Microprocessors & Microcomputers

ECE 3375

Arnav Goyal

Winter 2023

CONTENTS

1. INTRODUCTION & REVIEW

This chapter goes over a bit of the basic knowledge that we will build off-of for the rest of this document, almost everything here was taught in ECE-2277 Digital Logic Systems.

1.1. NUMBER SYSTEMS

In this course we will most often be using binary (base-2) and hexadecimal (base-16) number systems. We also assign some predefined words to binary numbers of differing lengths.

Definition A bit is a binary number of length 1 - in other words it is a binary digit.

Definition A nibble is a 4-digit binary number.

Definition A byte is an 8-digit binary number.

Definition A word is a larger binary number - most commonly a 32-bit binary number

Definition A double-word is an even larger binary number - in this case a 64-bit binary number

Converting from binary to decimal is cumbersome and rarely-useful, instead converting decimal to hexadecimal going nibble-by-nibble is much more useful as memory addresses and other things are usually stored as hexadecimal numbers.

> The number $(1101\,1010)_2$ is equal to $(\text{DA})_{16}$

1.2. BINARY CODED DECIMAL

Binary Coded Decimal (BCD) uses each nibble to represent a single decimal (base-10) digit. Because each nibble can store 16 different values and decimal only uses 10 values, using BCD is an inefficient way to store data.

> The number $(1653)_{10}$ can be written in BCD as $0001\ 0110\ 0101\ 0011$

There is no given way to tell if a given set of nibbles is a BCD number or a standard binary value. This is true for most numbers in this course. BCD is only really useful for converting binary to decimal - in case we need to hand-check computer outputs for example.

1.3. NEGATIVE NUMBERS

In a pure binary system, we can only use the symbols 0 and 1 for representation. In order to cope with the fact of not being able to use a negative symbol there are three ways ways to do this. The only one we will be using in this course is the Two's Complement as it is the one most practically used.

1. Signed Magnitude
2. One's Complement
3. Two's Complement

> *Definition* The Two's Complement of a binary number is found by flipping the bits and adding one. We define a number as negative (according to this convention) if the MSb is 1
>
> For example, 5_{10} is $(0000\ 0101)_2$, thus $(-5)_{10}$ is $(1111\ 1011)_2$
>
> The number $(1101\ 1001)_2$ is negative, thus we flip bits and add one to find its magnitude. We get a two's complement of $(0010\ 0111)_2$ which tells us the original binary number represents a $(-39)_{10}$

Given some random binary sequence such as: $1001\ 0110$ What is the number?

It could be an unsigned decimal number of 150, a twos complement decimal number of -106, or even a BCD representation of 96. An important thing to remember is that *there is no way to tell* without some additional information. These all look the same to the computer, and we impose meaning on the data by what we do with it.

1.4. BINARY ARITHMETIC

Binary addition is pretty easy, so I wont be going over it. The binary subtraction $A-B$ is performed by adding A to the two's complement of B, which is then just an easy addition.

> There is a major problem with binary arithmetic because binary numbers usually have a fixed (and finite) width
>
> *Definition* The width of a binary number is the amount of bits available to store that number.
>
> *Definition* Overflow occurs when the result of an arithmetic operation exceeds the available width for it to be stored.

There are two "kinds" of flags - overflow and carry, depending on if the operation is signed or unsigned. Essentially overflow represents a violation of width during signed addition, and carry represents a violation of width during unsigned addition.

When unsigned addition exceeds the available width it produces an extra bit called the carry bit, in computer systems this carry bit is essentially lost due it being larger than the available width, thus it returns a wrong result but enables the carry flag.

> Consider the following unsigned addition:
>
> $$(11100)_2 + (00110)_2 = (1\ 00010)_2$$
>
> This addition produces an extra bit and violates the width of 5 bits, thus there is a carry generated

During signed addition, the range of values we can store is basically halved to make space for negative numbers, Thus sometimes when adding signed numbers we exceed this range, which results in the enabling of the overflow flag

> Consider the following signed addition:
>
> $$(01100)_2 + (00101)_2 = (10001)_2$$
>
> This addition incorrectly changes the MSb from 0 to 1. If we interpret the operands as signed numbers according to the two's complement, the result must also be interpreted this way, and it results in the incorrect statement of $12 + 5 = -15$.
>
> The real result of this operation is $12 + 5 = 17$, which lies outside the representable range of 5-bit two's complements, Hence the overflow.

Some quick takeaways:

- An overflow can never occur when adding two numbers with differing signs
- During subtraction, carry = NOT borrow