CURIE Academy, Summer 2021
Lab 1 Notes: Computer Engineering – Hardware Perspective

Prof. Christopher Batten
School of Electrical and Computer Engineering
Cornell University

The field of computer engineering is at the interface between hardware and software and seeks to balance the tension between application requirements and technology constraints. In this lab, you will explore the field of computer engineering from a hardware perspective by assembling basic logic gates to implement a simple "calculator" for adding small binary numbers. In Lab 2, you will explore the field of computer engineering from a software perspective by incrementally programming a microcontroller in C++ to implement an IoT "smart light" system. These lab notes provide a brief survey of background information relevant to understanding the purpose and context for the lab.

As illustrated in Figure 1, computer systems can be viewed as a stack of abstraction and implementation layers from applications at the highest layer to technology at the lowest layer. In these lab notes we will briefly discuss the technology, devices, circuits, gate-level, register-transfer-level, and microarchitecture layers as they relate to our simple binary adder. In the actual lab session, you will have an opportunity to put what you have learned into practice. We will focus on some layers more than others, but by the end of this lab you should have a good understanding of how computer engineers can leverage these layers to implement higher-level arithmetic operations in hardware.

1. Devices and Technology: Resistors, LEDs, Transistors

Figure 2 illustrates the basic devices we will be using this lab: voltage sources (e.g., a battery), resistors, light-emitting diodes, NMOS transitors, and PMOS transistors. Figure 3 illustrates the abstract symbols we can use when drawing circuit schematics.

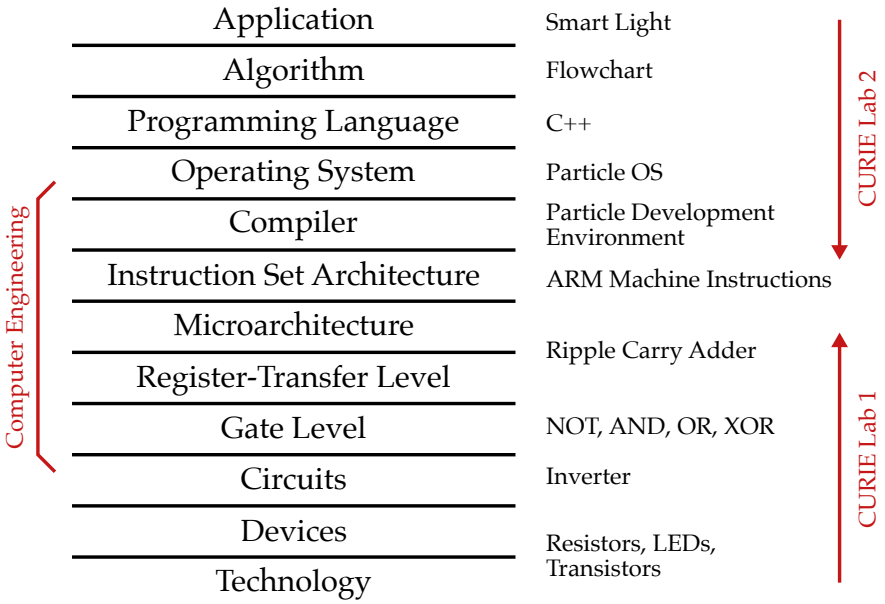


Figure 1: Computer Systems Stack

(a) LEDs (b) Resistors (c) Transistor

Figure 2: Various Devices

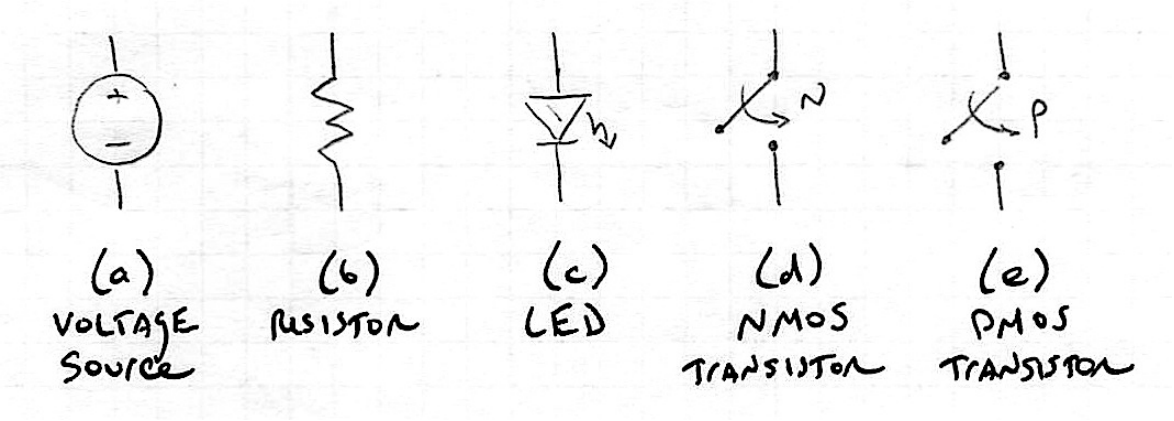


Figure 3: Symbols for Various Devices

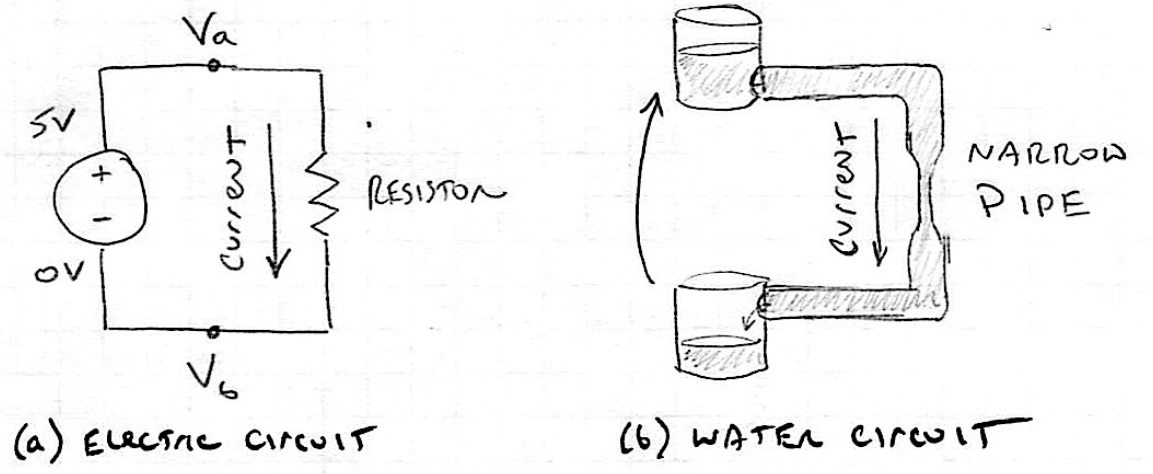


Figure 4: Basic Electrical and Water Circuit

Figure 4(a) illustrates a simple circuit where the positive terminal of a battery is attached to one end of a resistor, and the negative terminal of the battery is attached to the other end of the resistor. In order to build our intuition into how these devices work, we will use a simple analogy with the "water circuit" shown in Figure 4(b). The battery converts chemical energy into electrical potential energy; this is analogous to a human converting mechanical energy into gravitational potential energy by pouring water from the lower bucket into the higher bucket in Figure 4(b). Point V_a in the electrical circuit has higher electrical potential energy compared to point V_b, just like point P_a in the water circuit has higher gravitational potential energy compared to point P_b. The difference between two electrical potentials is called the *voltage* and is measured in *Volts*. The current in the electrical circuit has a tendency to move towards the point with lowest electrical potential energy, just like the current in the water circuit has a tendency to move towards the point with the lowest gravitational potential energy. A resistor in an electrical circuit is analogous to a narrow segment of pipe in the water circuit; both the resistor and the narrow pipe serve to restrict the flow of current. So the greater the resistance (narrower the pipe) the less current can flow through circuit. In the example, point V_a is at 5 Volts and point V_b is at 0 Volts. Since point V_a is at the positive terminal of the voltage source it is usually called VDD; since V_b is at the negative terminal of the voltage source it is usually called ground.

The current in both the electical and water circuits flows in a circle. In the electrical circuit: (1) chemical energy is converted into electrical potential energy by moving electrical current from point V_b to V_a; (2) the current tends to move from point V_a to V_b which has a lower electrical potential energy; (3) the current is obstructed when moving through the resistor (and the electrical potential energy is converted into heat); and (3) eventually the current reaches point V_b and the cycle starts again. In the water circuit: (1) mechanical energy is converted into gravitational potential energy by pouring water from the bottom bucket into the top bucket; (2) the current tends to move from point P_a to P_b which has a lower gravitational potential energy; (3) the current is obstructed when moving through the narrow pipe; and (4) eventually the current reaches point P_b and the cycles starts again. Note that this is a *very* simple analogy that breaks down pretty quickly, but it will be sufficient for building the intuition required in this lab.

Figure 2(c) shows a light-emitting diode (LED). LEDs have two terminals and emit light when current flows through them in a specific direction. If current flows through an LED in the "wrong" direction, then the LED will prevent any current from flowing through the circuit and will not light up.

For this lab we will be exclusively focusing on *digital circuits*. In a digital circuit, we usually do not care about specific voltages. By convention, we simply name VDD as *logic one* and ground as *logic zero*. Figure 2(d–e) illustrates NMOS and PMOS transistors. These transistors have three terminals and act as switches. An NMOS transistor is "closed" when its input is a logic one and is "open" when its input is a logic zero. A PMOS transistor is the exact opposite. A PMOS transistor is "closed" when its input is a logic zero and is "open" when its input is a logic one.

2. Circuits: Inverter

We can compose multiple devices together to create interesting circuits. Figure 5 shows a simple LED circuit which will cause the LED to light up. We use a resistor with the LED to limit how much current is flowing through the LED; without the resistor there would be too much current flowing through the LED and it would quickly burn out. To build our circuits we will use a *solderless breadboard*. A breadboard provides a convenient way to quickly prototype circuits by simply inserting components and wires into the breadboard holes. There is wiring inside the breadboard which internally connects all holes in each half-column as shown in Figure 6. Figure 7 shows the implementation of the simple LED circuit using the prototyping platform provided for you in this lab. The prototyping platform

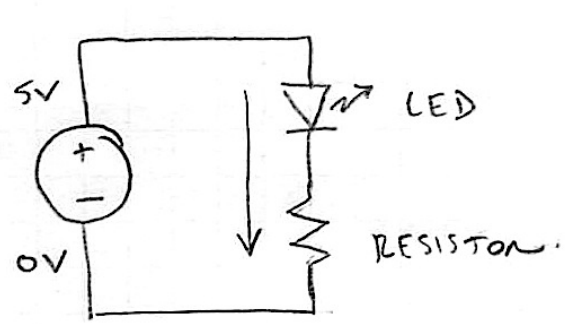


Figure 5: Simple LED Circuit

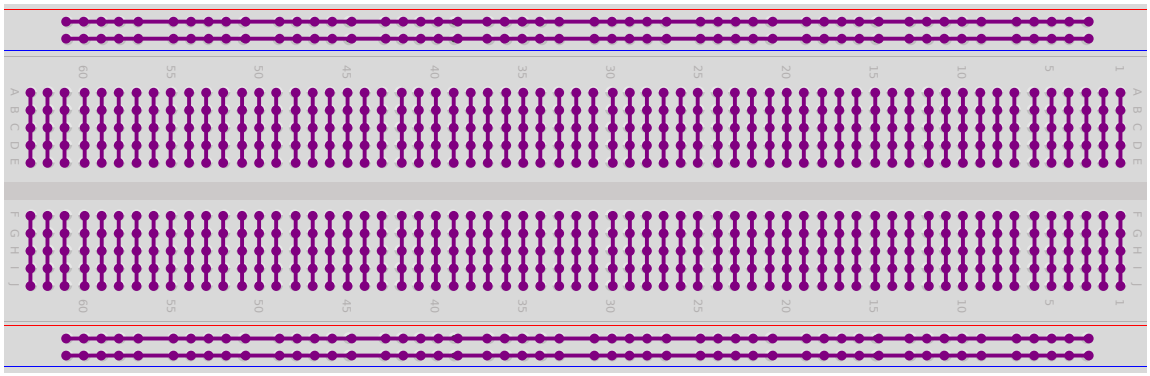


Figure 6: Connectivity Inside Breadboard Shown in Purple

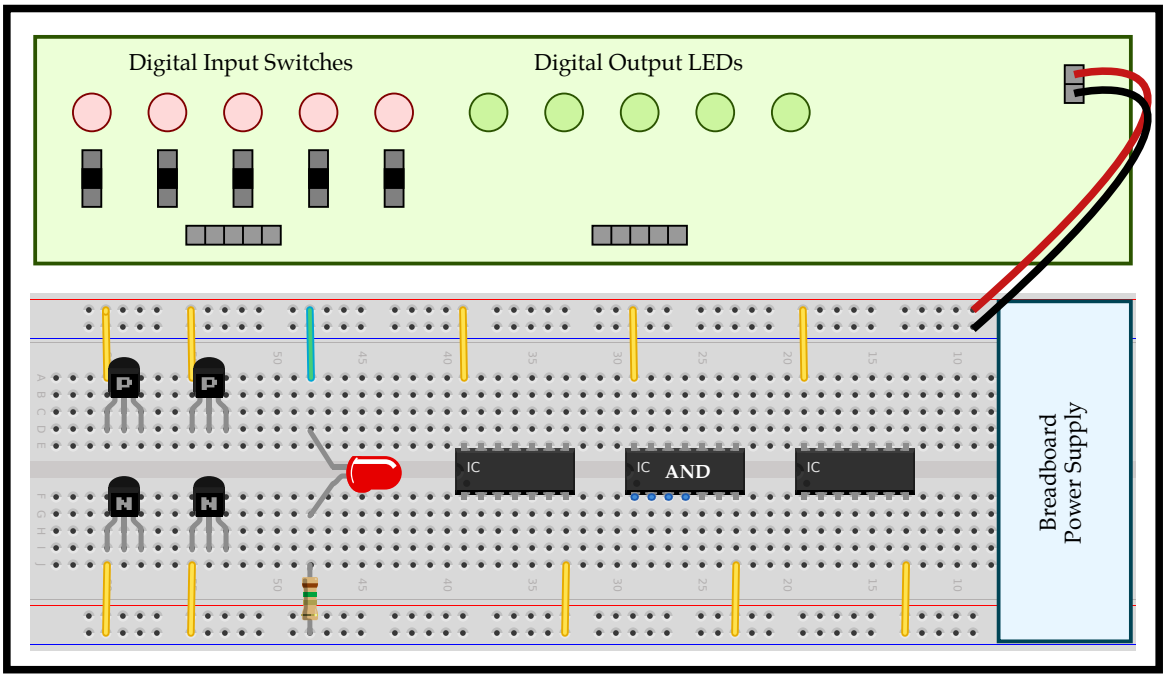


Figure 7: Simple LED Circuit Implemented on Prototyping Platform

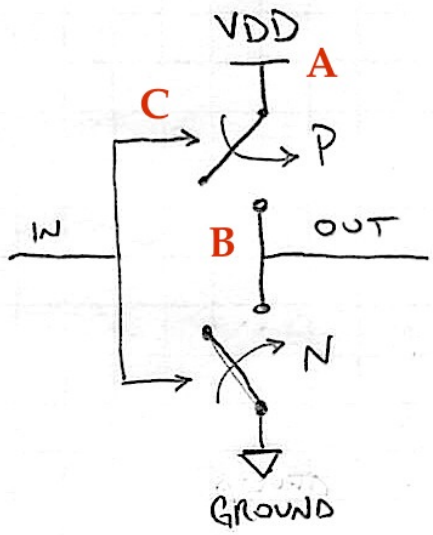


Figure 8: Inverter Circuit (P = PMOS transistor, N = NMOS transistor)

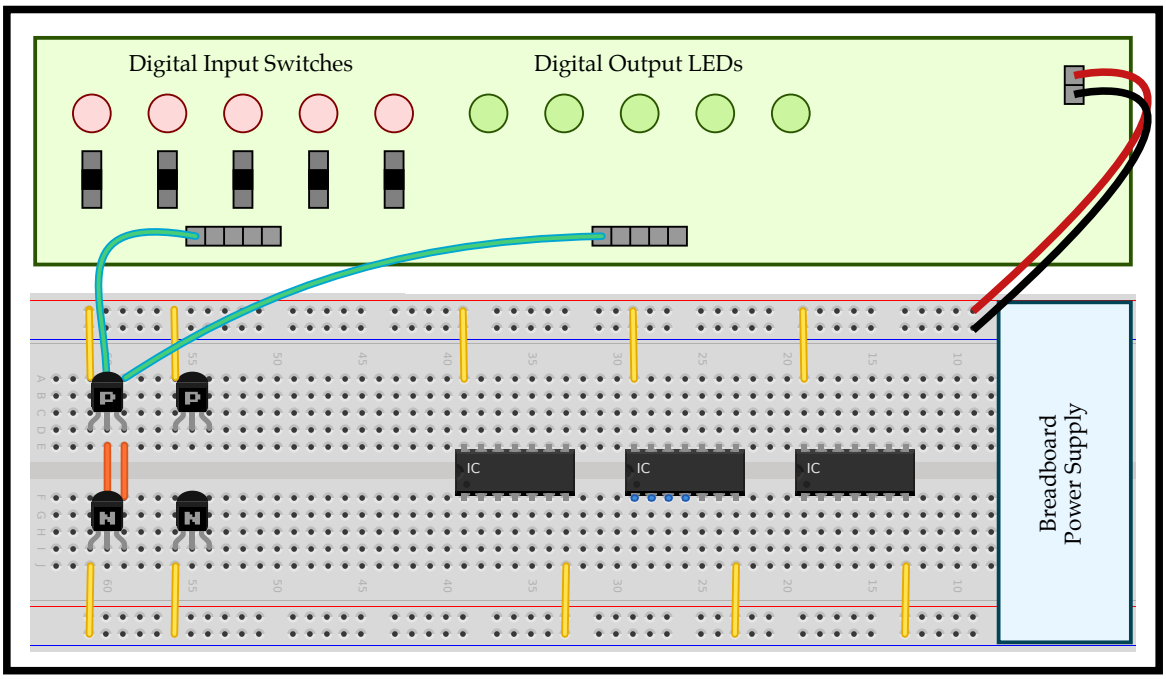


Figure 9: Inverter Circuit Implemented on Prototyping Platform

provides VDD (5 V) and ground (0 V) "rails" at the top and bottom of the breadboard, and also provides five digital input switches and five digital output LEDs.

Figure 8 shows a more interesting circuit called an inverter. This circuit uses a PMOS transistor and an NMOS transistor. When the input is a logic one then the PMOS transistor is open and the NMOS transistor is closed; this essentially causes the output to be "pulled down" to a logic zero. When

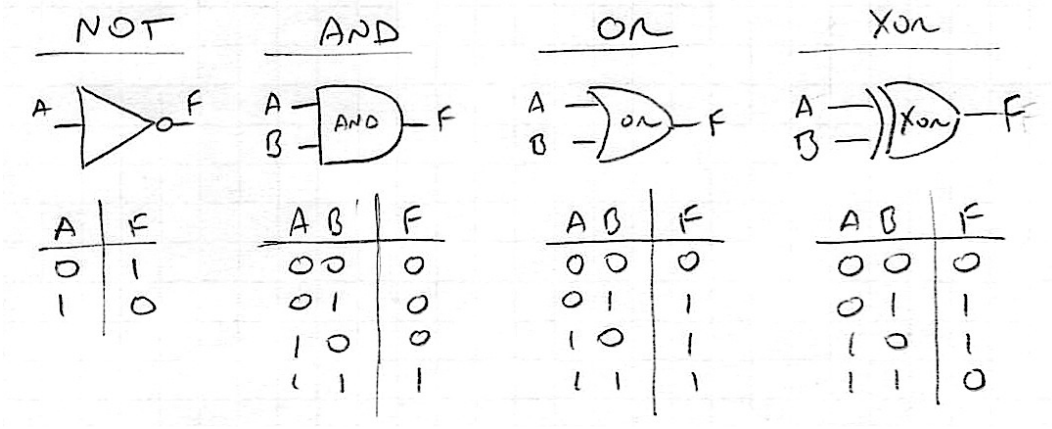


Figure 10: Symbol and Truth Table for NOT, AND, OR, XOR Gates

the input is a logic zero then the PMOS transistor is closed and the NMOS transistor is open; this essentially causes the output to be "pulled up" to a logic one. Figure 9 shows the implementation of the inverter using the prototyping platform provided for you in this lab. Notice how we have attached the input of the inverter to one of the digital input switches and the output of the inverter to one of the digital output LEDs. Turning on the input will turn off the output; and turning off the input will turn on the output.

3. Gates: NOT, AND, OR, XOR

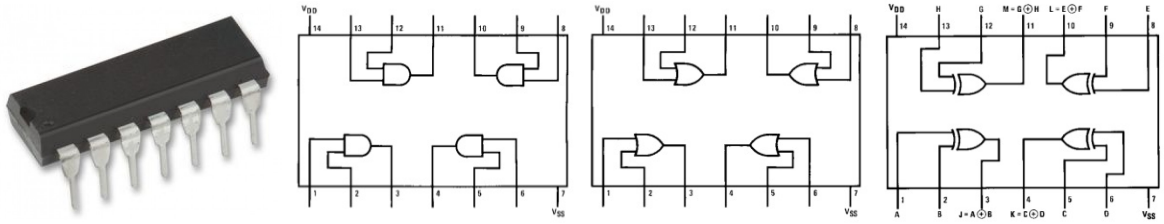
As computer engineers, we often use abstraction to hide implementation details and provide cleaner higher-level interfaces. Indeed digital signalling itself is an abstraction since we ignore the details of exact voltages and instead focus on logic one and logic zero values. To build more complicated circuits, we will create simple circuits and then abstract them into useful logic gates. For example, we can abstract the inverter discussed in the previous section into the NOT gate shown in Figure 10. If the input to a NOT gate is a logic one then the output is a logic zero; if the input to a NOT gate is a logic zero then the output is a logic one. Abstraction enables us to ignore the details of the specific implementation of a NOT gate using PMOS and NMOS transistors.

Figure 10 also shows three more useful logic gates. We can use a truth table to succinctly capture the functionality of each logic gate. The truth table shows what the output of the logic gate should be for every combination of inputs to the logic gate. For example, the output of an AND gate is only one when both of its inputs are one.

These logic gates are often implemented as *integrated circuits* with four gates per chip. Figure 11 shows what one of these chips looks like and illustrates how the logic gates are organized inside the chip.

4. Aside: Binary Arithmetic

In the next section, we will compose logic gates to create a hardware unit capable of adding two numbers together, but first we need to understand a bit about binary arithmetic. Computers use digital circuits which can only handle logic ones and zeros; in other words, computers work with *binary numbers* (i.e., base 2) and *binary arithmetic*. This is in contrast to how we usually use *decimal numbers* (i.e., base 10) and *decimal arithmetic*. Let's quickly review decimal numbers: each digit can be a number between zero and nine. The first digit is the "ones digit" (10^0), the second digit is the



(a) Integrated Circuit (b) AND Chip (c) OR Chip (d) XOR Chip

Figure 11: Four Logic Gates Implemented as an Integrated Circuit in a Single Chip

"tens digit" (10^1), the third digit is the "hundreds digit" (10^2), and so on. We multiply the number in each digit by 10^i where i indicates which digit. For binary numbers: each digit can only be zero or one. The first digit is the "ones digit" (2^0), the second digit is the "twos digit" (2^1), the third digit is the "fours digit" (2^2), and so on. We multiply the number in each digit by 2^i where i indicates which digit. Figure 12 illustrates the binary and decimal representations for all numbers between zero and 15.

Binary addition works exactly like decimal addition, except using binary numbers instead of decimal numbers. Figure 13 illustrates how to add three plus six using binary numbers and binary addition. In the first step, we add the "ones digit"; one plus zero is one. In the second step, we add the "twos digit"; one plus one is two, but remember we cannot use two in binary numbers. We can only use zero or one. We need to carry a one to the next digit. This is exactly the same as carrying a one when performing decimal arithmetic. In the third step, we add one (the carry bit) plus zero plus one to again get two, so we again must carry a one to the next digit. In the final step, we simply add one plus zero plus zero to get the final answer: 1001 which is nine in decimal. As expected three plus six is nine, but we have now demonstrated how to do this addition using binary arithmetic.

```
dec :    0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
bin : 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
```

Figure 12: Binary and Decimal Representation

```
  Step 1  Step 2  Step 3  Step 4

            1       11      11
    011     011      011     011
+   110   + 110    + 110   + 110
------  ------  ------  ------
     1      01      001    1001
```

Figure 13: Example Using Binary Addition for 3+6

5. Register-Transfer Level and Microarchitecture: Ripple-Carry Adder

Now we wish to use basic logic gates to implement a hardware unit that can add two binary numbers together. This is a relatively complicated task, so we will break it into a few smaller steps. Our first task is to implement a *half adder* capable of adding two one-bit numbers together. The reason it is called a half adder will become apparent a little later in this section. Figure 14 shows the four possibilities when adding two one-bit numbers.

As discussed in the previous section, if we add one plus one the answer is two which cannot be represented with a single bit. We must "overflow" from a one-bit result into a two-bit result. We call the right-most bit of the result the *sum bit* and the left-most bit of the result the *carry bit*. For example, when we add one plus zero, the sum bit is one and the carry bit is zero and when we add one plus one the sum bit is zero and the carry bit is one.

So how do we implement a one-bit half adder using boolean logic gates? We can think of the half adder as a black box which has two inputs and two outputs. The two inputs correspond to the two

input A		input B	result base 10	result base 2	carry bit	sum bit
0	+	0	= 0	00	0	0
0	+	1	= 1	01	0	1
1	+	0	= 1	01	0	1
1	+	1	= 2	10	1	0

Figure 14: Four Possibilities when Adding Two One-Bit Numbers

input A	input B	sum bit
0	0	0
0	1	1
1	0	1
1	1	0

input A	input B	carry out
0	0	0
0	1	0
1	0	0
1	1	1

Figure 15: Truth Tables for Sum Bit and Carry Bit

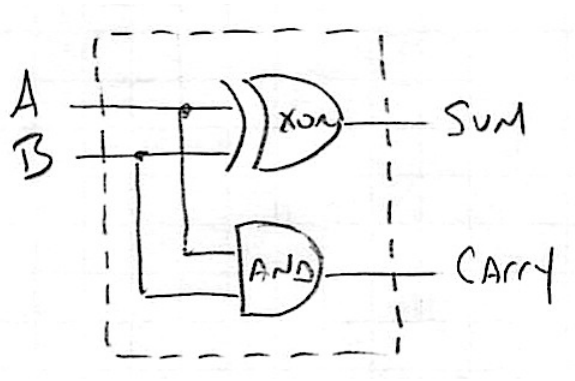


Figure 16: Half-Adder

input A		input B		input C		result base 10	result base 2	carry bit	sum bit
0	+	0	+	0	=	0	00	0	0
0	+	0	+	1	=	1	01	0	1
0	+	1	+	0	=	1	01	0	1
0	+	1	+	1	=	2	10	1	0
1	+	0	+	0	=	1	01	0	1
1	+	0	+	1	=	2	10	1	0
1	+	1	+	0	=	2	10	1	0
1	+	1	+	1	=	3	11	1	1

Figure 17: Eight Possibilities when Adding Three One-Bit Numbers

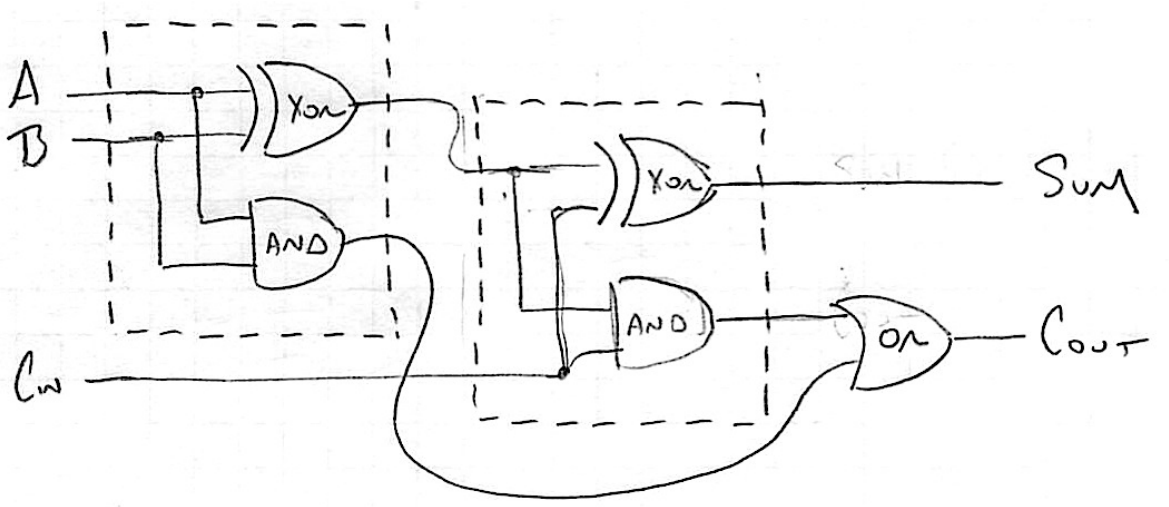


Figure 18: Full-Adder

one-bit numbers we wish to add and the two outputs correspond to the sum and carry bits. As a computer engineer, we need to choose a network of boolean logic gates which will always generate the desired outputs as a function of the inputs. We start by writing a truth table for each of the two outputs in isolation (see Figure 15). We can then look at our toolbox of logic gates to see if there are any matches and indeed it should be clear that we can implement the sum bit by connecting the inputs to a single XOR gate and that we can implement the carry bit by connecting the inputs to a single AND gate. The implementation of the half adder using boolean logic gates is shown in Figure 16.

We can use a similar approach to build a "full adder" which can add three one-bit numbers to produce a two-bit result. Figure 17 shows the eight possibilities when adding three one-bit numbers. We now wish to implement a one-bit full adder using boolean logic gates. The full adder has three inputs corresponding to three one-bit numbers we wish to add together and two outputs corresponding to the sum and carry bits. As in the previous section, we could now create truth tables for both the sum bit and the carry bit and carefully construct a network of boolean logic gates which will always generate the desired outputs as a function of the inputs; note that we will need to use multiple stages of boolean logic.

input A	input B	result base 10	result base 2	carry bit	sum bit 1	sum bit 0
00	+ 00	= 0	000	0	0	0
00	+ 01	= 1	001	0	0	1
00	+ 10	= 2	010	0	1	0
00	+ 11	= 3	011	0	1	1
01	+ 00	= 1	001	0	0	1
01	+ 01	= 2	010	0	1	0
01	+ 10	= 3	011	0	1	1
01	+ 11	= 4	100	1	0	0
10	+ 00	= 2	010	0	1	0
10	+ 01	= 3	011	0	1	1
10	+ 10	= 4	100	1	0	0
10	+ 11	= 5	101	1	0	1
11	+ 00	= 3	011	0	1	1
11	+ 01	= 4	100	1	0	0
11	+ 10	= 5	101	1	0	1
11	+ 11	= 6	110	1	1	0

Figure 19: 16 Possibilities when Adding Two Two-Bit Numbers

As a clever alternative, we can instead consider a way to leverage the half adder we already designed in the previous section to build a full adder. The half adder adds two one-bit numbers to create a two-bit result, so our full adder can use a half adder to add input A and input B and then use another half adder to add the result of the first half adder to input C. We just need to be careful how we implement the carry out for the full adder; if either of the carry bit of either of the half adders is one then the carry out for the full adder will also be a one. The implementation of a full adder using two half adders and an extra OR gate for the carry bit is shown in Figure 18.

Now we wish to implement a hardware unit that can add two binary numbers with many bits together. Figure 19 shows the sixteen possibilities when adding two two-bit numbers. Let's take a closer look at one of these possibilities. Assume we wish to add two plus one.

```
 10 (2 in base 10)
+01 (1 in base 10)
---
 11 (3 in base 10)
```

It might seem that we can consider each column in isolation to calculate the corresponding column in the two bit-output, but what happens if one column requires us to carry into the next column? For example, assume we wish to add three plus one.

```
CBA (column label)
 11 (3 in base 10)
+01 (1 in base 10)
---
100 (4 in bsae 10)
```

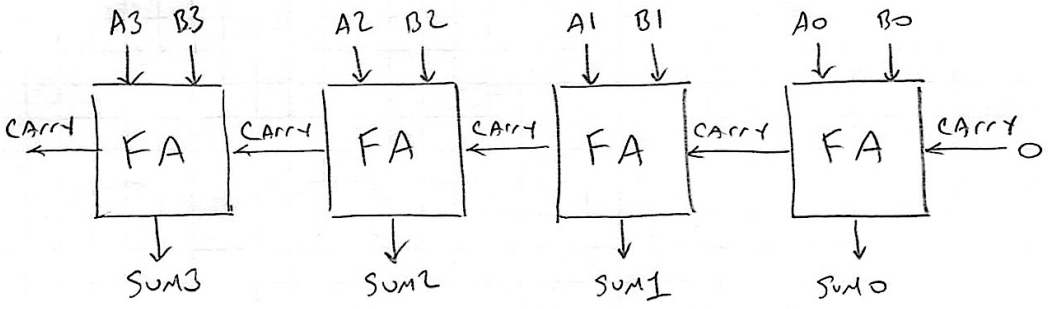


Figure 20: Four-Bit Ripple-Carry Adder (FA = full-adder)

Column A produces a carry bit which we must include when calculating the second bit of the sum, and similarly column B produces a carry bit meaning the result overflows into three bits. If we focus on column B, we will realize that we actually need to add *three* one-bit numbers together (one bit from the first input, one bit from the second input, and the carry bit from column A) and that we will produce two outputs (a sum bit and a carry bit for the next column). A full adder provides the exact functionality needed to calculate each column of a multi-bit addition. Figure 20 illustrates how we can chain a series of full adders to compute a four-bit addition. The carry bit output from a full adder is connected to one of the three inputs of the full adder to the left. Note that the third input of the right-most full adder should be set to zero and that the carry bit output of the left-most full adder allows us to detect overflow (i.e., the result cannot be encoded in just four bits).

This section illustrates modular design, a powerful design concept which is critical for implementing complex systems. We first designed and evaluated a small module (half adder) and then reused this small module to implement a larger and more complex module (full adder). We then chained multiple full adders together into a ripple-carry adder to enable multi-bit addition.