# Physics 310
# Lecture 10 – Microprocessors

| | | |
|---|---|---|
| Mon. 3/29 | **Project:** Component Shopping | **Project Progress Report** (due at the beginning of class) |
| Wed. 3/31 | **Ch 13:** Microprocessor Basics | |
| Thurs. 4/1 | **Intro to Projects** | |
| Fri. 4/2 | More Ch 13 | |
| Mon. 4/5 | **Projects** | |
| Wed. 4/7 | **Projects** | |
| Thurs. 4/8 | **Projects** | |
| Fri. 4/9 | **Projects** | |
| Mon. 4/12 | **Projects** | |
| Wed. 4/14 | **Projects** | |
| Thurs. 4/15 | **Projects** | |
| Fri. 4/16 | **Review** | |
| Mon. 4/19 | **Demonstrate Projects** | **Project Report & Notebook** |
| **Final Exam:** **Thursday April 22nd 3:00 p.m.** *or* **Saturday April 24th 9:00 a.m. / 12:00 noon** (we'll choose as a class) | | |

## Announcements:

- **Parts have been ordered; they should be in later this week.**
- **Most of you didn't actually 'turn in' your project progress reports. Please do so (I'll make photo copies and return them).**

## 13.1 Intro:

The microprocessesor is the natural conclusion of our study of digital circuitry. This is where all the 'thinking' or processing happens inside a computer or other common electronic devices. At the time that the text was written, mircroprocessors were finding their way into our daily lives, off the desktop and into our cars, ovens, etc. Today, they've found their way into our constant companions – phones and mp3 players.

## 13.2 Computer Arithmetic

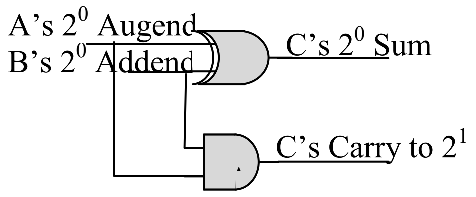
The book mentions Hexadecimal representation and bits (one data line) bytes (eight data lines) and nibbles (four data lines) as well as KB ($2^{10}$) and MB ($2^{20}$). For our purposes though, it's probably not worth focusing much on these.

Computers were originally for *computing / calculating*, and they still do a lot of this. So a natural bridge from logic gates to computers (and microprocessors) is considering how these gates can be used to perform common computational tasks: addition, subtraction, and multiplication.

### Binary Addition

You may recall that I'd introduced the simple "half adder" just after we'd met binary (to suggest that studying binary wasn't such a non-sequeter after studying logic gates). Now I'll use the text's notation

# Physics 310
# Lecture 10 – Microprocessors

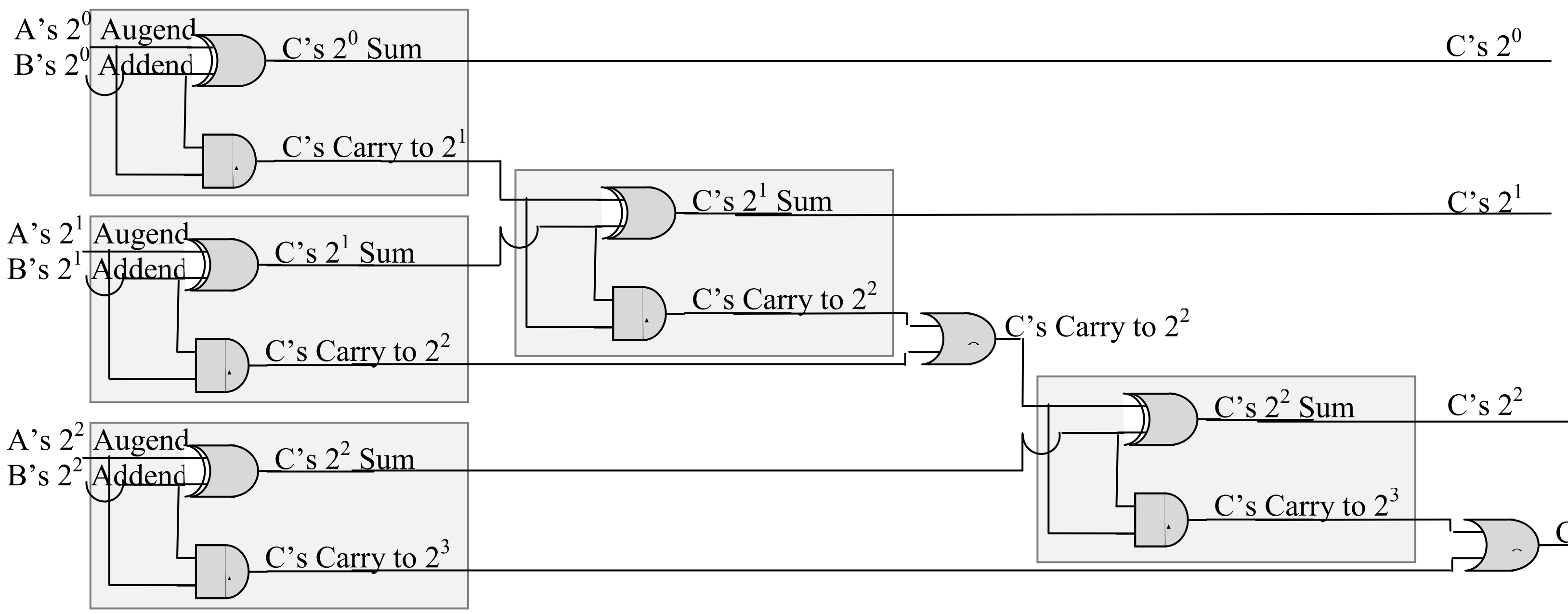


| Inputs | | Outputs | |
|---|---|---|---|
| Augend | Addend | Sum | Carry |
| A + | B = | C $2^0$ | C $2^1$ |
| 0 + | 0 = | 0 | 0 |
| 0 + | 1 = | 1 | 0 |
| 1 + | 0 = | 1 | 0 |
| 1 + | 1 = | 0 | 1 |

Alone, this is of fairly limited use – it can at most add two one-bit numbers and thus add one plus one to get two. Before we consider how we’d use gates to add *two*-bit binary numbers, lets first consider how we add ‘two-bit’ decimal numbers. It’s probably been a long time since you’ve actually thought through the words you were taught in your early days of addition:

$$\begin{array}{r} 1\,6 \\ +2\,5 \\ \hline \end{array}$$ “six plus five equals eleven, carry the one” -> $$\begin{array}{r} 1\phantom{\,6} \\ 1\,6 \\ +2\,5 \\ \hline 1 \end{array}$$ -> “two plus one equals three” $$\begin{array}{r} 1\phantom{\,6} \\ 1\,6 \\ +2\,5 \\ \hline 3\,1 \end{array}$$ “three plus the carried one equals four” $$\begin{array}{r} 1\phantom{\,6} \\ 1\,6 \\ +2\,5 \\ \hline 3\,1 \\ 4\,1 \end{array}$$

We essentially do the same thing when adding a two-bit binary number: we first add the lowest bits $2^0$, in the process we generate the answer’s lowest bit, and the “carry” which we carry over to add to the sum of the next lowest bit, and so on until we don’t have any more bits to add.



So, we add the $2^0$ bits and carry to the $2^1$; we add the $2^1$ bits and then add the carry in and determine the carry out, then we add the $2^2$ bit, add the carry in and determine the carry out,…

As for determining the "carry out," the OR gate is because two additions determine the answer's $2^1$ place value and *either* one of those (but not both) can be responsible for generating a carry to the $2^2$ addition. The same thing happens when we add familiar decimal numbers. Consider these two examples, again, spelled out as if we were in third grade,

```
                                        Note: the carry to
                                        the 100's place is
                                        generated now
                         1              1 1           1 1

   1 6 9                 1 6 9          1 6 9         1 6 9

  +2 5 3   ->           +2 5 3   ->    +2 5 3        +2 5 3    …
  ------                ------         ------        ------
                             2            1 2           1 2

                                                        2 2
```

Alternatively, if we'd added 149 + 253, then we wouldn't be carrying to the 100's place until we added in the carry from the 1's place. So, the carry to the next place can happen during either addition operation, but not during both (you only ever "carry the one" never "carry the two".)

In figure 13.4B, the book shows the inner workings of a 4-bit adder. The exact logic used is a little different from that illustrated here, but it gets the job done. With the help of some of the binary logic identities in chapter 11, one may be able to translate between the two; alternatively, the logic tables can be generated and compared.

**Binary Subtraction**

First think about how you do subtraction with multi-'bit' decimal numbers.

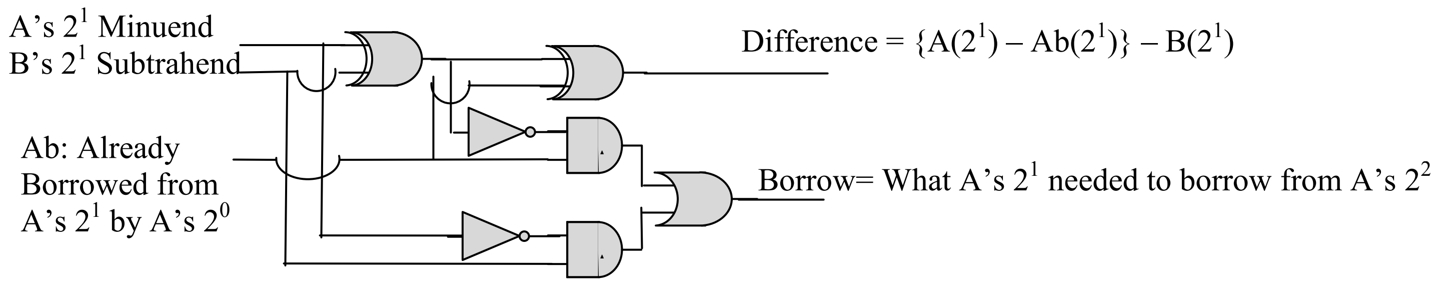
```
  Borrow 1 from 3 to make 14                    Borrow 1 from 7 to make 12

 7  3  4                  7  2 (14)    7  2  (14)              6 (12)  (14)

-6  4  8                 -6  4    8   -6  4   8               -6   4      8   …
--------                 ----------   ---------               -------------
                                              6                           6
```

So, we need a 'borrowing' mechanism for our subtraction. That has to essentially accomplish four things: a) subtract the value from the $2^{n+1}$ bit of the Minuend, b) add it to the next $2^n$ bit of the Minuend, c) only do this if the $2^n$ bit of the Minuend is less than that of the Subtrahend, and d) actually subtract the bits.

Here's the circuitry that accomplishes these tasks for one bit.


A's $2^1$ Minuend
B's $2^1$ Subtrahend
Ab: Already Borrowed from A's $2^1$ by A's $2^0$
Difference = {A($2^1$) – Ab($2^1$)} – B($2^1$)
Borrow= What A's $2^1$ needed to borrow from A's $2^2$

# Physics 310
# Lecture 10 – Microprocessors

Note: what the book confusingly labeled Bin, I've labeled Ab: what the previous bit of A already borrowed from this one. Let's see if we can reason out the truth table:

| Minuend A's $2^1$ | Subtrahend B's $2^1$ | Borrowed *from* A's $2^1$ by A's $2^0$ | Difference | Borrowed *by* A's $2^1$ from A's $2^2$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

**Two's-Complement Arithmetic**
From the perspective of minimizing the amount of hardware necessary, it would be nice to be able to handle subtraction with the *exact same circuitry* as we use for addition. Of course, subtracting a positive number is the same as adding a negative number, so the real question is, how can we represent a *negative* number in binary *such that* when we use our adding circuitry we get the appropriate result.

How do we need to represent a negative number?

1. First off, we give up the Most Significant Bit as denoting a number, instead it tells us the *sign* of the number, in that way we can represent both positive and negative numbers. The price we pay is that we have smaller range of numbers that we can represent. Okay, that sounds reasonable enough. Now, given that,
2. We "complement" all the bits – replace 1's with 0's and vice versa (easily done with inverters on the bit's lines.)
3. We add 1 to the last bit (easily done with our addition circuitry.)

First, let's perform this operation a few times, and then see that the resulting representation of a negative number adds to give the right answer.

Represent -63:

63 = 0011 1111
$\overline{63}$ = 1100 0000
$\underline{\qquad\qquad +1}$
-63=1100 0001

Represent -28:

28 = 0001 1100
$\overline{28}$ =1110 0011
$\underline{\qquad\qquad +1}$
-28=1110 0100

Now, let's add a negative number to a positive one:

63 = 0011 1111
$\underline{+(-28)}$ = $\underline{1110\ 0100}$
35 = (1) 0010 0011
(we just ditch the extra 'carried' 1)

# Physics 310
## Lecture 10 – Microprocessors

Chips like the 7483 can perform this task, they have ‘preconditioning’ input line that tells them whether or not to form the 2’s-complement (and thus whether or not to subtract).

**ALU** – Arithmetic Logic Unit.
A popular single chip has the basic logic and mathematical operations wired in, so depending on the ‘setting’ inputs, it can be set to perform one of the basic logic operations (say, NAND two 4-bit inputs) or mathematical operations (say, subtract them.)

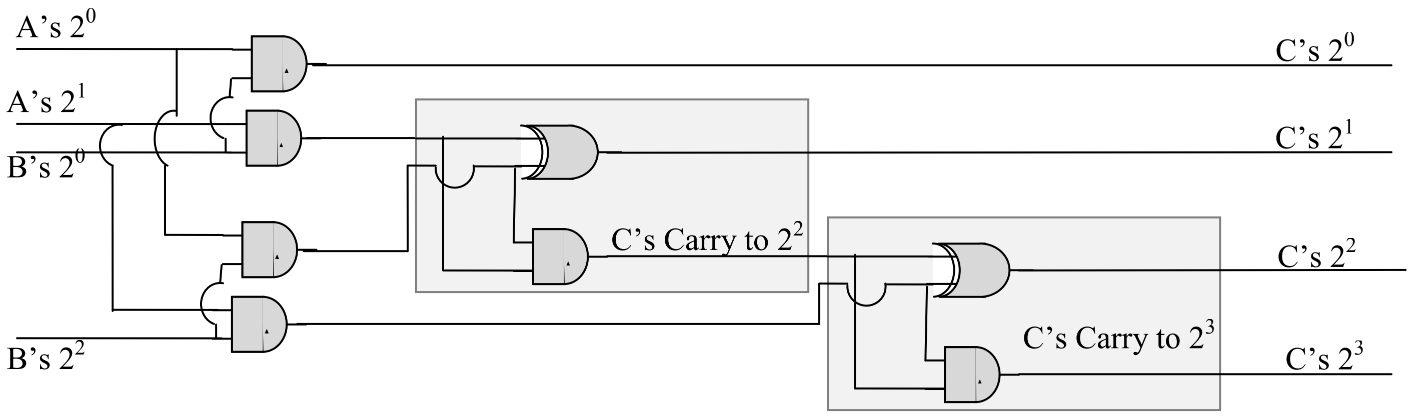
**Multiplication**
Think about how you do long multiplication – that’s how it’s done in binary.

```
                                       1      11      11      11      11
          1     11     11     11      11      11      11      11      11
  65     65    6 5     65     65      65      65      65      65      65
 ×23    ×23    ×23    ×23    ×23     ×23     ×23     ×23     ×23     ×23
          5     85     85     85      85      85      85      85      85
                     +110   +110    +110    +110    +110    +110    +110
                            195     195     195     195     195     195
                                     00     200    +200    +200    +200
                                                    395     395     395
                                                           1100   +1100
                                                                   1495
```

Okay, so we take the lowest bit of the one number and multiply all bits of the other number, then do the same for the next higher bit with an offset, and then add them. We do the exact same thing in binary

```
  65  =     01000001
 ×23  =    ×00010111
            01000001
           010000010
          0100000100
         00000000000
       +010000010000
1495 =   10111010111
```

Just looking at a simple two-bit multiplier, this could be performed like this: Note that the individual multiplications are simply AND’s: 1*1=1, 1*0=0 whether “*” means AND or Times.



A’s $2^0$
A’s $2^1$
B’s $2^0$
B’s $2^2$
C’s $2^0$
C’s $2^1$
C’s Carry to $2^2$
C’s $2^2$
C’s Carry to $2^3$
C’s $2^3$