Microcontrollers & Embedded Programming

Tom Briggs

August 25, 2019

Contents

1 General-Purpose I/O **1**

1.1 Introduction to GPIO 2

1.2 Anatomy of a GPIO Port 4

1.2.1 Logical Organization 4

1.2.2 Output Logic 5

1.2.3 Input Logic 7

1.2.4 Data and Control Lines 8

1.2.5 Tri-State Terminology 8

1.3 GPIO Electrical Characteristics 10

1.3.1 Mapping Voltages to Digital Logic 10

1.3.2 Voltage and Current Limits 13

1.3.3 Microcontroller as Source 19

1.3.4 Microcontroller as Sink 20

1.3.5 Microcontroller as Input 21

1.3.6 Driving Other Chips 22

1.3.7 Troubleshooting 23

1.3.8 Real World Example : Coffee Maker 25

1.4 GPIO Programming 27

1.4.1 Memory Mapped I/O 27

1.4.2 C Language Mapping 28

1.4.3 Port Configuration 29

1.4.4 Setting Port Direction 30

1.4.5 Reading from the GPIO Module 30

1.4.6 Output 32

1.5 GPIO Programming 37

1.5.1 Combinational Logic 37
1.5.2 Seven-Segment Display 42
1.5.3 Square Wave Output 43
1.5.4 Delay Loops 46
1.5.5 Pulse Width Modulation 47
1.5.6 Latency 48
1.5.7 Debouncing Switch Input 52
1.5.8 Switch Multiplexing 54
1.6 Source Code Quality 59

Chapter 1

General-Purpose I/O

, One of the most common applications of microcontrollers involves *General-Purpose Input and Output* (GPIO), where the microcontroller interprets voltage levels into logic values. One constant feature in the microcontroller domain, from the very first MCU to the most recent, is the support for GPIO. Over the subsequent years, the types, the capabilities, and the numbers of them may vary considerably, from vendor to vendor, family to family, and even within MCU family. This chapter will explore General-Purpose Input and Output from both an electrical and programming perspective.

This chapter assumes the reader has at least basic knowledge of electronics laws such as Ohm's Law, Kirchoff's Voltage and Current Laws; and the basic properties of simple devices includes resistors, capacitors, inductors, diodes, and transistors (both BJT and FET). This chapter also references a few digital logic devices including: AND, OR, and NOT gates; multiplexors, and flip-flop latches. If the reader is unfamiliar with these topics, review the basic ideas in Appendix **??**, and them come back to this section.

1.1 Introduction to GPIO

Through a clever use of digital logic devices, GPIO maps a bit in a register to a voltage on a physical pin of the microcontroller. The state of the GPIO line is totally controlled by the program running on the MCU. The line can be in one of three states: input, driving "high" voltage, and sinking to ground. It is this primitive, low-level operation that makes GPIO so flexible and powerful. With that flexibility comes the burden of writing code to make it work correctly.

Throughout the rest of this text, we will examine a number of different types of more specialized I/O devices. We will find a diverse range of specialized controllers. We will see controllers that drive electric motors, display text and graphics on displays, interact with other chips, and even communicate at high-speed with other computer systems. These special purpose controllers work with little programmer intervention. But we are constrained in using these hardware resources to exactly the purpose for which they were designed. We can only vary their operation by what the hardware engineering allowed when the chip was made. Here, we trade flexibility with the ease of allowing the hardware to do the work for us.

For those new to microcontrollers, GPIO can present a logical challenge: how do we connect our code to the real world? the short answer is through the memory. When we first learned to program, we learned to use a programming language with its different statements and variables. We also learned that those variables are stored somewhere in memory, and the compiler manage that for us. Ultimately, while the CPU is critical to *changing* the state of the computer, the memory is critical to remembering the state of the computer. But how does memory actually work?

Section **??** introduced the idea of building a single bit memory, an SRAM cell, using six transistors. With that hardware, we can store one bit, and we can retrieve one bit. We can put 8 of those SRAM cells together to store a byte, and we can put many bytes together to create a computer memory space, with each byte at a particular address. In that same section, we learned that the memory controller activates the proper memory cells in response to a load or store instruction's address. A load simply detects whether the SRAM cell's transistors are turned on (logic high) or turned off (logic low); and a write simply changes the transistor's state.

One way of looking at GPIO is that it is a memory device that is also connected to the physical world. If we were to add and additional wire to the 6T SRAM cell of Figure **??**, we could run that wire out from the CPU's core to one of the legs of the processor. By reading and

writing to that memory bit, we could interact with the memory cell's state and consequently, that one leg on the MCU. Logically, that is all that GPIO is, a memory cell whose value is connected to the outside world. Just like any other memory cell, what it represents and how it changes is controlled by the program running on the MCU.

This is a highly simplified view of the GPIO module. But with that simple understanding, we can go on to examine the actual details of a GPIO module. We will examine its electrical characteristics and limitations. We will study how to connect the GPIO legs into a circuit. Finally, we will look at writing code to configure and use the GPIO module.

1.2 Anatomy of a GPIO Port

This section explores the composition of the GPIO port. The section starts with a logical view of the organization and function of the digital logic elements that make up the I/O port.

1.2.1 Logical Organization

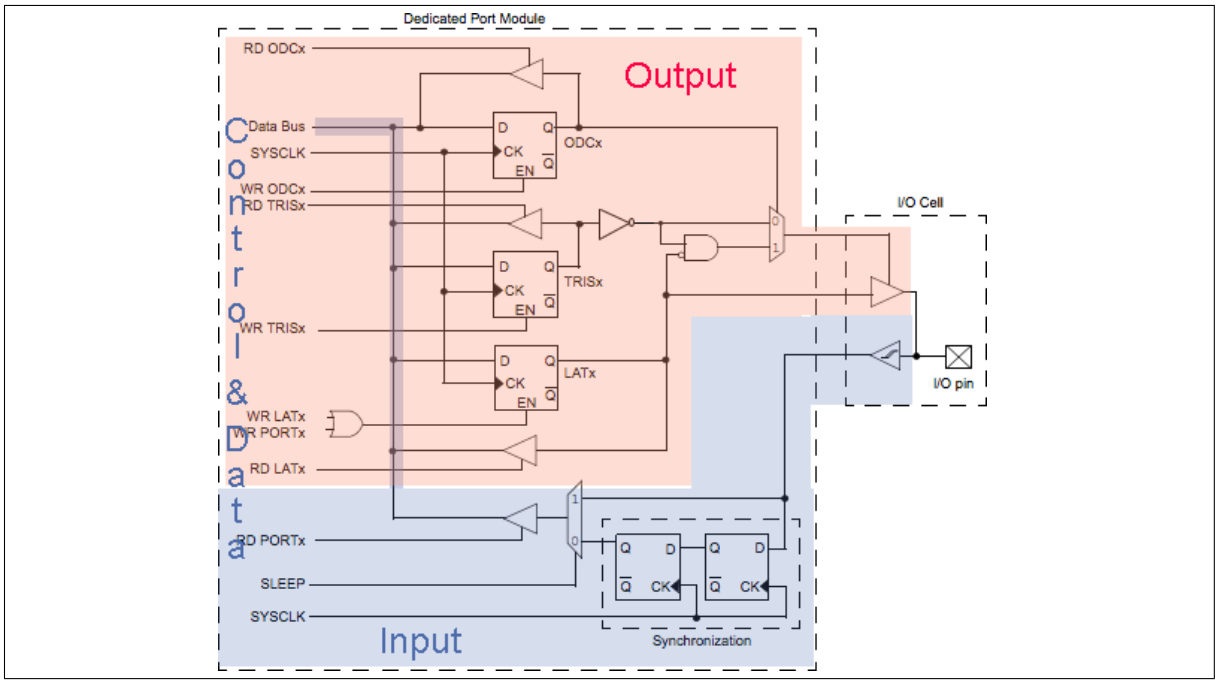

Figure 1.1: PIC32 GPIO Port Diagram[3]

Figure 1.1 shows a typical I/O cell for a GPIO module. This specific I/O cell is from the PIC32, but other microcontrollers will have a similar set of similar elements. Broadly, the diagram consists of four major parts:

- I/O cell and external pin connections (right hand side),
- output logic (highlighted in pink),
- input logic (highlighted in blue), and
- data and control lines interfacing to the CPU core (left hand side)

I/O Cell

The physical connection to an external pin of the microcontroller is represented in Figure 1.1 by the square with an X through it and is labeled "I/O pin." The pin is connected to the input

and output logic through a pair of *buffers*. These buffers will be explored in greater detail in the next two sections.

As we have already seen in Chapter **??**, there are never enough pins in a microcontroller, and their functions are often *multiplexed* with other peripherals (e.g. Figure **??**). So, we must ensure, through software configuration, that the I/O cell is available for use with GPIO.

Input Protection Because the I/O cell is directly connected to buffers, it is crucial to ensure that the voltages present on the I/O pin are within the accepted range of the device. Devices can sometimes tolerate higher voltages, but can suffer significant damage for even small negative voltages. For example, a $3.3\,\mathrm{V}$ microcontroller typically tolerates voltages between $-0.3\ldots5\,\mathrm{V}$. Voltages outside this range can cause permanent damage to the device. Two common reasons for this are bad circuit design and electrostatic discharge. There are a number of techniques to protect the I/O cell from damage explored in Section 1.3.2.

1.2.2 Output Logic

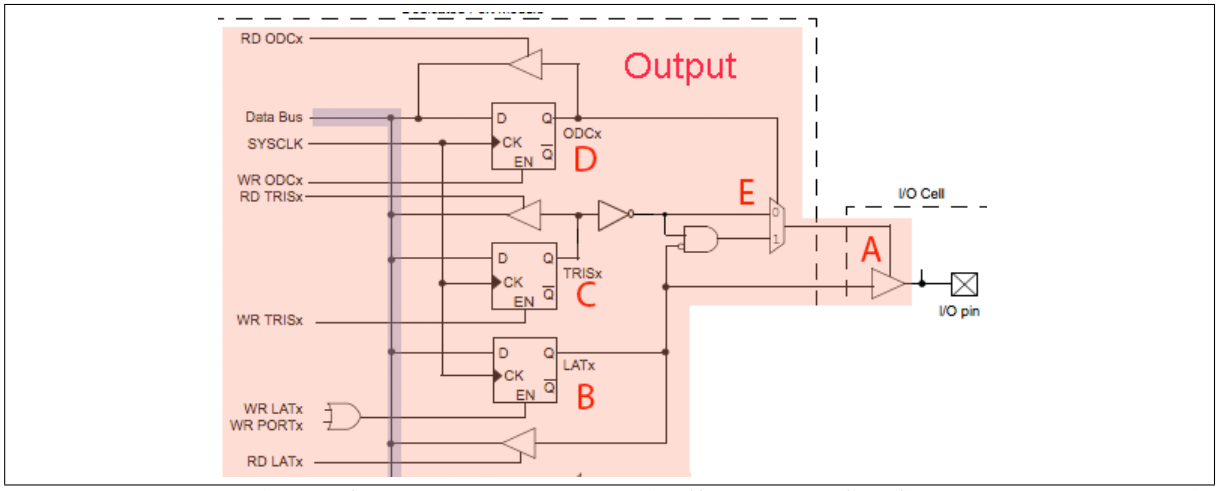

Figure 1.2: Output logic of GPIO port. A. Output buffer, B. Data flip-flop, C. Tri-State control, D. Open-Drain control E. buffer enable logic.

The output logic components are shown in Figure 1.2. Ultimately, the goal of all of these components is to configure the output buffer (A) into one of three states: enabled and connected to supply voltage; and enabled and connected to ground; and disabled. At its simplest, a *buffer* makes its output voltage match its input voltage (See Section **??** for more details).

When the output buffer is enabled (notice the signal entering the top of the buffer (A)). When the enable signal is high (a logic 1), then the buffer is enabled, and when the signal is low (logic 0) then it is disabled. When a buffer is disabled its output is effectively disconnected

from the circuit, a technique known as high impedance operation. When the buffer is enabled then its input, which is connected to the data flip-flop (B) will determine its output. If the input is a high voltage (logic 1) then its output will also be a high voltage (logic 1), likewise, when the input is low voltage (logic 0) then its output will also be low voltage (logic 0).

If we work backwards from the buffer (A) we find an MUX (E). This MUX will select from one of its two inputs, channel 0 is connected to a flip-flop (C) through an inverter; and channel 1 is connected through an AND gate to both flip-flops (B and C). Channel 0 is the normal operation while channel 1 is part of the *open-drain control*, and will explained shortly.

The flip-flop (C) holds a configuration bit written by the CPU. The output of the flip-flop (C) goes through an inverter, into the MUX (E) and then into the buffer. If the flip-flop (C) holds a logical 0, then the inverter will flip it to a logical 1 which will be fed into the MUX (E), into the buffer (A), and it will be enabled. Likewise, if the flip-flop (C) holds a logical 1, it will be inverted to a 0 and the buffer (A) will be disabled. Because this flip-flop is used to configure which of the three states the buffer can be in, it is also known as the *tri-state register*.

Open Drain Control Finally, we can look at the last flip-flop (D), which is also known as the *open drain control register*. During open drain operation, the I/O port will only alternate between sinking to ground and high impedance mode, it will not ever connect the output buffer to the supply voltage. The subject of open drain I/O will be further explored in Section 1.4.6. To see how this is implemented, look at the *AND* gate and see that one of its inputs comes from the negated *tri-state register* (C) while the second input is from the data flip-flop (B). The small bubble at the input of the *AND* gate indicates that this line is also negated. The following truth table shows the output of the AND gate:

Tri-State	Data	AND	Buffer
0	0	1	Pull to Ground
0	1	0	Disabled
1	0	0	Disabled
1	0	0	Disabled

So, if the open drain control register (D) is enabled, then the MUX (E) will use the output of the *AND* gate to control the buffer (A). And, from the truth table, we can see that when the tri-state control register (C) is configured for output and the data is 0 then the buffer is enabled and the since data is 0, the buffer's (A) will be to pull to ground, in all other cases, changing either data or the tri-state register will disable the buffer.

1.2.3 Input Logic

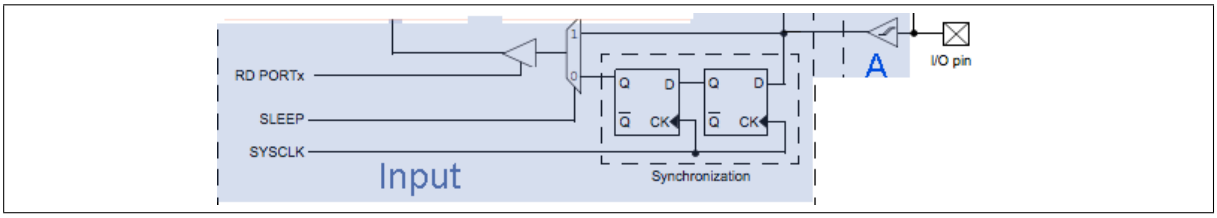

Figure 1.3: Input logic of GPIO port. A. Input buffer, B. Synchronization flip-flops, and C. Sleep mux

The input path starts with the input buffer (A). Notice that the input to buffer (A) shares a connection to the I/O pin and the output of the buffer from the output logic. The input to the buffer is known as a *high impedance* path and will draw a negligible amount of current from the circuit it is connected to. The output of buffer (A) is routed either to the sleep MUX (C) or to the synchronization flip-flops (B).

Metastability The the pair of synchronization flip-flops are used to provide synchronization of input signals to clock edges and to prevent *metastability*, a condition that allows the output of a flip-flop to change unexpectedly due to a glitch in its input clock or data lines. Metastability is a common problem with flip-flops. Figure 1.4 shows the impact of glitchy input to a flip-flop. The input to the flip-flop must be constant during its setup-and-hold time. If there is a glitch in the input to the flip-flop during this time, then it is uncertain what output value the flip-flop will take, and during this time, its output value will also be uncertain. The second flip-flop will also latch up its input on the same rising edge of the clock, but it will latch up the previous, stable output of the flip-flop. Using a pair of flip-flops manages the meta-stability

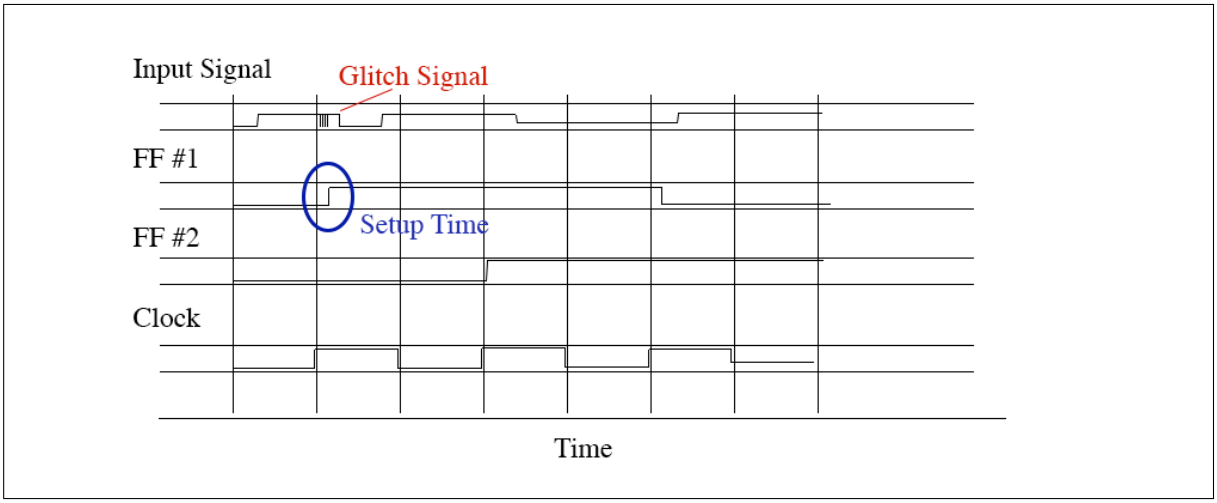

Figure 1.4: Flip-Flop Pair with Input Glitch

problem.

1.2.4 Data and Control Lines

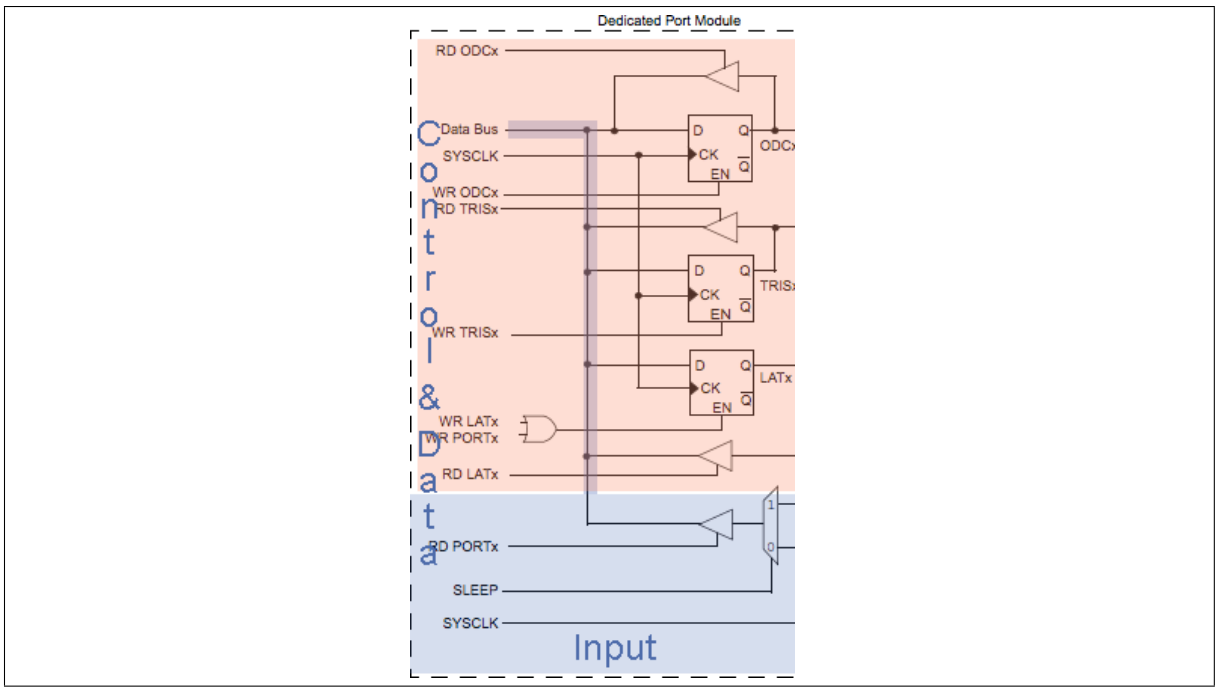

Figure 1.5: GPIO Port Control lines

Figure 1.5 shows the control lines between the CPU and the I/O port. When the CPU performs a load word or store word to one of the memory-mapped addresses, the control unit will enable one the control lines to select the correct data. For example, if the CPU performs a load word to read from the tristate register, then the `RD TRISx`, signal would be asserted. If we follow this signal we see that it enables another buffer. When this buffer is enabled it will make its output match the current value stored in the tri-state flip-flop. This is our first glimpse into how memory mapped I/O is actually implemented. From here, it is just a simple matter of mapping the proper memory address to activating the correct control lines.

1.2.5 Tri-State Terminology

As mentioned earlier in this section, microcontrollers (and many other digital logic devices) can be in one of three states, but humans prefer to think of electric things as being on or off. Unfortunately, when it comes to describing the pins of a microcontroller, on and off are insufficient and can lead to confusion. For example, is the I/O cell really "off" if it is actively pulling the I/O pin down to ground?

Remembering a few basic facts can clear up most of the terminological confusion. First, the input buffer is never disabled. Even though the port may be configured in output mode, the input buffer can still be read. Second, the output buffer can either be enabled or disabled depending on the tri-state configuration. To keep some clarity, it is best to use enabled or disabled to describe the output buffer.

The terms "on" and "off" can make sense when describing the behavior of a device that is connected to the microcontroller. For example, consider an indicator lamp that will emit light when current passes through it. If one leg of the lamp is connected to the microcontroller and the other is connected to ground, then the lamp will be on when the output driver connects that port to the voltage supply. Current will flow through the microcontroller, through the lamp, and enter ground to complete the loop. The lamp is "on" when the output is enabled and configured for logic high. The lamp is "off" when the output is enabled and set for logic low or the output buffer is disabled.

On the other hand, the exact same lamp could have one of its legs connected directly to the voltage supply and the other leg connected to the microcontroller. Now, to get current to flow through the lamp, the microcontroller must use its output driver to connect that leg to ground, so the output driver must be enabled but configured for logic low. To turn the lamp off, we can either disable the output driver or write a logic high to connect both legs of lamp to the same voltage supply and no current flows. Without knowing how the lamp is wired to the microcontroller there is no way to tell how to turn it on or off. In fact, using other logic devices, such as inverters, it is possible that the lamp will be on if the output buffer is disabled!

The terminology of "on" and "off" is a little dangerous, and depends entirely on the context in which these terms are used. Always keep that in mind when thinking about how the microcontroller will be used to drive other devices.

1.3 GPIO Electrical Characteristics

The GPIO ports of a microcontroller are said to be digital devices. In the previous section we described how the buffers used "high voltage" and "low voltage" to represent a logical 1 or 0, and we saw how these logical values can be mapped into values that can be read or written through software. But what do we really mean by "high" and "low" voltage? The answer depends on the fact that these complex devices, like buffers, muxes, and flip-flops are all built from transistors and it is the behavior of these transistors that determines the electrical behavior of a microcontroller (see Section **??** for a review of transistor fundamentals).

1.3.1 Mapping Voltages to Digital Logic

A digital logic system maps logical 1 or 0 to whether a transistor is conducting current or not. To keep the discussion focused on digital logic, let us focus only on one type of transistor, the "N-type MOSFET," a three terminal device consisting of a *gate*, *drain*, and *source*. When the voltage, as measured between the gate and the source, is greater than some *threshold voltage*, the MOSFET will be *saturated* and current will flow freely from drain to the source. When the voltage is less than some lower threshold, the MOSFET will be completely off and no current will flow. When the voltage is between these two thresholds then the MOSFET will act like a *voltage controlled resistor*, current will flow from drain to source, but it will be proportional to the gate voltage. Almost no current will flow through the gate. In fact, this is how the high-impedance inputs to buffers are constructed!

Table 1.1: Thresholds for sample microcontrollers

Part	VDD Nominal	V_{IH}	V_{IL}
Cypress CY7C6	$1.7 \ldots 3.6$	> 2	< 0.8
PIC 16F684	$2.0 \ldots 5.5$	$0.8 * \text{VDD} \ldots 5.5$	$0 \ldots 0.15 * \text{VDD}$
PIC32 MX	3.3	$0.8 \ldots 5.5$	$0 \ldots 0.2$
Tiva C TM4C123	3.3	$2.15 \ldots 5.5$	$0 \ldots 1.15$

The high voltage threshold is called V_{IH} (for input high voltage) and the low voltage threshold is V_{IL}. The actual values of these thresholds can vary and are reported in the datasheet for particular part. A sample section of a datasheet is shown in Figure 1.6. Table 1.1 shows different threshold ranges from different vendors. There can be significant difference in the thresholds between microcontrollers. For example, $1.0\,\text{V}$ would be interpreted as a logical 1 in the PIC32, a logical 0 for the Tiva C, and would be in the no man's land in the Cypress chip.

Table 24-6. Recommended GPIO Pad Operating Conditions

Parameter	Parameter Name	Min	Nom	Max	Unit
V_{IH}	GPIO high-level input voltage	0.65 * V_{DD}	-	5.5	V
V_{IL}	GPIO low-level input voltage	0	-	0.35 * V_{DD}	V
V_{HYS}	GPIO input hysteresis	0.2	-	-	V
V_{OH}	GPIO high-level output voltage	2.4	-	-	V
V_{OL}	GPIO low-level output voltage	-	-	0.4	V

Figure 1.6: Example electrical characteristics from the PIC32 datasheet [4].

Schmitt Trigger There is typically a dead band between the high and low voltage thresholds. For the PIC32 in Table **??**, the range $0.2 \ldots 0.8\,\mathrm{V}$ is in the "no man's land" between high and low. A *Schmitt Trigger* is a special type of buffer that uses *hysteresis* to hold its last state until the voltage crosses into one of the defined ranges. If the Schmitt trigger was in its low state, it will output a logic 0 until its input voltage crosses V_{IH}, and it will remain a logic 1 until the input voltage crosses the V_{IL} threshold.

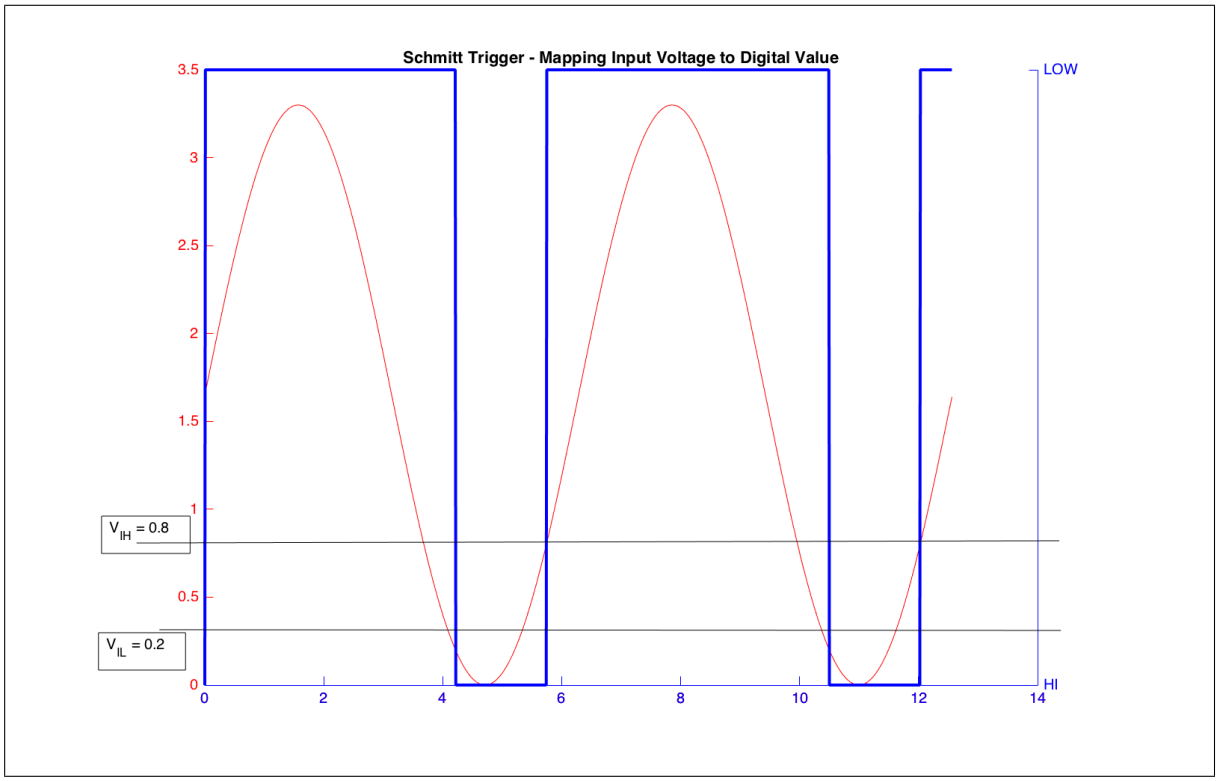

Figure 1.7: Input voltage to Logic Mapping of Schmitt Trigger

For example, if voltage was steadily rising from 0 to 3 volts, the logic level would start low and remain low until the voltage exceeded $0.8\,\mathrm{V}$ [2], and then it would switch to logic high through the rest of the voltage values. If the voltage were then to go from high to low, then we

Listing 1.1: Schmitt Trigger Logical Behavior

```
if (output == LO) {
   if (input > 0.8) output = HI;
   else output = LO;
}
else {  // input = HI
   if (input < 0.2) output = LO;
   else output = HI;
}
```

would start off reading logic high and would continue until the voltage fell below the $0.2\,\mathrm{V}$ value, and then we wold read logic low. This action is shown in Figure 1.7, which shows how a sinusoidal input voltage would be mapped into digital values.

The following code models the behavior of the Schmitt trigger, the actual construction of a Schmitt trigger has nothing to do with C language. Note the use of a state variable that models the hysteresis of the actual device. When the input crosses the logic level then the state changes and the output will reflect this change accordingly.

Burden Voltage One important design goal of the input buffer is that it should not have a significant impact on the circuit it is connected to. It should enable *sampling* the voltage levels on the circuit without measurably changing the voltage or the current on the rest of the circuit. This change is also known as the *burden voltage*. Typically, the input buffer of a MCU is listed as greater than $500\,\mathrm{k}\Omega$ and may effectively be in the mega-Ohms.

The burden voltage can be calculated using information from the device's datasheet and Ohm's Law. In an ideal MOSFET, no current flows through the gate. However, in a real MOSFET, there is some leakage current, typically just a few micro-amps. Given the leakage current and supply voltage, Ohm's Law will relate the equivalent resistance:

$$R = \quad V/I \tag{1.1}$$

$$R = \quad 3.3\,\mathrm{V}/1\,\mu\mathrm{A} \tag{1.2}$$

$$R = 3.3\,\mathrm{M}\Omega \tag{1.3}$$

.

For most circuits, this additional resistance will not cause any problems. However, there are some deranged examples that this can turn into a trap. Figure 1.8 shows three different circuits. In each case, there is a series resistor ($R1$). When the series resistor is small compared to the input port's equivalent resistance then there is little impact to the circuit, which is demonstrated in case (A). However, as the series resistance increases, then the burden volt-

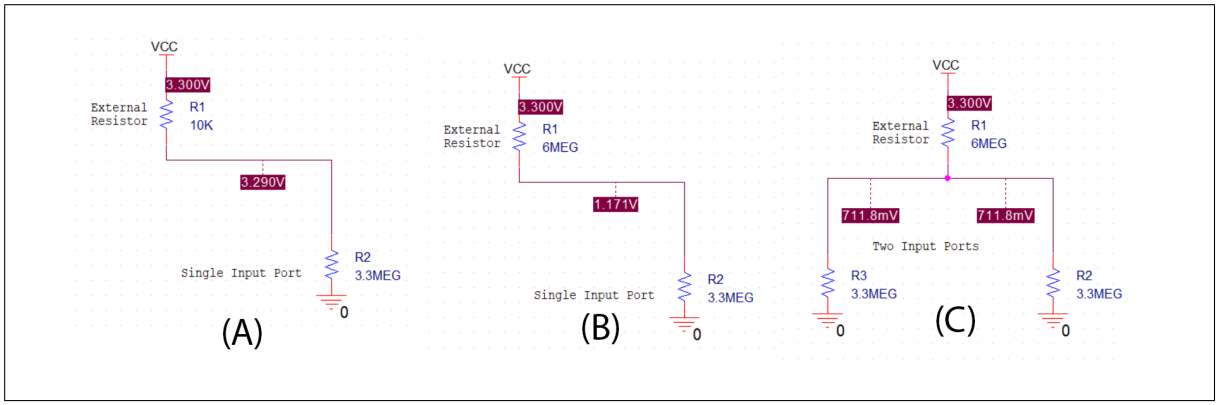

Figure 1.8: Three examples of burden voltage. (A) little impact, (B) noticeable impact, and (C) failure to read signal.

age of the microcontroller can start to have an impact on the circuit, which is demonstrated in (B). Some microcontrollers may read this signal as a logic high, others may not. The situation in (C) is far worse. Here, there are two high impedance inputs, which are equivalent to parallel resistors with the corresponding voltage drop. Again, the performance depends on the first series resistor. If it were smaller then the voltages may be within acceptable ranges for the circuit.

1.3.2 Voltage and Current Limits

In addition to the voltage thresholds described above, microcontrollers typically have strict electrical ratings regarding voltage and current that can pass through their I/O pins. Every microcontroller will have its own characteristics and they will be listed in the device's datasheet. This section will explore some of these important characteristics and what they mean for the way microcontrollers can be used. Table 1.2 shows some key characteristics from a PIC32 microcontroller as an example.

Table 1.2: PIC32MX GPIO Electrical Characteristics

Characteristics	Value
Maximum current out of V_{SS}	$200\,\mathrm{mA}$
Maximum current in on V_{DD}	$200\,\mathrm{mA}$
Maximum current on an I/O pin	$25\,\mathrm{mA}$
Voltage on any digital only pin with respect to V_{SS}	$-0.3 \ldots 5.5\,\mathrm{V}$
Voltage sourced, logic high	$2.4 \ldots 3.3\,\mathrm{V}$
Voltage sourced, logic low	$0 \ldots 0.4\,\mathrm{V}$
Current sink, input port	$\pm 1\,\mu\mathrm{A}$

Maximum Current and Power Microcontrollers are not designed to handle large current loads. Table 1.2 shows that this particular device can handle at most $200\,\mathrm{mA}$ through its supply and ground pins. This is the total for how much current it can provide and how much current it can sink. For example, when an output buffer is connecting the I/O pin to ground, the microcontroller will sink a certain amount of current. There is also a maximum amount of current that can pass through any single I/O pin. In this example, each pin can handle $25\,\mathrm{mA}$ (sourcing voltage or sinking ground). This does not mean that the microcontroller can have eight pins passing $25\,\mathrm{mA}$, as the CPU core and other peripherals also draw current which must pass through either the supply or ground path of the microcontroller.

As a consequence of Table 1.2, to avoid damaging the part, the following rules must always be followed:

- Voltage is never negative (reverse-bias)
- Voltage is never higher than the port can tolerate
- Current on any port is limited to $25\,\mathrm{mA}$, even during *in-rush*
- Total current through all ports (plus the chip) is less than $200\,\mathrm{mA}$

The rest of the section explores methods for protecting the GPIO ports on a microcontroller from damage. Primarily, we are concerned with methods to ensure the proper voltage, and methods to ensure the maximum current requirements are met.

Input Protection

Microcontrollers are connected to a wide variety of electrical components, therefore it is important to protect their I/O cells from damaging voltage and current spikes. Transient Voltage Suppression (TVS) devices can protect a circuit from voltage spikes such as electrostatic discharge, unregulated power supplies, or other short bursts of high voltage by providing an alternate, low-impedance path to ground. There are several methods of constructing a TVS, including: fuses, diode clamps, gas discharge tubes, spark gaps, varistors, and optical isolation.

Fuses The simplest and most common protection device is the humble fuse. Typically made from a sacrificial length of wire, when the voltage or current is greater than the design capacity of the fuse, the wire overheats and permanently breaks. For low voltage applications,

fuses provide safe, cheap, and reasonably effective input protection. Fuses with glass tubes can suffer a catastrophic failure where the glass shatters. For safety critical systems, fuses can be wrapped with a protective coating that will prevent the formation of glass shards. The primary disadvantage with fuses is that they can take relatively long and somewhat unpredictable amount of time to blow. By the time a fuse has blown its possible that extremely sensitive components (like an I/O cell of a microcontroller) have already been damaged.

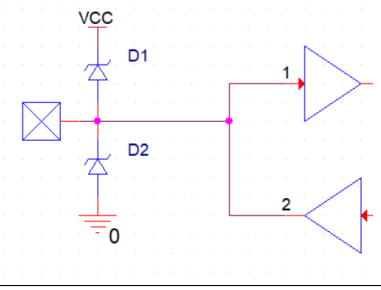

Figure 1.9: Protection diodes on an I/O port

Protection Diodes Figure 1.9 shows a pair of *protection diodes* or *clamping diodes* to protect the internal circuitry from transient voltage spikes. When the voltage on the I/O pin rises more than Vcc, diode D1 will begin conducting, and will dump the current into the global Vcc power supply (which could be dangerous to other devices!). When the voltage on the I/O pin drops sufficiently below ground, diode D2 will begin conducting and dump the current to ground. In either case, the voltage of the I/O pin will not exceed the thresholds set by the protection diodes. Typically, Zener diodes are selected because they can tolerate repeated reverse bias operation better than other diodes types. Internal protection diodes should only be used as a last-ditch effort to protect the I/O pin. A better design is to include external protection diodes and current-limiting series resistors on the I/O pin. Whether they are internal or external, the protection diodes are good choices for low voltage transients, but they will suffer permanent damage and ultimately fail in the presence of larger voltages and currents.

Gas Discharge Tubes Gas Discharge Tubes (GDT) (see Figure 1.10) can protect an I/O cell from large voltage and current spikes. The tube has two electrodes separated by a normally non-conductive gas. Typically, one of the electrodes is connected to part of the circuit to be protected and the other is connected to ground. Normally, no current can flow across the electrodes and the GDT has no effect on the circuit. However, when the voltage spikes high enough to form a plasma between the electrodes, it creates a low impedance path to ground and the voltage surge is dissipated. GDTs can typically handle thousands of volts

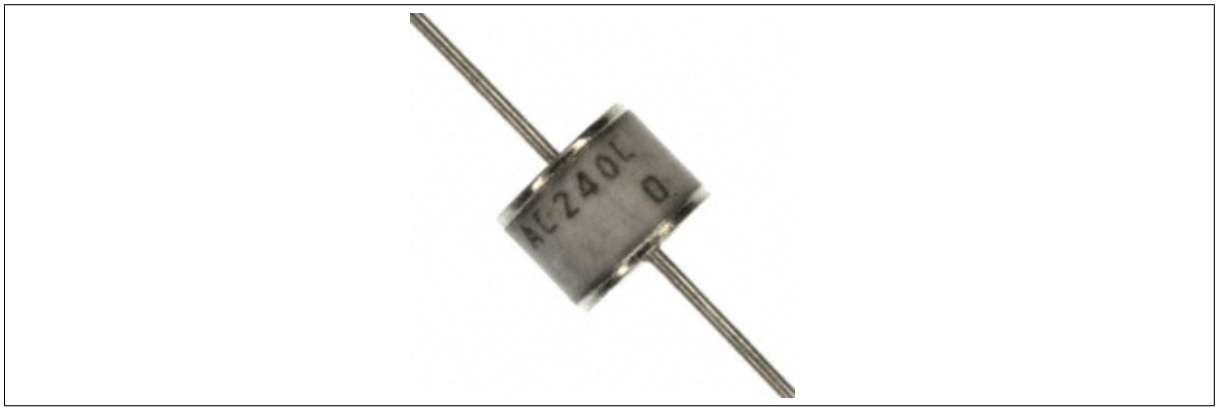

Figure 1.10: Gas discharge tube (Littlefuse)

and hundreds of amps of current, making them ideal lightening suppression. Remember, these devices are designed to handle transient spikes lasting a few microseconds and only happening occasionally. With each strike, the GDT has a chance of failing.

Figure 1.11: Spark gaps in a printed circuit board [**?**]

Spark Gaps Spark gaps (see Figure 1.11) are similar to gas discharge tubes for handling transients. The spark gap uses two contacts separated by a tiny gap in a circuit board. Air has an expected breakdown voltage of $3\,\mathrm{kV\,mm^{-1}}$ [**?**]. When the voltage exceeds the breakdown voltage, a plasma is formed and there is a low impedance path to ground. Typically, these gaps are about $0.2\,\mathrm{mm}$ apart, giving an expected breakdown voltage of $600\,\mathrm{V}$. The plasma formation typically blasts away electrons from the gaps causing them to wear out eventually.

Metal Oxide Varistors Metal Oxide Varistors (MOV) are another type of transient suppression. MOVs, like the clamping diodes, are semiconducting devices. They are constructed with grains of a semiconducting materials. In the presence of an electric field, the boundaries between grains begin conducting. As the voltage increases, exponentially more grains begin conducting and the MOV will shunt away the current. MOVs are extremely fast, dissipating a transient is as little as $10\ldots40\,\mathrm{ns}$. One major drawback to the MOV is that they cannot handle sustained high voltage and will suffer permanent damage as a result. Another major drawback is that through repeated and or long spikes the MOV can fail catastrophically. The

failure modes of a MOV include entering a thermal runaway state where the device can catch fire or set nearby components on fire. Very often, the MOV is used as one part of a protection system.

Optical Isolation One final type of input protection is to use a device called an optical isolator. Although they are typically contained in one package, there are really two separate electrical devices in the package. One side uses an electrical buffer to drive a light source. The other side has a light detector which drives an electrical buffer. There is no electrical connection between the two sides. The light detector can only output a fixed voltage, there is no possibility of a voltage spike or short circuit. On the other hand, the input side to the optical isolator can be damaged by transients, but the damage cannot propagate to the other side.

As a personal anecdote, many years ago I was troubleshooting a communications failure between a central server in our main office building and a remote cluster of terminals in another building. The two buildings were separated by about $500\,\mathrm{m}$ of copper cable that was run through PVC conduit in the ground. The system was installed in October, and it worked fine until early May, and then the communications controllers started failing throughout the late spring and early summer months. Eventually, we noted that whenever there was a thunderstorm anywhere nearby the voltage would travel through the water table which had filled the PVC pipe. A lightening strike, even a few miles away, could knock out our communications gear. One time, a strike actually created a spark inside the server and damaged a disk controller. We installed specially made optical isolators which had beefy input protection, and after that, we never had a problem again.

Designing proper input protection is complex and requires a significant design effort, especially for products that must meet safety standards. It is really vital to think about what impact the protection device will have on the circuit, what types of voltage transients we need to protect against, and importantly, what happens if the transient protection fails? For example, many of the items listed here can fail and simply stop protecting the circuit, which can create major problems to the rest of the system. For many of these reasons, designing and validating proper input protection is an important engineering speciality of its own.

Output Protection

In most cases, safely using a microcontroller as an output device means ensuring that the maximum current is not exceeded. The output voltage is typically driven by the microcon-

troller and will be controlled by the output buffer. However, while the input buffer has a high impedance input, the output buffer is a low impedance output. This means that there is little internal resistance provided by the buffer. If there is not enough resistance the output buffer will simply draw too much current and become damaged.

The electrical characteristics of the microcontroller stipulate the maximum current that will be tolerated per pin and per part. To protect each pin we must simply ensure that we obey Ohm's law. For example, if the pin cannot handle more than $20\,\mathrm{mA}$ at $3.3\,\mathrm{V}$, then use Ohm's Law to find the minimum resistance that can be driven on a pin:

$$I \leq VR \tag{1.4}$$

$$R \geq V/I \tag{1.5}$$

$$\geq 3.3\,\mathrm{V}/20\,\mathrm{mA} \tag{1.6}$$

$$\geq 165\,\Omega \tag{1.7}$$

So, we need to ensure at least $165\,\Omega$ resistance on each leg of the MCU. Of course, as the voltage changes, so does the minimum resistance.

The Output is Willing, but the Input is Weak The output of a buffer has low impedance, meaning it has little to limit its current. On the other hand, the input to a buffer has extremely high impedance. It is perfectly fine to directly connect the output of one buffer to the input of another. For example, two microcontrollers can be connected directly via a wire or circuit board trace. The high impedance input will limit the current from the low impedance output. This can work, until it doesn't. Very high speed devices may need to have carefully designed interconnections, with resistors to absorb reflections, filters to mitigate noise, and other techniques. Generally, signals that change with a frequency less than $100\,\mathrm{MHz}$ can maintain *signal integrity* without these techniques. Designing high speed interconnects and signal integrity is an advanced specialized area of electrical engineering that beyond the scope of this text.

Series Resistance If the circuit only has low impedance devices then there must be an additional series resistor added to limit the current. Devices like bulbs, LEDs, switches, and buttons are all low impedance. A resistor must selected to ensure that the current through every device is below its maximum allowable level. For example, a $3.3\,\mathrm{V}$ microcontroller may be able to handle $20\,\mathrm{mA}$, but if it is driving an LED that can only handle $5\,\mathrm{mA}$, then instead of

the $165\,\Omega$ resistor from Eqn 1.4, we would need a $R >= 3.3\,\mathrm{V}/5\,\mathrm{mA} \geq 660\,\Omega$ resistor.

Managing the current on a single branch of a circuit is generally straightforward. Unfortunately, there is no easy way to ensure that the MCU will not exceed its total current limit. The only way to really ensure this would be to add a fast-blow fuse to the circuit with the hope that it will blow before the MCU is damaged. Even this strategy is fraught will trouble since a transient current load might be acceptable (e.g. during startup) and will unnecessarily blow the fuse. The best engineers will study the behavior of their system and through careful simulation, analysis, testing, and verification will make sure that the device will never be taxed beyond its total current limit.

1.3.3 Microcontroller as Source

When the output buffer is active, the microcontroller either connects a pin to the microcontroller's power supply or to ground. We can use this operation to turn devices on or to turn them off, like a programmable light switch. One key difference between the MCU and the light-switch analogy is that when the MCU is in its "zero state" - it is still actively pulling the pin to ground whereas the switch actually breaks the circuit (very high-impedance!).

Consider the circuit shown in Figure 1.12. The circuit shows a GPIO port `RA0` of a microcontroller connected to the anode of a *diode* Port `RA0` means we are working with bit 0 of bank A of the MCU's GPIO module. The cathode of the diode is connected to a resistor, and the resistor is connected to ground. When the microcontroller port is on, then current flows through the microcontroller, out through the enabled pin, then through the diode, the resistor, and into ground. When the microcontroller port is turned off (still in output mode) then the port is connected to ground, and any current flowing from the anode of the diode would flow into the GPIO pin and finally through the microcontroller to ground. Since there isn't an external supply of current, there would be (virtually) no current flowing out from the anode of the diode.

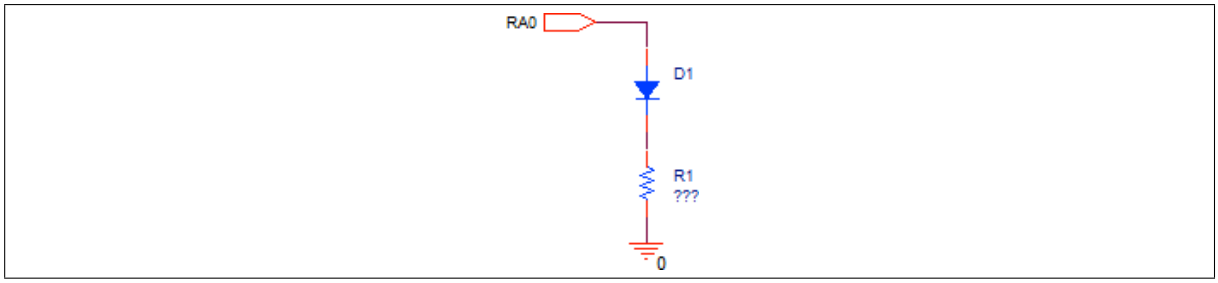

Figure 1.12: LED Circuit

To turn the LED on, we must configure the microcontroller to be in output mode to turn

on the output buffer. The voltage and current must be sufficient to make the LED work, but not greater than the electrical ratings of the microcontroller, the diode, or the resistor. Based on the previous section, for our Microchip PIC32 part, that means we must clear bit 0 of the `TRISA` register and set bit 0 of the `PORTA` or `LATA` registers. Section **??** describes how to clear or set individual bits (Bit Masking) using C language.

```
TRISA = TRISA & ~1;    // use AND mask to clear a port
PORTA = PORTA | 1;     // use OR mask to turn on a port
```

1.3.4 Microcontroller as Sink

Table 1.3: PIC32MX Current Sink Options

	GPIO	Open-Drain

Initialization	`ODCA = 0;`
	`PORTA = 0;`
	`TRISA = 1;`
		`ODCA = 1;`
		`TRISA = 0;`
		`PORTA = 0;`

Sink is Open	`TRISA = 0;` `PORTA = 0;`	`PORTA = 1;`
Sink is Closed	`TRISA = 1;`	`PORTA = 0;`

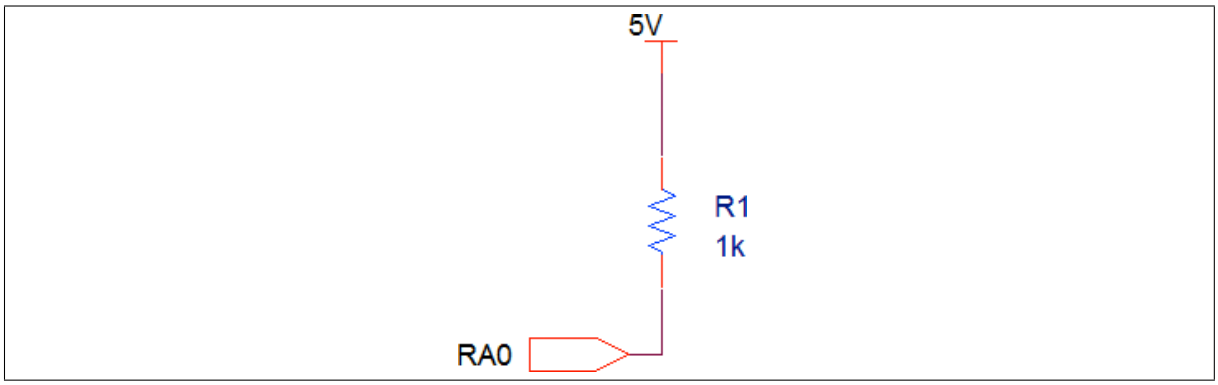

Figure 1.13: GPIO as Sink

The GPIO ports on a microcontroller can also be used as a sink to ground. In this mode of operation, the microcontroller operates as a logic switch, opening and closing to complete the path to ground of an externally powered switch. As mentioned before, the I/O port can handle higher voltage devices as a sink than as a source. Figure 1.13 shows a circuit where the ground connection is through the sink created by the microcontroller. Only when the microcontroller is sinking to ground will current pass through the resistive load.

In our PIC32 example, this makes use of the *output buffer* (see Figure 1.1) to create a path

to the microcontroller's ground. The sink cannot not use the input buffer, whose input port is always high-impedance. Instead, the MCU uses the output port to connect the GPIO port to ground. The microcontroller can create a sink using either GPIO or an Open-Drain configuration.

Let us call the state of current passing through the resistive load of Figure 1.13 the "on" state, and current being blocked the "off" state. Using this terminology, when the load should be on, we should be sinking current, and when the load is off, we should be disconnected from the circuit. The mapping of on and off states in this problem is because we are using the GPIO as a sink. Our on and off state mapping in the previous diode example (Figure 1.12) would map "on" to sourcing current and "off" to be sinking current. The mapping has less to do with the GPIO port than the external circuit.

As Table 1.3 shows, in GPIO mode, toggling between the "on" and "off" state in this example is accomplished by keeping the `PORTx` register set to 0, and toggling the `TRISx` register. This is very different than when using the port as a source where the state of the `PORTx` register determined on and off. Now consider the Open-Drain mode where toggling on and off means toggling the `PORTx` register (just like sourcing current).

In fact, there is no difference in functionality between using the open-drain mode and and setting up the registers as they were described here. The advantage to the ODC mode is that instead of toggling both `PORTAx` (or `LATx`) and `TRISx`, the ODC controller will do it for us.

1.3.5 Microcontroller as Input

The GPIO port of the microcontroller can also be used for input. In this mode, the output port is disabled and input buffer is connected to the port. This was described more full in Section **??**, but based on the voltage present on the pin the input buffer will map that voltage to either logic high or logic low. When the CPU executes a load instruction, those logical mappings are returned on the data-bus.

The tri-state control register is used to set the status of the output buffer. In the PIC32, the output buffers are controlled by the *tri-state* register. Setting the corresponding `TRISx` bit to a 1 enables input (*think 1 = i*) and simultaneously disables the output buffer.

After the microcontroller is setup for input mode, the values can be read through a load-word instruction on the *special function register* corresponding to the I/O port. For example, in the PIC32 instruction set, reading from one of the `PORTx` registers yields a 16-bit value, where bit n corresponds to port `Rx`n. The value can be copied into a General-Purpose register or

Listing 1.2: Simple Input Example

```
void maybe_do_something_cool( )
{
  // enable output on RA4
  TRISA = 0b1;

  if (PORTA & 0x01) {
    do_something_cool( );
  }
}
```

variable, and treated like any other value.

Listing 1.2 shows a very simple example of using GPIO input. The `TRISA` special-function register has `bit 0` set to `1`, indicating it is now in input mode. The `if` statement reads the value of the `PORTA` register. Because the register returns all of the bits in that port, the code uses an *AND-mask* to turn off all other bits except `bit 0`. If that bit is true, the `if` statement is taken, and we call the `do_something_cool()` function.

1.3.6 Driving Other Chips

One common use for microcontrollers is to use their GPIO lines to drive other chips. As a hold-over from the discrete logic days, devices often operate with *transistor-to-transistor level* (TTL) logic. A TTL input is expected to be high-impedance. A TTL output device is expected to be greater than a logic-high threshold or below a logic-low threshold. Older TTL levels used 1.7 V for a minimum logic-high threshold, and 0.7 as a maximum logic-low threshold.

For example, consider a larger system with a power-regulator, powerful CPU, and an MCU. The power-regulator has an *enable* line that will turn the power on to the rest of the system. The CPU has a *reset* line that will cause it to reboot. A reset/power switch can be connected to a microcontroller. If the power button is pressed the microcontroller will raise the *reset* line to reset the CPU and hold it 100 milliseconds and then release it (even if the button is still pressed). This will cause the CPU to reset. If the button is pressed and held for more than 3 seconds, the microcontroller will then turn on (or off) the machine by changing the enable status to the power manager.

This is a common use for the microcontroller: interface to some low-level I/O device and then drive logic to control other parts of a larger system. In fact, we could construct the same behavior using a collection of older *Jelly-Bean Logic* chips, but it would be a complex circuit design. Jelly-Bean Logic is relatively easy for *Combinational Logic*, where the output is determined by the input alone. But this circuit has a timing requirement - the same button is being used to either reset the processor or turn on/off the circuit. The MCU is ideally suited for this purpose, the code running on the MCU can determine which mode the user wants

and perform this purpose.

One concept that should not be missed here is that the MCU's input buffers can be connected to any voltage source, like a switch, or it could be connected to another TTL logic device's output. Its output could be connected to a humble LED, or it could be connected to another TTL logic device's input. The microcontroller itself is unaware of what it is connected to. The way that we program the MCU can ignore the type of device that it is connected to. But what matters is what the program actually does. This chapter focuses on simple I/O programming, but later we will rely on these input/output drivers to connect to other chips and perform advanced communications. So, this is this GPIO foundation is the base for the rest of our work with microcontrollers.

1.3.7 Troubleshooting

When working with GPIO, there are several common problems that can manifest themselves as either hardware or software problems. Debugging requires not only debugging the software but making sure the hardware is functioning properly as well. There are many different problems that can be encountered along the way. This list is a start, it isn't meant to be exhaustive. The key to debugging microcontroller applications is to make every attempt to isolate the problem as either hardware or software and use that to focus the debugging process – but never get "mission blindness" and fail to be willing to rethink where the bug occurs.

Sudden Reset The primary symptom of this problem is that enabling an I/O line causes the microcontroller to reset. For example pressing a button (that isn't intentionally a reset button!) makes the system reset. Another version of this is that while running code, enabling an output line causes the system to reset. The most common reason for this is a short circuit that is established only through the input or output device. For example, pressing a button causes a short through the switch which overloads the power-supply and causes it to *sag* and drop out.

The fix is to use a *continuity meter* or *continuity* function on a *multimeter*, and while testing the branches of a circuit involved at the time of the short, look for the short and add current limiting resistors as needed. If the problem is the microcontroller, make sure that the initialization is completed in the proper order - such that the microcontroller stays in Hi-Z mode until the output port is configured and ready.

Stuck Bit This problem shows up as a port that always reads as a one or a zero. This problem may be related to an electrical problem where the I/O port is always connected to part of the circuit that is directly connected to power or ground. If the hardware is tested and otherwise functioning properly, the problem is probably in software. If the software is misconfigured, the microcontroller may not have disabled the output buffer and the output buffer is driving the input value. Make sure the `TRISx` register is setup correctly. There is a chance that previous damage to the MCU has damaged the input buffer. Generally, damage to the input buffer will probably damage more than just that one buffer.

Random Input Values Most microcontrollers are very tolerant of a wide range of input voltages, but there are limits. When the voltage at the pin is between the `HI` and `LO` ranges, it is up to the trigger values to determine logic high and low, and small changes can "flip the bit". This particular problem can be difficult to track down. One common cause is that a branch of the circuit that is connected to the I/O pin gets connected to another path to ground, creating a voltage divider. With the I/O voltages already being small, even a small voltage divider can drop the voltage below the upper range. Check for the resistances between the power supply and ground along the branch of the circuit, checking for unintentional voltage dividers along the path. This problem is seldom caused by software.

Low Output The microcontroller isn't raising the output voltage to its advertised limit. Similar to the previous problem, there is most likely a voltage divider on the output circuit. Again, the problem is probably not hardware, however it is insufficient to rule out software. Microcontrollers are fast enough today that they can oscillate faster than a multimeter can properly measure. The side effect is that the voltage gets measured incorrectly. To rule this out, a high-speed *oscilloscope* can be used to inspect the wave form on the output line to ensure it is not oscillating, and that the output voltage is consistent. The low-output problem can actually be a design feature for controlling the brightness of LEDs - by pulsing the output line the LED appears dimmer in proportion to the frequency of its on to off time.

Lack of I/O Response Most microcontrollers are designed with some separation between I/O banks. If one bank of a microcontroller is damaged, it is possible that the rest of the microcontroller *may* be partially operational. Generally, the symptom is that the effected microcontroller will not work with any program, while the program will work on another microcontroller. This happens especially when prototyping, and the microcontroller is frequently

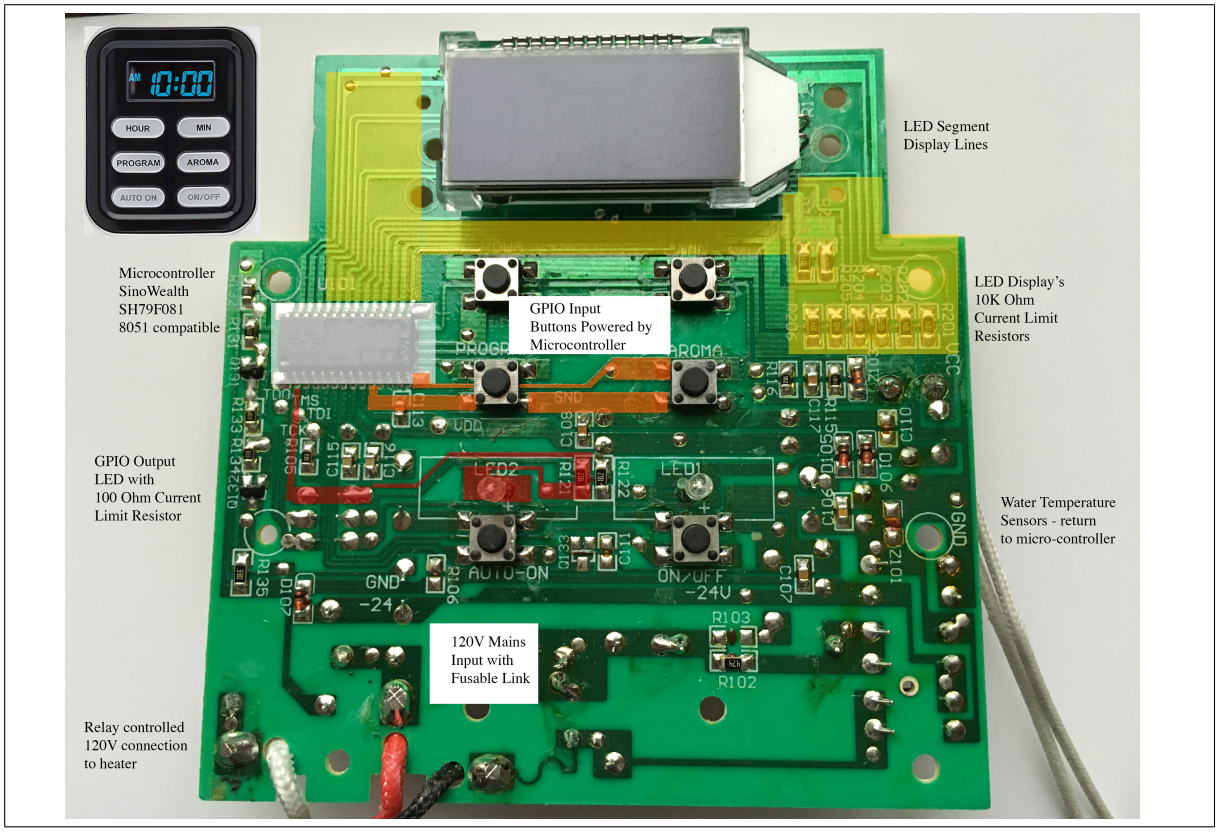

Figure 1.14: Coffee maker controller.

exposed to improper voltages or currents, and ultimately the part becomes damaged. After fixing the circuit, discard the chip and replace with a new one.

1.3.8 Real World Example : Coffee Maker

Programmable Coffee Makers have become common household appliances. Using a real-time clock, they start brewing at a pre-determined time so you can enjoy your morning coffee without having to wait. Like other electric coffee makers, they contain a heating element to boil the water and keep the carafe hot. Modern units contain a microcontroller to manage the start time, monitor and adjust the brewing temperature and strength, and will even remind you when its time to clean the unit.

Figure 1.14 shows the controller board from a popular brand of coffee maker. The controller gets its power from household mains AC (in the USA it is 120 V). This is needed for the 900 W heating element, but is also used to power the microcontroller. The AC to DC conversion happens using a *diode bridge rectifier*, a current limiting resistor, and a Zener diode to clamp the voltage to 5 V (all of this is on the other side of the circuit board).

The board is controlled by an 8051-compatible Sino Wealth SH79F081 microcontroller. Al-

though Sino Wealth is not one of the common manufacturer names, the 8051 architecture certainly is. The microcontroller has connections to the LED display, buttons, a relay, and the temperature sensor. Relays and temperature sensors will be covered in later chapters.

The LED controller is highlighted in yellow. We can see each of the GPIO lines on the PCB connecting the microcontroller to the LED module, and then we can see the series resistors used to limit the current through each of the LEDs. This style of hook-up is typical of the LED schematic described previously in the chapter, there are just several of them.

The push-button switches are highlighted in orange. The MCU provides power and ground to the switch (a slightly different arrangement than described above). Because the MCU provides both power and ground we eliminate the need for pull-ups or pull-downs. In addition, we see a surface mount capacitor, labeled `C13`, parallel to the path from the switch. This capacitor is used for *debouncing* the switch.

Finally, we see an LED indicator highlighted in red. The MCU uses the GPIO line to drive the LED. A series $100\,\Omega$ resistor limits the current through the LED. This is almost exactly like the LED circuit previously described. If you pay careful attention to the LED line at the MCU, we see that the LED line is shared with `TMS`, one of the *JTAG* signals. Although this board does not include a JTAG connector, there are *test-points* that would allow an engineer to solder wires to the JTAG connections and hook up an emulator to debug a component. The problem is that the series resistor would conflict with the JTAG emulator. As a result, the PCB layout includes small *jumper wires* which are soldered on the other side of the board. To debug the board, we would de-solder the jumper wires (or cut them), and to restore the LED operation we would re-install jumper wires.

While the actual printed circuit board (PCB) is a little more complicated than our previous examples, some elements should look familiar. Already we see how the MCU interfaces to its buttons, LEDs, and displays. The picture of the actual coffee maker shows the control pad as the user would normally see it. After stripping away the button caps and display bezel, we now see that the PCB is nothing more than the devices and the wiring between them. With some experience, almost any device can subjected to this type of *post mortem* examination, and just about every device will have some variation on these basic device connections.

1.4 GPIO Programming

So far, we have examined the construction of an I/O cell, and its electrical characteristics. This section shifts our attention away from the physical world to the programming environment and explores how to perform basic I/O operations on a GPIO port. This section introduces several important concepts, including memory mapped I/O, I/O banks, and bit masks.

1.4.1 Memory Mapped I/O

Input and output are always performed from the perspective of the central processing unit (CPU). Input to the CPU means reading data from a device, and output from the CPU means writing data to a device. Historically, there are two mechanisms for initiating communication between the CPU and the device: special instructions or memory mapped I/O.

Early computer systems, such as the Intel 8051 and even Intel's early microprocessors such as the 8086, used special instructions to access peripherals connected to the CPU. Intel included `IN` and `OUT` instructions to interface to an I/O peripheral. The disadvantage to this is that Assembly language routines must be written to initiate these instructions requiring code to be just a little less generalized and the control unit to be a little more complex.

The alternative is to use the standard memory loads and stores but direct them to the I/O peripheral instead of just the system's RAM. In a memory mapped I/O system every peripheral is given an *address range*. When a memory access for that range is detected, the device that has been assigned to that range will handle the request instead of the memory. In this way, I/O becomes a natural extension of the memory system of the processor.

Of course, this means that every peripheral is assigned some part of the system's memory address space. It also means that the peripheral must transfer a word of data at a time. Transferring a single bit requires transferring an entire word of data, typically 32-bits. For this reason, the individual I/O cells are grouped together into *banks*, typically 16-or 32-bits wide.

The GPIO port shown in Figure 1.1 was a single GPIO cell. It is one bit wide, and is connected to a single I/O port. Even early microcontrollers had more GPIO lines than could be referenced by a signal register. Modern devices such as the TI's TivaC and Microchip's PIC32 can have more than 100 GPIO lines. Consequently, the GPIO lines are grouped into *banks* of 16 or 32 lines each. Each bank is identified by a name, such as `PORTA`, `PORTB`, ..., `PORTn`. Using this name, all of the pins in the bank can be read or written as a group.

Listing 1.3 shows a C-like example of a GPIO peripheral controller. The three arguments to the function represents the memory bus, with its `address`, `data` (which can be either read

Listing 1.3: GPIO Peripheral Functional Model

```
gpio_port_t ports[8];

int gpio_peripheral(int sysclk, unsigned int address, unsigned int *data, unsigned int read_not_write)
{
   if ((address == GPIO_ADDRESS) && (read_not_write == READ)) {
       for (int i = 0; i < 8; i++)
       ports[i].read_port = 1;

     wait_next_clock( sysclk );

     *data = 0;
     for (int i = 0; i < 8; i++) {
         *data = *data | ports[i].data_bus << i;
        ports[i].read_port = 0;
    }
   }
}
```

or written by the function, so it is a pointer here), and `read_not_write` to select whether the data should be read or write. Of course, the peripheral controller (probably) isn't written in C language, this code snippet at least shows how the decisions would be made. This code could be triggered on each edge of a clock pulse. If the current memory address is a match for the specific GPIO peripheral the control lines that are selected to be read or turned on. After a clock cycle delay, each of the I/O cells that were selected for read are concatenated into a single word that be given back to the CPU for a load word value.

Each of these I/O banks are accessed using a unique memory address. For example, `PORTA` could be assigned a memory address (in hardware) as `0x1000_0000` and `PORTB` is assigned an address of `0x1000_0004`. If we were to read from that memory address the results would be from the I/O cells' flip-flops (as opposed to the values stored in the chip's memory). Writing to that memory address would store change the values in the I/O cells' flip-flops.

1.4.2 C Language Mapping

In C language, to access a specific memory address we use a pointer. In this case, we need a pointer as wide as the GPIO bank, which is typically an integer. The microcontroller's vendor will provide a mapping of meaningful names to these pointers. For example, the vendor will provide a standard header file with declarations such as:

```
#define PORTA (*(unsigned int *)0x10000000)
#define PORTB (*(unsigned int *)0x10000004)
```

The expansion of the `PORTA` and `PORTB` symbols is interesting. The hardware address is located at a known physical memory address. The address needs to be typecast to a pointer type to force C to treat it as a memory address as opposed to a literal integer. However, since it always just a pointer to a single integer, there is a *dereference* operator in front of each pointer.

Whenever the symbol `PORTA` the C pre-processor will translate this to be a load or store word instruction to the CPU.

Then, these I/O ports can be accessed as if they were variables. For example:

```
PORTA = 0x8001;
```

would be converted to a store word of a immediate value to the memory address associated with `PORTA`.

Since this is writing a value to an I/O bank, this will simultaneously turn on the 15th and 0th bits, and turn off all of the other bits in the same bank. Likewise, individual pins can be read or written using *masking* techniques (see Appendix **??**, Section **??**). For example:

```
1 PORTF = PORTF | (1 << 15) | (1 << 0);
```

uses the bit shifts to turn on the 15th and 0th bits of the `PORTF` register.

1.4.3 Port Configuration

There are a number of control signals shown along the left-hand side of Figure 1.1, such as `SYSCLK`, `Data Bus`, `RD PORTx`, and `WR ODCx`. These control signals are set by the GPIO *peripheral controller*, a device that is not shown. When it detects a load or store to one of its memory mapped registers it will set or clear the appropriate control lines to handle the request. For example, when the GPIO bank A peripheral detects a *load* from its `PORTA` memory address, then it activates the `RD PORTx` control lines for all eight of its GPIO ports. It connects each of the single-bit `Data Bus` lines to the CPU's 32-bit data-bus and signals to the CPU that the value is ready.

In this way, memory-mapped I/O allows for straightforward integration of the various peripherals in the computer system with the software running on the microcontroller. So, if we start with a C language statement like:

```
TRISA = 0x3;
```

The compiler will turn that C code into the following MIPS assembly which loads the address of TRISA, loads the immediate value and then executes a store-word instruction:

```
1   LA \$t0, TRISA
    LI \$t1, 0x03
3   SW \$t1, 0(\$t0)
```

Table 1.4: Tri-State Buffer Input to Output Mapping

TriState	Port	Direction	Output
0	0	Output	0
0	1	Output	1
1	0	Input	Z
1	1	Input	Z

1.4.4 Setting Port Direction

Section 1.2 introduced the organization of a typical GPIO port which includes two buffers, one for input and one for output. The input buffer, which is always enabled, is always in high impedance mode. The output buffer can either be disabled, leaving the I/O cell in high impedance mode only, or it can be enabled and then connect its output to the supply voltage (logic high) or connect its output to ground (logic low). Because of this, each I/O cell is said to be a *tri-state buffer*. Table 1.4 shows the truth table for a typical GPIO port.

Consequently, a GPIO port's direction must be set before they can be used for input or output. In some applications, the directionality of the GPIO port is constant. In this case, it can be set as part of the overall system initialization. In other applications, the directionality may change based on the behavior of the rest of the system. Either way, it is important to ensure that the direction is set in the tri-state register before reading or writing the port register.

1.4.5 Reading from the GPIO Module

Reading from a GPIO port require setting the tri-state register to the proper direction and then reading from the proper port register. The following example illustrates how to read from a switch. Figure 1.15 shows the schematic of a microcontroller connected to a switch. When the switch is closed, current can flow through the resistor and switch and there will be $3.3\,\mathrm{V}$ on the port. When the switch is closed, no current flows, and there will be $0\,\mathrm{V}$ on the port. The Schmitt triggers will capture the change voltage and store it in the flip-flop latches.

Listing 1.4 shows how to initialize the GPIO port for input, and then how to read the GPIO port. The code will spin in an infinite while loop and will turn on an LED when the switch is closed and turn of the LED when the switch is open.

The C compiler would compile the code in Listing 1.4 into the Assembly code shown in Listing 1.5. Note how reading from the `PORTA` peripheral is compiled into a simple `LW` instruction, just like any other memory address.

To keep the design of the CPU consistent, regular, and simple; when the CPU encounters

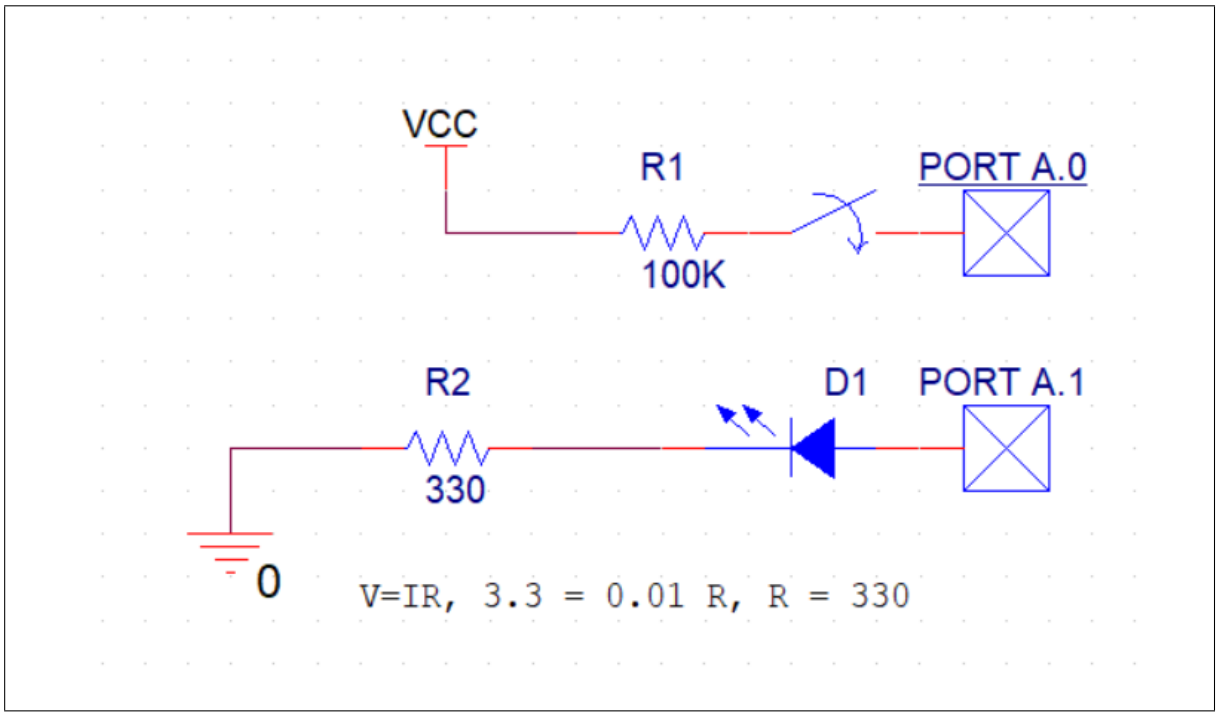

Figure 1.15: A simple switch and an LED circuit

Listing 1.4: Wait for a button press

```
void setup_bank_a( )
{
   // port A1 is input, A0 is output
   TRISA = (1 << 1) | (0 << 0);
}

int switch_is_pressed( )
{
  return PORTA & 0x01;
}

int main( )
{
 setup_bank_a( );
 while (1) {
  if (switch_is_pressed())
     turn_on_led( );
  else
     turn_off_led( );
 }
}
```

Listing 1.5: Assembled button press code

```
while_loop:  LA     $t0, PORTA          # load address of PORTA
             LW     $t1, 0($t0)         # load the value at PORTA
             ANDI   $t1, $t1, 1         # only look at bit 0 of PORTA
             BZ     $t1, while_loop     # if its zero, loop
```

the `LW` instruction it passes the address on to the memory controller. The memory controller steers the request to the proper device, but only as a normal memory read or write. When the peripheral detects the read or write it translates the memory request into its own internal implementation - in this case, reading from the ports. Ultimately, we get the 8-bit byte from the peripheral put onto the memory bus, the CPU's pipeline handles the result from the memory bus by storing the 32-bit word into `$T1` register, and the program continues as normal.

We have successfully traced the entire read operation - from C code, to Assembly, to CPU execution, to memory bus steering, to the peripheral module setting the controls on the ports, to the Schmitt triggers mapping the voltage of the switch into a logic-high or logic-low. This is the basis for GPIO, and in fact, is the basis for almost all other types of input to any computer system.

Registers versus Memory All the peripheral devices have addresses in the address space, and we communicate with them through the memory bus, and they may even store values, they are not memory. But they aren't CPU registers either. To help differentiate between memory and CPU registers, these peripheral devices are often referred to as *Special Function Registers* (SFR). Most embedded tool-kits give these SFR's special treatment. In Microchip's PIC eco-system, the SFRs are available as variables to C and Assembly coders. The debugging tools allow users to view the values of the SFRs, and will even interpret the configuration of the peripherals from the values of the SFRs.

1.4.6 Output

The GPIO port's output section, shown in Figure 1.16, has an *output tri-state buffer* (B), and *output latch* (F), *tri-state control* (G), and *open-drain control* (H) flip-flops. Together with the output control lines, these manage the output operation of the GPIO module.

Output Tri-State Buffer

Recall from Section **??**, a buffer's input is a high-impedance input, and the output could be one of three states: sinking ground (logic 0), sourcing current (logic 1), or high-impedance (Z). There are times that we want the GPIO port to be in one of the three modes. Since the input to a buffer can only ever be high-impedance, we cannot do this with just a single tri-state buffer. We will add another tri-state buffer (B) to handle the output modes for the GPIO port.

While the input buffer had its input connected to the I/O pin, the output buffer has its

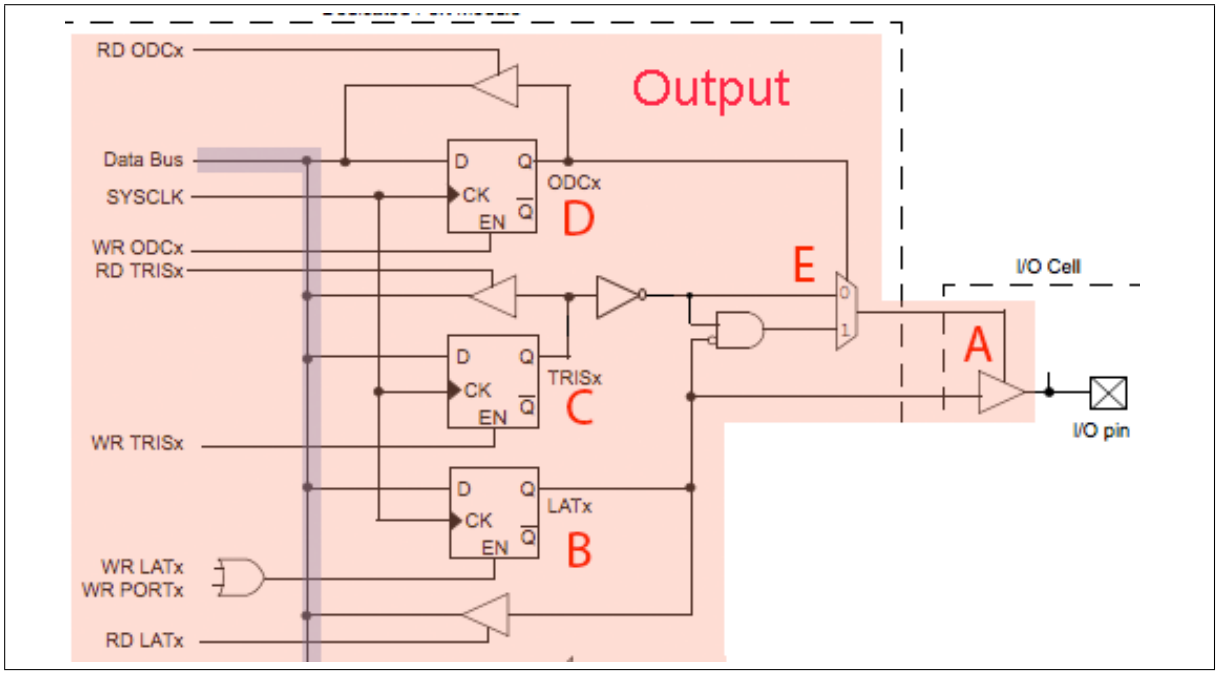

Figure 1.16: Output Section of GPIO Port

output connected to the I/O pin. When the GPIO port is configured for input then the output buffer should be disabled (its output is in the high-Z mode) so that it does not interfere with the operation of the input buffer.

The output buffer (B) is controlled by the output of a multiplexor. This multiplexor is part of the *Open-Drain Control* that will be discussed in Section 1.4.6. The only time that the multiplexor would not select the `TRISx` control line is when the Open-Drain mode is used, so we can safely ignore it for now.

The source of the `TRISx` line is another D-Flip-Flop Latch (G) whose output goes through an *inverter*, a buffer that always drives the opposite of its input. When the `TRISx` latch stores the value 0, the inverter flips it to a 1, and the output buffer is turned on. When the latch stores the value 1, then the inverter flips it to a 0, and the output buffer is turned off.

`WR TRISx` and `RD TRISx` Control Signals The `TRISx` flip-flop latch is shown having a control line connected to an enablement signal `EN` on the Flip-Flop. When the GPIO peripheral determines that we are writing (store-word instruction) to this flip-flop, it will put the value we are writing on the data bus and enable the `WR TRISx` control signal. After one clock cycle, that will cause the flip-flop to latch-up and store the new `TRISx` value.

The peripheral module will use a *Special Function Register* for the `TRISx` control signals, and unsurprisingly, it is called `TRISx`. Just like we had `PORTA` and `PORTE`, we have `TRISA` and `TRISE`. Bits 0 through n of the `TRISx` register corresponds to the first through the last

Listing 1.6: Toggle RA5

```
// toggle port RA5
void toggle( )
{
  TRISA = TRISA & ~0x04;  // set port RA5 as output
  int v = PORTA & 0x04;   // get old value
  v = (~v & 0x04);    // flip 5th bit

  // and mask out old value, or in new
  PORTA = (PORTA & ~0x04) | v;
}
```

ports in the bank. Like the PORTx SFR, the TRISx SFR will have its own memory address that corresponds to its bank and that will be recognized by the bank's peripheral controller. Reading or writing to this SFR will drive the appropriate control lines.

The use of a flip-flop here allows the last value that was written to be remembered by the GPIO module, the flip-flop is essentially 1-bit memory cell. The value stored in the latch will remain until the peripheral asserts one of the corresponding enable lines and the new value is latched up by the flip-flop. The flip-flop is a piece of hardware, its not software, and so as hardware, it never stops being hardware, and it never stops driving its output. So, the microcontroller can write a value to the flip-flop, it will be stored, and then constantly driven until the next time the flip-flop is changed. This allows the microcontroller to set the state of the port and then go on to run other code. The flip-flop will continue to drive its output lines, just like if you walked over to a light-switch and turned the lights on. You don't have to stand there and hold the switch to keep the lights on, the switch will do that for you.

When the GPIO peripheral is reading from this flip-flop the peripheral will assert the RD TRISx control signal, which will enable the tri-state buffer that connects the TRISx flip-flop's output to the shared data-bus. The GPIO peripheral can then use this to allow a program to read the value of the TRISx flip-flop.

The diagram shows that the TRISx flipflop Q signal will go through the inverter to the output buffer as an enable signal. The TRISx special function register uses a 0 as the value to indicate output, and a 1 to indicate input. As a mnemonic, think 0 looks like an Oh for output, and 1 looks like an I for input.

Using the TRISx values we can select the mode (input or output) of the GPIO port. Most programs modify both TRISx and PORTx to ensure they are configured correctly for the particular application. Listing 1.6 shows a sample of a C program that toggles the 5th bit of PORTA.

Output Latch

The *output latch*, `LATx`, stored in a D-Flip-Flop (labeled F in Figure 1.16) holds the value that will be conditionally driven to the output buffer. We can follow the line coming connect to the output (Q) of the flip-flop and trace it directly to the output tri-state buffer (B). Just like the `TRISx` flip-flop, when a value is written to this flip-flop it will be remembered until its changed by the MCU.

One difference from the `TRISx` FF is that instead of a signal write-control line, there are two write control signals, `WR LATx` and `WR PORTx` from the GPIO peripheral. Both of these signals enter an OR gate, and if either one is logic high, then the output of the OR will be high, and then the flip-flop will update its data data to whatever value is being driven onto the data bus. Because these control signals are OR'ed together, then we can write to either the `LATx` or `PORTx` SFR's and the peripheral will update the `LATx` register. So, why have these two different signals that both write to the same latch?

The answer is that there are also two different signals for reading - `RD LATx` shown in the output section of Figure 1.16 and `RD PORTx` shown in the input section of Figure **??**. Reading from the `PORTx` SFR will activate reading directly from the input port flip-flops - even if the port is in output mode! However, reading from the `LATx` SFR will always return what was previous written to the `LATx` register. So, the `PORTx` will return what is presently on the input port, while the `LATx` will return what was previously written to the flip-flop.

Open Drain Configuration

The *Open Drain Configuration* (ODC) is a special output mode supported by some microcontrollers. The term comes from a MOSFET, which has a gate, drain, and source. In a simplified model of an N-type FET, when the voltage between the gate and the source (V_{gs}) exceeds the devices threshold (typically $2.5\,\mathrm{V}$), then the FET will saturate and start conducting from the drain to the source. If the voltage is less than the threshold, the FET is closed and will not conduct from drain to source.

As the name suggests, the drain of an internal FET is not connected internally, but is connected to the I/O pin. When the internal FET starts conducting, then it opens a path to the source (usually ground) through the open drain, and then the internal FET is not conducting, the open drain is effectively a high-impedance point.

Open-Drain Configuration is used to allow a microcontroller to operate with devices that require a higher voltage than the MCU can source. For example, a special LED requires $5\,\mathrm{V}$ to

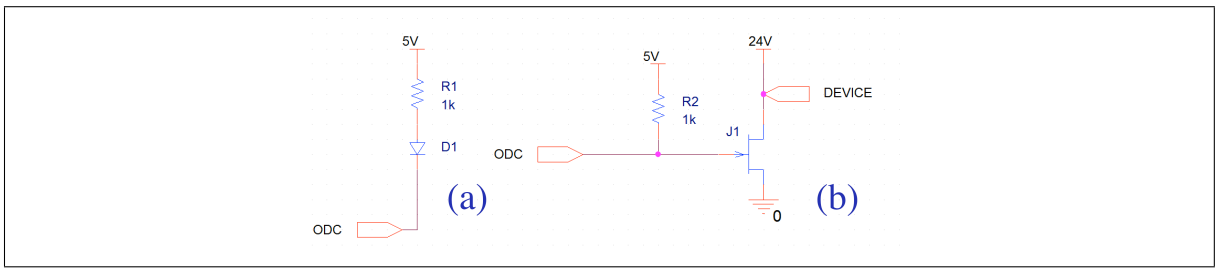

Figure 1.17: Open-Drain Configuration. (a) shows an ODC controlling a 5 V device, while (b) shows an ODC controlling a 5 V FET which controls a 24 V device.

Table 1.5: Open-Drain Configuration Possible Values

		Input		Output	
TRISx	LATx	Not TRISx	Not LATx	Enable	Value
0	0	1	1	1	Z
0	1	1	0	0	0
1	0	0	1	0	Z
1	1	0	0	0	Z

turn on, but the MCU can only drive 3.3 V. Figure 1.17 shows two circuit paths that demonstrate using the ODC port. This mode is especially handy for dealing with high-powered MOSFETs that typically have a higher turn-on voltage threshold.

The way this is implemented is that the output driver is togged between input and output modes depending on the value that is written to the output latch. If the port is configured for output (`TRISx = 0`) and open drain mode (`ODCx = 1`), then the output port is either sinking to ground or is disabled (high-Z).

The AND gate (I in Figure 1.16) takes the boolean AND of `TRISx` latch and the negation of the `LATx` latch, and that is passed into the multiplexor that controls the output buffer. Table 1.5 shows the possible values for the ODC port.

The ODC module is really just a convenience. By configuring ODC, when the program writes a 0 to the `LATx` latch, the port sinks to ground as normal (this is due to the negation of the input and the AND gate). However, when the program writes a 1 to the `LATx` latch, the negation flips it to zero, the AND gate outputs a zero, and the output buffer is disabled, and we are back to high-Z mode. This allows a program to configure a port for output using ODC and then write 0's and 1's to turn the port off and on as normal.

The same behavior can be realized using `PORTx` and `TRISx` registers (and will be explored as a homework question!).

1.5 GPIO Programming

The focus of this chapter so far has been on the GPIO hardware. This section introduces several techniques for developing applications that use GPIO modules. GPIO programming is the principle starting point for any microcontroller development. In fact, for most programmers, the first thing they do is write a 'hello world' program. The microcontroller equivalent to 'hello world' is the 'hello LED' code, where we get an LED to blink. From there, the section describes several more advanced techniques.

1.5.1 Combinational Logic

One of the simplest microcontroller applications is to implement combinational logic. Combinational logic describes a logical function whose output is determined only by the current input, with no memory of past input. Combinational logic circuits can be built using only Jelly-Bean Logic. Most importantly, because they don't involve any memory or clock, they are completely *reactive* to their input. This section compares implementations using Jelly-Bean gates and a microcontroller; and will look at when it is appropriate to use one over the other.

Logic Gate Version

Figure 1.18 shows a simple combinational logic circuit built using logic gates. There are three inputs: *Run* is connected to a switch that is logic high when the machine should be running, *MOTOR_SLOW* is driven by a speed sensor, so that when the motor is running too slow, it will be logic high, and when the motor is running above, it will be low, and finally a *HOOD_OPEN* signal that is connected to a switch, if the safety hood is raised up, then this signal is high, and is low otherwise. There are also two outputs, *MOTOR_POWER* will be high when the motor should get power, and *ALARM* which will be high if an alarm should sound. For safety reasons, there should be at most a $1\,\mathrm{ms}$ delay from when the hood opens and when the power

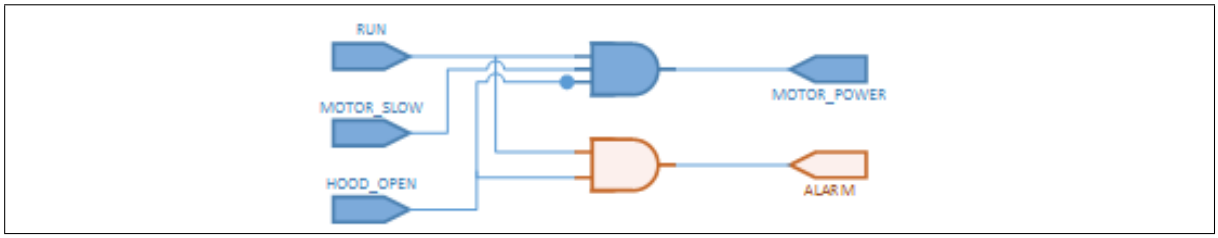

Figure 1.18: An alarm made up of combinational logic

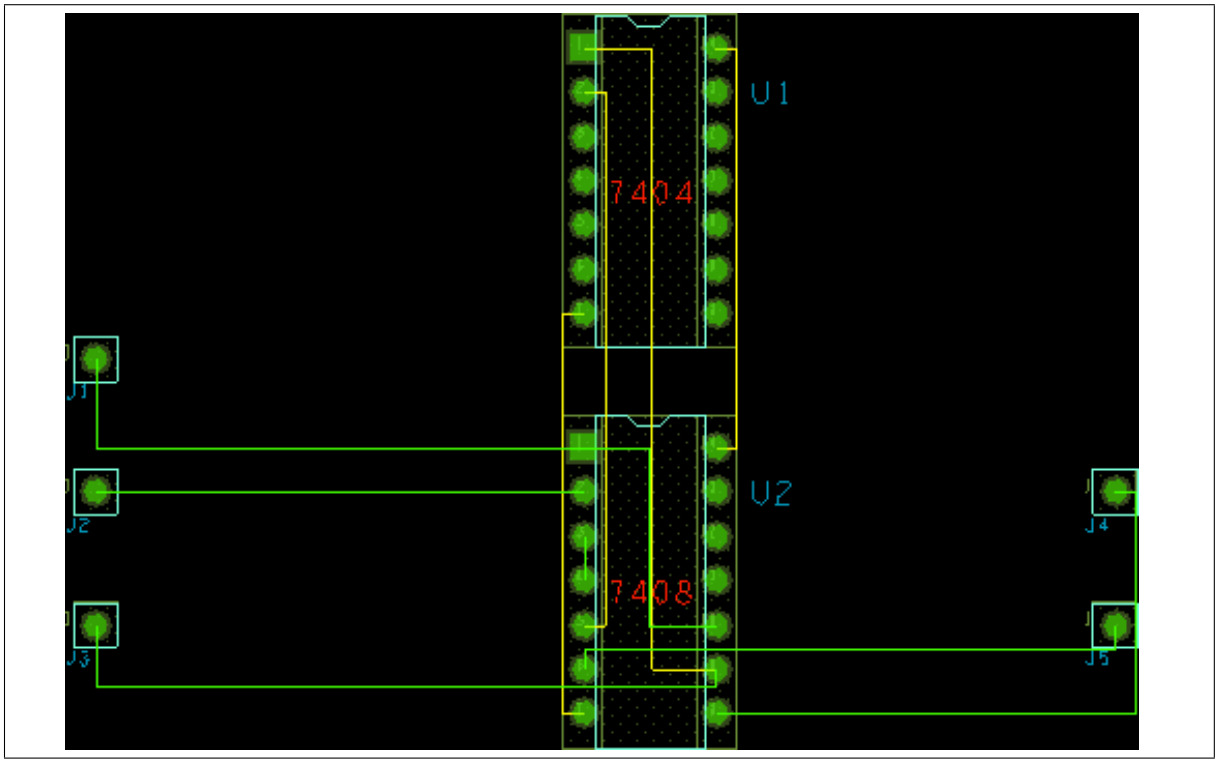

Figure 1.19: Circuit board layout for Combinational Alarm

is cut and the alarm goes off. Although highly simplified, this is a typical application of for digital logic.

In the days before microcontrollers, this circuit could actually be built using logic gates. The logic diagram of Figure 1.18 was mapped into a *7404 Hex-Inverter*, and a *7408 Quad AND-Gate*. Using the circuit shown in Figure 1.19 diagram, a circuit board could be built, the wires from the inputs and outputs could be fed into its connectors, and the circuit would function as designed.

Microcontroller Version

Combinational logic is easy to implement in an MCU. The simplicity of the logic helps expose important details about programming an MCU.

First, the designer needs to assign the input and output signals to appropriate (and unused) pins on the microcontroller. Many microcontrollers projects start with a *reference board*, a circuit board designed to support developers by placing a minimal set of components on a circuit board and opening the device to external input/output. If that is the case, then the choice of available pins may be limited to what the reference board has left open. It is good practice to document the mapping of logical signals to a pins, both in the source code as a source code comment and on the board or wires as labels.

Listing 1.7: Combinational Logic Example

```
#include <p32xxxx.h>

// B0 = Run, B1 = Motor Slow, B2 = Hood Open
// B4 = Motor Power, B5 = Alarm
int main( )
{
    PORTB = 0;

    TRISBbits.TRISB0 = 1;
    TRISBbits.TRISB1 = 1;
    TRISBbits.TRISB2 = 1;
    TRISBbits.TRISB3 = 0;
    TRISBbits.TRISB4 = 0;


    while(1) {
        PORTBbits.RB4 = PORTBbits.RB0 & PORTBbits.RB1 & ~PORTBbits.RB2;
        PORTBbits.RB5 = PORTBbits.RB0 & PORTBbits.RB2;
    }
}
```

Next, the microcontroller's GPIO module must be configured, which involves configuring selected tri-state buffers to match the assigned task of the pins. The GPIO port driver buffer should be initialized prior to enabling output to ensure that the output values are appropriate prior to actually driving those values on the port.

One-shot vs. infinite rounds Finally, the code should compute the output from the input. The logic chips of the previous section are always logic chips. As long as they are powered up, the AND gate will always compute the AND function of its inputs. The microcontroller should do the same thing. If the MCU code were to compute the output and leave `main()`, then the MCU will enter an idle state and won't continue computing the output state. This behavior is called *one-shot*.

One-shot programming is typical in conventional applications programming. In fact, it is hard to imagine any alternative. For example, in a typical, General-Purpose program, when we want to compile a program, we invoke the C compiler, it compiles the source code, and terminates (that is one-shot behavior). Since a conventional, General-Purpose computer is designed to run many different applications on demand, it doesn't make sense to keep a program running when it is finished. Therefore, it makes sense that as soon as a task is finished, it should exit and free up all of its resources to make room for other programs that the user might want.

Microcontrollers are special-purpose computers that only run one task, and only that one task. Usually, it is not a good thing for that one task to terminate - it means that the MCU won't ever run that task again without being reset. To make this happen, microcontroller code often includes an infinite while loop that spins around its loop, forever executing the same code over-and-over. This yields behavior similar to the logic chip.

MCU implementation An implementation of the combinational logic of Figure 1.18 is shown in Listing 1.7. This code follows the template described previously in this section. The code documents the assignments of pins to logical signal name. The port values and tri-state buffers are setup to match the directionality of the signals. The combinational logic computes the output inside an infinite while loop. So, as long as the chip has power, it will continue to read its input and compute the proper output.

Black Box Equivalence

Two implementations of the combinational logic were presented: one using discrete logic gates and the other using a microcontroller They both have the same input and they both take the same output. As a thought experiment, suppose both implementations were built, that is two circuit boards were built, one with the logic chips and the other with a programmed MCU. Both boards were each hidden inside a black box that only had connections for the input and output ports, and that both boxes were completely identical on the outside. Is there any way to tell which box contains the MCU and which contains the logic gates?

Functional Equivalence The two boards implement the same logic function and are functionally equivalent. Every combination of input to either board will produce an equivalent output. Because they are functionally equivalent, this cannot inform us which board is in the box. Functional equivalence is absolutely critical in establishing that two devices are comparable.

When deciding functional equivalence, it is equally vital to look at the functionality that the problem needs. For example, if we packed up a desktop computer into one of these black boxes, the computer could certainly do more than the either of the other devices, but as far as this problem is concerned, we are only interested in the combinational function described earlier.

Speed What if the two black boxes were connected to an oscilloscope which measured the time between when the input changes and when the output changes. This measures the *propagation delay* between input and output. After taking measurements, one device is found to have a delay of $45\,\mathrm{ns}$ and the other device has a delay of $40\,\mathrm{ns}$. Both devices are well within the $1\,\mathrm{ms}$ tolerance of the problem, but timing gives a clue to which device is which. Will the discrete logic gates developed in the 1970s be faster than a twenty-first century, fourth generation $80\,\mathrm{MHz}$ MCU?

Even though the MCU is running at $80\,\text{MHz}$, each pipeline stage still requires 2 clock cycles (Section **??**). There are no conditional branches, the PORT registers operate at SYSCLK speeds, and the instruction forwarding eliminates hazards. Listing 1.8 shows an optimized assembly code for this project's loop. Based on the listing, we see that the MCU implementation of our C code needs 16 instructions.

Listing 1.8: Assembly implementation of the combinational logic example

```
#PORTBbits.RB4 = PORTBbits.RB0 & PORTBbits.RB1 & ~PORTBbits.RB2;
#PORTBbits.RB5 = PORTBbits.RB0 & PORTBbits.RB2;
# s0 has addreses RB4
L1:     # top of the infinite while loop
    lw      $s1, 0($s0)     # load port B into memory
    ext     $t0,$s1,0,1     # get bit 0
    ext     $t1,$s1,1,1     # get bit 1
    srl     $t1, $t1, 1
    ext     $t2,$s1,2,1     # get bit 2
    srl     $t2, $t2, 2
    not     $t4, $t2        # ~ RB2
    and     $t4, $t4, $t1   # and $t5 and rb1
    and     $t4,$t4,$t0     # rb4 has value
    and     $t5,$t0,$t2     # rb5 has value
    sll     $t4, $t4, 4     # move rb4 over 4
    sll     $t5, $t5, 5     # move rb5 over 5
    ins     $s1,$t4,4,1   # insert the bit into S1
    ins     $s1,$t5,5,1     # insert the bit into S1
    j       L1              # jump to L1
    sw      $s1, 0($s0)     # branch delay slot!
```

We are running 40 million instructions per second, and in the worst case, we must execute 16 instructions:

$$T_{MCU} = \frac{1}{40,000,000} \cdot 16 = 400\text{ns} \tag{1.8}$$

The *critical path* through the logic gates involves passing through three gates: the NOT gate, and two AND gates. The NOT gate has a propagation time of $9\,\text{ns}$ and the AND gate has a time of $18\,\text{ns}$. The expected time is:

$$T_{GATE} = 9 + 18 + 18 = 45\,\text{ns} \tag{1.9}$$

So, the discrete logic chips from the 1970's were actually faster than the modern microcontroller! In fact, they were more than 8 times faster (for this problem). The logic delays are actually quite high by today's standards. If we used more modern devices with sub-nanosecond delays, the gap between the MCU and the logic gates would be even more apparent. The timing analysis for the source code was highly simplified, and will be examined in greater detail in Section 1.5.6.

Cost Ignoring the board costs, the discrete logic chips costs were about $\$0.75$ total. The MCU costs about $\$1.50$, twice the cost of the chips. So, we can use cost as another way to tell the two black boxes apart. But will this always be in the logic-chips favor?

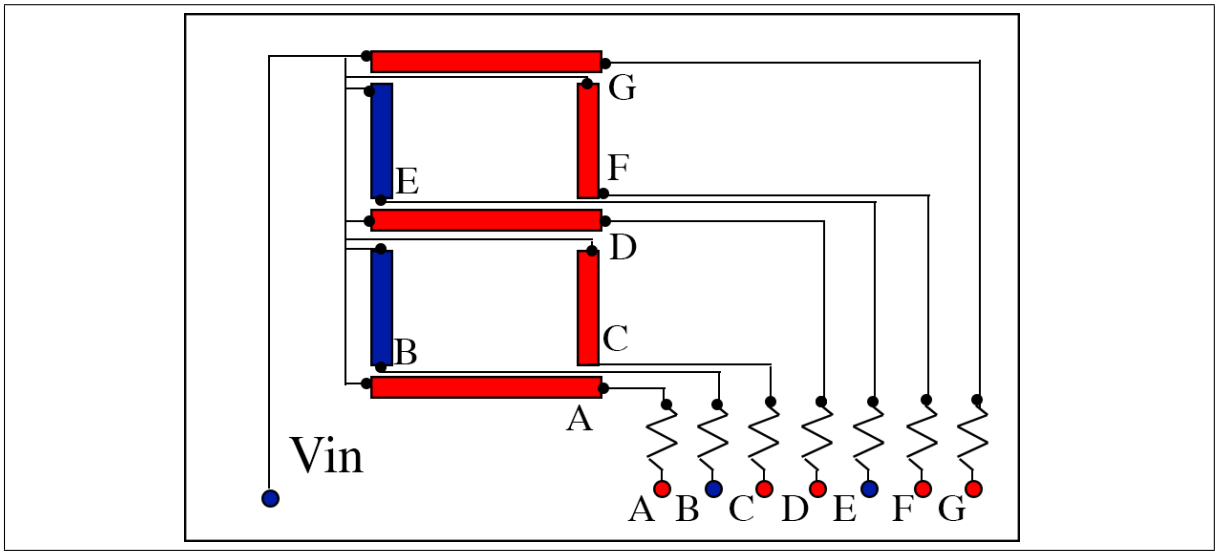

Figure 1.20: Seven segment display, displaying the number 3.

If the logic function were to change and require more gates, the price of additional chips will tip us over the cost of the small MCU. This little program probably uses less than 1% of the available program words in the MCU. The logic function can grow to include considerable complexity before the MCU must be replaced with a larger unit. So, applications that need one or two logic chips may be cheaper than an MCU, but beyond that the MCU will generally be more cost effective.

Additionally, if the logic function has any possibility of changing in the future, then the logic gates may need to be reorganized, needing a major revision to the circuit board, which can be a significant cost. The MCU can change by simply upgrading its firmware, in many cases the board may not need to change at all. Because they can be more easily reprogrammed to accommodate system changes, if the component costs for MCUs versus discrete chips are even remotely close, the MCU is still probably the most cost effective.

1.5.2 Seven-Segment Display

A seven-segment display uses 7 LEDs to display a number of digits, some letters, and symbols. Most common configurations include a common power source and seven control lines. Some units include current limiting resistors, others require them to be external to the module. An example is shown in Figure 1.20.

The display in Figure 1.20 is displaying the number three. A different configuration of ground or high-impedance values would allow us to display all of the decimal digits, and even some hexadecimal values.

Listing 1.9: BCD to Seven Segment Display

```
#include <stdint.h>
#include "port.h"

#define NUM_NUMBERS (10)
// store the 7 segment values as 7-bit value: ABCDEFG
static uint8_t numbers[] = {
  0b1110111,  0b0010010,  0b1101011,    // 0,1,2
  0b1011011,  0b0011110,  0b1011101,    // 3,4,5
  0b1111100,  0b0010011,  0b1111111,    // 6,7,8
  0b0011111   // 9
}


void main( )
{
  TRISB = TRISB & ! 0x7f;
  TRISA = TRISA & 0xf;
  while(1) {
    uint8_t bcd = PORTA & 0x0f;
    PORTB = numbers[bcd];
  }
}
```

Listing 1.9 shows an implementation of a BCD to Seven-Segment display driver. The C code use the first 4 pins on PORTA to read a binary-coded-decimal number. Since we are only reading one decimal digit, BCD just means the number is encoded in binary. For example, 5 would be encoded as `0b0101`. The code also uses a *Lookup-Table* (LUT) to map the BCD value to the bits that should be turned on or off on the display. The LUT expects that the display is mapped exactly to the first 7 bits of port B. This is typical *combinational logic*. As fast as the MCU can execute each loop iteration, it will read the input pins and drive the output pins.

While this example works, the source code depends heavily on the organization of the hardware. Suppose that the circuit board designer has to make a change and move one of the I/O lines from either PORTA or PORTB? The entire program must be scrapped. Section 1.6 will look at different methods of writing MCU code and will present several strategies that could improve this example.

1.5.3 Square Wave Output

When programming a microcontroller, the number of instructions that the CPU executes can directly impact the wave-form that is presented on a port. In this section, we look at creating a square-wave, an alternating high and low voltage. The following two examples show a square wave function using two different patterns. The first example, in Listing 1.10, uses an *if/else* control to check if the port was high then make it low, and make it high otherwise. The second example, in Listing 1.12, uses the boolean *not* operator to flip the port bit. Both function correctly, i.e. they both correctly toggle the port register high to low back to high again. But they both perform quite differently. The reasons for the different can really only be seen in the assembly language code. Both Figures 1.11 and 1.13 show the actual assembly

Listing 1.10: Square Wave Pulse - Using if/else control

```
void pulse( )
{
  while(1)
  {
    if (PORTAbits.RA0 == 0) PORTAbits.RA0 = 1;
    else PORTAbits.RA0 = 0;
  }
}
```

language generated by the PIC32 C compiler distributed by Microchip.

Listing 1.11: Disassembled version of Listing 1.10

```
  !    while(1) {
  !    if (PORTBbits.RB0 == 0)
  0x9D000258: LUI V0, -16504
  0x9D00025C: LW V0, 24656(V0)
  0x9D000260: ANDI V0, V0, 1
  0x9D000264: SLTIU V0, V0, 1
  0x9D000268: ANDI V0, V0, 255
  0x9D00026C: BEQ V0, ZERO, 0x9D000290
  0x9D000270: NOP
  !        PORTBbits.RB0 = 1;
  0x9D000274: LUI V1, -16504
  0x9D000278: LW V0, 24656(V1)
  0x9D00027C: ADDIU A0, ZERO, 1
  0x9D000280: INS V0, A0, 0, 1
  0x9D000284: SW V0, 24656(V1)
  0x9D000288: J 0x9D000258
  0x9D00028C: NOP

  !    else PORTBbits.RB0 = 0;
  0x9D000290: LUI V1, -16504
  0x9D000294: LW V0, 24656(V1)
  0x9D000298: INS V0, ZERO, 0, 1
  0x9D00029C: SW V0, 24656(V1)
  0x9D0002A0: J 0x9D000258
  0x9D0002A4: NOP

```

In Figure 1.11, the if statement is seven instructions (including the NOP in the branch-delay slot on line 9). If the port bit is 0, then the branch is not taken, so then lines 11-17 are executed, returning control back to the top of the while loop on line 3. This case used another seven instructions. In the case of the else logic on lines 20 to 25, there are only six instructions including the NOP. The CPU will spend 14 instructions making the output line go from low to high, and only 13 instructions going from high to low, crucially, the difference is before the value is stored in memory on line 15, so the extra instruction will keep the GPIO line low for 1 extra instruction time than it will stay high.

The second example, in Figure 1.12, is more efficient. It does away with the if/else and uses the boolean negation operator. Intuitively, it should be faster because it doesn't have the test/branch for the if/else. The actual assembly code is shown in Figure 1.13. Surprisingly, the execution time is always 14 clock cycles, which is actually the same speed as the previous if/then example. The reason why can be seen in the assembly code: note that while the C code looks like it should just be something like: LOAD PORTA, NEGATE PORTB, STORE PORTB, and it looks like it should only take three cycles. The problem is that we are only negating one of the 32-bit values, leaving the rest unchanged. So, this code has to isolate the one bit

Example 1.5.1. Suppose the processor's SYSCLK is set for $20\,\text{MHz}$. As described in Chapter **??**, the processor issues an instruction every two clock cycles, and the forwarding units eliminate pipeline stalls for dependencies. Compute the high, low, and total period for this square-wave.
The time period for an instruction is:

$$T_{INST} = \frac{1}{10,000,000} = 0.1\,\mu\text{s}$$

Next compute the high and low times:

$$T_{LO} = T_{INST} * 14 = 1.4\,\mu\text{s}$$

$$T_{HI} = T_{INST} * 13 = 1.3\,\mu\text{s}$$

Finally, the overall period is:

$$T_{OVERALL} = T_{INST} * 27 = 2.7\,\mu\text{s}$$

The frequency is about $370\,\text{kHz}$. The *jitter* in the hi/lo actually accounts for an almost 4% error rate if this signal were used as a clock.

■

Listing 1.12: Square Wave Pulse - Using branchless control

```
void pulse2( )
{
  while(1) {
    PORTBbits.RB0 = ~PORTBbits.RB0;
  }
}
```

through a series of and/or masks.

Listing 1.13: Disassembled version of Listing 1.12

```
!    while(1) {
!        PORTBbits.RB0 = ~PORTBbits.RB0;
0x9D000358: LUI V0, -16504
0x9D00035C: LW V0, 24656(V0)
0x9D000360: EXT V0, V0, 0, 1
0x9D000364: ANDI V0, V0, 255
0x9D000368: NOR V0, ZERO, V0
0x9D00036C: ANDI V0, V0, 255
0x9D000370: ANDI V0, V0, 1
0x9D000374: ANDI A0, V0, 255
0x9D000378: LUI V1, -16504
0x9D00037C: LW V0, 24656(V1)
0x9D000380: INS V0, A0, 0, 1
0x9D000384: SW V0, 24656(V1)
0x9D000388: J 0x9D000358
0x9D00038C: NOP
```

In general applications programming, students often learn about *loop invariants*, conditions that are true before and after each iteration of a loop. In this were a general program, an invariant would be that the port flips its state. In both examples, the loop invariant is met, but the timings are different. Consider the next example, in Listing 1.14, which obeys the loop invariant - with each iteration of the loop, if the value was high, it will be low; and vice versa.

Listing 1.14: Square Wave Pulse - Invariants aren't sufficient

```
void pulse1( )
{
  while(1)
  {
    int x= PORTBbits.RB0;
    PORTBbits.RB0 = 1;
    if (x)
      PORTBbits.RB0 = 0;
  }
}
```

Listing 1.15: A 10 µs delay using pre-processor and intrinsics

```
#define DELAY_1US()  \
  _NOP(); _NOP(); _NOP(); _NOP(); _NOP();\
  _NOP(); _NOP(); _NOP(); _NOP(); _NOP();

#define DELAY_10US() \
  DELAY_1US(); DELAY_1US(); \
  DELAY_1US(); DELAY_1US(); DELAY_1US(); \
  DELAY_1US(); DELAY_1US(); \
  DELAY_1US(); DELAY_1US(); DELAY_1US();
```

But, if the code were run, it would create a wave form where there were several extra cycles where the output was high, which would be even less balanced than either of the first two examples. This illustrates a common challenge when learning to code for microcontrollers: code frequently interacts with the real, physical environment; and timing matters.

1.5.4 Delay Loops

Sometimes, the microcontroller needs to stretch out the length of time a clock is high or low. This can usually be accomplished a delay loop. The loop executes a number instructions, usually a no-operation (NOP) instruction, for some fixed number of time. The speed of the processor determines the relationship between the number of NOPs and the length of time.

Suppose the processor is running at 20 MHz, then a single NOP instruction will require 2 clock cycles, or 0.1 µs. Ten instructions will create a delay of 1 µs, but it is important to remember that it is ten *machine* instructions, not C statements! Listings 1.15 and 1.16 both create a 10 µs delay, but they do it very differently.

Listing 1.15 creates the delay by using uses the `DELAY_1US` macro 10 times. Each `DELAY_1US` macro uses 10 `NOP` instruction. So while this is effective, it uses $10 \cdot 10 = 100$ instructions for each delay. Because this is a MIPS32 architecture it actually uses 400 bytes of instruction memory! With code this size, we also need to pay attention to the *prefetch cache* on the instruction memory - if the hit rates aren't 100%, then this code won't give the expected delay.

A second attempt is shown in Listing 1.16. This also uses the C pre-processor to define a function macro, `DELAY_X_US(X)`, that takes an argument for the number of microseconds to delay. The function macro starts off with a set of braces, which tells the C compiler to create

Listing 1.16: A 10 µs delay using pre-processor and intrinsics, with less code

```
#define DELAY_X_US(X) { \
  int __i; \
for (__i; __i < 10*X; __i++) Nop();  \
}


!    DELAY_X_US(10);
0x9D000134: J 0x9D00014C
0x9D000138: NOP
0x9D00013C: SSNOP
0x9D000140: LW V0, 0(S8)
0x9D000144: ADDIU V0, V0, 1
0x9D000148: SW V0, 0(S8)
0x9D00014C: LW V0, 0(S8)
0x9D000150: SLTI V0, V0, 100
0x9D000154: BNE V0, ZERO, 0x9D00013C
0x9D000158: NOP
```

a new, local scope. The code then declares a local variable, `__i`, and then uses a `for` loop that executes for `10 * X` times, and in the body of the loop is enough `NOP` instructions to make each loop iteration take steps, which gives a delay of 1 µs. The assembly code is shown in the listing, notice how each iteration is exactly 10 instructions. There is some wizardry in the arithmetic of `10 * X`. Notice that in the assembly code, there is not a multiplication, and in fact, in the assembly code, there is a `SLTI` instruction to compare the loop variable `i` to `100`. The C pre-processor will use *constant folding* across arithmetic, and will replace arithmetic expressions with their pre-computed constants wherever it can. Because it did that, the number of iterations will be computed at compile time rather than run-time (as long as the argument is a defined constant). This example is much better, using only ten instructions (40 instruction words) for the total delay loop.

The problem with both of these two attempts is that we assume that the instruction stream will not be interrupted with any other event. This will turn out not to be true as we move on. One final version of a delay loop uses code we developed in Chapter **??** for the *Core Timer* built into the MIPS32 (and most modern MCUs). Listing 1.17 uses two built-in functions: `ReadCoreTimer` and `WriteCoreTimer` to access the timer. Since the timer always runs at the same speed, this code will give the most accurate timing available, even if something interrupts the instruction stream.

1.5.5 Pulse Width Modulation

Using delay loops allows the microcontroller to generate a wide range of *pulse-width-modulation* (PWM) tasks, where the width of each pulse is controlled (or modulated) by the microcontroller. The brightness of an LED is controlled by the current that flows through it, but in most applications, the current limiting resistor is soldered into the board and it cannot be changed. This made LEDs easy to control - just turn them on or off. But what if we want to vary the

Listing 1.17: A microsecond delay using pre-processor and built-in functions for the core timer. This uses less code and is more stable.

```
#define DELAY_X_US(X) { \
  int __i; \
for (__i; __i < 10*X; __i++) Nop();  \
}


!    DELAY_X_US(10);
0x9D000134: J 0x9D00014C
0x9D000138: NOP
0x9D00013C: SSNOP
0x9D000140: LW V0, 0(S8)
0x9D000144: ADDIU V0, V0, 1
0x9D000148: SW V0, 0(S8)
0x9D00014C: LW V0, 0(S8)
0x9D000150: SLTI V0, V0, 100
0x9D000154: BNE V0, ZERO, 0x9D00013C
0x9D000158: NOP
```

Listing 1.18: A simple PWM loop to control an LED.

```
#define WIDTH_US (100)

void led_pulse(int percent)
{

  int ton = (WIDTH_US * percent) / 100;
  int toff = WIDTH_US - ton;

  if (ton > 0) {
    PORTAbits.RA0 = 1;
    DELAY_X_US(ton);
  }

  if (toff > 0) {
    PORTAbits.RA0 = 0;
    DELAY_X_US(toff);
  }

}
```

brightness of the LED?

A pulse-width modulation wave-form can vary the brightness of an LED. Based on the delay loop of Listing 1.17, we can construct a pulse-width modulation function that will turn the LED on and off for a varying amount of time. The LED will be on for some time (T_{ON}), and it will be off for some time (T_{OFF}). The total time will be $T_{TOT} = T_{ON} + T_{OFF}$. If the total period is more than $\frac{1}{100}$ of a second, humans will perceive a visible flicker. On the other hand, if the time is too fast, the LED will not full charge or discharge.

We will use a $100\,\mu s$ total time. Then, we can use a percentage of time that the LED should be on and use that to drive the delay loops.

1.5.6 Latency

The *latency* or reaction time of a system is how long it takes from the time an input event occurs until it responds. As noted in the previous section, the amount of time required to execute a stream of instructions is predictable. That fact was exploited to create predictable delay loops, and the same analysis can be used to determine the reaction time of a section of

Listing 1.19: Drive output to match input, test reaction time.

```
int main()
{
  while(1) {
    PORTCbits.RC1 = PORTBbits.RB0;
  }
}

/* Assembly Code ****************
0x9D000240: LUI V0, -16504    # load offset of PORTB
0x9D000244: LW V0, 24656(V0)  # load PORTB into V0
0x9D000248: EXT V0, V0, 0, 1  # Extract least bit
0x9D00024C: ANDI A0, V0, 255  # extra instruction by compiler
0x9D000250: LUI V1, -16504    # load ofset of PORTC
0x9D000254: LW V0, 24720(V1)  # load existing PORTC
0x9D000258: INS V0, A0, 1, 1  # insert bit into PORTC
0x9D00025C: SW V0, 24720(V1)  # store PORTC
0x9D000260: J 0x9D000240    # jump back to top
0x9D000264: NOP         # branch-delay slot
 ******************************** */
```

code.

Listing 1.19 uses an infinite while loop that will drive the PORTC bit 1 to be equal to whatever PORTB bit 0 is. The Assembly code shows what the Microchip XC32 C compiler generates for the while loop. There are a total of 10 instructions that comprise the loop. To assess the reaction time of the code, we need to consider both the minimum and maximum amount of time between when the input signal goes high and when the output signal goes high.

The minimum reaction time happens when the input signal change occurs immediately before the load-word instruction at address `0x9D000244`. For this to happen, the signal would have to change two clock cycles before so the new value would be read by that load word. The updated instruction will be written eight instructions later, which corresponds to 16 clock cycles. If SYSCLK is at $20\,\text{MHz}$, and each instruction takes $0.1\,\mu\text{s}$, then the fastest reaction time will be $1.6\,\mu\text{s}$.

The worst-case reaction time happens when the input signal occurs during the load-word instruction at address `0x9D000244`. Because of the input latch shown in the block diagram, the load-word will get the old value, even though the signal has already changed. Now the current iteration of the loop must finish, the loop must begin again, and then execute to the store-word instruction. This will require seventeen instructions, 34 clock cycles, or $3.4\,\mu\text{s}$.

If the processor were clocked up to its maximum $80\,\text{MHz}$, these times will be quartered, or a best case of $0.4\,\mu\text{s}$ ($400\,\text{ns}$). This seems quite fast, especially by human standards. But this isn't terribly fast by computer standards. Section 1.5.1 introduced combinations of logic gates. There is actually a logic-gate that does the same function - it is a device called a logic buffer, and its designed to make the output follow the input, without placing a current load on the input. They are used to every *level-shift*, where the input and output voltages are

different, or to manage *fan-out* where the large number of devices connected to a driver will cause an over-current on the driver, the buffer can be used to expand the number of devices. A typical 74-series buffer (e.g. 7407 hex buffer) is rated at a typical 20 ns switching time, or 20-times faster than the microcontroller. The microcontroller arrives at the same outcome as the logic gate, only it trades while-loops and assignment statements for a dedicated network of transistors.

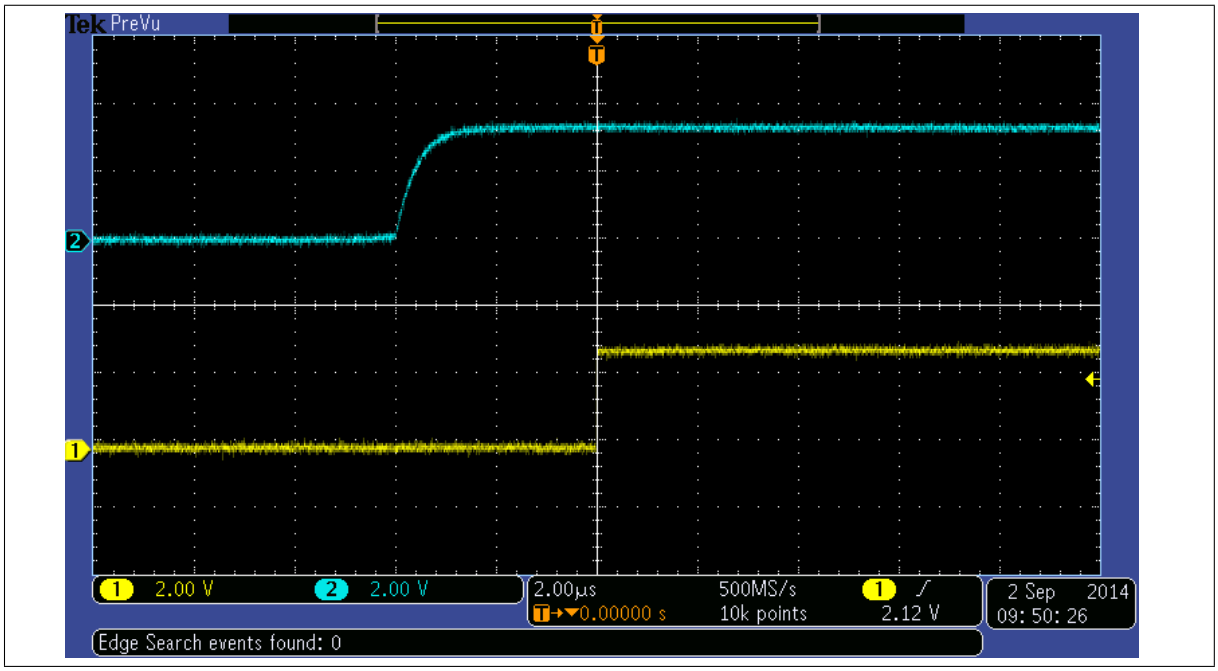

Figure 1.21: Oscilloscope capture of actual reaction time of code in Listing 1.19

The actual reaction time of code can also be measured using an oscilloscope. Figure 1.21 shows an oscilloscope capture of two signals. The blue signal shows the input to the program, and corresponds to a device (e.g a switch) raising the voltage on an input line. The yellow signal shows the output from the program. Each horizontal square is 2 µs. The observed reaction time is about 7 µs, due to the clock rate differences between the devices.

To be fair to the microcontroller, neither the C nor the Assembly code were optimal. Listing 1.20 shows a more optimized version of the program. By carefully setting the tri-state registers, the registers will ignore any writes to non-output values. Since now, there is only one PORTC bit set for output, the assignment statement does not need to restrict itself to just one bit, but can assign the whole integer word, and only the least-significant bit will be set. By manually re-coding the Assembly source code, the main loop can be reduced to three instructions (the `LUI` is not part of the loop). Now the minimum delay will be four instructions, and the maximum delay will be 6 instructions, which is more than 4 times faster than before.

Several key points were illustrated in this section. First, because the microcontroller is

Listing 1.20: Improved reaction time

```
int main()
{
  TRISB = 1;
  TRISC = ~1;
  while(1) {
    PORTC = PORTB;
  }
}

/* Assembly Code *****************
0x9D000240: LUI V0, -16504    # load offset of PORTB
0x9D000244: LW V1, 24656(V0)  # load PORTB into V1
0x9D000254: J 0x9D000244    # jump back to top
0x9D000248: SW V1, 24720(V0)    # store V1 into PORTC
 ******************************** */
```

Listing 1.21: Example of failed loop invariants

```
int main()
{
  while(1) {
    PORTCbits.RC1 = 0;
    if (PORTBbits.RB0)
      PORTCbits.RC1 = 1;
  }
}
```

running code through a CPU, the time it takes to process input into output will depend on the clock rate of that CPU, the efficiency of the machine language, and when the event occurs relative to when the change is detected. Second, even for trivial logic functions, such as a buffer, the overhead will be many times slower than a dedicated logic chip. Finally, the code must be optimized to maximize performance, including making best use of the hardware capabilities, organizing the C source code to promote better optimizations, and possibly manually re-writing the Assembly code to improve on the optimizations of the compiler.

Figure 1.22 shows the output captured by an oscilloscope from running the code of Listing 1.21. The blue trace of the figure was the result of a switch connected to an input port. The yellow trace is the output port connected to an LED. Each horizontal grid represents $4\,\mu s$, each vertical grid represents $2\,V$. The jump in the blue line is when a button is pressed, causing the input voltage to go high. There is an $8\,\mu s$ delay between the switch press and the first time the port output goes high. Similar to listing 1.14, the first step of the while loop sets the output port to 0, regardless of what it was before, and then executes the if statement. Because the if test is true, the branch is not taken, and the code sets the output port high. The port stays high for $2.4\,\mu s$, which is the time it takes for the program to loop back to line 5 of the listing. As long as the button is pressed, the output will continue to have a pulsed, square-wave output instead of a steady high output. Although the *loop invariants* are true, the *side effect* of driving an LED is not consistent. Interestingly, the LED does light up in this example, but just not as brightly as it did in the previous example - so while this is technical incorrect, it does actually reduce the power-consumption of the LED by 2/3 (it is on for $2.4\,\mu s$ and off for $4.8\,\mu s$).

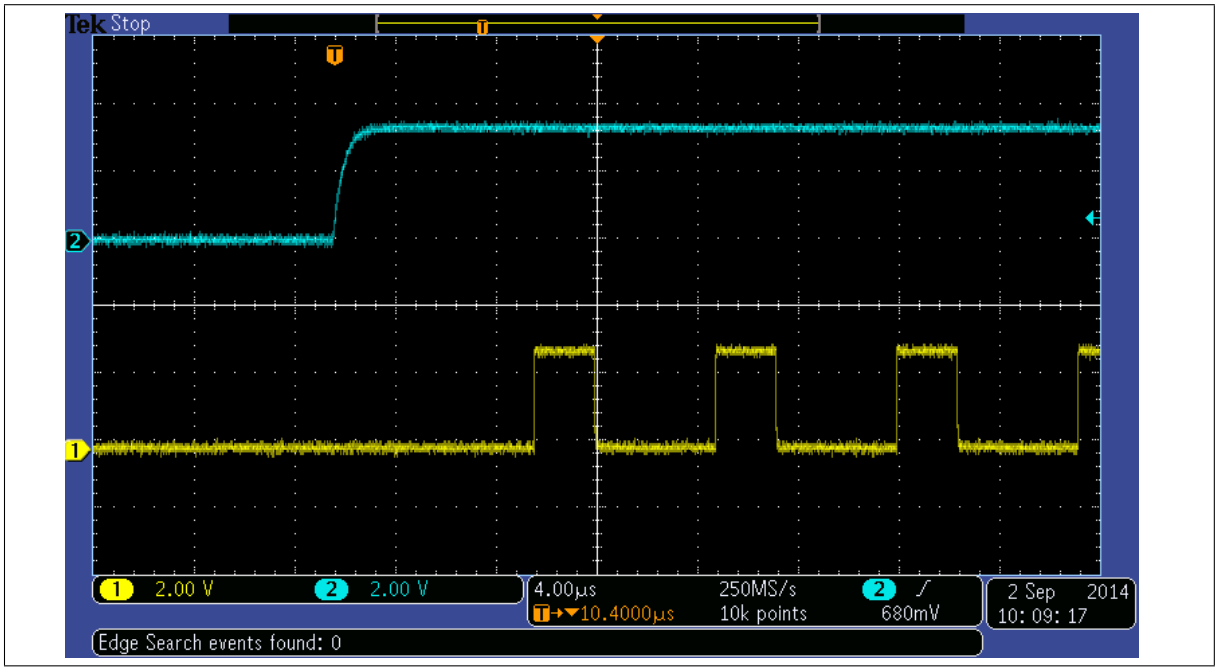

Figure 1.22: Port output from Listing 1.14. Blue trace is the input trigger, Yellow trace is the output to from the port.

1.5.7 Debouncing Switch Input

One of the most common input devices to a microcontroller is a mechanical switch. The switch typically has a pair of metal contacts that are brought together to make a circuit or pulled apart to break the circuit. As the metal contacts strike each other, they experience an elastic collision that causes them to momentarily deform. The collision energy generates a momentary oscillation where the metal surfaces are vibrate against each other. This phenomena is called *switch bounce*. A single activation of the switch can generate a complicated wave-form that bounces high and low until the contacts *settle* into their final state.

Figure 1.23 shows an oscilloscope trace (voltage over time) of a switch engaging. The vertical grids are arranged at 2 Volts per grid, and the horizontal grid is at $40\,\mu s$ per grid. The switch bounced for almost $200\,\mu s$ before it settled down. Note the three negative voltage drops in the trace, reaching almost $-2\,V$, which are caused by the rapid drop in voltage and the discharge of the parasitic inductance of the wires connecting the switch. This is one of the reasons for the diode protection mechanism built into the GPIO port.

There are several techniques for *debouncing* a switch, the most common being the introduction of a capacitor and resistor to create a low-pass filter to block the bounce noise. While this is reasonably effective, it can also be handled by the microcontroller. Instead of reacting to the switch as soon as a level change is detected, a *debounce loop* will poll the status of the

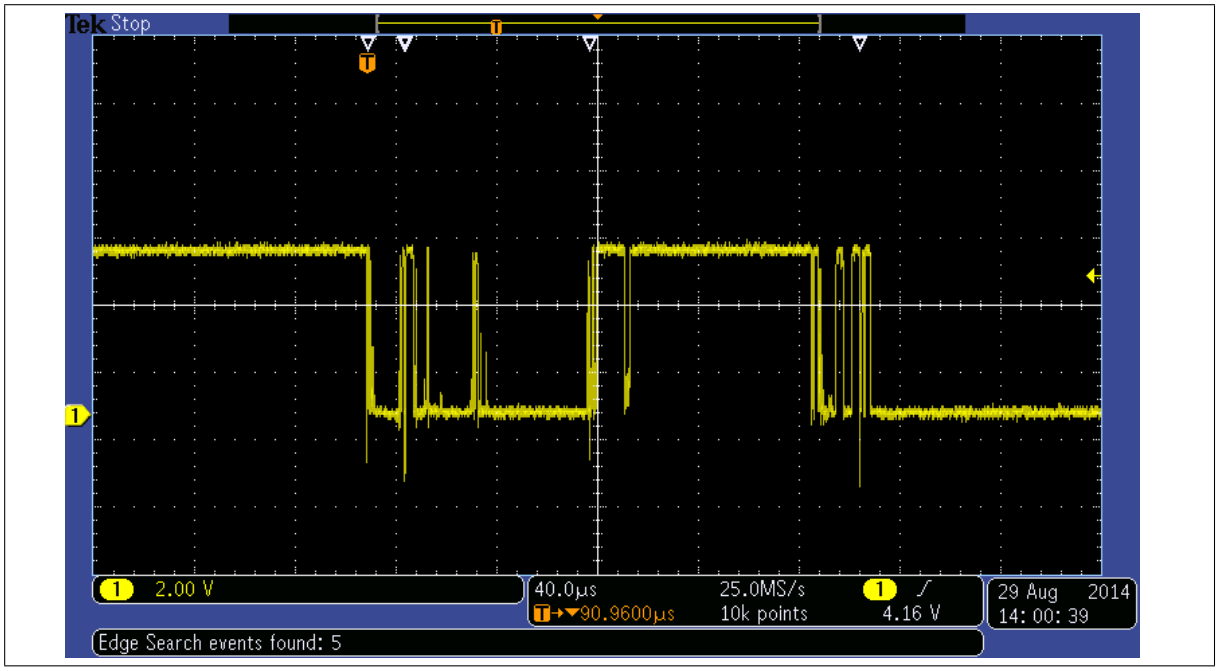

Figure 1.23: Oscilloscope capture of switch bounce.

Listing 1.22: Debounce a switch

```
#define DEBOUNCE_TIME 100

int get_switch( )
{
  int x = 0;
  int last_switch;

  while (x < DEBOUNCE_TIME)
  {
    last_switch = PORTBbits.RB0;
    for (x = 0; x < DEBOUNCE_TIME; x++)
    {
      if (PORTBbits.RB0 != last_switch)
        break;
    }
  }

  return last_switch;
}
```

switch, and then react only after the switch has been determined to settle down.

Listing 1.22 shows an example of a switch debounce function. The requires that 100 successive reads of the port must be the same value in order for the port to be considered *settled*. The way it works is that the *while* loop executes until `count` reaches `DEBOUNCE_TIME` which is a sufficiently large number to ensure that the settle time has been reached for the particular circuit elements. The *while* loop's body starts off by capturing the value of the port, and then entering the *for* loop. As the *for* loop executes, if the port status doesn't change, then it will complete normally, and the value of `count` will be equal to `DEBOUNCE_TIME`, so then the *while* loop will also terminate, and the value of the switch will be returned. However, if the switch has not yet settled, then there will be times when the current port status won't match the value it was at the beginning of the loop (see line 13). When this happens, the *for* performs

an early termination by issuing the `break` on line 14. Because the `count` variable is not equal to the `DEBOUNCE_TIME`, the while loop will restart, which resets the last port status and the count.

It is almost never acceptable to interface to switches, including throw-switch, push-buttons, or momentary buttons, without performing some sort of switch debouncing. Suppose we were writing an application that would count the number of times a button was pressed. With a fast microcontroller and without switch debouncing, the single physical switch throw in Figure 1.23 would actually appear as many switch activations, because the faster microcontroller will be react to more of the transient pulses.

1.5.8 Switch Multiplexing

The previous section used a single port on the microcontroller for a single switch. Consider the case of a integrating a numerical keypad on a simple calculator. The calculator would have 10 buttons for the digits, 4 buttons for arithmetic operations, a button for the equals sign, and another button for the clear. The simple calculator will have 16 buttons, meaning we would need to use 16 microcontroller ports to detect when one of the switches was pressed, which is going to be a large chunk out of any microcontroller's available input ports.

An alternative approach is based on the idea of *multiplexing*, where one line is used to connect multiple devices, but only one is active at a time. Figure 1.24 shows a 16-way *switch matrix*, where switches are grouped into rows and columns. Each switch is connected to a unique combination of rows and columns. The switches are *multiplexed* because only one row will be active a time, so the MCU can detect which switches of a row are pressed. By *strobing* through each row, the MCU can read out all sixteen switches one row at a time.

Suppose the user has depressed switches 6, 10 and 11. When the MCU starts a switch read-out, it sources power onto port `ROW0`. Since none of the switches in the row are down, there will be zero volts (logic-low) on each of the column ports. Next, the MCU sources power onto `ROW1`. Switch `SW6` is down, which allows voltage to pass `COL1`, but only that port will have voltage (logic-high). As the MCU continues to strobe the output rows, it sources power onto `ROW2`. This has two switches pressed, so there will be voltage at both `COL1` and `COL2`, so both will read logic-high. The strobe cycle will complete on the fourth row, which doesn't have any buttons pressed.

Electrically, the schematic shown in Figure 1.24 includes pull-down resistors on both the row lines as well as the column lines, ensuring the ports don't float when they aren't con-

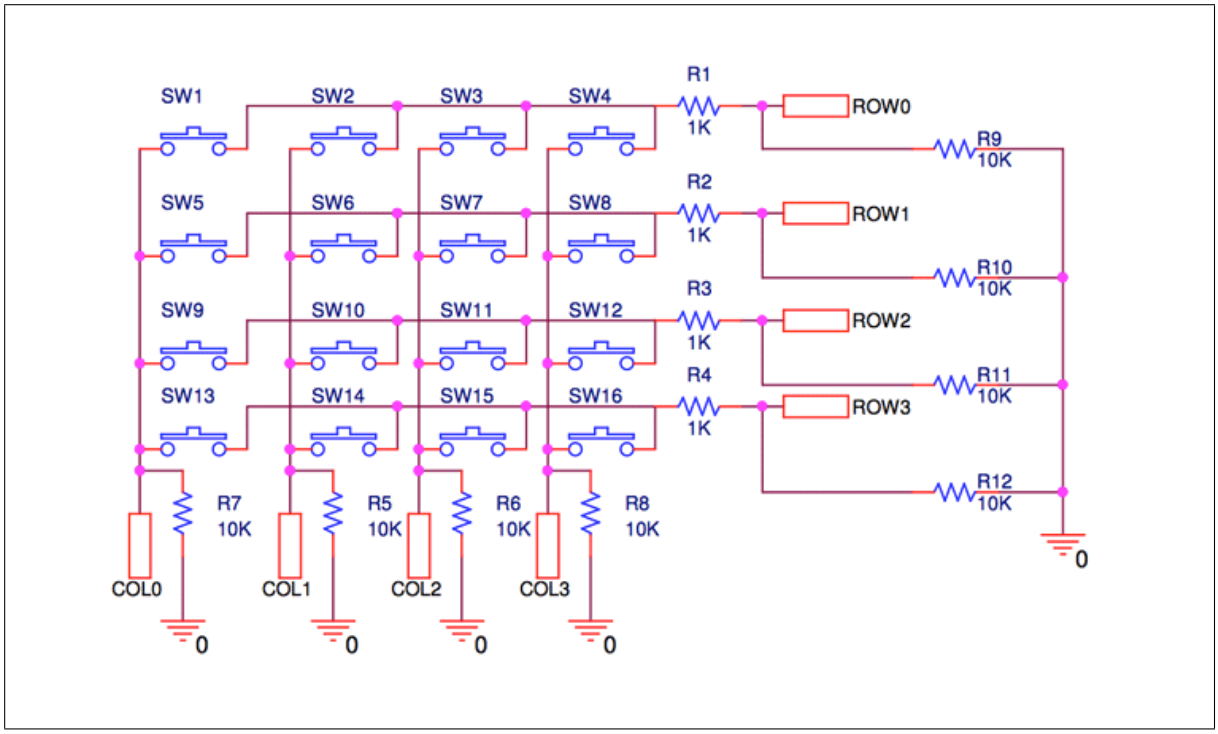

Figure 1.24: Switch multiplexing using 4x4 grid.

nected. Also, because the switches are connected in this matrix, it is important that one and only one row is sourcing current, and that the other rows are in *High-Z* mode to prevent them from sinking current from a connected switch. If that were to happen, the matrix would create a voltage divider, and its possible that the voltage at the input port would be above the V_{IH} threshold.

Listing 1.24 shows C code that will drive the rows, but will also debounce each switch. It uses an array of structs that keep track of the last port value of the switch, the number of consecutive times it has been in that state, and the current up or down status (an enumerated value). The function `get_switch` uses an unusual pair of *for* loops that increase by powers of two. This is an efficient way of building the row and port masks for sourcing the rows and checking the columns.

The code uses about 22 cycles to evaluate each column, and thus about 88 cycles for each row, and about 352 cycles to check the whole matrix. If each cycle takes $0.1\,\mu s$, then this code will require $35.2\,\mu s$ to check the whole matrix. Because only one key is being scanned at a time, if the key-press is held less than this time, it could be missed. On a ten-key calculator, a professional data entry clerk is expected to hit 10,000 keystrokes per hour, or 2.77 keystrokes per second. This works out to $0.361\,s$ or $361\,010\,\mu s$ between strokes. The microcontroller will be 10,000 times faster than the human typist, which is sufficient to ensure that no keystrokes

Listing 1.24: Switch multiplexing example

```
typedef enum {
    DOWN = 0, UP = 1
} switch_state_t;

typedef struct {
    int last;
    uint16_t count;
    switch_state_t current;
} switch_t;

switch_t switches[16];

// PORTB bits 0,1,2,3 = row, PORTC bits 0,1,2,3 = col
int get_switch( )
{
    int row, col, swnum;

    swnum = 0;
    // row = 1, 2, 4, 8 (16 ends loop)
    for (row = 1; row < 16; row = row * 2)
    {
        TRISB = ~row;       // current row output, rest high-z (input)
        PORTB = row;        // current row = drive courrent

        // col = 1, 2, 4, 8 (16 ends loop)
        for (col = 1; col < 16; col = col * 2)
        {
            int v = PORTC & col;   // small chance that PORTC could change during loop

            // check to see if the switches match
            if (v != switches[swnum].last)
                switches[swnum].count = 0;
            else if ( ++switches[swnum].count > DEBOUNCE_TIME) {
                if (v == 1)
                    switches[swnum].current = UP;
                else
                    switches[swnum].current = DOWN;
            }
            // store state of switch and move to next switch number
            switches[swnum].last = v;
            swnum++;    // 0 .. 16
        }
    }

    // all back to input
    TRISB = 0xffff;
}
```

will be missed.

Switches and Ports

The previous example uses 4 output and 4 input ports to arrive at 16 total ports. Adding one additional output row or input column would add the capability for 4 additional switches. As a generalization of this relationship, Equation 1.10 shows the N the number of switches, R the number of rows, and C the number of columns.

$$N = R \times C \tag{1.10}$$

To get the number of rows and columns approximately balanced, we get:

$$R = \lfloor \sqrt{(N)} \rfloor \tag{1.11}$$

$$C = \lceil N/R \rceil \tag{1.12}$$

$$\tag{1.13}$$

For example, suppose we wanted to build a PC keyboard with 104-keys, using Equation 1.11, to keep the the rows and columns approximately balanced, we get:

$$\begin{aligned} R &= \lfloor \sqrt{(104)} \rfloor \\ &\approx \lfloor 10.19 \rfloor \\ &= 10 \end{aligned}$$

and then solving for C:

$$\begin{aligned} C &= \lceil N/R \rceil \\ &= \lceil 104/10 \rceil \\ &= 11 \end{aligned}$$

Our PC keyboard will actually have capacity for $N = 10 \cdot 11 = 110$ keys. Most importantly, with this technique we would need a microcontroller with 21 I/O legs to address the rows and

columns.

Discrete Decoder and Encoder

Another technique can reduce the number of I/O legs even further. An *M:N* decoder is a device that takes an M-bit binary input and drives exactly one of its N-output lines. Figure 1.25 shows an example $2-4$ bit binary decoder [1]. The number of binary output lines is really 2^M, so that a 4-bit input will drive 16-output lines.

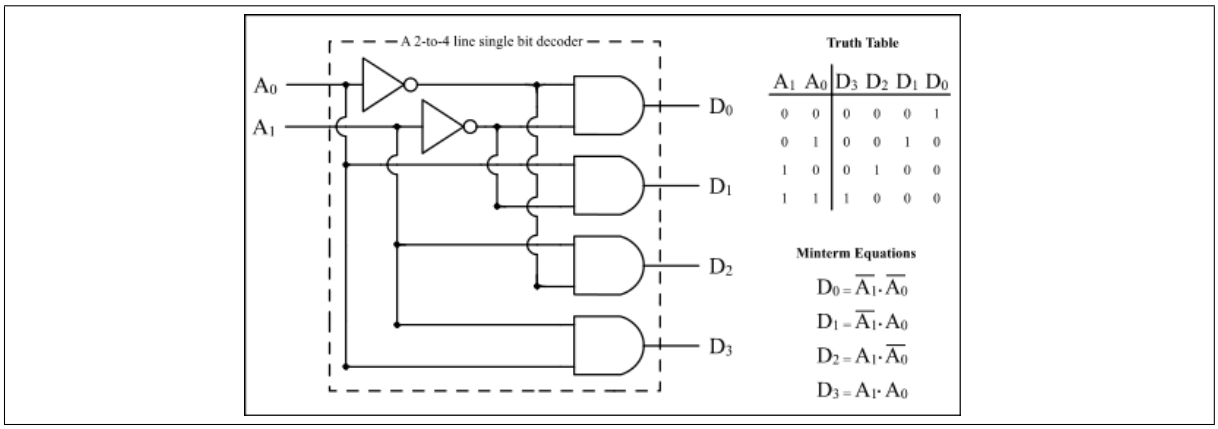

Figure 1.25: A 2-4 line decoder

In the PC keyboard example, if we use a 4:16 decoder it no longer makes sense to balance the rows and columns. Four MCU ports will be converted by the decoder to source voltage on exactly 1 of 16 rows. So, with 16 rows, we need 7 columns to get $16x7 = 112$ possible switches, which is still more than our required 104. This design needs $4 + 7 = 11$ ports on the MCU to control a 104 (or even a 112) key keyboard, which is only three more ports than we used for the 16-key keyboard! The decoder can be very useful for reducing the number of ports required for the MCU, which can possibly reduce the size of the MCU needed for a task (and thus reduce costs for a design) at the cost of adding the decoder, which typically costs less than one dollar US.

1.6 Source Code Quality

The code shown in Listing 1.7 is a faithful implementation of the combinational logic circuit, but does it earn the elusive moniker of "good code?" Unfortunately, microcontroller source code has a well-earned reputation for being of poor quality. There are several reasons for this. One of them is that the nature of microcontroller programming is historically more difficult than conventional programming; although that is changing with the fourth and fifth generation MCUs. Another reason is that many hardware engineers who may have had little interest in programming have been repurposed as programmers. Reconfigurable devices, including MCUs, are replacing the exotic analog circuits of years gone past, and the result is that more and more work is being done with software than with a soldering iron.

Toyota's Unintended Acceleration How bad can this be? The tragedy surrounding the unintended acceleration of some of Toyota's cars sheds some light on the importance of code quality. Starting as early as 2007, reports emerged of some of Toyota's cars experiencing full-throttle acceleration that could not be stopped with the car's brakes. Initially blamed on loose floor mats, Toyota recalled cars and replaced the mats. Reports continued, and eventually there were fatalities. At one point, *tin whiskers*, which can form between the contacts of a switch, were blamed for the problem. By 2010, there were 6200 complains, 89 people were killed, and Toyota was under investigation.

The problem was linked to the software that controls the throttle, which was found to be 1300 lines long and was so poorly structured that it had a McCabe Cyclomatic Complexity score of 146 (there were 146 paths through that block of code), meaning the code was untestable and unmaintainable.

Toyota's code had 11,253 global variables, meaning the values could be read or written by any part of the program, without any controls. The goal should be zero global variables. Even worse, those global variables were *expected* to be modified by other parts of the program but weren't marked as *volatile*, so the readers would never changes from the writers!

MISRA-C, a set of industry standard safety rules for developing embedded systems, forbids using recursive functions, but Toyota did this anyway. The stack for the control system was already 94% full, but then the code made extensive use of recursive function calls. Eventually the stack overflowed and corrupted the embedded real-time operating system's memory.

Ultimately, the engineering team at Toyota made contradictory statements. Team mem-

bers believed others to be responsible for testing, but no one was. Others were not aware of Toyota's own policies. As a consequence, Toyota's public statements and statements to investigators were inconsistent and fraudulent. While we can argue about the culpability of the developers, Toyota as a company faced billions of dollars in fines, legal settlements, and recall fees; and 89 people were killed.

Throughout the rest of this text, there will be an emphasis on developing code that is well-structured. In this chapter, we look at different ways of organizing the code to manage the GPIO ports, and introduce a software metric called *Fragility*.

Fragility

Fragility, in software, describes the concept that poorly structured code is difficult to maintain because a change requires touching the code in many, many places. When code is fragile making a change is likely to require large changes to the code base, and will be likely to introduce new bugs. Programmers are notorious for patching up the code to just make it work, and one consequence is that fragile code tends to beget more fragile code as fixes and patches stack up. Typically, fragility can only be fixed with a major overhaul of the code base.

One aspect of MCU programming that makes it more susceptible to being fragile is that the interface between hardware and software may not be fully controlled by the developers. The mapping of input signals to I/O can change throughout a product's life cycle. In fact, the MCU can change throughout a product's life cycle. Because the code is tied so tightly to the low-level hardware of a specific MCU and the external port mappings, these changes can have major impact on the structure of the code.

Another common programming pitfall is the use of *magic numbers* throughout a program. For example, PIC32's tri-state register used 1 for input, and 0 for output, but not all GPIO module follow that convention. To put a port into input mode, we would typically write: `TRISA = 1;`. In this case, the value 1 is a magic-number. The program doesn't work without it, and its not clear where that came from. If we were to move this code to another MCU, we might have to change hundreds of 1's into 0's to work with the different MCU. Magic numbers are just one example of a common (but bad) programming technique leads to fragility.

We will take another look at the previous combinational logic example, shown again in Listing 1.25. This code used the vendor provided port names and bitfield entries along with the magic numbers to set directionality on the ports and drive the port registers. While this is common practice, it isn't always good practice.

Listing 1.25: Combinational Logic Example

```
#include <p32xxxx.h>

// B0 = Run, B1 = Motor Slow, B2 = Hood Open
// B4 = Motor Power, B5 = Alarm
int main( )
{
    PORTB = 0;

    TRISBbits.TRISB0 = 1;
    TRISBbits.TRISB1 = 1;
    TRISBbits.TRISB2 = 1;
    TRISBbits.TRISB3 = 0;
    TRISBbits.TRISB4 = 0;


    while(1) {
        PORTBbits.RB4 = PORTBbits.RB0 & PORTBbits.RB1 & ~PORTBbits.RB2;
        PORTBbits.RB5 = PORTBbits.RB0 & PORTBbits.RB2;
    }
}
```

Example 1.6.1. The microcontroller example shown in Listing 1.25 had the RUN switch connected to I/O port B0. Due to a hardware change, the port is now switched to I/O port B7. Three lines of code: 9, 17, and 18 need to be changed out of a total of 10 lines code. To make this one change, 30% of the code had to be touched!

■

Named Constants

Listing 1.26 uses the C Pre-Processor to give named constants for the port registers. The code is more readable because it now uses the symbols instead of the port registers. The code is also less fragile. To change a port register, there are only two changes to make: the named port and its associated tri-state buffer configuration. This illustrates a small problem with the microchip libraries that is really more of an annoyance: the port names, e.g. `PORTBbits.RB0` and `TRISBbits.TRISB0` cannot be referenced by the same #define symbol. Because of the tri-state buffer, changing any port register requires changing both the `PORTx` and `TRISx` registers, so this is largely unavailable.

Function Macros

Listing 1.27 moves to the next level in sophistication with the C Pre-Processor. Two *functional macros* are defined, `T` and `P`, that expand to the tri-state and port registers suffixed with the name of the I/O port. For example, `T(B0)` gets expanded by the pre-processor to `TRISBbits.TRISB0`, while `P(B0)` will be expanded to `PORTBbits.RB0`. The symbolic port names now only contain the last part of the port name e.g. `B0`. The advantage to this approach is that the tri-state and port registers can be driven with the same symbolic port name (e.g. lines 19 and 27). The disadvantage with this version of the program is that now the function macros must be used instead of just the symbolic port names, but nothing in the

Listing 1.26: Named values with #define

```
#include <p32xxxx.h>

#define RUN PORTBbits.RB0
#define MOTOR_SLOW PORTBbits.RB1;
#define HOOD_OPEN PORTBbits.RB2;
#define MOTOR_POWER PORTBbits.RB4;
#define MOTOR_ALARM PORTBbits.RB5;

int main( )
{
    PORTB = 0;

    TRISBbits.TRISB0 = 1;
    TRISBbits.TRISB1 = 1;
    TRISBbits.TRISB2 = 1;
    TRISBbits.TRISB4 = 0;
    TRISBbits.TRISB5 = 0;


    while(1) {
        MOTOR_POWER = RUN & MOTOR_SLOW & ~HOOD_OPEN;
        MOTOR_ALARM = RUN & HOOD_OPEN;
    }
}
```

Listing 1.27: Functional defined macros to handle TRIS and PORT registers

```
#include <p32xxxx.h>

#define P(x) PORTBbits.R##x
#define T(x) TRISBbits.TRIS##x

#define RUN B0
#define MOTOR_SLOW B1;
#define HOOD_OPEN B2;
#define MOTOR_POWER B4;
#define MOTOR_ALARM B5;

#define IN 1
#define OUT 0

int main( )
{
    PORTB = 0;

  T(RUN) = IN;
  T(MOTOR_SLOW) = IN;
  T(HOOD_OPEN) = IN;
  T(MOTOR_POWER) = OUT;
  T(MOTOR_ALARM) = OUT;


    while(1) {
        P(MOTOR_POWER) = P(RUN) & P(MOTOR_SLOW) & ~P(HOOD_OPEN);
        P(MOTOR_ALARM) = P(RUN) & P(HOOD_OPEN);
    }
}
```

Listing 1.28: `port.c` - Mapping GPIO port to logical function C implementation

```
/* port.c - implements the input/output to the port. */
#include <p32xxxx.h>
#include <xc.h>
#include "port.h"

// B0 = Run, B1 = Motor Slow, B2 = Hood Open
// B4 = Motor Power, B5 = Alarm

int getMotorSlow()
{
  TRISBbits.TRISB1 = 1;
  return PORTBbits.RB1;
}

int getHoodOpen( )
{
  TRISBbits.TRISB2 = 1;
  return PORTBbits.RB2;
}

void setMotorPower(int state)
{
  TRISBbits.TRISB4 = 0;
  TRISBbits.TRISB4 = state;
}

void setMotorAlarm(int state)
{
  TRISBbits.TRISB5 = 0;
  TRISBbits.TRISB5 = state;
}
```

Listing 1.29: `port.h` - Mapping GPIO port to logical function header file

```
#ifndef PORT_H
#define PORT_H

int getMotorRun( void );
int getMotorSlow( void );
int getHoodOpen( void );
void setMotorPower(const int state);
void setMotorAlarm(const int state);

#endif
```

programming environment will force us to do that. One advantage is that the code is less fragile - changing a port only requires changing the symbolic port name on one line, but is it better?

When choosing a style, there are many things to consider. Using metrics such as fragility can be a good guide, but ultimately, they are only guides. For example, using the *named constants* approach may actually make the code more readable as it does not loose the ability use the port names without the `T` and `P` prefixes, can easily be misused.

Seperation of Functionality

Instead of playing games with the pre-processor, we can turn to Structured Programming and separate the functionality into different parts of the program. In this example, we split the program into two parts - one part that maps the external GPIO ports into an internal logical name, and the main control loop.

Listing 1.28 is its own file, `port.c`, that maps the external GPIO ports to a logical signal.

Listing 1.30: `main.c` - Main combinational logic

```
#include <xc.h>
#include "port.h" // local port implementation

void main( )
{
    while(1) {
    int power = getMotorRun( ) & getMotorSlow( ) & ~getHoodOpen( );
    setMotorPower(power);

    int alarm = getMotorRun( ) & getHoodOpen( );
    setMotorAlarm(alarm);
    }
}
```

The rest of the program doen't need to know that the motor run signal is connected to port B0, only this file does. In fact, as far as the rest of the program is concerned, the port functionality is *opaque*, no other part of the program can see the implementation of the port logic. Since the functionality must be used by other parts of the program (the control logic), a *header file*, shown in Listing 1.29, will be shared across the rest of the program.

Listing 1.30 shows the `main()` function implementation. Like our original example, it includes an infinite while loop, but instead of directly modifying the port registers, this implementation makes calls to the named functions in the `port.c` file.

The implementation is successful in maps the symbolic name (`setMotorAlarm`) to its functionality, but disconnects it from its implementation. The control logic doesn't care how the motor alarm is set, it just needs to be set.

How fragile is this code? Suppose we have to change a port mapping like we did before. We only need to change two lines of code, which is the minimum number of code changes we can possibly make. That is certainly an improvement. Suppose we have to change the combinational logic? This implementation makes that straightforward as well. So, on all of the previous metrics, this is a good implementation.

One argument against using this type of code stems from a mistaken understanding of how function calls are made. The C compiler contains an optimizer that improves the run-time or space used by the Assembled program. One of the optimization steps is called *auto-inlining*, where small functions are replaced by the instructions contained in the function, without actually making a function call. So, for all of these small functions, the C compiler won't make a function call anyway; but the source code is still well organized.

Another advantage to the code in Listing 1.30 is the use of the local variables for the output states. These variables will be allocated and can be inspected during a debug operation; but during an optimized compilation, they will be optimized away. In fact, the resulting assembly for this version will be almost identical to the hand-written assembly presented earlier in this

chapter!

Exercises

Exercise 1.1 (5)(§1.2) The PIC32's output port includes an enable signal, which means it's output can be on or off. Why doesn't the input buffer need a similar enable control signal?

Exercise 1.2 (5)(§1.3) Define *Schmitt Trigger* and explain why it is especially useful for *digital* input. Note: If your answer includes the word *hysteresis*, define it.

Exercise 1.3 (5)(§1.3) Define flip-flop latches and explain why they are used for constructing registers

Exercise 1.4 (5)(§1.3) Define metastability for flip-flops

Exercise 1.5 (5)(§1.3) Find the data-sheet or reference manual for at least two other microprocessors and find their version of the GPIO Port Diagram. Compare and contrast their capabilities.

Exercise 1.6 (10)(§1.3) Find at least two other data-sheets from two other vendor's microcontrollers (e.g. Texas Instruments or AVR), and fill out the table below:

Characteristics	Family:	Family:
Maximum current out of V_{SS}		
Maximum current in on V_{DD}		
Maximum current on an I/O pin		
Voltage on any digital only pin with respect to V_{SS}		
Voltage sourced, logic high		
Voltage sourced, logic low		
Current sink, input port		

Exercise 1.7 (5)(§1.4)

Section 1.5 gave these two examples to change GPIO bits:

```
1 PORTA = 0x8001;
  PORTF = PORTF | (1 << 15) | (1 << 0);
```

They both do something the same, but they also have a (possibly) important difference. Explain.

Exercise 1.8 (5)(§1.4) The GPIO port used two buffers (Figure 1.16), but only the output was a tri-state buffer with an enable signal. Why?

Exercise 1.9 (5)(§1.4) Define high-impedance. Why is it important to ensure that MCU digital input is high-impedance?

Exercise 1.10 (5)(§1.4) In Section 1.4.6 two different SFRs were introduced: `LATx` and `PORTx`. If the port is in output mode (`TRISx = 0`), would there ever be a reason where a read from `LATx` would be different than a read from `PORTx`? Explain your answer.

Exercise 1.11 (10)(§1.4) This question is based on the I/O port diagram in Figure 1.1. The following table lists the control signals for the I/O port, and then lists a series of instructions. Show the values for each of the control signals as the processor executes each instruction. For all signals except the I/O pin, use 1 for high/true, 0 for low/false, and X for don't care (doesn't matter). For the I/O pin, use HI for source voltage, 0 for sink voltage, Z for hi-impedance, IN when reading input, ? for don't know. For each of instruction, assume only the port that corresponds to the least significant bit of the value being written. As an example, the values for `TRISA = 0` are filled into the table.

State	Data Bus	RD PORTx	WR PORTx	RD LATx	WR LATx	RD TRISx	WR TRISx	RD ODCx	WR ODCx	SLEEP	I/O PIN
TRISA = 0	0	0	0	0	0	0	0	0	0	0	?
`PORTA = 1;`											
`PORTA = 0;`											
`TRISA = 1;`											
`int x = PORTA;`											
`LATA = 1;`											
`ODCA = 1;`											
`LATA = 0;`											

Exercise 1.12 (5)(§1.4) Suppose you are working to design this microprocessor. Your task

is to set the initial state of the processor after a power-on-reset (POR). Since you cannot make any assumptions about what the I/O ports are connected to, we have to make sure that the initialization state is safe - it won't over-load input or output modules. What values would you set in each of the latches in Figure 1.1?

Exercise 1.13 (5)(§1.4) Write the instructions, in the right order, to take the port from its default power-on state (see previous exercise) to driving a logic high on the I/O pin.

Exercise 1.14 (5)(§1.4) Recall from Figure 1.1 that the input value goes through a pair of flip-flops that are trigged by `SYSCLK`. Suppose `SYSCLK` is running at $40\,\text{MHz}$. What is the maximum *latency* for an input value to be available on the data bus? With the linked flip-flops, is it possible that the microcontroller could see a change every clock cycle?

Exercise 1.15 (10)(§1.5) Suppose a PIC32 MCU has a SYSCLK at $20\,\text{MHz}$. Experimental data shows that longest observed time for a given switch to settle is $220\,\mu\text{s}$. Using either the *disassembled* MIPS implementation of the code in Listing 1.22, or write your own Assembly routine (provide the code either way) and determine what the minimum value for `DEBOUNCE_TIME` to ensure sufficient time for the switch to settle.

Bibliography

[1] Decoder. URL http://en.wikipedia.org/w/index.php?title=Decoder&oldid=621948206. Page Version ID: 621948206.

[2] Schmitt trigger. URL http://en.wikipedia.org/w/index.php?title=Schmitt_trigger&oldid=606157100. Page Version ID: 606157100.

[3] Microchip Technology, Inc. I/O Port, . URL http://ww1.microchip.com/downloads/en/DeviceDoc/61120E.pdf.

[4] Microchip Technology, Inc. PIC32MX Family Datasheet, . URL http://ww1.microchip.com/downloads/en/DeviceDoc/PIC32MX_Datasheet_v2_61143B.pdf.

Index

Architecture
    Special Function Register, 32, 33

Black-Box Equivalence, 40

Combinational Logic, 37

Fragility, 60

General-Purpose I/O, *see* GIO1
GPIO, 1
    Debounce, 52
    Delay Loops, 46
    Electrical Characteristics, 10, 13
    High-Impedance Input, 12
    Input, 21
    Latency, 48
    Low Output, 24
    Multiplexing, 54
    No Response, 24
    Open Drain Configuration, 35, 36
    Open-Drain Control, 33
    Output Buffer, 20
    Output Port, 33
    Programming, 37
    Pulse Width Modulation, 47
    Random Values, 24
    Sinking Current, 20
    Square Wave Output, 43
    Stuck Bit, 24
    Sudden Reset, 23
    Tri-State Buffer, 21, 32
    TTL, 22
    Voltage Source, 19

LED
    Seven-Segment Display, 42

Metric
    Fragility, 60

Special Function Register, 21
Switch
    Debounce, 52
    Multiplexing, 54