General Input/Output and Peripherals

Prof. John McLeod

ECE9047/9407, Winter 2021

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 input/output, seven-segment displays, timers, and analog-to-digital converters are discussed.

Memory Mapped Peripherals

The ARM®Cortex-A9 is a **von Neumann** architecture, **memory mapped** microcontroller. This means that all peripherals attached to the microprocessor are assigned a fixed address in memory space. Controlling or recieving 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.

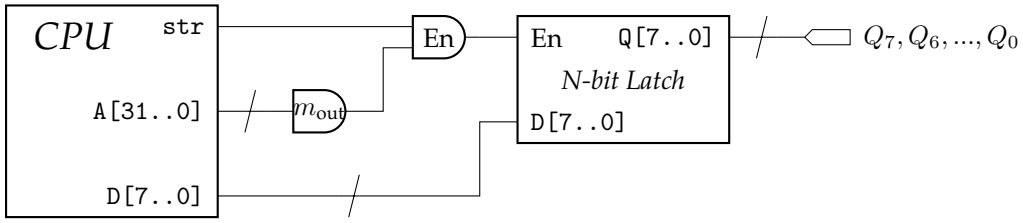

Figure 1: Simplified schematic of an 8-bit output port.

- 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.

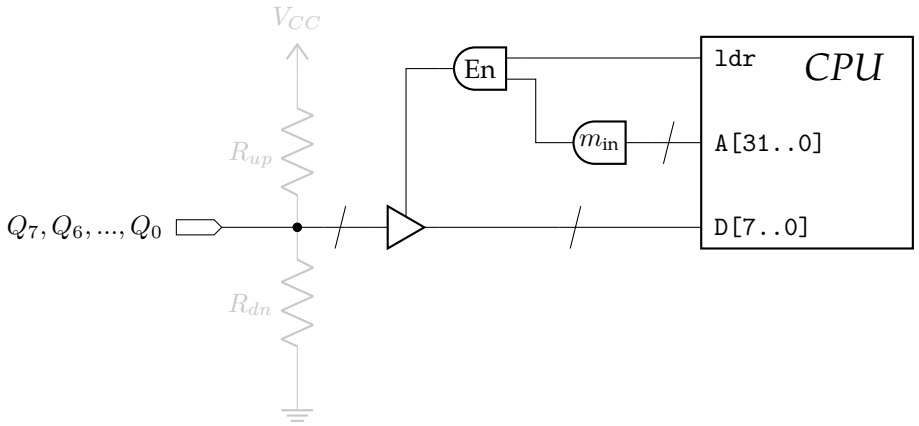

Figure 2: Simplified schematic of an 8-bit input port. Either a "pull-up" resistor R_{up} or a "pull-down" resistor R_{dn} is used to ensure a digital input is present even when the input pin is floating.

- Again, the peripheral is enabled by the correct address and a control bit indicating that a `ldr` instruction was provided.

[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.

- 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.

- 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.,

[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.

zero is the "off" or "resting" state), can cause a lot of unnecessary power loss when connected to a pull-up input 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 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 making 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

[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.

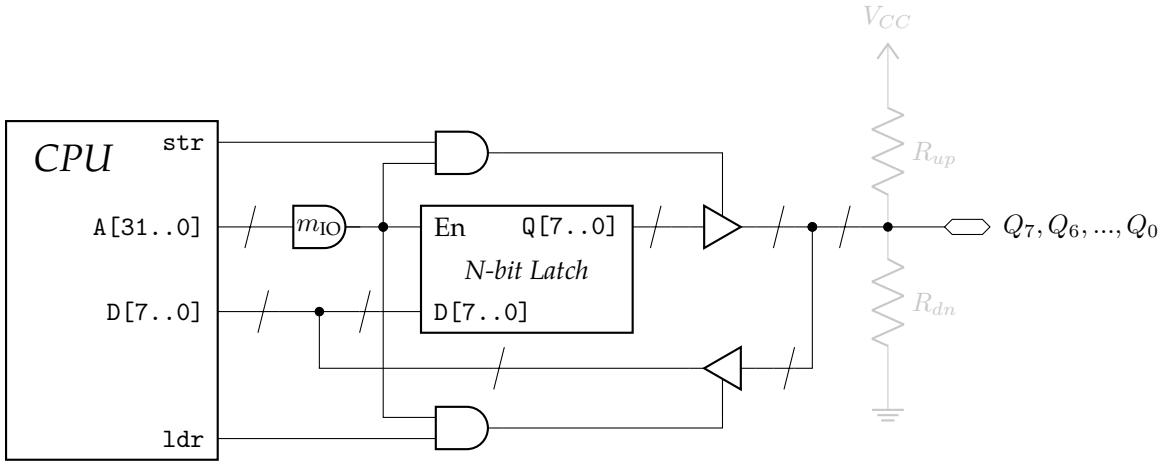

Figure 3: Simplified schematic of an 8-bit bidirectional port. Either a "pull-up" resistor R_{up} or a "pull-down" resistor R_{dn} is used to ensure a digital input is present even when a floating pin is configured as input.

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.

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 7 of the **status register** for the *hand grenade* peripheral that comes standard with every DE1-SoC 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 hand grenade 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 7 are zero.

One implementation of this is given below.

```
@ read in the status register
ldr r0, hand_grenade_sr
ldr r1, [r0]
@ prepare a bit mask, where only
@ bit 7 = 1, all others are 0
mov r2, #1, lsl #7
@ 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.

Example: The "evil flag" is bit 13 of the **control register** for the *moral compass* peripheral that comes standard with every DE1-SoC 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 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 1's with a few 0's at select bits, such as using an *exclusive or* (`xor`) or the *bitwise clear* (`bic`) mnemonics.

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 4

- 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.

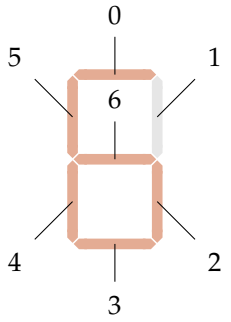

Figure 4: The mapping between the control bits and the segments on the display. The MSb control bit, 7, is unused.

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

- 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 5.

Figure 5: 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 4). 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 `0x0077` to the two left-most panels at address `0xff2000030`, and `0x7776` to the four right-most panels at address `0xff200020`.

The code is shown below.

```
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]
```

Timers

Every microcontroller contains on or more timing peripherals. These are essential for synchronizing the microcontroller with other peripherals, or for allowing controlled delays in program execution. Every microcontroller has a **watchdog timer** to prevent the system from freezing.

- The watchdog timer is always counting, and if the count ever **overflows** (or "wraps around"), the microcontroller completely resets.
- Under normal operation, the microcontroller always *rolls back* the watchdog timer before it overflows.
- If the system ever gets stuck, the watchdog timer will eventually cause a reset.
- This process is entirely transparent to the user, and the watchdog timer cannot be accessed by software.

Most microcontrollers have additional timers used in software. A timer is a reasonably complicated peripheral, and has some **structure**.

- Most timers have a set of **control bits** (in a control register) that determine how the timer operates.
- Timers also have a **counter** register (a data register) that is adjusted to keep track of the time.

- Many timers also have a register that can store a time limit which tells the counter when to stop (another data register).

Clearly a timer needs to accept both input and output, as control flags and count limits must be set and the current time needs to be read. Timers don't actually measure time directly, rather they count **clock cycles**. Some timers operate using the system clock, others do not. Some count up, others count down, and some are configurable.

- The general-purpose *interval timers* on the DE1-SoC board operate at 100 MHz, which is considerably slower than the ARM®Cortex-A9 CPU clock.

- The general-purpose *interval timers* on the DE1-SoC board always **count down** to zero.

Consequently, you set a timer by writing a *multiple of the timer clock cycles*, rather than an quantity in units of time.
For the labs in this course, one or both of the *interval timers* should be used whenever a timer is needed. These are memory mapped to the base addresses `0xff202000` and `0xff202020`. For simplicity, the **structure** of these timers is based on 32-bit **words**. However, due to a bunch of reasons, none of the actual inputs or outputs from the timer exceed 16-bit **half-words**.

- The first word at the base address of the timer is the **status register**. This uses only two (2) bits!

    - Bit 0 is a flag that indicates whether or not the timer has reached a **timeout** — i.e., counted down to zero.

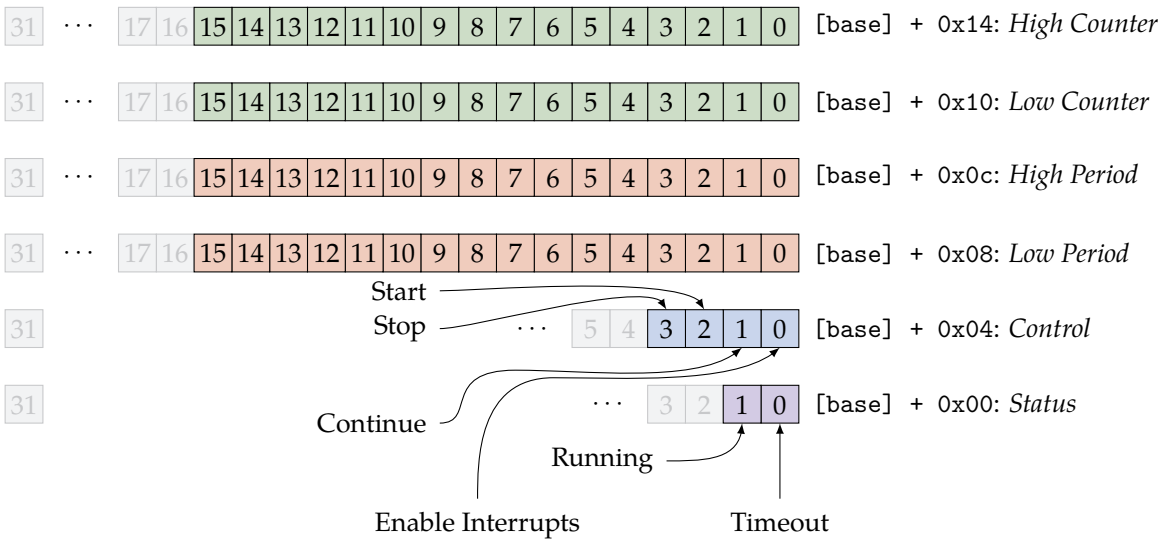

Figure 6: Structure of the interval timers on the DE1-SoC board. The base addresses are `0x0xff202000` and `0x0xff202020`. Note that although the timer is structured in words (32 bits), for legacy reasons at most only half-words (16 bits) are used for each part.

  - Bit 1 is a flag that indicates whether or not the timer is running.

  This **status register** can be read to determine the current state of the timer. This register can also be *partially* written to: if the **timeout flag** was previously set, writing *anything* to this register will clear it.

- The second word after the base address of the timer is the **control register**. This uses a whopping four (4) bits.
  - Bit 0 is a flag that enables the timer to send **interrupts** to the CPU. We will discuss interrupts in a later lesson.
  - Bit 1 is a flag that tells the timer to count continuously, "wrapping around" whenever the present interval is reached.

    - Bit 2 is a flag that tells the timer to stop counting.
    - Bit 3 is a flag that tells the timer to start counting.

  This register can obviously be written to, as this is how the timer is controlled. It can also be read from, if you forget what the controls were.

- The third and fourth words after the base address are the **period registers**. These are a type of data register, and are used to set the count interval. Although the count interval is a 32-bit value — and the registers themselves are 32-bits — for legacy reasons this is split into the least significant 16 bits of two registers.

- The final two words after the base address are the **counter registers**. These are also a type of data register, and are used to record the current count state. Again, for legacy reasons the 32-bit state is split into the least significant 16 bits of two registers.

The structure of the time is pretty frustrating — why, for a 32-bit timer, on a 32-bit architecture, is it necessary to split the timer interval across two half-word registers? Presumably these timers were first developed for 16-bit CPUs, and for "reasons" needs to keep it that way. [8]

- My griping aside, reading a writing to the timer isn't a huge problem as long as you remember how to use the barrel shifter to switch two half-words for word.

[8] Incidentally, this timer silliness is an issue with the DE1-SoC development board, *not* the ARM®Cortex-A9 microcontroller, as the timer is an on-board peripheral, not an on-chip peripheral. So direct your complaints to Terasic, not ARM.

For example, here is how to initialize the timer to count an interval of 5 s:

```
.data
@ to set the timer for 5 s
timer_interval: .word 500000000

.text
@ initialize the timer
ldr r0, =0xff202000 @timer address
ldr r1, adr_timer_int
ldr r2, [r1]
str r2, [r0, #8] @write interval to timer
     @low-period register
     @only lowest-16 bits will be written
mov r2, r2, lsr #16 @shift right by 16 bits
str r2, [r0, #12 ] @write rest of interval
     @to timer high-period

adr_time_int: .word timer_interval
```

To continue complaining about this timer, I find operating it also very confusing.

- I can't think of a reason why the **control register** has one bit for "start" and another bit for "stop".
- Setting the start bit to 1 will start the timer counting down.
- It seems that setting both the start and stop bit to 1 will also

start the timer — it seems the "start bit" takes priority.

- If the timer is running, it will continue to run if the the entire **control register** is cleared to zero — it seems the "running bit" in the **status register** takes priority.
- You can't write directly to the **status register**, so it seems the only way to turn off the timer (other than letting it count down to zero if the "continue bit" is not set) is to set the "stop bit" to 1 and the "start bit" to 0 in the *control register*.

Finally, one last complexity! If you want to read the current state of the timer, *do not* read the **period registers**. The current count is stored in these registers, but attempting to read them directly will use one or more clock cycles and will make the actual timed interval less accurate.

- Instead, write some value — any value, as it is ignored — to either of the **counter registers**.
- This will cause the timer to mirror the current count to the **counter registers**.
- After that, you can read from the **counter registers** normally.

In this simulator, the timer is not completely accurate but it is still pretty close. Note that if you start the timer to run continuously [9] it will do so — even if you stop, or pause the assembly program. This usually isn't a problem, but can lead to unexpected results if you are trying to time a precise interval and also stepping through your code line-by-line to debug it.

[9] i.e. write `0x06` to the **control register** to set both the "start bit" and the "continue bit"

Example: We want to use the timer to measure a 5 s interval, and also sequentially light up LEDs for each 1 s subinterval. After the 5 s period is over, the LED bank should be reset and the process should repeat.

A working implementation is given below. Those of you who have a shred of skill at coding will recognize that the example below is *not* a particularly good solution to the problem. For one thing, it would be easier (and probably better coding practice too) to have the timer just count a 1 s interval and use a separate counter to count to 5 — but then that would not demonstrate how to read the current count from the timer.

```
.global _start

.data
timer_T:   .word   500000000
timer_sT:  .word   100000000


.text
_start:
@ get the addresses
ldr r4, =0xff202000
ldr r5, =0xff200000
ldr r6, adr_T
ldr r7, adr_sT

@ initialize the 5 s count
```

```
ldr r0, [r6]
str r0, [r4, #8]
mov r1, r0, lsr #16
str r1, [r6, #12]

@ start the timer for continuous
@ counting (0b0110)
mov r1, #6
str r1, [r4, #4]

@ get the 1 s interval
ldr r1, [r7]

@ initialize LED pattern
mov r8, #0
str r8, [r5]

@ main loop
main_loop:

        @ get the current count
        @ first write junk to counter
        str r1, [r4, #16]
        @ now get the low count
        ldr r2, [r4, #16]
        @ get the high count
        ldr r3, [r4, #20]
        @ combine them
```

```
        add r2, r3, lsl #16

        @ check if current count has
        @ passed 1 s
        cmp r2, r0
        bhi main_loop

        @ increment LED pattern
        lsl r8, #1
        add r8, #1
        str r8, [r5]
        @ check if we have passed 5 s
        @ #63 is bit pattern 111111
        cmp r8, #63
        @ if not, subtract 1 s from interval
        @ and loop back
        sublo r0, r1
        blo main_loop
        @ if we passed 5 s, reset interval
        ldr r0, [r6]
        @ blank LEDs
        mov r8, #0
        str r8, [r5]
        b main_loop

adr_T:          .word   timer_T
adr_sT:         .word   timer_sT
```

Debouncing

Consider the simplified input port with a push button switch for the external peripheral, as shown in Figure 7. 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.

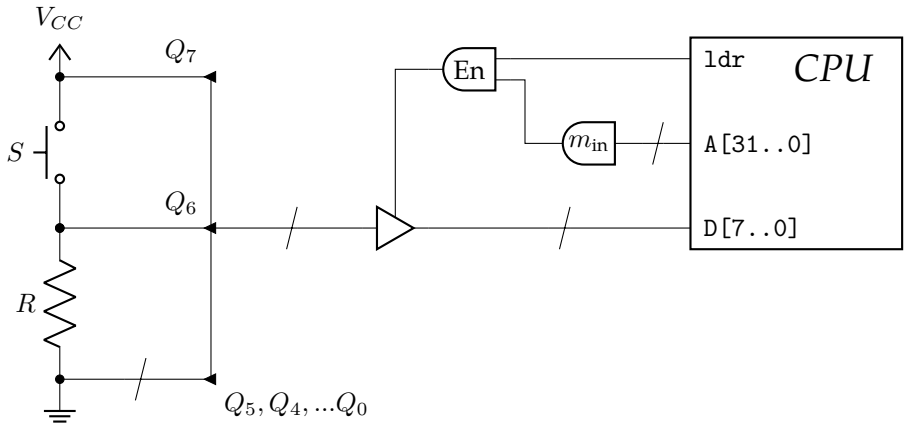

Figure 7: Simplified schematic of an 8-bit input port with a push button switch as a peripheral. A pull-up or pull-down resistor may also be added between the peripheral and the tri-state buffer, but that has been omitted in this diagram.

The follow code could be used to test when S is pressed. [10]

```
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
```

[10] 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
ldr r4, adr_timer    @memory address for timer
/* a timer 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
```

```
    @ clear timeout flag
    str r1, [r4]
    @ start timer
    mov r2, #4
    str r2, [r4, #4]
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
    @ check for timeout
    ldr r3, [r4]
    cmp r3, #1
    bne timer_loop

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

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 8.
Sometimes a combination of hardware and software filtering is also necessary, depending on the peripheral and how important it is for

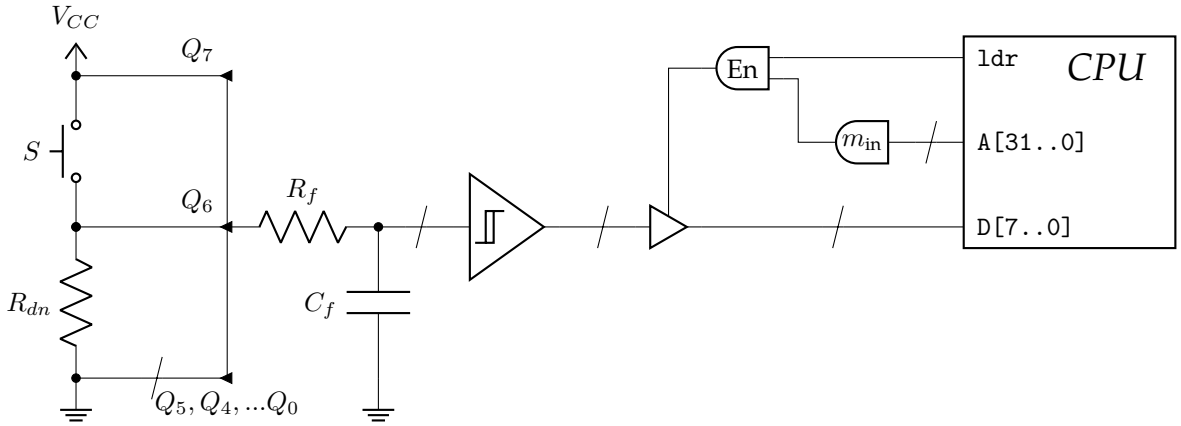

Figure 8: Simplified schematic of an 8-bit input port with some hardware debouncing. A low-pass filter $R_f C_f$ removes high-frequency noise, and a Schmitt trigger digitizes the input to V_{CC} or 0.

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.

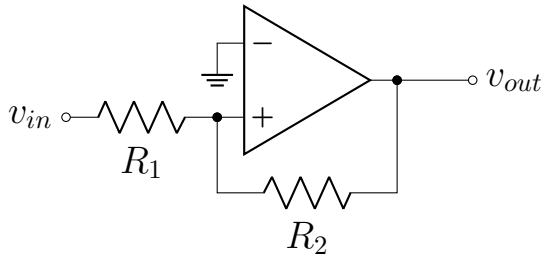

Figure 9: One way to implement a Schmitt trigger is with an op amp in which the feedback resistor R_2 is connected to the $+$'ve input terminal (instead of the usual $-$'ve input terminal for amplifier circuits). Electrical instability causes the output voltage to be pushed up (or down) to the power 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.

Analog-to-Digital Conversion

Most real-world information is analog. An analog signal $v_s(t)$ varies smoothly and continously in voltage and time. This signal must be converted to digital in order to be processed by a microcontroller. For this reason, almost all microcontrollers have at least one **analog-to-digital converter** peripheral ("A/D" or "ADC") built-in.

Definition: An analog signal is **sampled** if it is only measured at discrete time intervals Δt (or measured and averaged over that interval).

Definition: An analog signal is **discretized** if the voltage amplitude is rounded to some multiple of the voltage resolution ΔV.

Definition: An analog signal is **digitized** when it is both sampled in time and discretized in amplitude.

Definition: A digital signal is **binary** if only two discretization levels (equivalent to 0 and 1) are used.

Technically, most ADC are actually analog-to-binary converters (or ABC?), as they take a single analog input and produce an n-bit binary output. But whatever... the name ADC is what everyone uses. A representation of sampled, discretized, and digital signal is shown in Figure 10.

- Time sampling is usually done in multiples of the CPU clock cycle.

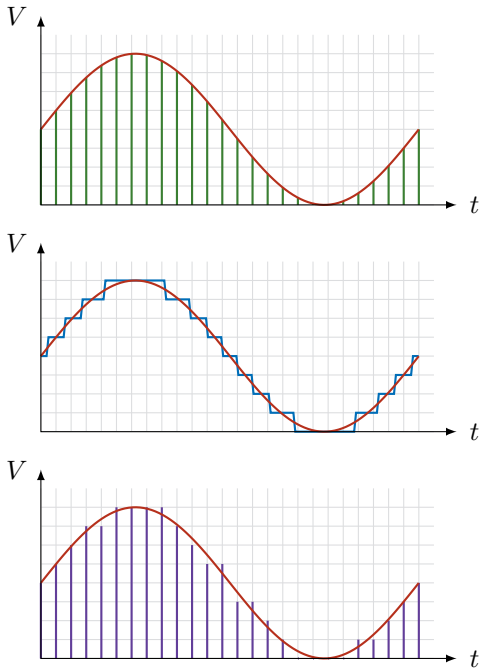

Figure 10: Various representations of an analog signal (a sine wave). Top: The signal is sampled at a given time interval. Middle: The signal is discretized into a given voltage increment. Bottom: The signal is fully digitized.

- Exactly how the voltage is discretized depends on the hardware implementation of the ADC, we will discuss some implementations later.

A discretized analog signal can only be understood if the **reference voltage**, **bias voltage**, and **data width** are known. A given analog voltage V can be discretized into n bits as:

$$V = \left(\frac{b_{n-1}}{2} + \frac{b_{n-2}}{4} + \cdots + \frac{b_0}{2^n}\right) V_{ref} + V_{bias},$$

where b_i is the value of digital bit i, V_{ref} is the reference voltage (or scale voltage), and V_{bias} is the bias voltage.

- Note that in this definition, the "bias voltage" refers to the *minimum voltage* in the analog signal, as the sum of the bit values is strictly positive. [11]
- Often this bias is removed before ADC, so $V_{bias} = 0\,\mathrm{V}$.

From the above expression it should be clear that the **resolution** ΔV of the discretized signal is:

$$\Delta V = \frac{V_{ref}}{2^n}.$$

This is the smallest non-zero analog signal that can be represented in n bits. With a typical reference voltage of $V_{ref} = 5\,\mathrm{V}$, discretizing a signal into $n = 16$ bits gives $\Delta V = 76.3\,\mu\mathrm{V}$, which is often well below the noise threshold for a typical circuit. Consequently, ADC is rarely done with more than $n = 16$ bits of width. [12]

[11] Or the *maximum voltage*, if $V_{ref} < 0$.

[12] Even that is a bit extreme — if $n = 8$ then $\Delta V = 19.5\,\mathrm{mV}$ which is already pretty small for most circuits.

Ladder Converters

The first circuit we will consider is an ADC turned on its head: a *digital-to-analog converter* (DAC). This actually takes a n-bit binary input and produces a digital output (so a BDC?), but in the microcontroller world anything that is not binary is considered analog. A simple and practical implementation of a DAC is a R-$2R$ *ladder converter*. This circuit is typically based on an op amp, as shown in Figure 11. Those of you who paid close attention in your undergraduate electronics classes will recognize this as a particular implementation of a *weighted inverting summer*. [13]

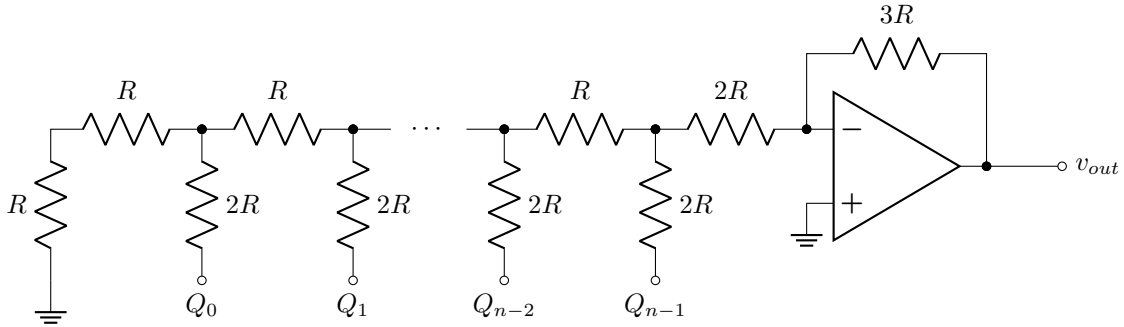

Figure 11: An n-bit ladder converter circuit for DAC. The digital signal $Q[n-1, ..., 0]$ is converted to the analog output v_{out}.

[13] The word "summer" means a "circuit that performs summations," not the season.

Analyzing this circuit is straightforward, assuming you remember the principle of superposition.

- Because of the operating principles of the op amp, the $-$'ve terminal is at essentially zero potential — as it is tied to the $+$'ve terminal by a virtual short circuit.
- If only one input Q_m is considered, and all other inputs are set to ground, the equivalent circuit for that input is shown in

Figure 12. This occurs because the resistances were carefully chosen: Note that $2R \parallel 2R = R$.

- The equivalent resistance seen by Q_m is $2R + 2R \parallel 2R = 3R$, so the current is:

$$i_m = \left(\frac{Q_m}{3R}\right) V_{ref},$$

where V_{ref} is the voltage of binary signal 1.

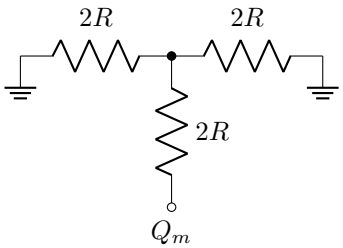

Figure 12: Equivalent circuit by the principle of superposition for any input Q_m in the ladder converter from Figure 11, when all other signal sources are replaced with grounds.

- The equivalent circuit for input Q_m also makes it clear that any current which flows from signal Q_m gets split in half at the node.

- The half current that flows right towards the op amp will further be split in half each time it reaches a node for Q_p (where $m < p < n - 1$). The current from signal Q_m that reaches the $-$'ve terminal of the op amp is therefore:

$$i_{m,-} = \left(\frac{Q_m}{2^{n-m}}\right) \frac{V_{ref}}{3R}.$$

- By Kirchoff's current law, the total current reaching the op amp is just the sum of all the signals:

$$i_{tot,-} = \frac{V_{ref}}{3R} \sum_{m=0}^{m=n-1} \frac{Q_m}{2^{n-m}}$$

- This current can't enter the op amp, as the impedance at the input is almost infinite. Instead it must flow across the $3R$ resistor. The output voltage is therefore:

$$v_{out} = -3Ri_{tot,-} = -V_{ref} \sum_{m=0}^{m=n-1} \frac{Q_m}{2^{n-m}}$$
$$== -V_{ref} \left(\frac{Q_0}{2^n} + \frac{Q_1}{2^{n-1}} + ... + \frac{Q_{n-2}}{4} + \frac{Q_{n-1}}{2} \right).$$

This is a digital signal, as it has 2^n discrete voltage levels, and can only change in value when the inputs Q_m change — and they are controlled by the system clock. However for a sufficiently fast clock (as is usually the case) and reasonably large n (as mentioned above, $n = 8$ is usually plenty), this signal is often indistinguishable from a true analog signal.

Integrating ADCs

Converting an analog signal into a binary sequence is slightly more complicated than converting binary to analog. One method for ADC is a **single-slope integrator**, as shown in Figure 13.

- The first stage of this circuit is an *inverting integrator*, where charge accumulates on the capacitor C.
- The input "signal" for this integrator is just the negative of the reference voltage V_{ref}, [14] so the output of this integrator is a straight line sloping up from 0 to V_{ref} as a function of time.

$$v_{int}(t) = \left(\frac{V_{ref}}{RC}\right) t$$

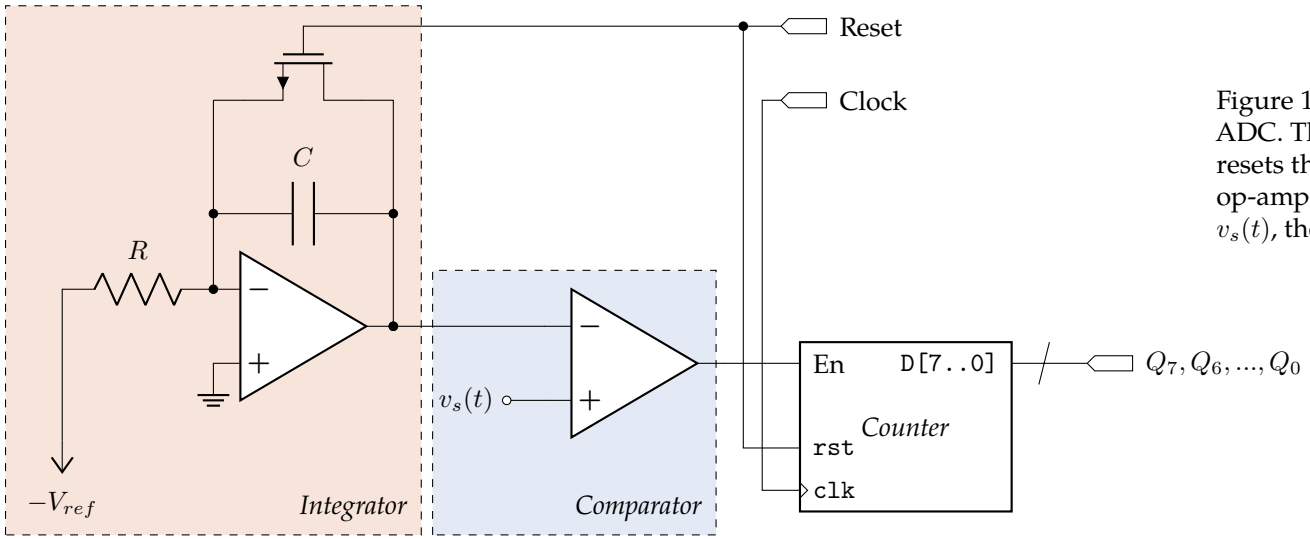

Figure 13: An 8-bit single-slope integrator for ADC. The reset signal from the microcontroller resets the counter and shorts the capacitor in the op-amp integrator circuit. The analog input is $v_s(t)$, the digital output is $Q_7, Q_6, ..., Q_0$.

[14] Because op amp integrators are always inverting, using $-V_{ref}$ provides a positive output.

- The integrator output goes into a second op amp. The signal voltage $v_s(t)$ is used as the input to the $+$'ve terminal. This op amp has no *feedback loop*, so the output is always **saturated** to the limits set by the op amp power supplies — usually V_{ref} and ground.
- Because an op amp is a *differential amplifier*, whenever $v_s(t) - v_{int}(t) > 0$, the comparator output will saturate to V_{ref}. Whenever $v_s(t) - v_{int}(t) < 0$, the comparator output will saturate to ground.
- The comparator output is used as the enable for a n-bit counter, it will stop counting as soon as $v_{int}(t) > v_s(t)$.
- The value after the count has stopped is used as binary equivalent of the analog signal at the given time interval.
- To read the next time interval, the CPU sends a reset signal to clear the capacitor charge and reset the counter.

It is important to recognize that the entire process described above must occur to read the signal for a *single* time interval: the operation of an integrating ADC is necessarily significantly slower than the system clock. This is usually not a huge problem, but is something to be aware of.
The operation of a single-slope integrator is shown in Figure 14. Two example input signal voltages are given as v_1 and v_2.

- At $t = 0$, $v_{int}(t)$ is less than the signal, so the counter is enabled.

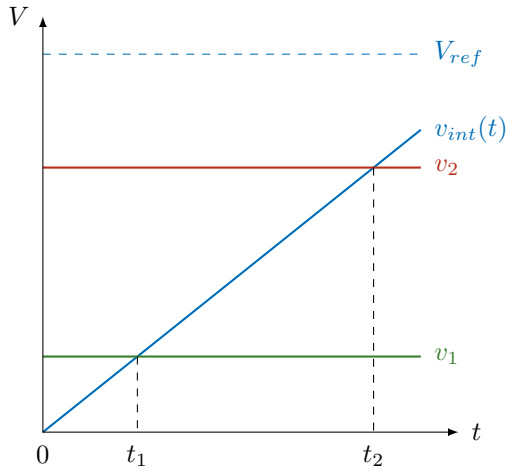

Figure 14: Example of the operation of a single-slope integrator with two different signal voltages (v_1 and v_2) given. The integrator output $v_{int}(t)$ cannot exceed V_{ref}.

- The counter stops when $v_{int}(t) > v_s(t)$, so the time spent counting (t_1 and t_2) is proportional to the magnitude of the signal.
- If the single-slope integrator circuit was designed appropriately such that $RC = 2^n\tau$, where τ is the clock frequency for the counter, then the counter stops at a time:

$$v_1 = v_{int}(t_1) = \left(\frac{V_{ref}}{RC}\right) t_1 = \frac{V_{ref}}{2^n}\left(\frac{t_1}{\tau}\right)$$
$$t_1 = 2^n\tau\left(\frac{v_1}{V_{ref}}\right),$$

- This means the final value on the counter is:

$$D = 2^n\left(\frac{v_1}{V_{ref}}\right).$$

This circuit works in principle, but it suffers from two flaws:

1. The conversion time depends on the magnitude of the signal voltage (the smaller input v_1 is converted in a time t_1 that is shorter than the time t_2 for the larger input v_2).
2. If RC is not exactly equal to $2^n\tau$ (which is likely) then conversion errors start creeping in.

The second flaw can be addressed using a **dual-slope integrator**, as shown in Figure 15.
In this circuit, a switch is used to control whether the signal $v_s(t)$ or the constant $-V_{ref}$ is fed into the integrator.

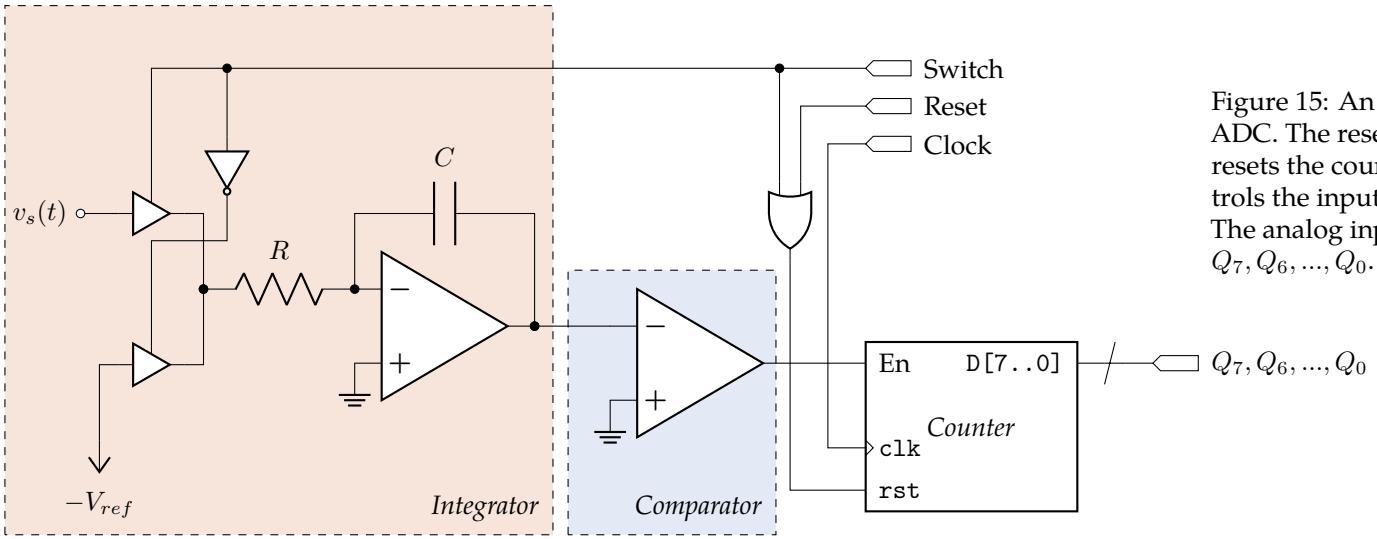

Figure 15: An 8-bt dual-slope integrator for ADC. The reset signal from the microcontroller resets the counter, while the switch signal controls the input to the op-amp integrator circuit. The analog input is $v_s(t)$, the digital output is $Q_7, Q_6, ..., Q_0$.

- At the start of the conversion, the switch is set to use $v_s(t)$ as the input. The switch remains in this position for a known interval $\Delta t = K\tau$, where τ is the counter clock pulse and K is some integer.
- Since the integrator is inverting, this drives the integrator output to the negative voltage:

$$v_{int}(t = K\tau) = -\left(\frac{v_s}{RC}\right) K\tau$$

- The comparator +'ve input is grounded. As $0 - v_{int}(t) > 0$ when the integrator output is negative, the comparator output saturates to V_{ref} and the counter is enabled. However the counter doesn't actually start counting, as the reset signal is held.

- After an interval of $K\tau$, the switch is flipped and $-V_{ref}$ is used as the input. This drives $v_{int}(t)$ linearly back up to zero. Flipping the switch also turns the reset to the counter off. As the counter enable is still set, the counter starts counting.
- After an *unknown* time interval $\Delta t'$, the integrator output goes slightly above zero: $v_{int}(t = K\tau + \Delta t') > 0$. At this point the $-$'ve input to the comparator exceeds ground, so the comparator output saturates to 0 and the counter stops.
- This unknown interval can be expressed in clock pulses: $\Delta t' = L\tau$. We therefore have:

$$v_{int}(t = K\tau + L\tau) = 0 = -\left(\frac{v_s}{RC}\right) K\tau + \left(\frac{V_{ref}}{RC}\right) L\tau$$

$$\frac{v_s}{V_{ref}} = \frac{L}{K}$$

If we chose $K = 2^n$, then the value on the counter is:

$$D \approx 2^{\log_2 L}.$$

This circuit still has the problem that the conversion time depends on the magnitude of the signal (which is proportional to the unknown constant L), but the dependence on RC has been removed. These **dual-slope integrators** are not often used as-is for ADC, but rather as the first stage of a **sigma-delta** ADC. Here an additional logic stage controls the switch and reset inputs, and reads the counter output D. This additional logic stage does further comparisons between the input signal and the output to reduce errors. [15]

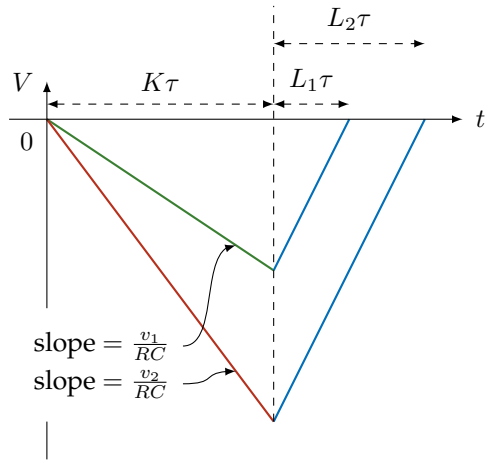

Figure 16: Example of the operation of a dual-slope integrator with two different signal voltages (v_1 and v_2) given, where $V_{ref} > v_2 > v_1$.

[15] The dual-slope integrator as presented has a tendency to always "round down" the converted signal.

Successive-Approximation Converters

The time-dependence on conversion time that was a persistent issue with integrator-based ADC can be avoided using a different approach. This is implemented by a the **successive-approximation converter**, as shown in Figure 17. This circuit implements a *binary search* for the correct value that digitizes $v_s(t)$, by comparing the current estimate of that value $(Q_7, Q_6, ..., Q_0)$ to $v_s(t)$ using a DAC.

- The heart of this circuit is again an open-loop op amp comparator. If $v_s(t) > v_A$, the output saturates to V_{ref}. If $v_s(t) \leq v_A$, the output saturates to ground.

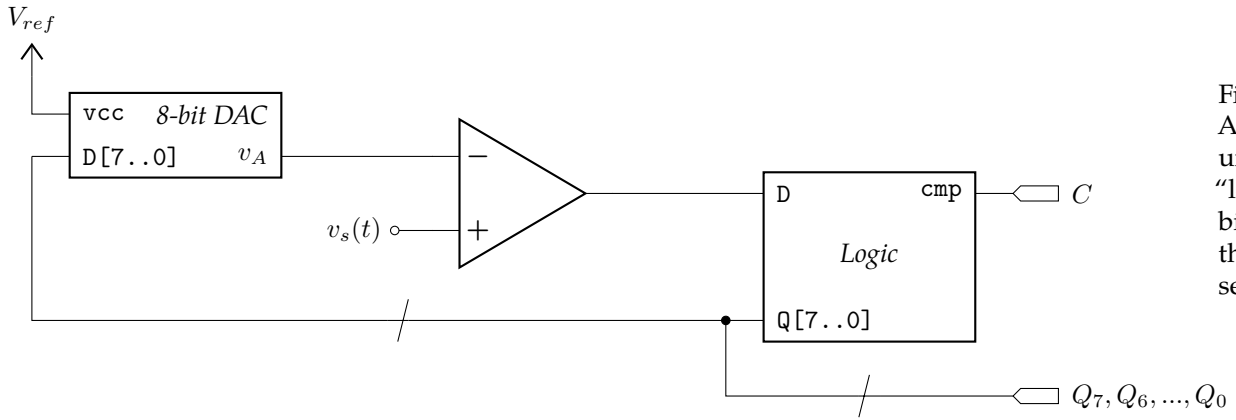

Figure 17: An 8-bit successive-approximation ADC. Here the "8-bit DAC" is based on Figure 11, with the output scaled by V_{ref}. The "logic" block checks the input D and adjusts the bits in Q from MSb to LSb to keep D low. After the LSb in Q is checked, the comparison flag C is set, indicating complete conversion.

The "logic" part of the circuit is more complicated than a simple counter (and that is why I drew it as a block), it is responsible for performing the binary search. The logic circuit processes the output bit-by-bit, starting with the MSb and working down to the LSb. It

starts with $Q_n, Q_{n-1}, ..., Q_0 =$ `0b00...0`. It then iterates through this algorithm for all bits m in $n > m \geq 0$.

- Assume bit m is high, so the value is $Q_n Q_{n-1} ... Q_{m+1} 10...0$.
- Check the output from the comparator.
- If this guess is $\leq v_s(t)$, keep bit m high. Otherwise, flip bit m back to low.
- Repeat the process for bit $m - 1$.

An example of this search for a 4-bit successive approximation ADC is shown in Table 1. Starting with the MSb, each bit is successively tested (shown in red). If the resulting voltage exceeds the analog signal, that bit is cleared back to 0, otherwise it is retained as 1.

- The successive approximation ADC will always take n iterations for n-bit conversion, regardless of the magnitude of the input signal.

Successive approximation ADCs are quite common, as the constant conversion time makes it easier to synchronize analog signal processing with other processes in the microcontroller.

Guess	Voltage	Result
`0b1000`	2.5 V	$v_A < v_s$
`0b1100`	3.75 V	$v_A > v_s$
`0b1010`	3.125 V	$v_A < v_s$
`0b1011`	3.3475 V	$v_A > v_s$

Table 1: Example of using a binary search to find the 4-bit binary equivalent of $v_s = 3.33$ V when $V_{ref} = 5$ V The final result is `0b1010`.

Flash ADC

It is important to stress that the analog signal is a function of time: $v_s(t)$, but all of the ADC methods discussed above treat it as constant over the sampling interval.

- ADC needs to be performed for each sample time in the signal.

This is not a problem if the analog signal varies slowly. However, one should remember that the *ADC clock speed* is not the same as the *ADC conversion time* — rather n-bit conversion takes up to 2^n clock cycles. [16]

- You can *never* sample analog signals frequencies that are faster than the clock speed.
- However as accurate ADC usually requires at least $n = 8$ bits, it is quite possible to have signals of interest that have key frequencies that are between $(2^n\tau^{-1})$ and τ^{-1}.

An easy method for rapid ADC is called **flash ADC**. [17]
The operating principle of a flash ADC is straightforward.

- A voltage divider ladder composed of resistances R and $2R$ is used between V_{ref} and ground to obtain all the intermediate voltages for digitization.
- Each of these intermediate voltages is compared with the signal $v_s(t)$. The output of the comparators is V_{ref} if the intermediate voltage is larger than $v_s(t)$, otherwise it is ground.

[16] Always 2^n iterations for successive approximation, for the integrators 2^n is the worst-case.

[17] This has nothing to do with Flash memory!

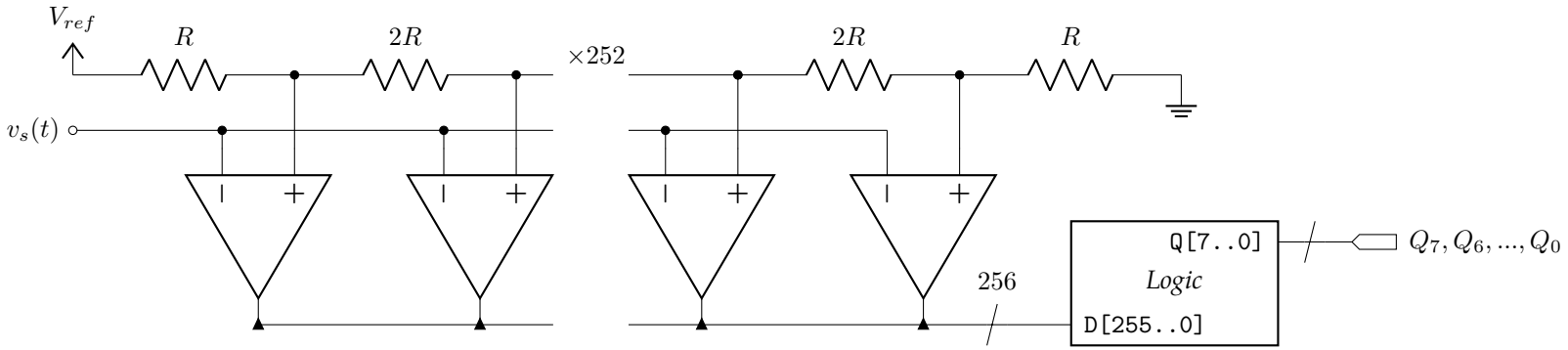

Figure 18: An 8-bit flash ADC. The input signal is compared to a voltage divider ladder with $2^8 = 256$ stages. A combinational logic circuit encodes these 256 inputs into an 8-bit sequence.

- A combinational logic circuit combines the binary output from the comparators into a binary sequence.

Flash ADC does not integrate the signal, and it processes all bits in parallel rather than sequentially, so it is extremely fast. However it is also extremely large: 2^n comparator stages are required for a n-bit ADC. This takes up a lot of physical chip space, and also uses a significant amount of power. Flash ADCs beyond 8-bits are prohibitively expensive, and even 8-bit flash ADCs are too large and power-hungry for many microcontrollers. Flash ADCs are typically only used in specialist applications that require high-speed ADC.