EECS-140/141 Introduction to Digital Logic Design
Lecture 5: Number Systems and Arithmetic

I. UNSIGNED NUMBER SYSTEMS: THE COUNTING NUMBERS

I.A Motivation

Digital circuits are almost always *binary* circuits.
Binary means only:

Why? Transistors! These are binary switches that are incredibly:

So we want to use them to represent/manipulate numbers.

I.B Decimal Representation (System) for Whole Numbers

For now, consider only the case of the counting (whole) numbers *starting with 0.* We will deal with negative numbers and numbers between integers later.

I.B.1 Decimal System Basics

This is the system humans use in everyday life.

I.B.1.a Ten Symbols

The symbols 0 through 9 represent the first 10 whole numbers. These are referred to as:

I.B.1.b Representing Larger Numbers

We use combinations of the 10 basic symbols and *positional significance* to represent larger numbers:

I.B.1.c Limits on Numbers Represented

If no limit on the number of digits (positions):

But, if the number of digits is restricted, so are the possible numbers:

I.B.1.d The Decimal System Is Not the Only Way

Our "human" (decimal) system is built around 10:

The computer system is built around 2:

I.B.2 Radix (or Base)

I.B.2.a Notice the Prominence of 10 in Decimal System

I.B.2.b 10 Is Known as the Radix of the Decimal System

I.C Unsigned Binary System for Whole Numbers

I.C.1 Binary: Radix is 2

1. 2 symbols *only*:
2. Positions imply power of 2 multipliers.
3. *k*-bit binary string ("word") can represent:

I.C.2 How to Distinguish Different Systems

— Use radix as subscript:

— Use words:

— Use context.

I.C.3 Converting Binary to Decimal

Just use positional significance:

I.C.4 Converting Decimal to Binary

I.C.4.a First Question

How many bits are needed to represent the (decimal) number N?

Examples: $N = (31)_{10}$ requires:

I.C.4.b Decimal to Binary Conversion Process

Suppose we know N can be represented with 5 bits. Then:

$N =$

Now suppose we divide N by 2:

$N/2 =$

So:

Now if we divide the *whole result* of $N/2$ by 2:

See the pattern? Just keep repeating!

Example: $N = (19)_{10}$ requires:

I.D Closely Related Whole Number Systems

I.D.1 Motivation

For large numbers, the binary system is cumbersome for humans.

So, humans need more compact, but closely related, number systems.

I.D.2 Unsigned Octal System (Base 8)

Symbols are:

I.D.2.a Binary to Octal Conversion

— Group bits in sets of 3, starting:
— Fill with 0's at left as needed.
— Convert each triple individually.

I.D.2.b Octal to Binary Conversion

Expand each octal digit into 3 bits:

I.D.2.c Decimal to Octal

Divide successively by:

I.D.3 Hexadecimal (Hex): Radix Is 16

Need 16 symbols:

II. UNSIGNED BINARY ADDITION

II.A Introduction

II.A.1 Motivation

Number manipulation is fundamental to computing.

A basic number manipulation operation is *addition*.

II.A.2 Unsigned Addition Basics

For now, assume numbers to be added are whole (counting) numbers:

We can use *combinational logic* to construct adding circuits. The approach will mimic the way humans do addition.

II.B Adding 1-bit Numbers

Start simple!

II.B.1 Inputs and Outputs

Just two 1-bit inputs:

But we need *two* 1-bit outputs as well:

II.B.2 Truth Table and Logic Network

a	b	c (msb)	s (lsb)
0	0		
0	1		
1	0		
1	1		

II.C Generalize: Full Adder

II.C.1 Introduction

— To add 2 n-bit numbers, we have $2n$ inputs, so a Truth Table for such a function would have:

— Better to use a *modular* design:

— Approach is similar to how humans do addition -- add one column at a time:

II.C.2 Inputs and Outputs

To add i^{th} bits of 2 binary numbers, need:

a_i: b_i: c_i:

but still only 2 outputs, since max could be:

s_i: c_{i+1}:

II.C.3 Truth Table and K-Maps

a_i	b_i	c_i	c_{i+1}	s_i
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

II.C.4 Logic Expressions

$c_{i+1} =$

$s_i =$

Note:

and:

So,

II.C.5 Logic Network for Full Adder

II.C.6 Equivalent Logic Network (You Verify)

II.D Multi-Bit Adder

Use the Full Adder (FA) as a *replicated module* for an n-bit adder:

Called a "ripple-carry" adder since the carry bit must:

III. MULTIPLY/DIVIDE BY 2 (UNSIGNED)

III.A Multiply by 2

Suppose you want to multiply an n-bit unsigned binary number by 2:

$$a = a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_1 2^1 + a_0 2^0$$

Then: $2a =$

Result: $2a$ is an $(n+1)$-bit number formed by shifting all a bits *left* by 1 place and filling with:

Generalize: For n-bit unsigned binary number, $2^k \cdot a$ is:

Example:

III.B Divide by 2

Result is exact *only* if the least significant bit (LSB) is 0.

Then just shift all bits *right* by 1 place and LSB=0 "falls off" the end. Fill from left with 0's.

Example:

IV. SIGNED INTEGER BINARY SYSTEMS

IV.A Introduction

- Next step is to represent positive and:
- Signed binary representations are *always* for a specified *fixed word length* (e.g., 8 bits, 16 bits, etc.).
- Leftmost bit is *always* reserved for representing + or - (details follow).

  So, an 8-bit signed binary representation has only 7 bits to represent magnitude.
- *All* systems represent positive numbers as 0xxx, where xxx is the $(n-1)$-bit unsigned binary representation of the number.

IV.B Signed Integer Representations

IV.B.1 Signed-Magnitude (or Sign-and-Magnitude)

- This is the "human" way.
- Leftmost bit is sign bit: 0 for positive, 1 for negative.
- Example: In 4-bit signed-magnitude system:
- *Range*: with n-bit word, this represents:
- *Note*: both +0 and -0 are represented (not a good thing).
- *Not* a convenient representation for computers because arithmetic is awkward (for humans, too!).

IV.B.2 1's Complement (1C)

This is another way to represent negative integers.

IV.B.2.a 1C Additive Inverse Operation

To find the 1C additive inverse of an n-bit positive number P, subtract P from $(2^n) - 1$ (the all-1 n-bit word).

Example: n=4 $P = (3)_{10} = (0011)_2$.

Then:

Note: Since $(2^n) - 1$ is the all-1 n-bit word,

Clearly, then, for each bit position, $1 - p_i = \overline{p_i}$.

So, $(-P)_{1C} = \overline{p_{n-1}} \cdot \overline{p_{n-2}} \cdot \cdots \cdot \overline{p_1} \cdot \overline{p_0}$

Note that this holds for the previous example (with n=4):

IV.B.2.b Some Characteristics of 1C System

1C also represents both +0 (all-0 word) and -0:

$(-0)_{1C}$ = all-1 word Example: n=4 -0 = -(0000) = 1111 = -0

Also, $\left\{-\left[(2^{n-1}) - 1\right]\right\}_{1C} = -(0111 \cdots 11) = 1000 \cdots 00$

Note: since $\overline{\overline{x}} = x$, it is clear that $-(-P)_{1C} = P$.

Range: Same as signed-magnitude: $-\left[(2^{n-1}) - 1\right]$ to $(2^{n-1}) - 1$

Finally, left shift does *not* work for negative 1C numbers, even considering the result as an $(n + 1)$-bit 1C number:

IV.B.3 2's Complement (2C)

Although the 1C additive inverse operation is *very* easy, 1C is not very good for addition (see this later). But there is a *better* format: 2's Complement (2C).

IV.B.3.a 2C Additive Inverse Operation

To find the 2C additive inverse of an n-bit positive number P, subtract P from 2^n:

Example: n=4, $P = (3)_{10} = (0011)_2$

Then:

There are 2 easy ways to find $(-P)_{2C}$ from binary representation for P:

— Using the basic definition of the 2C additive inverse operation, note that $(-P)_{2C} = (-P)_{1C} + 1$

  So, *first* find $(-P)_{1C}$ by flipping every bit, *then* add 1 to that result.

  Example: n=4 P=3:

— Start at the right (p_0, the LSB) and moving left:

  — Copy each bit that is a 0 and the *first* bit that is a 1.

  — After that first 1, flip the remaining bits.

IV.B.3.b Some Characteristics of 2C System

2C has only *one* representation for 0:

What does $(100 \cdots 00)$ represent in 2C system?

Range of 2C system is:

Multiply by 2 via left shift *does* work for 2C, considering the result as an $(n+1)$-bit 2C number:

IV.C Addition and Subtraction for Positive and Negative Integers

IV.C.1 Subtraction

Note that $A - B$ is the same as $A + (-B)$ regardless of whether A and B are positive or negative. So, all we need is addition and the ability to form the additive inverse.

IV.C.2 Addition

We will illustrate with $n = 4$ and decimal magnitudes 3 (0011) and 2 (0010).

We *want* to use the same basic method (circuit) to add positive and/or negative numbers as is used to add 2 positive numbers (earlier section).

IV.C.2.a Signed-Magnitude

Terrible: What works for pos+pos does *not* work for other cases *at all.*

IV.C.2.b 1C and 2C Addition

Decimal	1C	2C

Conclusion: 1C *might* work, but maybe would need to add 1 when $c_n = 1$? 2C just works!

IV.D Understanding 2C Arithmetic

IV.D.1 Introduction

2C arithmetic may seem like "magic":

— Why is $(-P)_{2C} = (2^n) - P$?

— Why does it work with an adder circuit designed only for *positive* integers?

— Why is there a carry out sometimes?

— When there is, why can we just ignore it?

The answers to all these are apparent when we understand that 2C arithmetic is just *modulo*-2^n *arithmetic!*

IV.D.2 2C as Mod-2^n

Illustrate with $n = 4$, $2^n = 16$.

IV.D.2.a Put the 16 numbers around a circle

With $N = 4$, we can represent only 16 numbers.

IV.D.2.b Movement Around Circle

Move *clockwise* when adding a _______________ number and *counter − clockwise* when adding a _______________ number.

IV.D.2.c What if we add 16 to or subtract 16 from any number?

One full rotation around circle, so get:

This is a basic feature of modulo arithmetic.

IV.D.2.d Subtraction as Addition

Note that *subtracting* P is the same as:

Example:

IV.D.2.e Final Interpretation of Numbers

Our final interpretation must be according to the 2C system: for $n = 4$, the 16 numbers are (in decimal):

If our result is not in this range, just add or subtract 16 to get it in this range.

Example (continued):

Note that on our mod-16 circle, -7 is the *same* number as:

This is just the 2C additive inverse operation: $(-P)_{2C} = 2^n - P$.

Remember: Calculating $2^n - P$ is simple *in binary:*

Key: Treat negative numbers as *mod*-2^n *positive equivalent when doing arithmetic:* this lets us use an adder circuit designed for positive numbers!

IV.D.2.f Carry-Out

If the *addition* of 2 positive (or pos-equiv) numbers is ≥ 16, we have a *carry − out* that can be:

Ignoring the carry-out is the *same* as subtracting 16 from the result:

IV.D.2.g Final Adjustment

If the n-bit pos-number result (after ignoring any carry-out) is >7 (largest positive 2C number: in general $(2^{n-1}) - 1$), then *adjust* by subtracting 16 (2^n in general) to convert/interpret it as a *negative* number:

IV.D.3 Summary/Re-Cap

a. n bits can represent *only* 2^n numbers:

2C says these are:

b. Mod-2^n arithmetic applies:

c. 2C uses *positive-equivalent* for negative numbers:

which is easy to compute in binary.

And since $A - B = A + (-B) = A + (2^n - B)$, *all* addition/subtraction can be done by adding positive numbers!

d. If pos-number result is $\geq 2^n$ (n=4: 16), a carry-out occurs, which can be:

This is the same as subtracting 2^n (mod-2^n).

e. If pos-number result *after* ignoring carry-out is > $(2^{n-1}) - 1$ (largest pos 2C number, 7 for n=4), *interpret* as neg number by:

IV.D.4 Examples with n=4 (Handout)

These include the previous example calculations, but include both $A + B$ and $B + A$.

IV.D.5 Overflow

Since all arithmetic is mod-2^n (the result of adding 2 n-bit words is an n-bit word), some results will be *invalid* -- this is called *overflow*.

Note: Overflow is *not* the same as carry-out (c_n)!!!!!

Examples: n=4 and all combinations of +/- 6 and +/- 3.

(Continued from previous page)

Note: Overflow (an invalid result) happens when we get an "impossible" result:

pos + pos = neg? Overflow!!
neg + neg = pos? Overflow!!

So, we can detect overflow with:

But note a simpler expression for overflow that seems to work from the above examples:

You will prove this in homework this week.

IV.E 2C Adder/Subtractor Circuit

IV.E.1 Introduction:

- We can convert the *n*-bit adder circuit from before into an adder/subtractor by adding some gates.
- We have already showed that we can *add* a pair of 2C numbers, any combination of pos and neg.
- So we just need an efficient way to form the 2C additive inverse for subtraction:

IV.E.2 Forming the 2C Additive Inverse

We will use the method of flipping all bits and adding 1.

IV.E.2.a Flipping the Bits

This could be just an inverter for each bit, but we *only* want to flip bits if we are subtracting. Better to use:

This is useful because:

So, the first input to XOR *controls* whether to flip or not.

IV.E.2.b Add 1

We can do this with the c_0 bit (the carry-*in* to the LSB).

IV.E.2.c Control Bit

Since we want to be able to *either* add (A+B) or subtract (A-B = A+(-b)), we need:

IV.E.2.d Resulting Circuit

V. Fast Adders (Carry Lookahead)

V.A Intro

Recall: the problem with ripple-carry adder design is long delay as carry bits "ripple" through from LSB to MSB. From the Full Adder circuit in Fig 5.4, this is 2 gate delays for *each* bit position (and they accumulate).

Can we reduce this delay???

V.B Basic Form Lookahead Adder

Start with CSoP for c_{i+1} (carry out of position i).

Now let:

Then:

g_i:

p_i:

Note: g_i and p_i do *not* depend on c_i:

Instead, delay results from propagation:

Expression is valid for *any* i from $n - 1$ down to 0. So:

Result: SoP form, so just 2 gate delays for *every* c_i result, including c_n!

Need one more gate delay to calculate p_i and g_i, plus another (XOR) to produce s_i. (See Fig 5.16).

Total:

This could be called a *monolithic* carry-lookahead adder.

This adder delay for s_{n-1}:

Ripple-carry adder max delay for s_{n-1}:

Problems with monolithic carry-lookahead adder:

1. *Massive* logic circuit:
2. *Large* fan-in required to get low delay:

3. No longer modular

V.C Modular Lookahead Adders

V.C.1 Truly Modular

Can combine lookahead and ripple-carry.

— Design a k-bit lookahead adder as above, with k relatively small (e.g., k=4 or 8)

— Interconnect with ripple-carry (See Fig 5.17). This gives *partial* benefit from lookahead.

V.C.2 Semi-Modular Form

V.C.2.a Modular Block (k=8)

Example: j=0, k=8

(Continued) Modular block computes G_0 and P_0, extra block computes c_8:

Now, for Block 1 (j=1), all p_i and g_i and c_i subscripts increment by 8:

and we need another (*unique*) block to calculate c_{16}:

No Ripple! Calculation of c_{16} does *not* need c_8 as input.

See Figure 5.18.

VI. MULTIPLICATION

VI.A Introduction

Review how humans do decimal multiply:

Similar in binary, but:

— Only every multiply M by 0 or 1, and 1-bit multiplier is just:

— For n-bit Q, must add together n shifted-multiplied results.

Conclude: For binary, each R_i is just M (shifted) or 0, so multiplying is just shifting and adding!

VI.B Unsigned Numbers: Array Multiplier

Note first: multiplying 2 *n*-bit numbers requires:

Example: *n*=3:

This design takes the following approach for the sum:

See Figure 5.31.

Example with *n*=4 bits:

Circuit to do this needs only 2 replicated blocks (although could do it with single replicated block plus AND gates).

See Figure 5.32.

VI.C Signed 2C Multiplication

Below is an outline of the changes needed to deal with 2C numbers.

— If Q is negative, form the negative of *both* Q and M using the 2C additive inverse operation, since $Q \cdot M = (-Q) \cdot (-M)$.

— Now (since $-Q$ is positive), we are just shifting/adding positive or negative numbers.

— We can *avoid* overflow when adding 2C numbers here by making each operand 1 bit longer.

— To lengthen a k-bit number to a $(k+1)$-bit number, use *sign extension.*

This does not change the *value* of the number (see Homework).

— See book for details.

VII. REPRESENTING REAL NUMBERS (vs. INTEGERS)

VII.A Binary Fixed-Point Number Representation

VII.A.1 Basics

Similar to decimal, we can represent a number with n integer bits and k fractional bits as:

VII.A.2 Decimal to Binary Conversion

— Already know how to do integer part (successive divide by 2).

— For the fractional part:

Example: Represent $(5.45)_{10}$ as 8-bit word with 4 integer bits and 4 fractional bits.

Integer Part:

Fractional Part:

Final Result:

Note: conversion is *seldom* exact due to word-length limit!

VII.A.3 Fixed-Point Arithmetic

— Every fixed-point $(n + k)$-bit number *can* be converted to an integer by:

This just moves the binary point.

— We can then do 2C arithmetic on integers and convert back (multiply by 2^{-k}). This just moves the binary point back.

— *But*, why even move the binary point over and back??

Conclusions:

a. Can *treat* fixed-point numbers as *integers* for arithmetic, and

b. Can represent pos and neg numbers with 2C number system.

Caution: the 2C additive inverse operation *must* be applied to the entire $(n+k)$-bit word, *not* just the integer part!

Example:

VII.B Binary Floating-Point Number Representation

VII.B.1 Introduction

In decimal, when we want to represent really *big* and/or really *small* numbers, we use scientific notation:

We can do something similar with binary:

Just need a specific system to represent sign, Mantissa, and Exponent.

VII.B.2 IEEE Floating Point: Single Precision

This is a 32-bit word format with 3 components:

E: Interpreted as an unsigned binary number. It is a representation of the (signed) exponent in what is known as *excess-127 format.*

Special values:

M: Interpreted as a binary fraction:

So:

VII.B.3 IEEE Floating Point: Double-Precision

VII.B.4 Floating Point Arithmetic

VII.B.4.a Addition

Must shift one mantissa to get common exponent:

VII.B.4.b Multiplication

— Multiply mantissas
— Add exponents
— Adjust sign

