General Purpose Input/Output

Prof. John McLeod

ECE3375, Winter 2022

This lesson continues the discussion of assembly language, with specific emphasis to the code base for the ARM®Cortex-A9 processor. The basics of general purpose input/output is discussed. This is of limited utility for class-room demonstration, as I can't plug a cool custom peripheral into an online simulator, but it is an important topic to understand none-the-less.

Memory Mapped Peripherals

The ARM®Cortex-A9 is a **memory mapped** microcontroller. This means that all peripherals attached to the microprocessor are assigned a fixed address in memory space. Controlling or receiving input data from these peripherals is therefore as simple as writing to, or reading from, that memory address. For example, consider the DE1-SoC development board that uses the ARM®Cortex-A9.

- There are 10 LEDs assigned to address `0xFF200000`. Each of these LEDs is equivalent to a single bit in the 4-byte (32-bit) memory block starting at address `0xFF200000`. [1] To light up the 3 right-most LEDs, simply write `0x7` (i.e., `0b111`) to that address. These LEDs will remain illuminated until another value is written to the LED band address, or until the microcontroller is powered off.
- There are 10 switches assigned to address `0xFF200040`. Again, each of these switches is equivalent to a single bit. To accept toggled switches as input to your program, simply read from that address.

Most peripherals are more complicated than LEDs and switches, and will have some **structure**. This means that more than one word in memory is assigned to that peripheral, and different words have different purposes. Finally, it is common to refer to the different memory addresses of the peripheral as *registers*, even though they may not have much in common with the registers in the CPU.

[1] Because a **word** is 32 bits, each peripheral is assigned am integer multiple of 32-bits in address space — whether or not that much space is needed — so writing or reading a register's worth of data to or from that peripheral doesn't "leak" out into adjacent memory space.

Input/Output Pins

All **memory mapped** peripherals built into the microcontroller chip have the same basic connection structure as the memory cells discussed previously.

- The peripheral is physically **enabled** when the **address bus** activates the appropriate **minterm** that corresponds to the pre-defined memory address of that peripheral.
- Additional logic will combine the addressing minterm with some control flags to distinguish between reading and writing to that peripheral.
- The **data bus** will also connect to that peripheral.

An example of a simplified **output** port is shown in Figure 1. This is representative of any peripheral that can only accept output from the microcontroller — for example, the LED bank mentioned above. Figure 1 is a generic output port, as just an output pin is shown instead of a hardware peripheral.

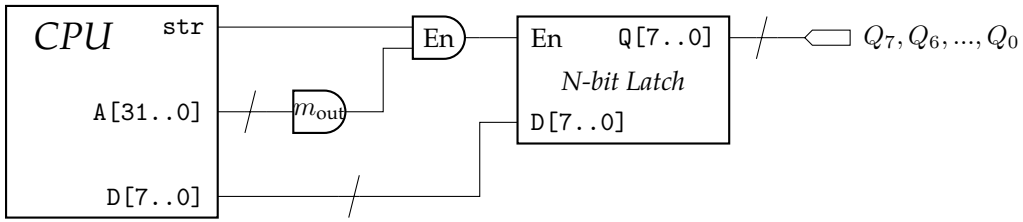

Figure 1: Simplified schematic of an 8-bit output port.

- Here the peripheral is enabled by supplying both the correct address to the address bus, and a control bit indicating that a `str` instruction was provided.

- A flip-flop or latch is used (actually 8 parallel latches, as there aree 8 data lines) to control the output instead of a tri-state buffer. This way, when output is *not* enabled, the value on the $Q_7, ..., Q_0$ lines persists as the last value written. [2]

An example of a simplified **input** port is shown in Figure 2. This is representative of any peripheral that can only accept input from the microcontroller — for example, the set of switches mentioned above. Again, Figure 2 represents a generic input port, as just an input pin is shown instead of a hardware peripheral.

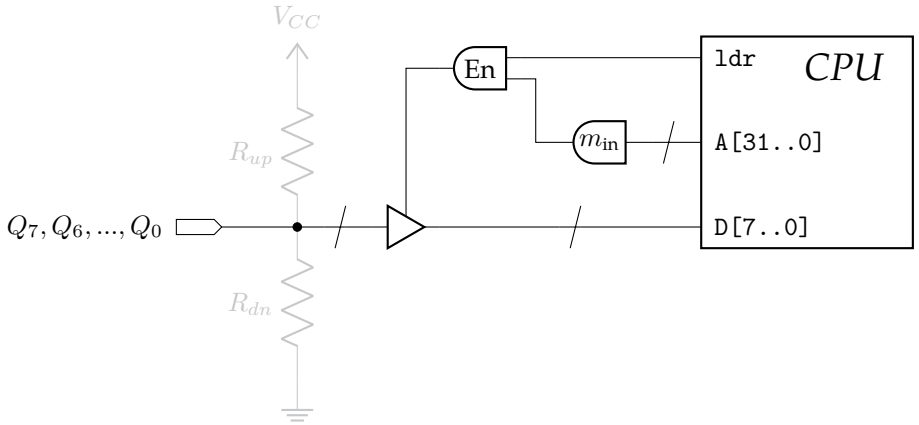

Figure 2: Simplified schematic of an 8-bit input port. Either a "pull-up" resistor R_{up} or a "pull-down" resistor R_{dn} is used to ensure a digital input is present even when the input pin is floating.

- Again, the peripheral is enabled by the correct address and a control bit indicating that a `ldr` instruction was provided.
- Here a tri-state buffer is used (actually, set of 8 parallel tri-state buffers, as there are 8 data lines) to separate the data bus from the input port. Note the orientation of the tri-state buffer allows the external value $Q_7, ..., Q_0$ to be placed on the data bus, but not vice-versa.

[2] This is evident when using the online-simulator: After writing a value to the LED bank and illuminating some LEDs, those LEDs remain illuminated while the program continues, until a subsequent write to the LED bank changes the values.

- This circuit schematic assumes the input from the peripheral is already **digital**.

A digital input port needs either a **pull-up** or a **pull-down** resistor, otherwise the result of a read operation is unpredictable if the microcontroller attempts to read from a floating port. The difference between pull-up and pull-down is basically a matter of taste: with a pull-down resistor an unconnected pin will always read zero, while with a pull-up resistor an unconnected pin will always read one. Either state is fine, and simply needs to be accounted for in software when probing that port. The only situation in which there is a difference between pull-up and pull-down is when there is an external signal at that port.

- Whenever the signal is the complement of the normal floating state of the port, [3] a current will flow across the resistor.
- To minimize power loss, the resistances used are usually relatively large.
- However, if you know that the external signal typically "rests" in a particular digital state, choosing a microcontroller with the equivalent input will minimize power loss. A peripheral that only rarely measures a signal, and is **active high** (i.e., zero is the "off" or "resting" state), can cause a lot of unnecessary power loss when connected to a pull-up input port.

[3] So whenever a signal of 1 is sent to a pull-down port, or a signal of 0 is sent to a pull-up port.

Another consideration is if the external signal comes from a circuit with a large intrinsic capacitance. In that case the combination of large capacitance and large pull-up/pull-down resistance creates a *slow transient*. This configuration can take a long time (in terms

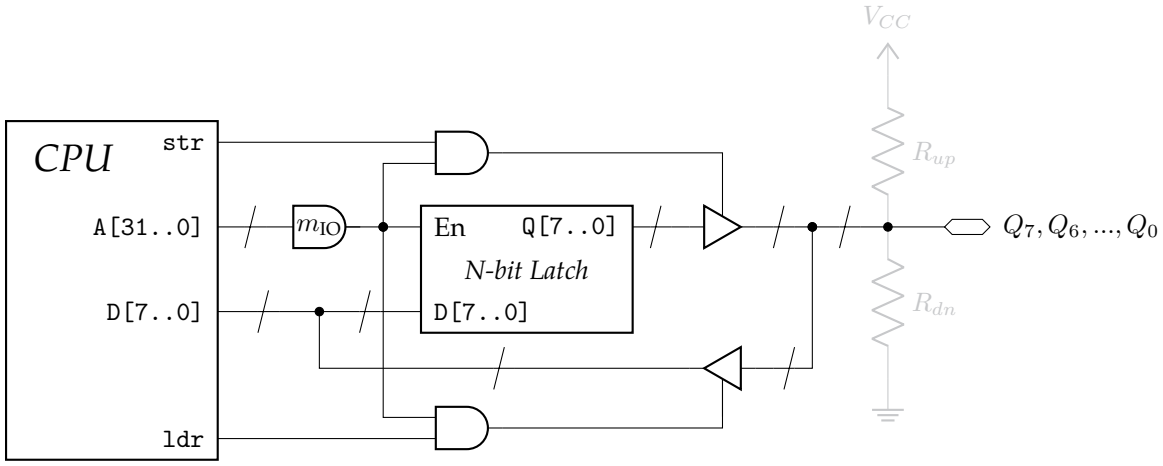

Figure 3: Simplified schematic of an 8-bit bidirectional port. Either a "pull-up" resistor R_{up} or a "pull-down" resistor R_{dn} is used to ensure a digital input is present even when a floating pin is configured as input.

of microcontroller processor speeds) to settle to a "good" digital value.

Many of the ports on a microcontroller are configured to be **bidirectional**: able to act as either input or output ports, and configurable in software. Although the LED bank and set of switches previously discussed as peripherals obviously only have output and input functionality, [4] respectively, these peripherals are not physically on the microcontroller chip — rather they are connected to some ports on the microcontroller. Making particular ports limited to input or output by the physical hardware present limits the functionality of the microcontroller, and so is something a manufacturer would typically try to avoid.

An example of a simplified **bidirectional** port is shown in Figure 3. This circuit assumes that the CPU ensures that only one of the `str` and `ldr` control bits can be set at a time.

- The `str` and `ldr` control bits cannot both be set without mak-

[4] I tested it out on the simulator: you actually can read from the LED bank — it just returns the state of the LEDs. So basically (in the simulator, anyway, I don't know about the actual hardware) the LED bank acts as a memory cell that can light up. However, while you can write to the switches in software without triggering an error message, it doesn't do anything.

ing both tristate buffers active — if the peripheral is providing an input $Q_7, Q_6, ..., Q_0$ at the same time the Q_{out} flip-flop is providing output to the pin, bad things can happen.

- Otherwise, the Q_{out} flip-flop preserves the state in between `str` operations while the pin is acting as an output, and the pull-up/pull-down resistors ensure that a floating pin provides a digital signal when acting as an input.
- The pull-up/pull-down resistors will drive some (unnecessary) current when the pin is acting as an output. Additional tri-state buffers could be added to block this, but usually aren't — the complexity (and intrinsic power loss) of additional tri-state buffers exceeds the cost of some parasitic power loss.

General Purpose Input/Output

As mentioned above, we refer to the memory addresses assigned to memory-mapped peripherals as *registers*. There are three basic kinds of registers for an peripheral.

Definition: A **status register** is the part of the peripheral that indicates the current operating status of that peripheral. Status registers are typically read-only.

Definition: A **control register** is the part of the peripheral that defines the present operating method. Control registers are typically write-only.

Definition: A **data register** is the part of the peripheral that accepts or produces data. These are usually read/write.

Depending on the complexity of the peripheral, it may have some or all of these types of registers, and may have more than one of each type.

Most external peripherals connect to a microcontroller the same way: through a general purpose input/output port. These ports have a simple standard architecture.

- The physical pins in the port on the microcontroller chip are mapped to a **data register** in memory space, with at least one bit in memory for each pin on the port.
- A **control register** of the same width is mapped in memory space. This register allows the *individual pins* on the port to be set to input or output. Usually this register is a word higher (or lower) than the **data register** in memory space.

- Usually these registers are addressed by word, even though the physical port is often less than 32 bits.
- The general purpose input/output control registers on the DE1-SoC board interpret bitwise values of 1 to set the corresponding pin as an output, and a value of 0 to set the corresponding pin as an input.

Since everything is done in the simulator there isn't really the option to plug in cool custom peripherals this year, so these details are of limited importance. However, the basic concept of reading/writing *individual bits* is useful for both general purpose input/output and for specific peripherals.

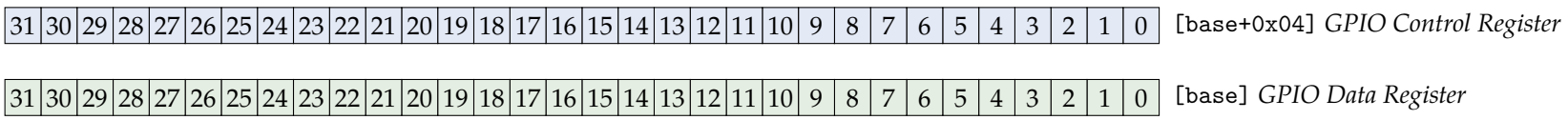

Figure 4: Structure of an ARM GPIO port.

The structure of a GPIO port on an ARM system is shown in Figure 4. As mentioned above, for a generic microcontroller you may expect the control register to be above or below the data register. For an ARM system the control register is above the data register. The simulator has two GPIO ports mapped to parallel ports (at addresses `0xFF200060` and `0xFF200070`).

- On the simulator, these ports are represented by 32 check boxes.
- When the check boxes are dark black, they are configured as input. Input can be set by checking/unchecking each box.

- When the check boxes are greyed out, they are configured as output. The output will be shown as a checked/unchecked box.

I do recommend playing around with the simulator GPIO port if you are unsure of any of the conceptual details discussed in this lesson.

Example: Consider an 8-bit GPIO port that is connected to a peripheral that is a mix of LEDs and push buttons. The top four bits are connected to LEDs, and the bottom four bits are connected to push buttons.

- Assume the peripheral is constructed in a reasonably idiot-proof fashion. Attempting to read from an LED always returns zero, and attempting to write to a push button does nothing.

To properly use this port, we should set the **control register** to `0x1111 0000`, as shown in Figure 5. This defines all of the LEDs as output and the push buttons as inputs.

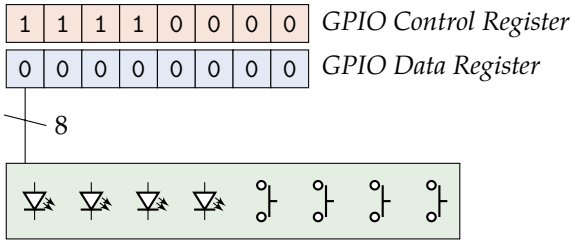

Figure 5: A mixed LED/push button peripheral, connected to a GPIO port with a properly-defined control register.

Now assume you did not listen very well during this lesson (hard to imagine, I know!), and instead set the **control register** to `0x1010 1010`.

- You press down on all the switches.
- The control register has only assigned two switches as input, and the peripheral always returns zero from the LEDs improperly set to inputs.
- Therefore, only the value `0x0000 0101` is returned, rather than the expected `0x0000 1111`.

This is shown in Figure 6.

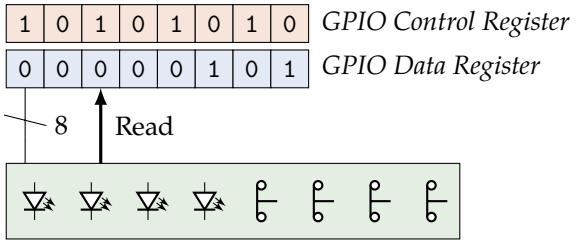

Figure 6: A mixed LED/push button peripheral, connected to a GPIO port with a poorly-defined control register. When attempting to read, only two switches are properly configured.

With the same poorly-set **control register**, you now try to illuminate all the LEDs by writing `0x1111 1111` to the peripheral.

- Writing a value to a push button does nothing, regardless of whether or not the pin is set as output.
- Only two of the LEDs are set as output, so only two LEDs will light up.

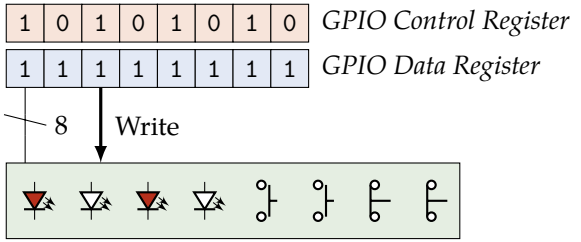

Figure 7: A mixed LED/push button peripheral, connected to a GPIO port with a poorly-defined control register. When attempting to write, only two LEDs are properly configured.

Seven-Segment Displays

A seven-segment display is nice example of an output-only peripheral that is slightly more complicated than a bank of LEDs. The seven-segment display is operated exactly the same way a bank of LEDs is operated — by using `str` instructions to write data to the device.

- A seven-segment display has only **data registers**. The number of data registers present depends on the number of panels in the display.

The complexity with the seven-segment display lies in *how* the data is displayed.

- A set of seven-segment displays are typically used to display human-readable numbers in decimal or hexadecimal.
- Each individual segment in each seven-segment display panel is controlled by a single bit.
- Consequently, the *bit sequence* required to display a particular digit is different than the *binary representation* of that digit.

The DE1-SoC board has a 6-panel seven-segment display. Each individual panel is controlled by a single byte, as shown in Figure 8

- Since each panel has only seven segments (obviously), and each segment is toggled by one bit, there is an extra bit available that does nothing. For the DE1-SoC, this is the MSb of the byte.

As each panel uses a byte, the 6-panel display therefore requires six bytes in address space.

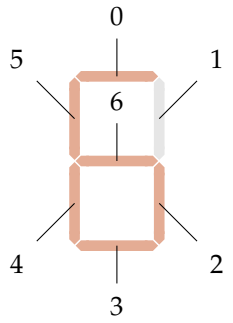

Figure 8: The mapping between the control bits and the segments on the display. The MSb control bit, 7, is unused.

- The base address of the seven-segment display is `0xFF200020`.
- The first four panels in the display map to the first word at this address.
- Somewhat confusingly, the remaining two panels *do not* map to the subsequent bits — rather they start at `0xFF200030`.

The online simulator will allow you to write to the display byte-by-byte, but it will also throw up a lot of warnings that the real device may not allow bytewise access, and writing to the first four panels with a single word is preferred.

- Remember that the output is persistent. To clear the seven-segment display, write zeros to both `0xFF200020` and `0xFF200030`.

Example: We want to display our feelings about this course to the seven-segment display: "AAAAAH". This is shown in Figure 9.

Figure 9: Using the 6-panel seven-segment display to express your frustration with this course.

- The letter "A" can be written on a display panel by turning on every segment except the bottom one (bit 3, as shown in Figure 8). As there are only seven segments but eight bits in a byte, the MSb is unused and irrelevant, so "A" can be represented by `0b0x0111 0111` = `0x77`.
- The letter "H" is similarly displayed by turning on every segment except the top and bottom, so it can be represented by `0b0x0111 0110` = `0x76`.
- To write to the display in units of **words**, we should therefore write `0x00007777` to the two left-most panels at address `0xFF2000030`, and `0x77777776` to the four right-most panels at address `0xFF200020`.

The code is shown below in Assembly.

```
ldr r0, =0xFF200020 @first 4 7seg panels
ldr r1, =0x77777776 @AAAH in hex
ldr r2, =0x7777     @AA in hex
str r1, [r0]        @write to 7seg
str r2, [r0, #16]
```

Bit Masks

The bits in registers or memory cells cannot be accessed individually. Usually this isn't an issue, but often the control and status registers of peripherals have distinct meanings for individual bits, and it is useful to be able to write or read to these bits without influencing the others. This is where the concept of **bit masking** is helpful.

Definition: A **bit mask** is a sequence of bits that is designed to be used with a logic operation to isolate only particular bits in a register.

The choice of logic operation depends on what is needed. Consider a bit x with an unspecified value:

- The `and` logic operation can be used to **clear** x to zero: $x \cdot 1 = x$, while $x \cdot 0 = 0$.
- The `or` logic operation can be used to **set** x to one: $x + 1 = 1$, while $x + 0 = x$.

These logic operations, combined with the appropriate bit mask, allow us to do the basic task of setting or clearing an individual bit while leaving all other bits untouched.

Example: The "self-destruct activated" flag is bit 4 of the **status register** for the *self-destruct* peripheral that comes standard with every DE10-Standard board. [5] Before students are ready for in-person labs, it is important they know how to check this flag.

- To complicate matters, each of the 32 bits in the the status register for the self-destruct is an independent flag for one purpose or another, which may have any arbitrary value during normal operation.
- To check the self-destruct flag, we need to load the contents of the status register into a general CPU register. Then **mask** this value so that all bits except bit 4 are zero.

One implementation of this in Assembly is given below.

```
@ read in the status register
ldr r0, self_destruct_sr
ldr r1, [r0]
@ prepare a bit mask, where only
@ bit 4 = 1, all others are 0
mov r2, #1
lsl r2, #4
@ mask the bits
and r1, r2
@ check result
cmp r1, r2
@ get ready to run!
beq get_outta_here
```

[5] Note that this peripheral is very poorly documented, so don't bother looking for it in the manual.

It is worth pointing out that bitmasking operation itself can also be used to update the status flags by using `ands` instead of `and`. However now the `Z=1` condition applies if the fourth bit of the self-destruct peripheral's status register is *not set*.

```
@ mask the bits and update flags
ands r1, r2
@ get ready to run!
bne get_outta_here
```

Example: The "evil flag" is bit 13 of the **control register** for the *moral compass* peripheral that comes standard with every DE10-Standard board. [6] By default this flag is set, it is important to clear it if you want your program to run without adding to the sum of human misery.

- Again, each of the 32 bits in the control register for the moral compass is an independent control of one purpose or another, which may have any arbitrary value during normal operation.
- To clear the evil flag, we need to write to the control register with the *existing contents* of every bit *except* with bit 13 cleared to zero.

One implementation in Assembly is given below. Note the use of *rotate right* to obtain a bit sequence of all 1's except for a zero at bit 13 (a *right* rotate of $32 - 13 = 19$). [7]

```
@ read in the control register
ldr r0, moral_compass_cr
ldr r1, [r0]
@ prepare a bit mask, where all bits are 1
@ except bit 13
mov r2, #0
sub r2, #2
ror r2, #19
@ clear the evil bit
and r1, r2
str r1, [r0]
```

[6] This is another peripheral that is, bafflingly, very poorly documented!

[7] There are other alternatives to obtaining a sequence of mostly 1s with a few 0's at select bits, such as using an *exclusive or* (`eor`) or the *bitwise clear* (`bic`) mnemonics.

Example: The DE10-Standard board has a Cyclone V chipset that includes an FPGA on one side and a dual-core ARM®Cortex-A9 on the other side. This chip is physically unbalanced, as the FPGA is heavier than the CPU. Good programming practice is to keep *weighted registers* in the CPU to help balance it out: These are registers with mostly 1s instead of 0s. [8]

- The DE10-Standard board has a bank of 10 switches memory mapped to `0xFF200040`, the top 22 bits of this register are unused since they do not map to any physical switch.
- If we always keep the top 22 bits recorded as 1s every time we read from the switches, it will help balance the chip. [9]

One implementation in Assembly is given below. Here I use an *exclusive or* to obtain a bit sequence of all 1s: for some reason, the mnemonic `eor` is used for *exclusive or* instead of `xor`.

```
@ the address of the switch data register
ldr r0, sw_base
@ prepare a bit mask
mov r2, #-1     @32-bits of 1
ldr r3, =0x3ff  @10-bits of 1
eor r2, r3
@ read from switches
ldr r1, [r0]
@ make register heavier
orr r1, r2
```

[8] Most textbooks and reference manuals don't bother explaining this, as it is simply common sense that a binary 1 is heavier than a binary 0.

[9] Again, it is plain and simple common sense that a MSb 1 is heavier than a LSb 1.

Example: The DE10-Standard board has a bank of 10 LEDs memory mapped to `0xFF200000`. It is widely known that the lucky 7th LED is the "lighthouse" — its brilliant red glow guides students to success in their laboratory. [10]

- Whenever your write a program that uses the LED bank, you should always make sure to keep the lighthouse LED on.
- This means whenever you write to the LED bank, you should always keep bit 7 set.

One implementation in Assembly is given below.

```
@ the address of the LED data register
ldr r0, led_base

@ prepare a bit mask where all bits
@ are 0 except bit 7
mov r2, #1, lsl #7

@ r1 already contains the LED pattern
@ but make sure bit 7 is always set
orr r1, r2
@ now write to the LED bank
str r1, [r0]
```

[10] Although this is widely known, as you may have guessed it is very poorly documented.

Example: As a less-silly example, consider an 8-pin GPIO port, where pins 2 and 6 are bidirectional. The remaining pins are either input or output-only. Also make the following assumptions:

- The **control register** for this port was previously appropriately defined for all pins except 2 and 6.
- Pins 2 and 6 act in tandem: we need to either toggle both to accept input, or both to accept output, without changing the state of the other pins.

Assuming that the base address of the GPIO port is memory-mapped to `GPIO_base`. This port has only 8 pins (a byte of data), but as with other peripherals each register is memory mapped to an entire word. [11] Also, remember that in the ARM-family of microcontrollers the control register for a GPIO port is at the base address $+4$.

- At some point in the source code, the control register needs to be set in order to define whether pins 0, 1, 3, 4, 5, and 7 are input or output.
- We are not told what those pins do, so we'll just assume they are correctly set.

A suitable bit mask for this port is `0b0100 0100` $= 68_{10}$. To read data from this port, we should define pins 2 and 6 as input, which means we should *clear* control bits 2 and 6, and leave all other bits the same.

```
@ get the hardware address
ldr r4, GPIO_base
@ get the bitmask
```

[11] Similar to how there are only 10 LEDs on the DE10-Standard, but are memory mapped to a 32-bit word.

```
mvn r1, #68  @ 0x0100 0100
@ apply bitmask to control register
ldr r0, [r4, #4]
and r0, r1
str r0, [r4, #4]
@ read data from GPIO port
ldr r0, [r4]
```

Notice how the logical compliment of the mask was used by the `mvn` ("move negative") mnemonic.

To write data to the port, we should define pins 2 and 6 as output, which means we should *set* control bits 2 and 6, and leave all other bits the same. Assume the data to be written to the port is in register `r1`.

```
@ get the hardware address
ldr r4, GPIO_base
@ get the bitmask
mov r1, #68  @ 0x0100 0100
@ apply bitmask to control register
ldr r0, [r4, #4]
orr r0, r1
str r0, [r4, #4]
@ write data to GPIO port
str r1, [r4]
```

Here we use the original mask.

Debouncing

Consider the simplified input port with a push button switch for the external peripheral, as shown in Figure 10. This peripheral uses an 8-bit data bus, and is constructed such that Q_7 always reads 1, $Q_5, ..., Q_0$ always read 0, and Q_6 reads 1 when the button S is pressed.

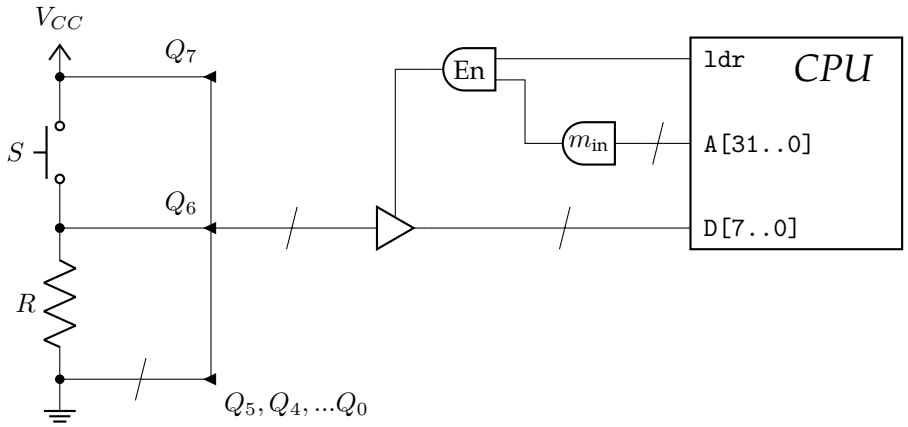

Figure 10: Simplified schematic of an 8-bit input port with a push button switch as a peripheral. A pull-up or pull-down resistor may also be added between the peripheral and the tri-state buffer, but that has been omitted in this diagram.

The follow code could be used to test when S is pressed. [12]

```
ldr r0, adr_button  @memory address for button

@ loop until button is pressed
push_button_input:
    ldr r1, [r0]
    cmp r1, #192
    bne push_button_input
```

[12] Note that `#192` is `0b11000000`.

But what will this code actually do? The CPU clock is $> 800\,\text{MHz}$: the time it takes you to physically push the button is an eternity for the CPU. Furthermore, reading a signal of `0b11000000` from the peripheral does not necessarily mean the button was actually pressed — it could be a random electrical oscillation. It is important to remove any **bounce** from peripherals — especially mechanical ones.

Definition: **Bouncing** is electromechanical noise in the system that could be interpreted as a meaningful signal if the CPU is acting too quickly.

A simple method of **debouncing** might be to continuously measure the button for a physically-relevant interval of time (say, $100\,\mu\text{s}$). If the button registered as "pressed" for 75% (say) of that interval, then it probably actually was physically pressed. One possible implementation of this is given below.

```
ldr r0, adr_button  @memory address for button
/* assume we have a timer peripheral that is
   initialized to a 100 us interval */

@ some threshold for the button
ldr r5, push_threshold

@ loop until button is pressed
push_button_input:
   @ initialize counter
   mov r1, #0
```

```
    /* code to start timer */
    /* timer starts counting to 100 us */
timer_loop:
    @ read button
    ldr r3, [r0]
    @ isolate button state (bit 6)
    and r3, #64  @2^6=64
    lsr r3, #6
    @ add button state to counter
    add r1, r3

    /* loop until timer is done */
    /* need to check timer status register
       to see when 100 us is done */
    bne timer_loop

    @ now check if button was pushed enough
    cmp r1, r5
    blo push_button_input
```

Note: I cribbed these lesson notes from some other material that I had previous prepared for an M.Eng. class. In the video lesson it shows the example code that explicitly controls the timer. We have not studied timers yet, so I removed that part of the code from this note. Hopefully that clarifies the code for debouncing.

Debouncing can also be done in hardware. A simple low-pass RC filter can remove most electrical noise from an input signal, and a Schmitt trigger can clean up the resulting analog signal, as shown in Figure 11.

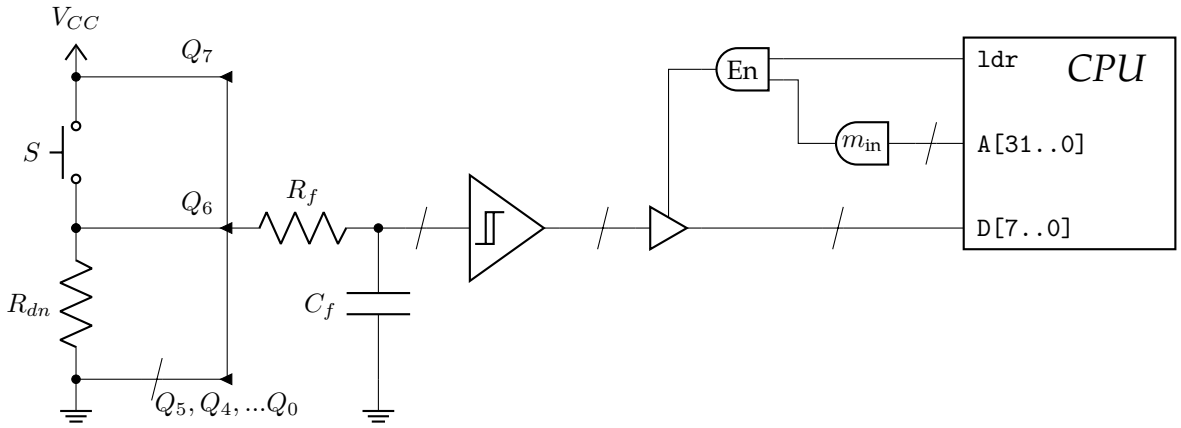

Figure 11: Simplified schematic of an 8-bit input port with some hardware debouncing. A low-pass filter $R_f C_f$ removes high-frequency noise, and a Schmitt trigger digitizes the input to V_{CC} or 0.

Sometimes a combination of hardware and software filtering is also necessary, depending on the peripheral and how important it is for the microcontroller to receive accurate input.

- If the input peripheral is a smoke detector, which is output to an annoying buzzer, then you can get away with minimal (or no) filtering: the buzzer may only "beep" quickly if it is randomly triggered by high-frequency noise.
- If the input peripheral is a push button, and when pressed the microcontroller will launch all of the nuclear weapons, then some filtering is probably a good idea.

Often, the input port of a microcontroller may have some hardware filtering built-in. Schmitt triggers are particularly common for input ports that are supposed to accept a digital signal.

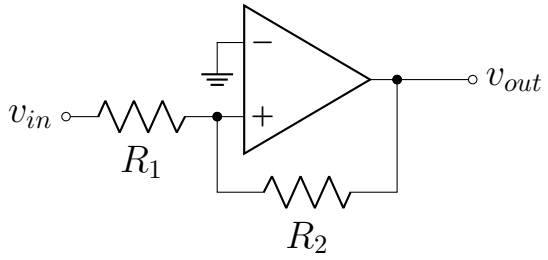

Figure 12: One way to implement a Schmitt trigger is with an op amp in which the feedback resistor R_2 is connected to the $+$'ve input terminal (instead of the usual $-$'ve input terminal for amplifier circuits). Electrical instability causes the output voltage to be pushed up (or down) to the power supply limits of the op amp. If the op amp has a high voltage of V_{CC} and a low voltage of 0, the output v_{out} is basically digital. The finite response time of the op amp also serves to filter out some electrical noise.

GPIO Peripheral: Joystick

Just about any digital-output peripheral can be connected to a GPIO port. One example of such a peripheral is a joystick.

- The simplest joystick is a shaft that connects to four or five contact switches at the base.
- When a user steers the shaft around, one or more switches may be closed (see Figure 13).
- Reading from the GPIO pins informs the microcontroller which in which direction the user is pushing the joystick shaft, allowing the program to adjust accordingly.

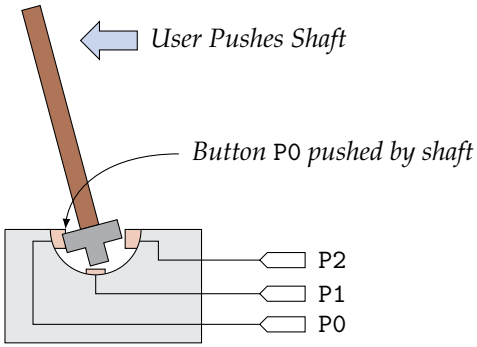

Figure 13: Visual representation of a simple joystick. The relative position of the shaft depresses various push buttons. By reading the input from these push buttons, the microcontroller knows the relative position of the shaft.

A simple joystick is commonly included on some variants of development boards that use a Cortex-M series CPU (such as the STM32L4, discussed in the textbook and in Prof. Reyhani's slides). Although this hardware isn't available on the DE10-Standard, since it interfaces with GPIO it is simple to write a software driver that interacts with the GPIO pins *as if* they were connected to the joystick.

Typical joystick hardware [13] has six pins: one to connect to the power supply, four switches oriented in the *up*, *down*, *left*, and *right* directions, and a fifth switch connected to a button on the shaft (called the *center*). These kinds of joysticks are called "4-way directional with center select".

By way of example, consider this genius idea: a smart garage door-opener that uses a joystick so the garage door isn't just moved up or down, but also side to side! Pretty awesome, eh? I'll use a JS 5208

[13] See Digikey's list of navigational switches, joysticks for a rather long list of options.

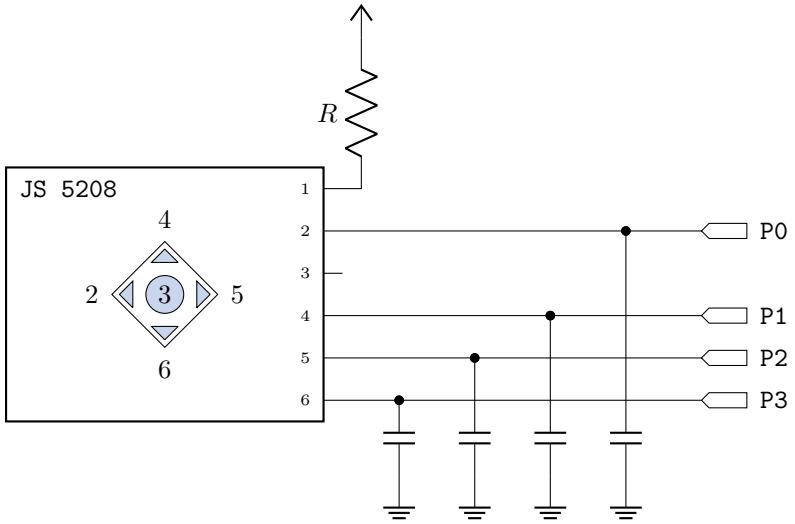

Figure 14: Example "4-way directional with center select" joystick peripheral, connected to four GPIO pins. Here the center select output is left floating.

switch as the joystick. In this hardware, pin 1 connects to the power supply, pin 2 connects to the left button, pin 3 connects to the center button, pin 4 connects to the up button, pin 5 connects to the right button, and pin 6 connects to the down button. [14] For my design, I'll ignore the center button: this garage door opener will spring into action the instant the joystick is moved. Perhaps later designs will use the center button to active a "turbo" feature to make the door move faster.

This hardware can be connected to a GPIO port as shown in Figure 14. One of the reasons for using only the four directional inputs is that this hardware can be easily faked using the four push buttons on the DE10 Standard. Software can be written to identify the direction of the joystick by checking which pin has an active input.

[14] See the datasheet for the JS 5208 on Digikey. It is important to stress that this is just one example, different hardware has different pin assignments. In particular, the joystick on the STM32L4 discovery board is different — see Prof. Reyhani's slides.

Assuming the address of the joystick GPIO port is in `r0`, the following subroutine in Assembly code checks the direction.

```
check_joystick:
  @ get joystick input
  ldr r1, [r0]
  @ left is wired to pin 0, use bitmask 0b0001
  tst r1, #0b0001
  bne move_left
  @ right is wired to pin 2, use bitmask 0b0100
  tst r1, #0b0100
  bne move_right
  @ up is wired to pin 1, use bitmask 0b0010
  tst r1, #0b0010
  bne move_up
  @ down is wired to pin 3, use bitmask 0b1000
  tst r1, #0b1000
  bne move_down
exit_joystick:
  bx lr

move_left:
  @ do something
  b exit_joystick
@ and so on... for remaining directions
```

Note the use of `tst r1, #`⟨*bitmask*⟩ to perform the logical `AND` of the data with the bitmask, and then `bne` ⟨*label*⟩ to branch if the result is **not** zero — i.e. if the appropriate bit in `r1` is set to 1.

GPIO Peripheral: Keypad

Another common peripheral is the 12-button keypad. This is basically a set of push buttons, and as such could be implemented as a 13-pin device: one pin for the power supply, the other 12 for the buttons. There are some hardware implementations that use this method of wiring the buttons to the pins, but most implementations use a "matrix read-out" method for connecting the pins. This approach requires 7 (or 8) pins.

- There is one pin for each of the rows in the keypad (usually 4).
- There is one pin for each of the columns in the keypad (usually 3).
- Some keypads are **active** devices and will have an additional pin for a power supply.

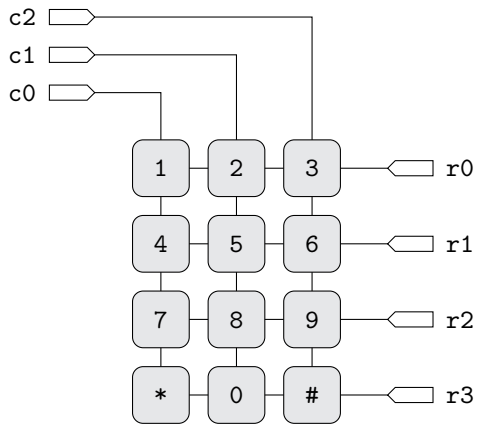

Figure 15: Typical keypad with matrix read-out.

Pressing one of the buttons shorts the corresponding row and column pins together. The software interface for the keypad needs to scan the row/column combinations to determine which pin is pressed. We'll look at this peripheral in more detail: because the software driver needs to combine input and output for a single device, this is an excellent educational example of using GPIO ports. First, consider a passive keypad that does not have a power supply pin.

The circuit diagram for a passive 12-button keypad is shown in Figure 16. The microcontroller typically writes **output** to the row pins and reads **input** from the column pins. The following assumes

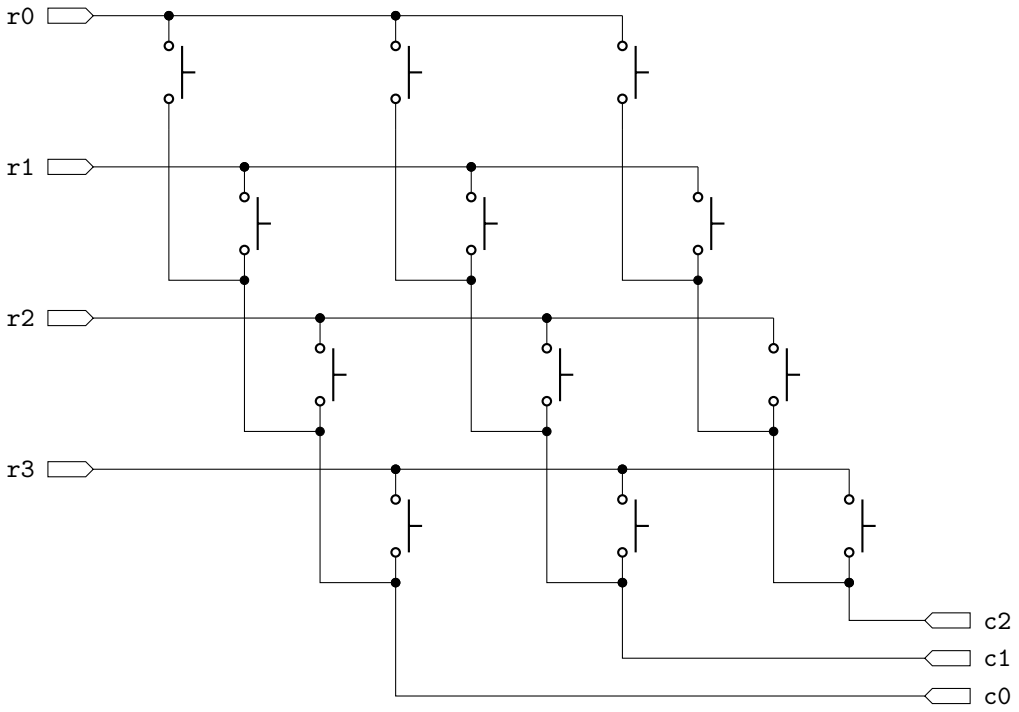

Figure 16: Circuit diagram for a passive 12-button keypad.

the input pins have **pull down** resistors, or otherwise interpret a floating value as zero. [15]

- To test if the button in column c and row r was pressed, the output for row r is set to 1 while all other outputs are zero.
- The input is read for column c. If that is 1, then the button in column c and row r is pressed.
- By cycling through all rows, providing each with an output of 1 one at a time, and checking all columns each time, the system can identify any and all buttons that have been pressed.
- If no buttons were pressed, the inputs read for all columns will always be zero, regardless of which column has output 1.

This kind of keypad connected to a GPIO port therefore requires some of the port pins to be set to input, and others to be set to output. To check if a button was pressed, software must be written to cycle writing output to each column and reading from each row. An example in Assembly language is shown below, assuming the address of the keypad GPIO port is in `r0` and the keypad uses pins 0 to 6.

First, the following subroutine initializes the GPIO pins as output and input, as appropriate. Notice that this subroutine takes care to avoid modifying any of the other GPIO pins, in case there are other peripherals attached to this port.

[15] If the input pins have **pull up** resistors or otherwise interpret a floating value as one, just use the complements of all input/outputs described below.

```
init_keypad:
  @ use pins 0-2 as columns, pins 3-6 as rows
  @ get current GPIO pin direction
  ldr r1, [r0, #4]

  @ mask for output pins
  mov r2, #0b01111000
  @ apply mask for output
  orr r1, r2
  str r1, [r0, #4]

  @ mask for input pins
  mvn r2, #0b00000111
  @ apply mask for input
  and r1, r2
  str r1, [r0, #4]

  @ mask for output data
  mvn r2, #0b01111000
  @ blank output pins
  and r1, r2
  str r1, [r0]

  @ return
  bx lr
```

The following subroutine scans the keypad to determine which

button was pressed. It returns a value in `r1` that indicates which button was pressed: row r is indicated by setting the $(r+2)$th bit to 1 ($0 \leq r \leq 3$), and column c is indicated by setting the cth bit to 1 ($0 \leq c \leq 2$). If no button was pressed, `r1` stores 0.

```
scan_keypad:
  @ start with row 0
  @ remember row 0 is pin 3
  mov r1, #0b00001000

  @ get current port data
  ldr r3, [r0]

  @ loop until we reach row 3
scan_loop:
  @ output to current row while preserving
  @ any other outputs
  orr r2, r3, r1
  @ set output on row
  str r2, [r0]
  @ read input
  ldr r2, [r0]
  @ mask input
  and r2, #0b00000111

  @ check column 0
  tst r2, #0b0001
  addne r1, #0b0001
```

```
    bxne lr

    @ check column 1
    tst r2, #0b0010
    addne r1, #0b0010
    bxne lr

    @ check column 2
    tst r2, #0b0100
    addne r1, #0b0100
    bxne lr

    @ otherwise move to next row
    lsl r1, #1
    @ check if last row is done
    cmp r1, #0b10000000
    bne scan_loop

    @ nothing was pressed
    mov r1, #0
    bx lr
```

Notice how this subroutine is careful to avoid changing the output on any other pins (as it combines the new output with the old output using `orr r2, r3, r1`). One draw-back with this software implementation is that it won't register when two buttons are pressed simultaneously — only upper-leftmost button will be identified. You can think of how to modify this code to fix that problem. [16]

[16] Assuming it *is* a problem — depending on what you need the keypad for, you may *not* want to register multiple buttons.

There is one major problem with an active keypad as we have implemented it here: if two or more buttons are simultaneously pressed *in the same column*, then a short circuit will be created between the row pins when cycling through the output pattern.

- One way to solve this problem is to configure the hardware of the output pins to be a high-impedance state ("open drain"). This isn't always an option — the DE10-Standard does not seem to have configurable hardware on the GPIO pins, for example. [17]
- Another way to solve this problem in software is to switch all row pins to **input** except the one that is currently used as output.

It is worth stressing that, just like joysticks, there are lots of different hardware variations of keypads. A passive keypad is symmetric: it doesn't matter whether rows or columns are set as input. But an active keypad has either columns set to a common voltage (as presented here) or rows. If the rows are common, the search algorithm in software will need to be transposed: the columns set as output and the rows set as input. Finally, it is difficult to mimic a matrix read-out keypad on the GPIO port, because the input needs to change based on the current stage of the search algorithm.

[17] Others, like the STM32L4 discovery board discussed in the textbook and Prof. Reyhani's slides, allow extensive GPIO hardware customization. The STM32L4 has extra **control registers** mapped to the GPIO that allow the user to configure the hardware of the pins to be push-up, pull-down, or open-drain.