB). The right-hand bit has the least weight (it only changes the encoded number by 1), and so is called the *least significant bit* (LSB).

[3] In the sense that changing the bit from 0 to 1 creates a greater change in the magnitude of the encoded number.

In general, with N bits, if we start from and include zero, the maximum unsigned value that we can represent is $2^N - 1$.

If we label the LSB as b_0, the next column as b_1, and so on until the MSB is labeled with b_{N-1}, we can write an equation for the encoded number in terms of the value of each bit:

$$\begin{aligned} v &= b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0 \\ &= \sum_{i=0}^{N-1} b_i \times 2^i \end{aligned}$$

Looking Deeper: Python Here's the binary-to-decimal conversion implemented in Python:

```python
def binary_to_decimal(bits):
    """Takes a list of bits (0 or 1), MSB first, and returns the
    corresponding decimal (base 10) value.
    """
    # Note that because we assume MSB first, we reverse the bit list
    # prior to enumeration so that the indices work out correctly
    return sum(bit*(2**i) for i, bit in enumerate(reversed(bits)))
```

Executing this function gives results like this:

```
>>> binary_to_decimal([1, 0, 0, 0])
8
>>> binary_to_decimal([0, 1, 0, 1])
5
>>> binary_to_decimal([1, 1])
3
```

Other Representations

Because a binary (base-2) sequence is positional we can easily convert it into other positional values, such as decimal (base-10) and *hexadecimal* (base-16)[4]. We use these other representations because they're more compact, i.e., take less space to write.

Aside: In a sense, by using a different representation we're *encoding* the binary code...

[4] See https://en.wikipedia.org/wiki/Positional_notation#Base_conversion for more.

Example: Hexadecimal is particularly common in embedded systems since it's much more compact than binary notation, and easier than decimal numbers to convert to and from bit patterns. So let's see how we can relate hexadecimal digits to binary sequences. We use 4-bit binary sequences, since that gives us the $2^4 = 16$ sequences needed to represent the 16 possible values of a hexadecimal digit.

Exercise: Given the table to the left, work out the following conversions:

- `0x8F` to binary
- 259_{10} to hex
- `0x1FB` to decimal
- 16-bit binary pattern `0011101110011110` to hex

pattern	hexadecimal	decimal	pattern	hexadecimal	decimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	*A*	10
0011	3	3	1011	*B*	11
0100	4	4	1100	*C*	12
0101	5	5	1101	*D*	13
0110	6	6	1110	*E*	14
0111	7	7	1111	*F*	15

In C, a hexadecimal number is indicated by the prefix '`0x`', thus the decimal value 17 is represented as `0x11`. .

Similarly, binary numbers in C begin with '`0b`'. For example, `0b1101` represents decimal value 13.

Because a single hexadecimal digit corresponds to a 4-bit sequence, we can represent an 8-bit byte using two hex digits. Thus the possible values of a single byte range from `0x00` to `0xFF`.

Looking Deeper: Base Conversion The classic method of performing base conversion involves iterated (integer) division of the number to be converted by the new base. At the i^{th} step you store the remainder of the division as the value for the *base*^{i} position. The iterated division continues until the results of the division is 0. Here's an example of using this approach to convert from base-10 to base-16:

```
def decimal_to_hex(decimal):
    """Takes a decimal number (digits 0-9), MS digit first, and returns the
    corresponding hexadecimal (base 16) value as a string.
    """
    lookup = {0:'0', 1:'1', 2:'2', 3:'3', 4:'4', 5:'5', 6:'6', 7:'7', 8:'8',
              9:'9', 10:'A', 11:'B', 12:'C', 13:'D', 14:'E', 15:'F'}
    hexdigits = []  # We'll build a list of hex digits

    while not decimal == 0:
        remainder = decimal % 16       # Find the remainder
        hexdigits += lookup[remainder] # Convert to hex digit and save
        decimal = decimal // 16        # Integer division (//)
    return "0x" + ''.join(reversed(hexdigits))
```

And here's a few examples of the fucntion in use:

```
>>> decimal_to_hex(245)
'0xF5'
>>> decimal_to_hex(254)
'0xFE'
>>> decimal_to_hex(10)
'0xA'
```

Signed Values

For *signed* integers, we could use one of the bits to indicate whether the number is positive or negative. For example, we could assign values -3 to 3 using :

000	0
001	1
010	2
011	3
100	-0
101	-1
110	-2
111	-3

Here the MSB is 1 if the represented number is negative.

The problem with this scheme (called *sign-magnitude*) is that we have *two representations* for zero. To avoid this (and for a few other reasons), computers use an encoding called *two's complement*:

000	0
001	1
010	2
011	3
100	-4
101	-3
110	-2
111	-1

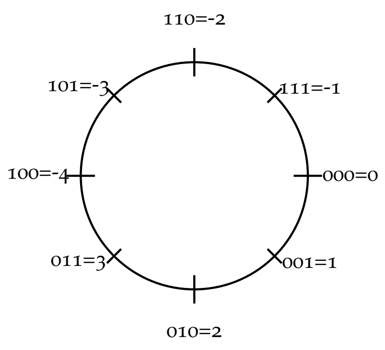

Figure 3: The 2's complement number circle.

Fig. 3 illustrates the "number circle" for 2's complement representations.

As with the sign-magnitude representation, in a 2's complement representation the MSB is 1 if the represented number is negative.

Note that there is one more negative value than there are positive values (ignoring zero). In general, with N bits the maximum two's complement value we can represent is $2^{N-1} - 1$ and the minimum two's complement value we can represent is -2^{N-1}. For example,

To find the 2's complement representation of a negative number, take the bitwise logical complement of the corresponding positive number and add 1. E.g., 3 is 011, so -3 is $011 + 001 = 100 + 001 = 101$.

Bits	Max. unsigned	Max. signed	Min. signed
N	$2^N - 1$	$2^{N-1} - 1$	-2^{N-1}
3	7	3	-4
8	255	127	-128
16	65535	32767	-32768
32	4294967295	2147483647	-2147483648

Bitwise Logic in C

The C language provides two distinct mechanisms for dealing with Boolean logical operations. Both mechanisms offer the sufficient set of operations: AND, OR and NOT. The difference between the two mechanisms lie in whether to treat variables as logical (i.e., TRUE or FALSE) or as a multiple-bit sequence; this gives rise to the logical and *bitwise* operators respectively.

Recall that we have already seen that in C the Boolean logical operators are: NOT: `!a` , OR: `a || b` , AND: `a && b` . Here the variables `a` and `b` are interpreted as either TRUE or FALSE.

The **bitwise operators** take into account the multi-bit nature of variables and the data-types in C. So the C bitwise operators perform multiple-bit operations in parallel. These operators are:

- NOT: `~a` also called 'complement'
- OR: `a | b`
- AND: `a & b`
- XOR: `a ^ b` (as noted earlier, XOR is a useful extra operation)

Example: The following example code shows various bitwise operations, with the results of each assignment operation shown in the comments:

```
uint8_t a,b;
uint8_t result;

a = 0x99;           // 1001 1001
b = 0x55;           // 0101 0101
result = ~a;        // 0110 0110
result = a | ~b;    // 1011 1011
result = a & b;     // 0001 0001
result = a ^ b;     // 1100 1100
result = a || b;    // TRUE   (0x01)
result = a && b;    // TRUE   (0x01)
```

Masking

By using a bitwise operator in combination with a *mask* (i.e. a bit pattern with only the specific bit or bits nonzero) we can read or modify individual bits within a byte or word. For example, an 8-bit mask to read or modify bit 0 is `0x01 = 0000 0001` , while one for bit 3 is `0x08 = 0000 1000` .

Example: This fragment of C code illustrates the use of masks:

```
    #define  PIN3_ON    0x08       // 0000 1000
    #define  PIN2_ON    0x04       // 0000 0100
    #define  PIN1_ON    0x02       // 0000 0010
    #define  PIN0_ON    0x01       // 0000 0001
    /* 'portb' is the address of an externally addressed 8-bit interface -
     * a 'port'; the bits within a port are often called 'pins'.   */

    uint8_t mask1, mask2, result;

    mask1 = PIN2_ON;               // 0000 0100
    mask2 = PIN1_ON | PIN3_ON;     // 0000 1010
    ...
    result = portb & mask1;        // 0000 0#00 (# is the state of the selected
    pin)
    if (result)
       {  ...  }                   // Do this only if Pin 2 of Port B is 1
    portb = portb | mask2;         // Set Pin 1 and Pin 3 of Port B without
                                   //  changing any other pins
```

Shifting

In addition to using the bitwise logical operators, we can modify the bits in a byte or word using the *shift* operators. These operators move bits about:

1. Shift Left (<<) :
2. Shift Right (>>) :

A left shift is equivalent to multiplying by a power of 2.

A right shift is equivalent to dividing by a power of 2.

Example: Some examples of the shift operators in action:

```
// Shift Left

uint8_t a, b = 2, c;
a = 1;       //  0000 0001
c = a << 1;  //  0000 0010
c = a << b;  //  0000 0100
```

```
c = a << 3;  //  0000 1000

// Shift Right
b = 16;      //  0001 0000
c = b >> 1;  //  0000 1000
c = b >> 3;  //  0000 0010
```

Bytes, Words, and Memory

Computers are built to handle binary sequences that are of a fixed length, or are a multiple of that fixed length[5]. This is done for reasons of efficiency and economy. The fixed length for a particular architecture is known as the *word size*.

[5] Using a fixed length (and multiples of that length) means that the size of data can be assumed instead of having to detect the end of each binary sequence, and that the design can use a handful of standardized components.

The smallest referencable ("addressable") unit in most computer architectures is the *byte*. Although historically the word "byte" could refer to other numbers of bits, in modern[6] usage a *byte* is a sequence of 8 bits (also known as an "octet").

[6] You're unlikely to encounter a byte that isn't 8 bits outside of a museum.

The *memory* in a computer is a collection of bytes. Each byte is assigned a unique number, called its *address*[7], that can be used to access (read or write) the data contained by that byte. You can think of memory as a giant *array* of bytes. The address of each byte is like an *index* into that array.

[7] As you might guess from the name, the *addresses* of bytes are analogous to the numbers of the houses in a street (a rather long street).

Although it's possible to refer to stored data directly using the address of the bytes where the data is stored, when we write a program it's often easier to use meaningful variable names to identify pieces of data. Behind the scenes, the compiler can convert these names to the addresses of the bytes where the data is actually stored.

Example: Here's an example of an area of memory, with ascending addresses on the left. Each box represents a byte of data. The names of some variables are on the right.

`0x100`	byte 0	varA
`0x101`	byte 1	varB
`0x102`	byte 2	temperature[0]
`0x103`	byte 3	
`0x104`	byte 4	temperature[1]
`0x105`	byte 5	
`0x106`	byte 6	temperature[2]
`0x107`	byte 7	
`0x108`	byte 8	
...	...	

In this example, two 8-bit variables, `varA` and `varB` and a 3-element array, `temperature` of 16-bit values are stored in the first 8 bytes.

If we think of the memory as a giant array called `memory`, then `varA` is equivalent to `memory[0x100]`, `varB` is equivalent to `memory[0x101]`, and so on. That isn't *actually* how we directly access memory (there isn't an array called `memory`), but it's conceptually similar.

Thus a reference in the program to `varB` means the byte at address 0x101, while a reference in the program to `temperature[2]` means the two bytes at addresses 0x106 and 0x107, treated as a 16-bit value.