Chapter 4

Introduction to Digital Logic

You are surrounded by the fruits of the information revolution, from the smart phones that you carry around to the web services that you depend upon. This chapter will look inside these devices to show you that these extremely complex systems are built out of very simple elements that operate on even simpler signals. These signals have only two legal values, which are sometimes called true and false, and other times called 0 and 1. To introduce you to this world of 0s and 1s we will start with a very silly machine, called a useless box.

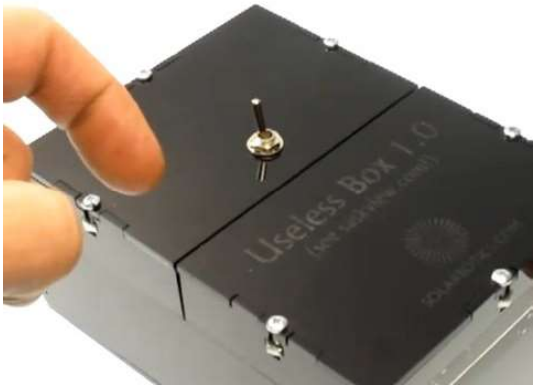


Figure 4.1: A useless box

4.1 Useless box

The origins of this machine date back to the 1930s. Once turned on, the boxs sole purpose is to perform the mostly useless task of turning itself back off. In a little more detailed description, the useless box is a machine with a simple two state switch which, when flipped to the on position, causes an internal lever to peek out from inside the box and flip the switch back to the off position, after which the lever retreats back inside the box. In its most simple implementation the box contains two switches (the flip switch that serves as the user interface, and a limit switch), a motor, and a battery pack. Smarter versions of the boxes, which can be programmed to execute their

useless tasks in more creative ways, will also include a microcontroller of some sort, in our case an Arduino. You will build both version of this box in lab, but this section will focus on the original, simple box.

We would like to describe how this machine works, in a more formal way. One useful way of describing machines like this is to enumerate the different actions that the box can be doing at any time. For many systems these diagrams become very complicated, but the base useless box can only do two things, or nothing. Since we want to represent every action including no action, this box has three possible states. Here state means a specific configuration that causes some action to happen. The three states for the box are: finger is moving forward, finger is moving back into the box, and the finger is not moving. Now that we have the different states that the box can be in, we need to determine what causes the box to change the state that it is in.

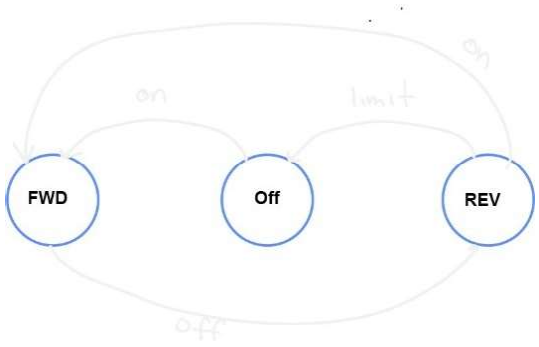


Figure 4.2: The FSM of a useless box

Lets trace through the typical movement of a useless box. The most common state, and the resting state of the box will be during its OFF state. When the master switch is flipped on, the box transitions to the FWD state, where it will stay until the switch is flipped back off, generally triggered by the boxs lever though possibly triggered by an outside force such as the user causing a transition into the REV state. The box will stay in the REV state until one of two things happens: either the lever reverses far enough to hit the limit switch, putting the box into the OFF state, or the master switch is flipped back into the on position, putting the box back into the FWD state. We can visualize this behavior using a state diagram, which shows all of the states and the possible transitions between them. The state diagram for a simple useless box is shown in Figure 4.2.

Although fairly simple, the state diagram provides a powerful framework for the design of a digital system or a program that needs to keep track of where it is in a complex sequence of tasks. It is so important that it is given a fancy name: finite state machine or FSM. If you take more hardware design classes, like EE108, you will learn much more about FSM. But we will end this short discussion of FSM to look at little more closely at the signals on the motor of the useless box, because they have a very interesting property.

Our state machine says that the motor can be in one of three states, so lets measure the voltage on the motors leads in each of these three different states. M+ is the lead of the motor that gets positive voltage for the motor to move the finger forward, and M- is the lead that must be positive for the motor to move in reverse. If we measure the voltage on these wires with a meter, we measure the following voltages (the reference is the negative end of the battery). Notice that the voltage on the motor has only two stable values. It is either at the voltage of the reference value, which

we call gnd, or it is at the voltage of the battery pack. For historical reasons, this voltage is often called Vdd, and we will use this name as well. Since the voltages only have two values they can be thought of as a Boolean signal, which are described next.

State	M+	M-
Forward	4.5V	0V
Reverse	0V	4.5V
Off	0V	0V

4.2 Boolean Signals and Logic

While the useless box is kind of a silly toy, it serves as an introduction to a very powerful idea that has shaped the world that we live in today: digital logic. We will continue with the example of the useless box to show how the signals to the motor can be considered as two binary signals and how the control of the box can then be expressed as Boolean equations. A Boolean signal is one that has only two values, true or false. If we are going to represent a Boolean signal in an electrical circuit, we can use voltage to indicate the value of the signal. Generally we use the following mapping:

TRUE	1	HIGH	4.5V (Vdd)
FALSE	0	LOW	0V (GND)

What we have just created is a way to map an electrical voltage to a logical value. As in all electrical circuits, the way you find the voltage is through nodal analysis, or some analysis short-cut, but once you do this analysis you will find that the resulting voltage will either be very close to the power supply (Vdd) or have a value that it very close the the reference, gnd. Since node voltage generally will only have one of two values, we say it is a Boolean signal, and contains one bit of information.

Lets look again at the voltages found on the motors terminals. Using this mapping of voltages to logical values, we get the following table to the right. Notice that with this electrical mapping, we can say that the motor moves forward when M+ is 1 (true) , and M- is 0 (false). Similarly if M- is 1 and M+ is 0, the motor runs in reverse. In this binary view, it makes sense to call the M+ wire Forward, and the M- wire Reverse, since when one is true and the other is false, the motor will move in the direction of the signal that is 1.

State	M+	M-
Forward	True	False
Reverse	False	True
Off	False	False

When we look at the useless box circuit a little further, we see that the voltage on the switches

also form electrical binary signals. The voltages at the output of these switches will also only be gnd or Vdd, so they are binary. This means that we can represent them in our circuit as a Boolean signal too. Lets represent the state of the two switches in the box by two Boolean signals, onSwitch, which represents the state of the on/off switch, and limitSwitch, which is the state of the limit switch. Now we can represent the operation of the useless box using simple Boolean expressions.

Just like there are operators that work on numbers (+,-,*,/) there are a number of Boolean operators, also referred to as logic operators, that are used to combine and modify Boolean information. The three main operators are AND, OR, and NOT.

Operator	Programming Notation	Meaning
AND	A&&B	Are A **AND** B both true?
OR	A || B	Is at least one of either A **OR** B true?
NOT(INVERSE)	!A	Is **NOT** A true? (Is the **INVERSE** of A true?)

Table 4.1: Truth table for AND operation

A	B	A && B
1	1	TRUE (1)
1	0	FALSE (0)
0	1	FALSE (0)
0	0	FALSE (0)

Table 4.2: Truth table for OR operation

A	B	A || B
1	1	TRUE (1)
1	0	TRUE (1)
0	1	TRUE (1)
0	0	FALSE (0)

Table 4.3: Truth table for NOT operation

A	!A
1	FALSE (0)
0	TRUE (1)

In addition, NOT can be combined with both AND and OR to create NAND and NOR respectively. When this happens, you first do the Boolean operation (AND or OR) and then invert the output. So for a NAND gate, the output is false (0) only if both inputs are true(1). So a NAND gate just inverts the output of an AND gate. Notice how each operator does exactly what its name means in English.

We will now use these operators to define the behavior of our useless box. What combination of the switch states must occur in order for the motor to be going forward? For the box to start moving in the forward direction, the user has to have flipped the user switch, so we know that SwitchOn must be true. We dont want to look at the limit switch in this case, since we want to finger to move forward whether the finger is fully retracted (limitSwitch is true), or if it is still moving back into the box. What makes the box fun is that the finger reverses as soon as you flip the switch. Because of this, the only condition for Forward to be true is SwitchOn to also be true. The Boolean equation is therefore:

Forward = SwitchOn

Now, we ask the same question for the reverse direction. For the box to start moving in reverse, the user switch must have been flipped back again (either by the boxs lever or by the user). The box moves in reverse until it has hit the limit switch, which in this case should shut off the box, otherwise the lever would forever be in reverse. This means that the box is only in reverse while both the user switch and the limit switch are off. The Boolean equation for the reverse motion is therefore:

Reverse = !SwitchOn && !Limit

In the case of the useless box, we have created this logic using mechanical switches and motors. While this contraption does work, and it is fun to play with, each switch is pretty big, and they dont switch very rapidly. Wouldnt it be great if we had a new type of switch that we could control with another electrical signal? There is such a device, and it is called a MOS transistor.

4.3 MOS transistors

MOS transistors are a very interesting new type of electrical device. There are two different flavors of MOS transistors, and they are called pMOS and nMOS. There are different conventions used to symbolize both, some of which are shown to on the right.

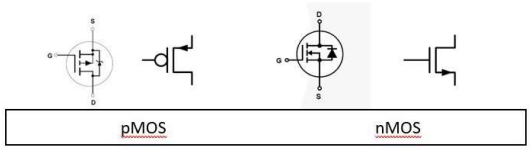


Figure 4.3: PMOS and NMOS symbols

A MOS is a three terminal device. These terminals are named Gate, Drain, and Source, indicated by the G, D, and S, on the images above. We will typically only use the less complex symbols on the right. Note that the orientation of gate, drain, and source are the same for both PMOs symbols, and similarly for both NMOS symbols (the source terminal always has an arrow drawn on it in the simpler symbols we will be using).

Typically, and for the purposes of the class, the drain serves as the output terminal of the transistor. In order to understand the device, we will characterize the iV curve for each pair of terminals. If we look at both the gate-to-source and gate-to-drain, shown in Figure 4.4, we see that there is no current flow through the gate terminal. The iV curves show an open circuit. These curves shown V_{gs} vs I_{gs}.

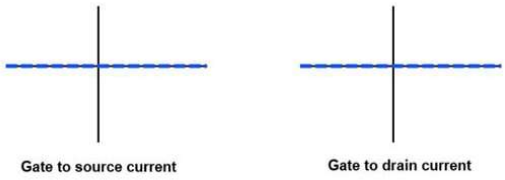


Figure 4.4: iV curve for Gate-Source terminals and Gate-Drain terminals

What this means is **no current flows from the gate to the source in neither the nMOS nor the pMOS.**

For the remaining pair of terminals, drain-to-source, the pMOS and the nMOS exhibit different behavior. We will discuss this next.

We will first look at the nMOS transistor. We will briefly look at the characteristics of a real transistor, then move on to discuss the simplified model we will be using in E40M, as we are only using them as switches.

The drain-to-source iV curve is controlled by the voltage between the gate and the source terminal, as shown in Figure 4.5. If the gate-to-source voltage is less than the threshold voltage no current flows. For our external transistors, his threshold voltage is around 1V, but it can be smaller in other transistors. So when the gate to source voltage is 0V, no current can flow - notice how you don't see a 0V curve, that is because it lies at 0A always. When the gate-to-source voltage is +5V, plenty of current flows, particularly for higher V_{ds}. The amount of current depends on both the gate to source voltage (Vgs in the figure below), and the voltage from the drain to source.

Note: For nMOS transistors the source is often connected to the reference node, GND, so when this happens you can simply look at the gate voltage, but you should always remember it depends on

the gate to source voltage.

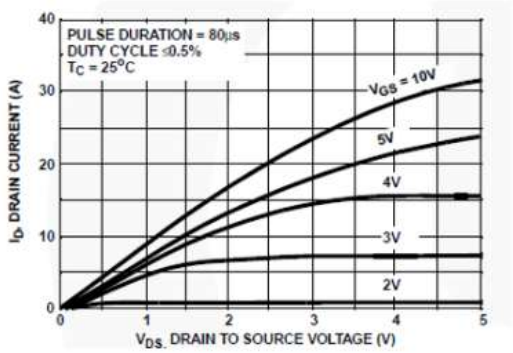
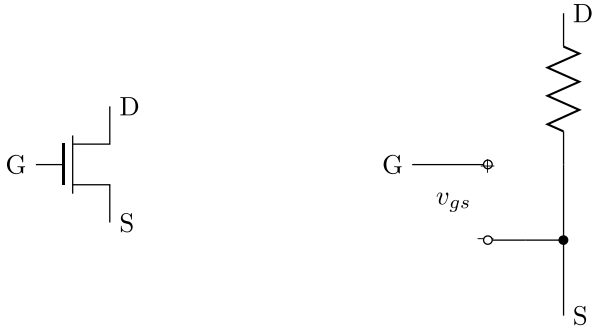


Figure 4.5: iV curve for Drain-Source terminals - V_{ds} vs I_{ds} for different applied V_{gs}

However, this is more detail than we need to know to build logic gates. As we are only interested in building logic gates, we will be using these transistors only as switches.

For logic gates the voltages will be only Vdd (the power supply) or gnd, and so we really only need to model the device behavior at these voltages. With this constraint, **we can model an nMOS device as a voltage controlled switch in series with a resistor.** When the gate is at GND or, if we want to relate this back to Boolean, is LOW, then the switch is off If the gate is at Vdd, or is HIGH, then the switch is on. This case is shown below, where if v_{gs} were a value larger than the threshold voltage, then we can model the transistor as shown in the circuit on the right - drain are source are now connected by a (small) on resistance.



An nMOS serves as a great switch to GND. However, we also need a way to connect our output to Vdd as we need a way to output both a HIGH and a LOW from our circuit. If we try to use an nMOS as a switch to Vdd it wont work. This is because in order to turn the transistor on, the gate needs to be at a higher voltage than the source. We want the source to connect to Vdd. To achieve this we'd want a voltage higher than Vdd, which is more complicated to achieve (we would have to create this higher voltage, instead of just uding Vdd in the case where the nMOS connects to GND). To simplify things, we can use a pMOS instead!

A pMOS is just like an nMOS, but upside down. It turns on when the gate-to-source voltage is less than its threshold voltage, and a pMOS threshold voltage is negative. Also notice that for a pMOS device the drain to source voltage should be negative too. So for a pMOS device the source

should be a high voltage in the circuit. This is exactly the opposite of an nMOS device, where its source should be a low voltage.

Again we will simplify this iV curve since our voltages will be only Vdd (the power supply) or gnd, and we only need to model the device behavior at these voltages. With this constraint, we can model an pMOS device as a voltage controlled switch in series with a resistor. When the gate is at GND or, if we want to relate this back to Boolean, is LOW, then the voltage from the gate to the source is -5V and the switch is on (note this is the opposite of the nMOS and is the reason our gate has an inversion bubble on its input). If the gate is at Vdd, or is HIGH, then the gate to source voltage is zero, and the switch is off.

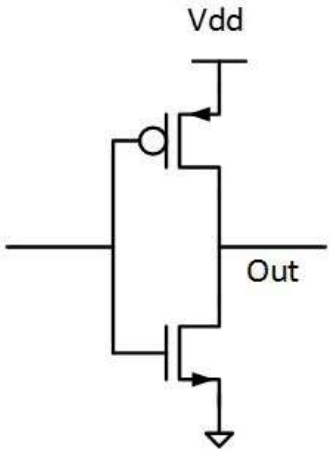
In summary:

1. The nMOS is on when the gate-source voltage is POSITIVE and off when the gate-source voltage is 0 or less. The nMOS is typically used to connect the output to GND when its on.

2. The pMOS is on when the gate-source voltage is NEGATIVE, and off when the gate-source voltage is 0 or more. The pMOS is typically used to connect the output to Vdd when its on.

4.4 Building CMOS logic gates

Now that we know how CMOS transistors work, we can now use them to build logic gates. First well start with a simple inverter. The inverter is the logic gate that corresponds to the NOT, or INVERSE, operator. An inverter takes an input, which should be a HIGH or a LOW voltage, and outputs the opposite. So how do we build this? Well we need to create a circuit that connects the output to GND when the input is HIGH, creating a LOW output, and connect the output to Vdd when the input is LOW. Notice that this is exactly what the nMOS and pMOS transistors do. When the nMOS gate goes HIGH, it connects the output to GND, producing a LOW. When the pMOS gate goes LOW, it connects the output to Vdd, producing a HIGH. In order to build an inverter we simply connect an nMOS and a pMOS from input to output, like so:



What is amazing about transistors is that one can use them to create many different logic functions. A logic function only ever outputs '1' or '0'. Hence, to create logic gates with transistors, we need to follow the following two rules:

1. The output should always be connected to either VDD (a value of '1') or GND (a value of '0').

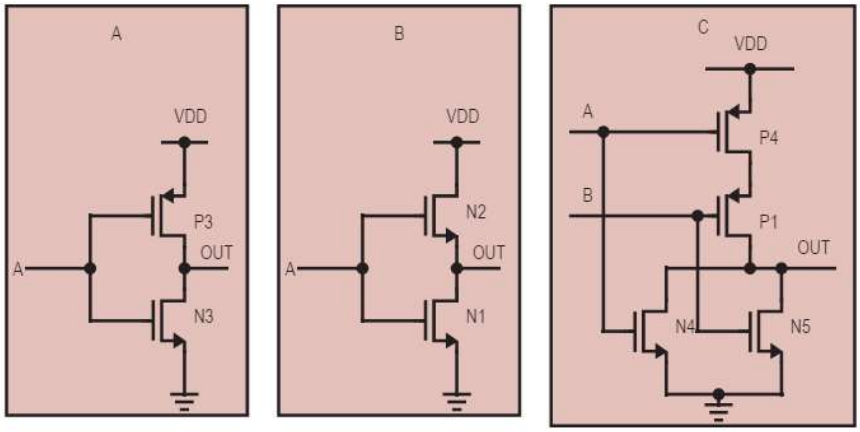
2. The output should never be connected to both VDD and GND at the same time.

This means that when designing a logical function, we need to create a collection of pMOS transistors that connect the output to Vdd when the output should be 1 and a set of nMOS transistors that connect the output to GND when we want the output to be 0. We also need to make sure that the output is always driven, and it is never driven to both Vdd and GND. Notice that in our inverter, for each input case, at least one of the transistors is on and they are never both on, so they satisfy these rules.

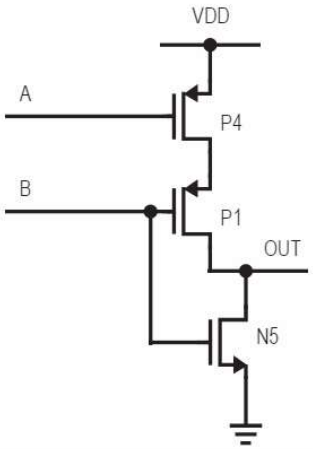
If you violate these rules then one of two bad situations happen. If there is an input condition where the output is not connected to Vdd or GND then the output node is left floating unconnected to any other node. Since the node has no resistive paths to any supply, any node voltage is possible (there are no device current equations to constrain it), so we cant say whether the output will be 0 or 1, and it might even be in a state that is illegal (say at a voltage that is Vdd/2). If you put

Vdd/2 into an inverter, both transistors can turn on, and that inverter will get hot. Transistors will also get hot if you create a gate where some input combination creates a path from the output to both Vdd and GND. Now we have a resistive path through the MOS transistors between Vdd and Gnd, and current will flow through these transistors. Now the output voltage will depend on the resistance ratio between these paths, and again the output wont be good digital values.

Problem 4.1 Which are valid logic gates?
By using the rules described above, determine which of the following three CMOS circuits are valid logic gates.



Problem 4.2 More practice: Is the following a valid logic gate?

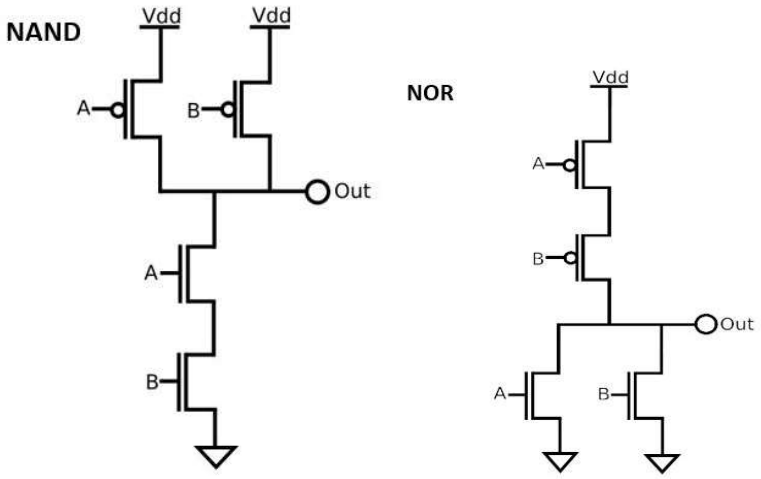


4.4.1 How to create a logic gate?

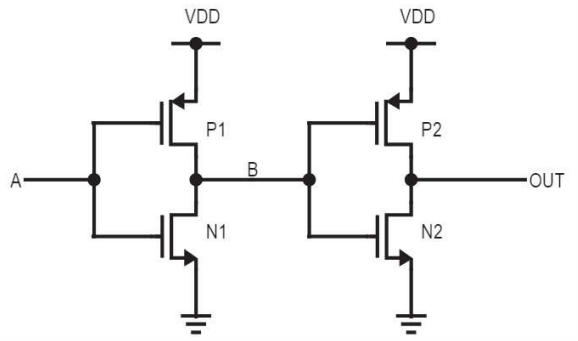
It is not that complicated as long as you follow a few simple rules. First notice that nMOS transistors connect the output to GND and turn on when the inputs are HIGH, and pMOS transistors connect the output to Vdd and their input is LOW. This means that 1 inputs can only cause the output to be 0, and 0 inputs can only cause the output to be 1. Said a different way, all MOS gates will invert. They can only make inverters, or NAND gates or NOR gates. To make an AND gate, you need to make a NAND gate and then add an inverter to its output! When building a logic circuit, its helpful to ask two questions: What conditions should cause the output to be 0? We use the answer to this question to create the nMOS transistor circuit that connect the output to GND. If two inputs both must be true for the output to low, then we need to create a circuit that only connects the outputs when both transistors are on. This condition occurs when the two transistors are connected in series. Then there is a path to from the output to GND only when both transistors are on. If the output should be low if either of the inputs are on, then we should create a circuit where the two transistors are put in parallel, where the source of both transistors goes to GND, and the drain of both transistors connects to the output. In this case when either transistor is on, the output will be connected to GND.

What conditions should cause the output to be 1? We use the answer to create the pMOS transistor circuit that connects the output to Vdd. If two inputs both must be LOW for the output to HIGH, then we need to create a circuit that only connects the outputs when both transistors are on. This condition occurs when the two transistors are connected in series. Then there is a path to from the output to Vdd only when both transistors are on. If the output should be HIGH if either of the inputs are LOW, then we should create a circuit where the two transistors are put in parallel, where the source of both transistors goes to Vdd, and the drain of both transistors connects to the output. In this case when either transistor is on, the output will be connected to Vdd.

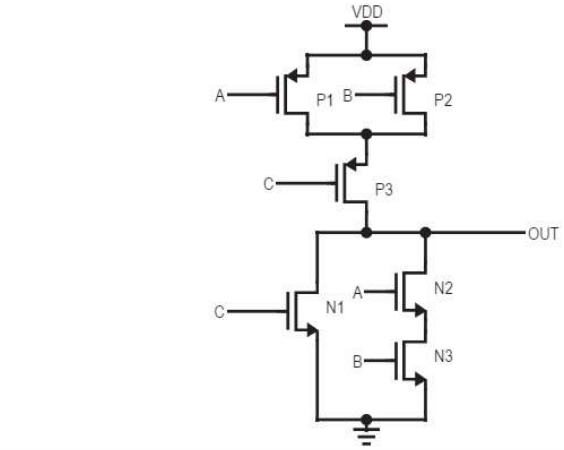
Using these rules gives the following transistor circuits for a NAND and NOR gate.



Problem 4.3 Write out the truth table for the following circuits. Then, write a logical expression relating the inputs and the outputs.
1.)



2.)



4.5 Coding with the Arduino

4.5.1 Arduino

"Arduino" is the combination of a software development environment, a series of microcontroller boards, and a library which provides a common set of useful functions across all of the hardware boards. The fact that the design and development package are open-source, combined with its emphasis on easy to use programming model has helped make Arduino the standard for hobbyist electronics projects in recent years. While there are now Intel and ARM powered Arduino designs, most Arduino boards still use Atmel microcontrollers.

Your lab kit includes an Arduino Micro, which contains everything you need to make an embedded computer-controlled system - an Atmel micro-controller, power supply and regulation, a USB connection for power and programming, and input/output pins which make it easy to connect to both digital and analog circuits.

The Micro is designed to be very compact. Larger Arduino boards include a standard set of header pins positioned precisely to be "plug and play" compatible with a set of accessories called Shields. Through shields, you can add everything from wireless networking to displays to your Arduino.

You can program your Arduino using your laptop or desktop computer through its micro-USB port. Development for Arduino is all done through the Arduino IDE, discussed at:
http://arduino.cc/en/Guide/Environment.

4.5.2 Arduino IDE

The Arduino IDE has been crucial in the Arduino's widespread success. It is designed to allow people with little to no programming or hardware experience quickly get software up and running. It has created a set of library routines that make driving values on pins of the chip, and reading values from other pins of the chip relatively easy. They also have libraries to communicate results back to your computer, and other stuff that makes building computer controlled hardware easier.

[Testing Blink (before Friday 10/10)]

1. Download the Arduino development environment from

   http://arduino.cc/en/Main/Software and install it.

2. Run the IDE and configure it for the Arduino Micro: Click "Tools → Board" and select "Arduino Micro". If you configure for another board, the IDE will hang when you try to download.

   The other thing you might need to set is the serial port used to communicate with your Arduino. This again is under the "Tools" tab, and if you are lucky, there will only be one port listed. Note that the port name can change (or dissappear entirely) depending on which USB port you plug the Arduino into, and the IDE won't automatically correct it. Worse, it will sometimes appear to download code correctly, even when you have the wrong port selected (this occurred most frequently on Macs). If unplugging, re-plugging, and selecting the right port doesn't work, you may have to reboot.

   Please let us know if you have problems with getting the IDE to talk with you Arduino.

3. Open the "Blink" example and run it. The LED on the Arduino should blink.

There is nothing you need to turn in for this prelab, but please make sure you have everything running.

The Arduino programming language is a slightly restricted dialect of C++. If you're not familiar with C++, that's fine - it has a very similar syntax to Java.

One advantage of using Arduino is that there is a lot of code out on the web for Arduinos, so when you are trying to do something, there often is code that is close to what you are trying to do that you can use and modify. In this class we encourage you to start with a program that does something related to what you need, and then modify it, to do exactly what you want. All we require is you acknowledge where the source of code, and that you don't just copy a friend's code. Under the "File" tab, you will find the "Examples" dropdown menu. In that section you will find many programs that do interesting tasks. We encourage you to look at these examples, or search the website to better understand the programming language, and some of the special functions that Arduino provides.

Thorough documentation for all of the built-in functions is on the Arduino reference page: http://arduino.cc/en/Reference/HomePage.

When you start a new program, the Arduino IDE provides two functions `setup()` and `loop()`. The `setup()` function is called once when the Arduino boots up (during power-up or when reset), and `loop()` gets called repeatedly after that. Generally, `setup()` will contain initialization code, and `loop()` will contain some code the reads inputs and responds by setting some outputs.

[The code behind the IDE] When you click "Verify", the IDE combines the code you've written with a template `main.cpp`, which you can find under the installation directory as `hardware/arduino/cores/arduino`.

The meat of `main` is just:

```
setup();

for (;;) {
    loop();
    if (serialEventRun) serialEventRun(); // Handle serial port I/O
}
```

It passes this to the AVR-GCC compiler, which is Atmel's version of GCC for AVR microcontrollers.

If you are familiar with C++, you might want to know that most standard C++ syntax is supported, including classes, references, etc. However, some important things are not:

- There is no heap memory allocation. We've only got 2.5kB of RAM!
- There is no implementation for the C++ standard library (i.e., `iostream`, `string`, etc.)
- Exceptions are not supported.

If you want all of the details, look at Atmel's documentation for the AVR-GCC compiler: http://www.atmel.com/webdoc/AVRLibcReferenceManual/index.html

4.6 Input/output pins on the Arduino

An *input/output pin*, or *I/O pin*, is the interface between a microcontroller and another circuit. It can be configured in the microcontroller's software to be either an input or an output. On the Arduino, this configuration is accomplished using the `pinMode()` function.

This handout concerns three functions: `pinMode()`, `digitalWrite()` and `digitalRead()`. The details of all of these can be found in the Arduino reference at https://www.arduino.cc/en/Reference/HomePage. (If you're reading a printed copy of this document, the links all go there.)

4.6.1 Output pins

When configured as an *output pin*, a pin provides V_{DD} or 0 V to whatever is connected to it (if anything). When the pin provides V_{DD}, we say that it is in the `HIGH` state. When the pin provides 0 V, we say that it is in the `LOW` state. We set the state of an output pin using the `digitalWrite()` function.

An output pin achieves this by making a connection to V_{DD} or a connection to ground internally, inside the microcontroller. This, in turn, is accomplished with transistors whose drains are connected to the output pin, as shown in Figure 4.6.

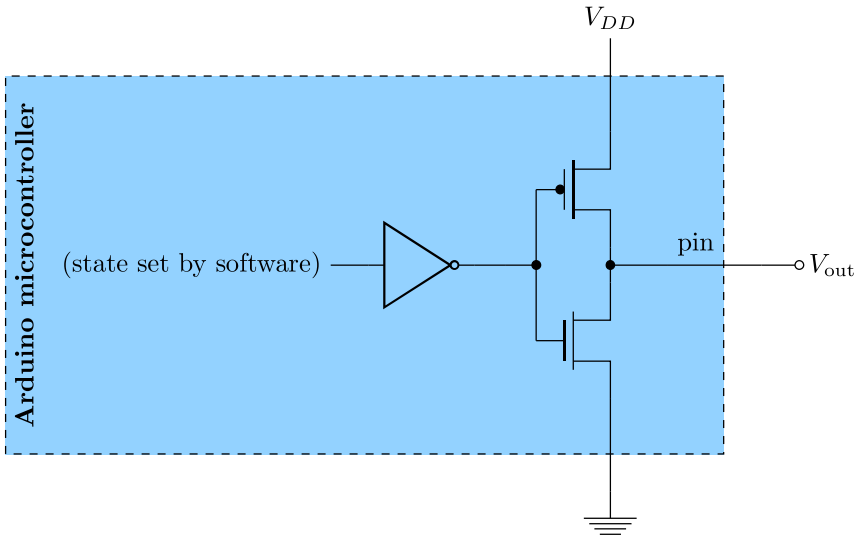


Figure 4.6: Schematic of an I/O pin when configured as an output

When the output is set to `HIGH`, the PMOS transistor turns on and provides a connection to V_{DD}. When the output is set to `LOW`, the NMOS transistor turns on and provides a connection to ground.

Output resistance

If these transistors were ideal, then they would provide a direct connection to V_{DD} or ground, depending on which output state (`HIGH` or `LOW`) was set by software. Of course, these transistors

aren't ideal; as we understood from our discussion of transistors, they have some (hopefully small) *on resistance*. Furthermore, the outputs of microcontrollers are typically designed to power low loads, so this resistance isn't negligible.

The resistance thus "seen" by a device connected to a pin is called the *output resistance* of the pin. Of course, there is not just one output resistance—it depends on the state of the pin. When the output is `HIGH`, it is the resistance to V_{DD} that is relevant ($R_{\texttt{HIGH}}$ in Figure 4.7); in our model, this is the on resistance of the PMOS transistor. When the output is `LOW`, it is the resistance to ground; in our model, the on resistance of the NMOS transistor.

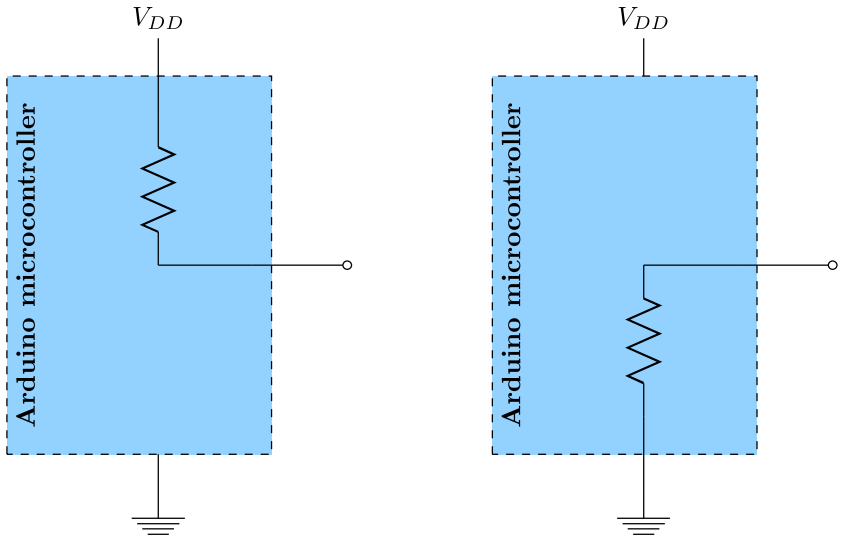


Figure 4.7: Equivalent circuits showing the output resistance when the output is high (left) and low (right)

Measuring the output resistance

Say we wished to measure the resistance of an output pin when it is set to `HIGH`. First, we would need to set the output pin to high, using the `pinMode()` and `digitalWrite` functions. Then, we can measure the resistance with an ohmmeter, as shown in Figure 4.8.

Recall, however, that what we're measuring isn't a resistor, but a transistor that is on. Therefore, it is polarized, and unlike with a resistor, the polarity of the leads matters. The ohmmeter will try to push a test current from the red lead to the black lead, so we should place our leads so that this test current goes in the expected direction. When measuring the resistance of a PMOS transistor, this means putting the red lead at the source (VDD), and the black lead at the drain (the pin). Similarly, when measuring the resistance of the NMOS transistor, the red lead should be at the drain (pin), and the black lead at the source (ground).

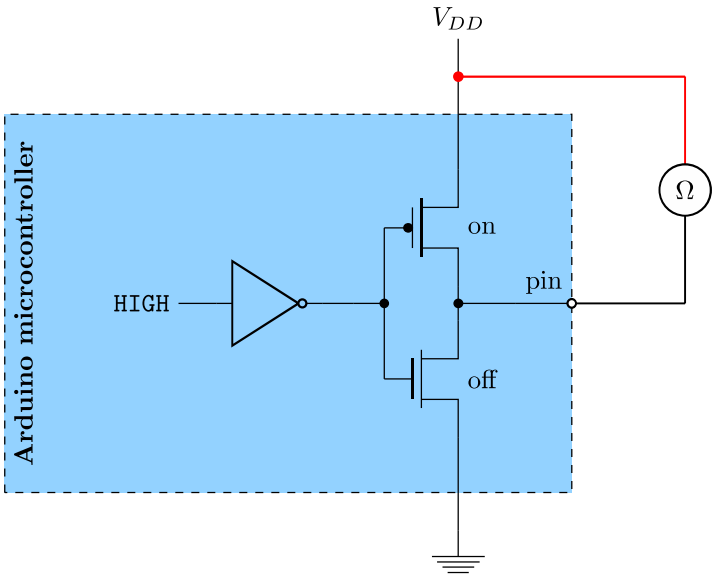


Figure 4.8: Measuring the output resistance of a pin when it is set to `HIGH`

For devices like LEDs, this output resistance isn't too much of an issue: we don't wish to draw that much current from the pin anyway. For motors, however, this output resistance is prohibitively high, and we need to place a driver circuit in between the output pin and the motor, so that the motor can draw enough current. Such a driver circuit often uses *power transistors*, transistors that are designed to provide higher current (lower drain-source resistance) than those in the microcontroller.

4.6.2 Input pins

An *input pin* reads the voltage on the pin as if it were a voltmeter, and returns either `HIGH` (`1`) in software if the voltage is close to V_{DD}, or `LOW` (`0`) if it is close to 0 V. An input pin can be read using the `digitalRead()` function.

Note that the value returned by `digitalRead()` is only well-defined when the input pin voltage is close to V_{DD} or 0 V. The precise meaning of "close" varies between microcontrollers, but for the Adafruit Metro Mini[1] as it's used in our circuit, the input pin voltage needs to be at least $0.6V_{DD}$ to qualify as `HIGH`, and at most $0.3V_{DD}$ to qualify as `LOW`. In the middle (say, at $0.45V_{DD}$), the behavior of the pin is undefined.

An (ideal) input pin takes (approximately) no current, like the gate of a transistor or a voltmeter. In the projects we do in this class, modeling the input pin as ideal is a good approximation.[2]

[1] More precisely, it's the ATmega328, which is the microcontroller used in the Adafruit Metro Mini. The full datasheet can be found at http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf; this information is on page 313.

[2] The input leakage current is specified in the datasheet to be at most 1 µA.

Connecting a switch to an input pin (pull-up resistors)

How do we connect a switch to an Arduino? This is not as obvious as it sounds: remember that switches govern connections, not voltages. So simply connecting the switch between the pin and some other point in the circuit, say, ground, would *not* work (Figure 4.9). When the switch is closed, the input pin would be shorted to ground, and V_{in} would be 0 V, which is fine. However, when the switch is open, the input pin is floating, and we have no idea what its voltage is. (In practice, it will "remember" the last voltage it was driven to, which in this case is 0 V.)

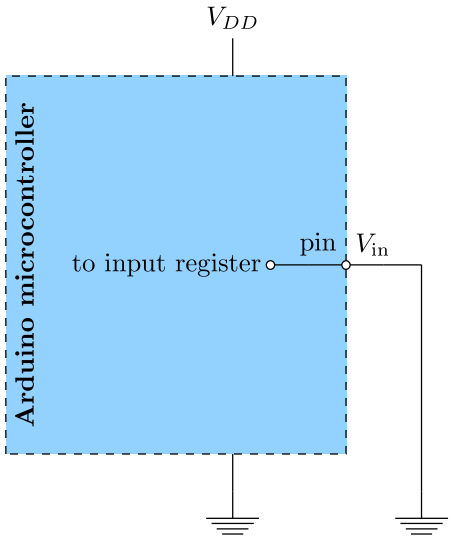


Figure 4.9: A switch circuit that does not work

Consider, instead, the circuit shown in Figure 4.10. When the switch is closed, the pin is still shorted to ground, so $V_{\text{in}} = 0\,\text{V}$. However, this time, when the switch is open, no current flows through the resistor R_{pu} (there's nowhere for it to go, since the input pin takes no current), so the voltage drop across the resistor is zero, so $V_{\text{in}} = V_{DD}$. Thus, we can reliably distinguish between the closed and open states of the switch: when the switch is closed, the input pin will read `LOW`, and when the switch is open, it will read `HIGH`.

You can think of the role of the resistor as being to enforce a "default" state on the pin. When we directly connect the pin to ground, V_{in} is set to 0 V by that connection—the resistor won't get in the way (current will flow through it, but that doesn't affect the input pin voltage). In the *absence* of such a connection, though, the resistor comes into play, *pulling up* the voltage V_{in} to V_{DD}. For this reason, the resistor R_{pu} is often referred to as a **pull-up resistor**.

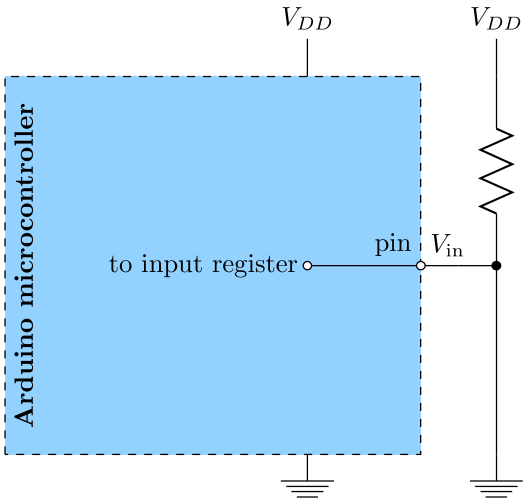


Figure 4.10: Using an external pull-up resistor

We could also do this the other way round, connecting the switch between V_{DD} and the pin, and the resistor between the pin and ground, so that would be a "pull-down resistor". However, connecting the switch to ground and the resistor to V_{DD} tends to be the more common practice. It's so common, in fact, that many microcontrollers (including the Adafruit Metro Mini) implement a pull-up resistor internally, inside the microcontroller, as shown in Figure 4.11. The pull-up resistor can be enabled or disabled in software. To enable it, we pass the `INPUT_PULLUP` constant as the second argument to `pinMode()`: `pinMode(pin, INPUT_PULLUP)`. To disable it, we pass the constant `INPUT` instead: `pinMode(pin, INPUT)`.

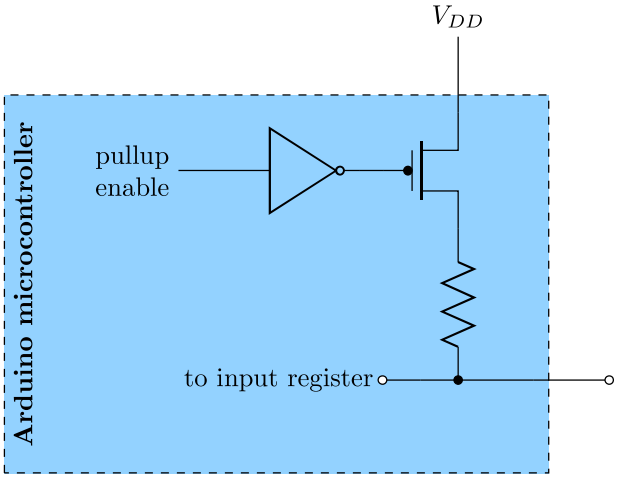


Figure 4.11: Internal pull-up resistor schematic

The enabling/disabling of the internal pull-up resistor is implemented using—you guessed it—

a PMOS transistor, connected to VDD. When the transistor is off, it looks like an open circuit, effectively removing the pull-up resistor from the circuit.

If you use the internal pull-up resistor, you naturally don't need the external one. This makes it acceptable to connect your switch as in Figure 4.9, so long as you enable the pull-up resistor, making the equivalent circuit in Figure 4.12.

Finally, a couple of cautionary notes. First, we haven't yet talked about the value of the resistance R_{pu}. This is partly because it doesn't really matter—for any reasonable resistor value, this circuit will function as we described; the precise resistance only affects how much current flows when the switch is closed. But there is another aspect to this: the internal pull-up resistor isn't specified to have a precise value. Typically, it's on the order of tens of kiloohms; for the Adafruit Metro Mini, it's specified to be between 20 kΩ and 50 kΩ. So you shouldn't use the internal pull-up resistor if you need a precisely-known resistance.

Secondly, the pull-up resistor obviously creates a path to V_{DD}, which can sometimes make a pin you intended to be an output pin seem like it's working even when you forgot to include `pinMode(pin, OUTPUT)` in your `setup()` function. This is because, in a slight quirk of the Arduino, when a pin is configured as an *input* pin, the `digitalWrite()` function actually enables (`HIGH`) or disables (`LOW`) the pull-up resistor. (That is, `pinMode(pin, INPUT_PULLUP)` is actually just shorthand for `pinMode(pin, INPUT); digitalWrite(pin, HIGH)`.)

So if you forget to configure a pin to output, you're actually just enabling and disabling the pull-up resistor. For a multitude of reasons this is a *bad idea*: First, as discussed above, the pull-up resistor does not have a precisely-defined resistance. Also, when the pull-up resistor is disabled, the pin is floating—which is acceptable for an LED, but if you're using this pin to drive the gate of a power transistor, it will mean you have a floating gate. For this reason, it's important always to remember to configure the `pinMode()` of all pins that you're using in your code.[3]

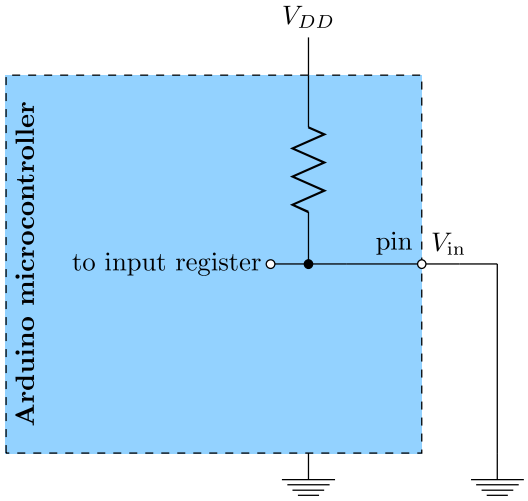


Figure 4.12: Using an internal pull-up resistor

[3] If you don't configure a pin, it defaults to being an input. However, to make your code easier for others to read, you should explicitly configure your input pins, too.

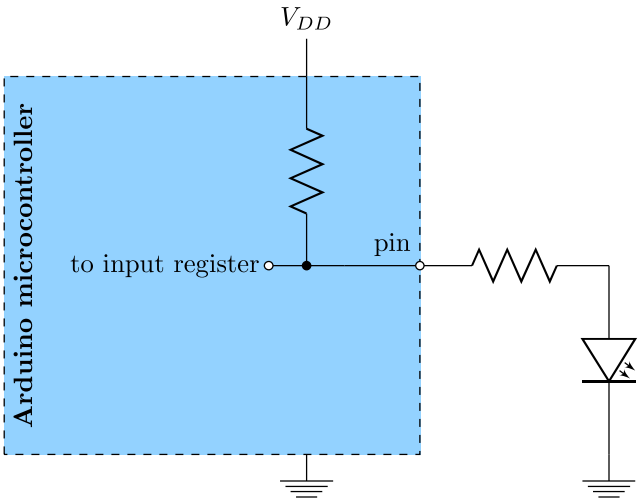


Figure 4.13: When you forget to `pinMode(pin, OUTPUT)` (bad, don't do this)

4.7 Solutions to practice problems

Solution 4.1:

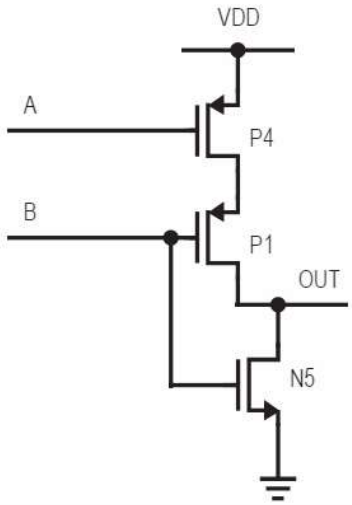
Circuit A is a valid logic gate. When A = VDD, only the NMOS is turned on, conncting the output to GND. When A = GND, only the PMOS is turned on, connecting the output to VDD. For valid inputs ('1' or '0'), the output is never connected to both VDD and GND at the same time. Hence, it fulfulls both requirements for a logic gate. (This circuit is the commonly used inverter.)

Circuit B is not a valid logic gate. When A = VDD, the output is connected to both VDD and GND at the same time (both the NMOS turn on). This breaks the fundamental rule of logic gates (and in fact could also break your circuit if you actually tried to build it.)

Circuit C is a valid logic gate.

To check this, you need to consider all combinations of valid inputs for A and B and determine which gates are turned on for each case in order to find what the output is. The table below summarises the output for all possible input combinations. The output is a valid value for all of them, and at no time connected to both VDD and GND, hence it is a valid logic gate.

A	B	OUT
1 (VDD)	1 (VDD)	0 (GND)
1 (VDD)	0 (GND)	0 (GND)
0 (GND)	1 (VDD)	0 (GND)
0 (GND)	0 (GND)	1 (VDD)



Solution 4.2:
It is not a valid logic gate.

We can check by using the same method as before, testing the output for all valid input combinations, as summarized in the table below. We find that for one of the combinations, the output is left floating and connected to neither VDD nor GND. Hence, it is not a valid gate.

You might realise that this problem is fixed in circuit C of the previoous problem. For every PMOS on the upper half pulling the output up to VDD, there must be a corresponding NMOS on the bottom pulling the output to GND in the correct way to ensure this state does not occur.

A	B	OUT
1 (VDD)	1 (VDD)	0 (GND)
1 (VDD)	0 (GND)	Floating! (not connected to VDD or GND)
0 (GND)	1 (VDD)	0 (GND)
0 (GND)	0 (GND)	1 (VDD)

Solution 4.3:

1.)
The truth table is shown below. The logical expression is OUT = (A!)! = A.

A	B	OUT
1	0	1
0	1	0

2.)

The truth table is shown below. The logical expression is $OUT = (C + A \cdot B)!$. There are many equivalent expressions, but you should always be able to write out an equivalent truth table for whatever you come up with.

A	B	C	OUT
1	1	1	0
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1
0	0	1	0
0	0	0	1