Review of Digital Logic

Prof. John McLeod

ECE9047/9407, Winter 2021

Review of Numbers

Hopefully you are at least somewhat familiar with binary and hexadecimal values in a computer system.

Definition: A **bit** is a binary digit, either 0 or 1. Often, a bit is physically encoded as a low voltage for 0 and a high voltage for 1.

Definition: A **n-bit binary number** is a sequence of n binary digits b_i, which represent the number:

$$\begin{aligned}(b_{n-1}b_{n-2}...b_1b_0)_2 =& b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + ... \\ &+ b_1 \times 2^1 + b_0 \times 2^0.\end{aligned} \tag{1}$$

Definition: The **most significant bit** (MSb) in an n-bit binary number is the one representing the largest magnitude (i.e., the "2^n's place"). Typically this is the left-most bit in the sequence.

Definition: The **least significant bit** (LSb) in an n-bit binary number is the one representing the smallest magnitude (i.e., the 1's place"). Typically this is the right-most bit in the sequence.

Definition: A **hex** (hexadecimal) number encodes a value in a base sixteen system. Individual digits are from the set (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F).

Definition: A **n-digit hex number** is a sequence of n hex digits h_i, which represent the number:

$$(h_{n-1}h_{n-2}...h_1h_0)_{16} = h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + ... + h_1 \times 16 + h_0.$$

Definition: A **nibble** is a four bit binary number.

There is a simple mapping between binary numbers and hexadecimal numbers, as each nibble in the binary number corresponds to a single hex digit.

- It is straightforward to convert from hex to decimal, or from binary to decimal, but it is a bit cumbersome (especially as numbers get larger). It is also rarely useful.
- It is much more useful to convert from binary to hexadecimal (or vice versa)

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

Table 1: Mapping between nibbles (4-bit binary numbers), decimal numbers, and hex digits.

- Converting from binary to hexadecimal (or vice versa) is also easy to do if you use the nibble-to-hex conversion shown in Table 1.

To finish of this set of definitions:

Definition: A **byte** is an eight bit binary number (or two **nibbles**). Many (most?) computer systems use a **byte** as the size of the basic memory cells for data storage.

Definition: A **word** is a larger binary number, with various definitions depending on the context. The most common definition is that a **word** is the size of the memory registers in the computer processor. In the microprocessor (ARM®Cortex-A9) used in the lab/homework for this course, a **word** is 32 bits.

Review of Negative Numbers

In a pure binary system, the *only* symbols available are 0 and 1. Consequently all forms of information must be encoded using those symbols. This presents a challenge when trying to represent negative numbers: sticking a $-$ sign in front of the string of digits is not an option.

- There are several different ways of representing negative numbers in binary: **signed magnitude**, **one's complement**, and **two's complement**.

- **Two's complement** is the best and most common method, and the only one you need to know in this course.

Definition: The **complement** $\bar{b}$ of a bit b is found by "flipping" or "inverting" the bit — meaning to change its value from 1 to 0, or from 0 to 1.

Definition: The **two's complement** representation of a negative binary number is found by taking the complement of each bit, then adding 1.

As an example, consider some 8 bit binary numbers:

- Decimal 5_{10} is $(0000\,0101)_2$, so -5_{10} in two's complement is $(1111\,1010)_2 + 1_2 = (1111\,1011)_2$.

- Decimal 39_{10} is $(0010\,0111)_2$, so -39_{10} in two's complement is $(1101\,1000)_2 + 1_2 = (1101\,1001)_2$.

The most significant bit in the a two's complement number indicates the sign of the number: if that bit is a 1 the number is negative, if it is zero it is positive. Consequently, a n-bit two's complement binary number can store values from -2^{n-1} (represented as a one followed by $n-1$ zeros) up to $2^{n-1}-1$ (represented as a zero followed by $n-1$ ones).

Definition: An alternative definition of **two's complement** is to consider the MSb as a negative number. In an N-bit number, this has a value $-b_{N-1} \times 2^{N-1}$:

$$(b_{n-1}b_{n-2}...b_1b_0)_2 = -b_{n-1} \times 2^{n-1} + b_{n_2} \times 2^{n-2} + ... \\ + b_1 \times 2^1 + b_0 \tag{2}$$

The red colours is used to highlight the difference between this definition of a two's complement number the definition of an unsigned binary number from Equation 1.

As an example, consider the same binary numbers from before in two's complement.

- Decimal -5_{10} in two's complement is $(1111\,1011)_2$, which is $-(1000\,0000)_2 + (0111\,1011)_2 = -128_{10} + 123_{10} = -5_{10}$.
- Decimal -39_{10} in two's complement is $(1101\,1001)_2$, which is $-(1000\,000)_2 + (0101\,1001)_2 = -128_{10} + 89_{10} = -39_{10}$.

This alternative definition also works for two's complement numbers which are not negative.

- Decimal $+97_{10}$ in two's complement is $(0110\,0001)_2$, which is $-(0000\,0000)_2 + (0110\,0001)_2 = 97_{10}$.

Important! Suppose you find the binary sequence $(1001\,0110)_2$ stored in a computer system's memory. As a hex code, $1001_2 = 9_{16}$ and $0110_2 = 6_{16}$. What is the corresponding number?

- Is it an unsigned decimal number, 150_{10}?
- Is it a two's complement decimal number, -106_{10}?
- Is it a BCD decimal code, 96_{10}?

The answer is that *there is no way to tell* without some additional information. They all look the same to the computer — we impose meaning on the data by what we do with it.

Review of Binary Arithmetic

Binary values can be added following the normal rules for arithmetic. To perform the **subtraction** $A - B$ in binary, we should first take the two's complement of B and then **add** the two values, [1] and then of course the normal rules for arithmetic addition are followed.

[1] This is equivalent to $A - B = A + (-B)$.

- There is a major potential problem with binary arithmetic, as binary sequences usually have finite **width**.

Definition: The **width** of a piece of information is the number of bits available to store that information.

Definition: **Overflow** occurs when an arithmetic operation exceeds the available width available for binary values.

There are two kinds of overflow, depending on whether the operation is signed or unsigned arithmetic.

- When unsigned addition exceeds the available width, the highest order bit is lost. As an example, consider the following unsigned addition using 8-bit binary numbers:

$$(255)_{10} + 5_{10} = (260)_{10}$$
$$(1111\,1111)_2 + (0000\,0101)_2 = (0000\,0100)_2 + (1\,0000\,0000)_2.$$

  The last value (in red) is lost, because it is larger than the available width. Therefore the solution to $255 + 5$ is incorrectly return as 4_{10}.

- When signed addition improperly changes the MSb, the result of the arithmetic suddenly changes sign. As an example, consider the following signed addition using 8-bit binary numbers:

$$(127)_{10} + 1_{10} = (128)_{10}$$
$$(0111\,1111)_2 + (0000\,0001)_2 = (1000\,0000)_2.$$

  The values did not exceed the available width, and the answer is correct if the values were *unsigned*. But if we are expecting an answer in two's complement form, the result is incorrectly returned as $(-128)_{10}$.

Unsigned overflow is handled by passing a **carry-out bit** back with the result of the arithmetic operation.

Definition: In a n-bit system, a **carry-out bit** can be thought of as the $(n+1)$th digit (of value $b_n \times 2^n$).

Unsigned overflow can be accommodated by allowing a single number to be spread across multiple data elements.

- Since memory cells are often in units of **bytes**, we can have a single number stored in multiple bytes.

- Each time an arithmetic operation returns a non-zero carry-out bit, we know we need to add another **most significant byte** (MSB) to store that number.

Signed overflow is more difficult to deal with, as the problem is not related to a lack of memory width. When adding signed numbers, the following rules apply for positive **P** and negative **N** numbers:

- **PP**: Adding two positive numbers should *never* cause a carry-out, but if the sum is negative the result is invalid.
- **NN**: Adding two negative numbers should *always* cause a carry-out (which is ignored), but if the sum is positive the result is invalid.
- **NP**: Sometimes there will be a carry-out (which is ignored). The result is always valid.

How these concepts apply to microprocessors will be discussed in more detail later on.

Review of Combinational Logic

Hopefully you remember the symbols and truth tables for **AND** and **OR** gates, as shown in Figure 1 and Table 2. We will not make very heavy use of these logic gates in this course. Hopefully you also remember **minterms**, as they are rather important.

Definition: A **minterm** is a AND operation on n inputs (or their complements). It returns true (1) when all inputs are true (1) and all complemented inputs are false (0).

Definition: An alternative definition of a **minterm** is that it detects a single binary value.

To demonstrate the second definition, consider the Boolean expression:

$$f(A_3, A_2, A_1, A_0) = \overline{A}_3 A_2 A_1 \overline{A}_0.$$

This returns an output 1 when ever $A_3 = A_0 = 0$ and $A_2 = A_1 = 1$ simultaneously. If the inputs are interpreted as a 4-bit binary number $(A_3A_2A_1A_0)_2$, then this minterm detects the value $(0110)_2 = 6_{10}$.

- Consequently, we usually identify this minterm by the index 6 (as m_6 or "minterm 6").
- Every possible value of a n-bit binary number corresponds to one and only one n-input minterm.

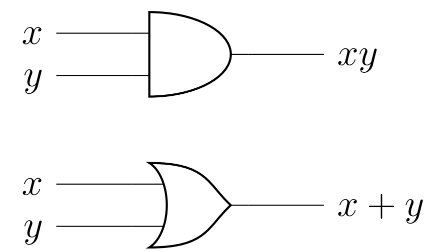

Figure 1: Circuit symbols for the binary logic operations AND (xy) and OR $(x+y)$.

x	y	xy	$x+y$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Table 2: Truth tables for the binary logic operations AND (xy) and OR $(x+y)$.

A **decoder** extends this concept: we can use decoders to detect a set, or range, of numbers. Consider the following Boolean expression:

$$f(A_3, A_2, A_1, A_0) = \overline{A}_3 A_1.$$

We can interpret this as a circuit that returns 1 when $A_3 = 0$ and $A_1 = 1$, or by considering implementing this circuit as a decoder we can describe this as a circuit that detects all numbers in the set $(0x1x)_2 = \{0010, 0011, 0110, 0111\}_2$. We will see how this is useful for microprocessors later on.

Review of Sequential Logic

Hopefully you remember something about **flip flops**. In this course, flip flops are considered **one-bit memory cells**. Flip flops are always **clocked**, and usually have an asynchronous preset and reset (or clear).

Definition: A **clock** is a single bit that switches between 0 and 1 at a regular, periodic interval. A clock is a special input for most of the components in a microprocessor. A clock helps keep all parts of the microprocessor synchronized.

A set of flip flops is a **register**. An n-bit register loads a n-bit binary value on a **rising clock edge** when the load input (LD) is 1.

- Depending on the context, you can consider a n-bit register as either storing a single n-bit number, or just a collection of n different bits held together for convenience.
- In other words, depending on the application, the contents of a register are not necessarily a single data value.
- For example, the ARM®Cortex-A9 microprocessor used in the lab have 32-bit registers (the size of a **word** for this system), while the memory has 8-bit cells. A register can hold a single 32-bit number (from four memory cells), or it could hold four different 8-bit numbers. Or it could hold thirty-two 1-bit control flags, or any combination thereof. We will see these situations later in this course.

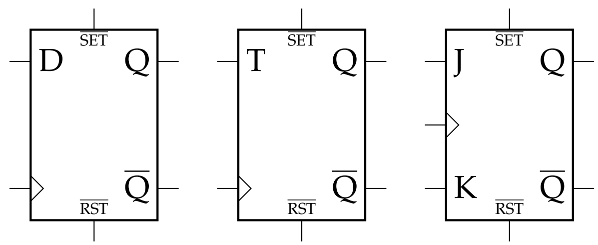

Figure 2: Circuit symbols for the three most common flip flops.

A **counter** is a special kind of register that can be used to periodically increment or decrement the bit sequence stored inside (this necessarily presumes that the contents of the counter are a single number, not a collection of independent bits).

- Counters with an enable input (EN) can be used as a standard register when EN is 0.
- Counters will count up (or down, as configured) on every rising clock edge.
- When a counter reaches the maximum (or minimum) value, they wrap around to zero (or maximum) and emit a carry-out (or borrow).

As we will discuss later, a microprocessor has several counters. The ones that are user-accessible usually consist of more than one register — at least one for holding the count value, and at least one more for holding the instructions that control the counter's operation.

- Only the counting register (the one that holds the count) would be called a "counter" in terms hardware circuitry, but in this course is it convenient to call the entire set of registers a "counter".

Circuit symbols for registers and counters are shown in Figure 3. Please note that the label "D[n]" is used for **input data**, regardless of how the register or counter is implemented — i.e., it is not necessarily the case that D-type flip flops are used.

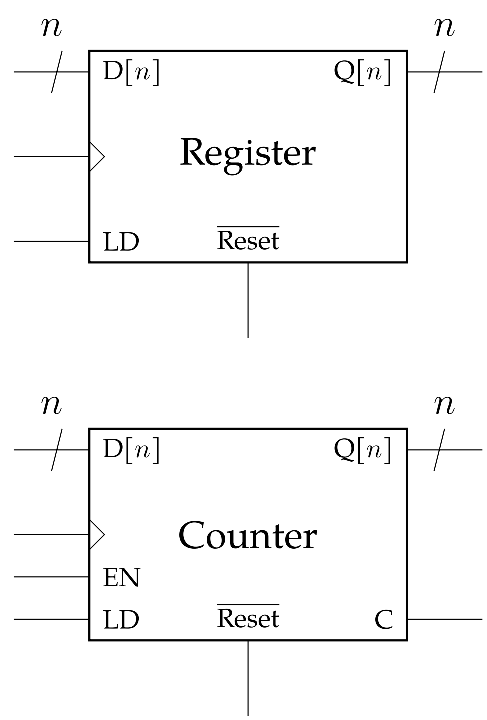

Figure 3: Circuit symbol for a n-bit register and an n-bit counter. Here this counter allow the current count value to be read from Q[n]. Note that the line with a slash through it (for D[n] and Q[n]) is the symbol for a **data bus** — a set of n lines.

Computer Bus Structures

As a final topic to review, consider the circuit shown in Figure 4. This is something you can easily build on a breadboard — but you should not! What is the output F?

- V_{CC} is logical 1 and ground is logical 0.
- The AND gate will provide an output of 0, as $1 \cdot 0 = 0$.
- The OR gate will provide an output of 1, as $1 + 0 = 1$.
- Wiring the two outputs together creates an ambiguous, and possibly dangerous, configuration (the high-voltage from the OR gate will try to force current to flow backwards into the AND gate).

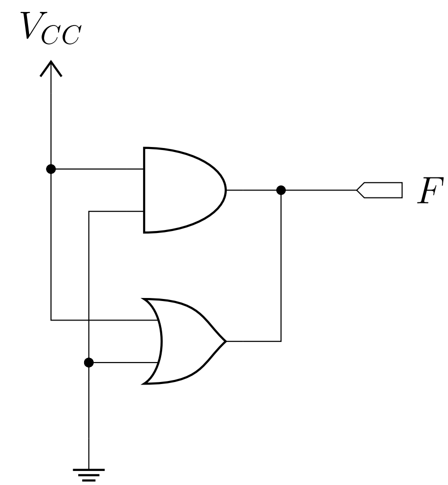

Figure 4: A simple, but ill-advised, logic circuit.

The circuit from Figure 4 is a good circuit to build if you want an excuse to go shopping for new electronics. However the design principle behind this circuit is sound: often we want multiple components to connect to output.

- Obviously each component can't "talk" at once, which is why we need some control element to select which component controls the output.
- This control element is called a **tri-state driver** (or "tri-state buffer"). [2]

[2] We covered these briefly in ECE2277.

Definition: A **tri-state driver** is a circuit element with a single input bit, a single control bit, and a single output bit. Unlike most digital electronics, however, it has three possible output states: 1, 0, and **high impedance**.

It is this high impedance state that allows multiple components to connect to the same output. The high impedance state is basically a "shut up" state, effectively disconnecting the input from the output line — so the other components in the circuit can set the logic value for that line. To fix the circuit in Figure 4, we should connect tri-state drivers to the outputs from each gate, and have those drivers controlled by a common line so only one can be active at a time. This is shown in Figure 6.

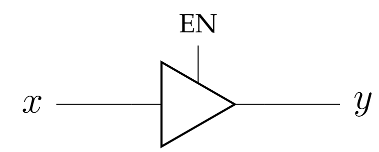

Figure 5: A tri-state driver. When $\text{EN} = 1$, $y = x$. When $\text{EN} = 0$, y is floating — it is disconnected from x.

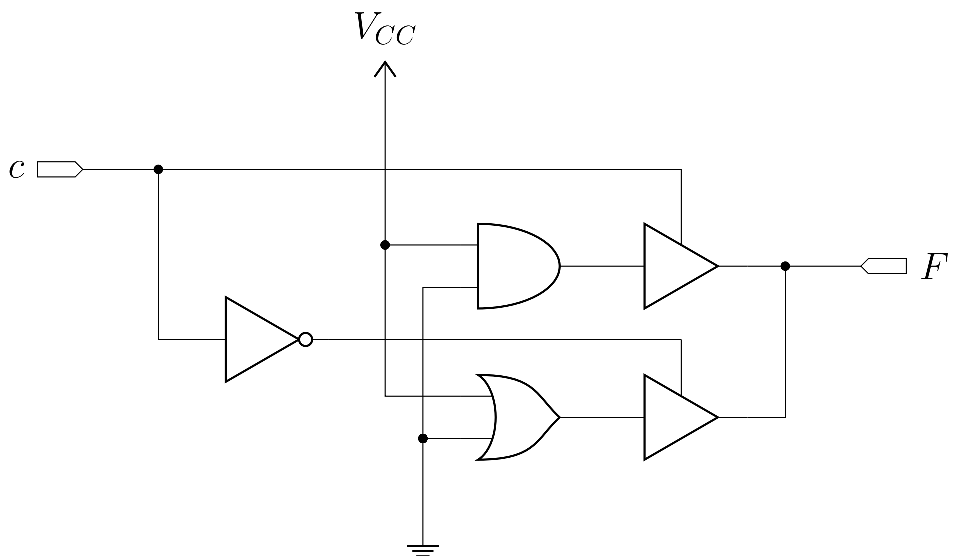

Figure 6: Fixing the pointless logic circuit fr
Figure 4. Now the control line c allows the
output from only one of the two logic gates
selected for F.