Notes for ECE 36200 - Microprocessor Systems And Interfacing

Zeke Ulrich

October 22, 2025

Contents

Course Description

An introduction to basic computer organization, microprocessor instruction sets, assembly language programming, and microcontroller peripherals.

GPIO

General Purpose Input/Output (GPIO)

Input/output (I/O) is the interface between our digital microcontroller (MCU) and the rest of the world. Each MCU has pins which its software uses to read input and write output.

Some pins are general purpose I/O, but some are used to power the MCU (VDD, GND) and some pins are special (oscillator, clock pins), can to analog to digital I/O or vice versa, or more complex digital I/O (SPI, I2C, UART).

Digital input reads the pin to check if an external voltage is applied. Digital output drives the voltage on the pin. More advanced circuity performs digital-to-analog conversion (DAC) or analog-to-digital (ADC).

The layout of every GPIO pin port looks like Figure 1

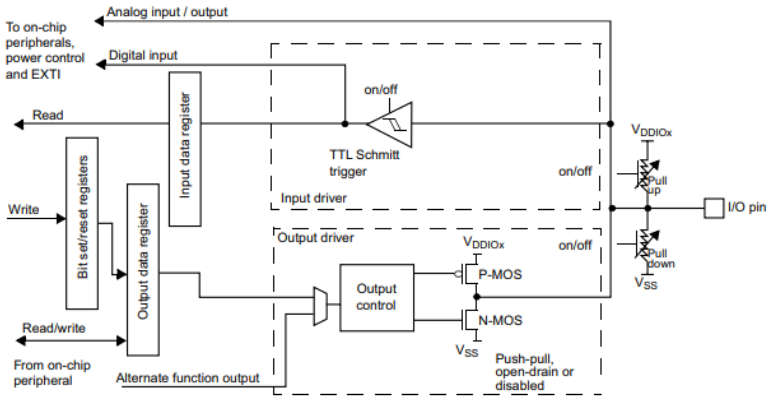


Figure 1: GPIO Layout

Each MCU has multiple GPIO ports A, B, etc. Each port has multiple GPIO bits (typically 8).

Output

The way output works is that an output bit is written into the Output Data Register (ODR). Then the output driver reads the ODR and drives the pin either high or low depending on the value of the ODR bit. The voltage then appears on the I/O pin.

The GPIO can be configured as either read or write, output mode of push-pull or open drain, output speed, and more.

In push-pull mode, we have need of a buffer to increase the output voltage of the MCU so that when the output of the software is 1 the output of the GPIO pin is VCC. This is accomplished with two NOT gates. If the software write a 1, then the controller outputs a 0 to a

NOT gate, which pulls the output pin to 1. Basically, in push-pull mode the MCU actively drives the pin low for 0 or high for 1.

In open drain mode, the MCU drives the output to low, but floats in 1. That is the GPIO pin can be 0 volts, but it cannot be VCC. Basically, it can actively drive a pin low for 0, but leaves the pin floating for 1. Open drain mode is useful when there are multiple outputs. Two output pins can be tied together, which isn't possible with push-pull outputs because one could be high and one low, causing a short circuit. Thus, all tied pins must be open drain to avoid short circuits. Any one of them can drive the shared output low, while a pull-up resistor passively pulls the wire up to high when no MCU is driving it to low.

This is how I23 works. More details to follow in the unit of I2C.

Another configurable is output speed, the speed of voltage rising and falling. A faster GPIO is good for fast communication, but higher speed increases electromagnetic interference and power consumption.

A related concept is slew rate, defined as

$$\max(\frac{\Delta V}{\Delta t}) \tag{1}$$

Input

With input, an external circuit applies a voltage to the I/O pin. The input driver converts the voltage to either 1 or 0. The input is then sampled into the Input Data Register (IDR) every clock cycle.

In real life, the input voltage is noisy and messy. We add a Schmitt trigger to smooth it, reduce noise, and increase the slew rate to make it suitable for our digital circuit. Recall that a Schmitt trigger is just a buffer that is immune to oscillating issues because it has a low and high threshold. When the input signal crosses the high threshold, the output of the Schmitt trigger is high. It stays high until the input signal crosses the low threshold, at which point the output of the Schmitt trigger goes low.

Electricity in the real world is unfortunately messy. Consider the circuit in Figure 2.

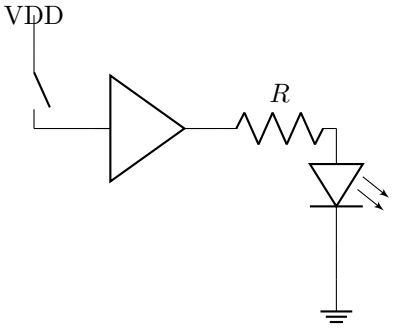


Figure 2: Faulty LED Control Circuit

You would expect that the circuit turns off when the switch is open.

However, the input to the buffer would be floating, so the output of the buffer is unpredictable.

We can remedy this by adding a pull-down resistor, as in Figure 3.

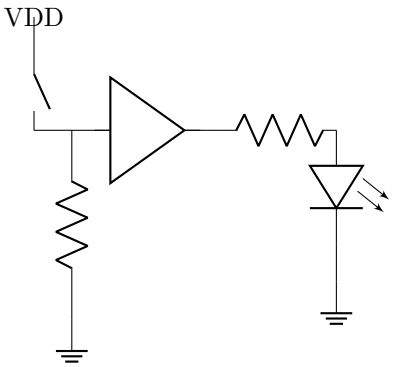


Figure 3: LED Control Circuit

Pins face a similar issue. A floating pin's voltage is unknown. We need to add a pull-up or pull-down resistor so that when the voltage isn't actively being driven low or high then the pin is at a known voltage. The resistors have to be weaker than however the pins are being driven, or the pin would always be high/low, but if it's too weak then the pin will be slow and unresponsive.

Many MCUs have on-chip PU/PD resistors, which are software configurable. Off-chip PU/PD can also be on the PCB instead, in which case it is not configurable.

We have two options or using software to set up a pin.

- Port-mapped I/O, which uses special instructions in the processor where every device is assigned a unique port number (used in x86).
- Memory-mapped I/O, which has special addresses which can be read from or written to in order to perform I/O.

This is true for all on-chip modules, not just GPIO.

Memory-mapped I/O

If we have a 32-bit CPU, each address is 1 byte and there are up to 4 GBs of addressable memory in a 32 bit system. MCUs have limited read/write memory (less than a few MB). Memory is *byte-addressable*. Each byte has a unique address. Addresses go from 0x00000000 to OxFFFFFFFF. The bottom sections have on-chip flash memory, for code and data. The next sections have SRAM, on-chip RAM, for the heap, stack, and some code. At the top is system memory dedicated to external devices like NVIC, system timer, SCB, other vendor-specific memory. Also in high memory (but not the highest) is memory for external devices like SD cards. In the middle is external RAM, off-chip memory for data. See Figure 4.

Most modern computers are 64-bit, but MCUs are still mainly 32-bit.

When the CPU needs to talk to RAM (read/write variables) it's connected via the address bus and data bus. It reads from ROM (read-only memory). When the CPU wants to write to the RAM, it gives an address to the address bus and data to the data bus. The RAM looks at the address bus and if it sees an address that belongs in RAM space, the RAM will take the data from the data bus (if's it's a write signal) and write it at the specified address. When the CPU wants to read it puts an address on the address bus and signals read. The RAM sees the read signal and puts data at the specified address on the data bus.

While RAM is read and write, ROM is read-only. The reason for this separation is that ROM is cheaper. If you turn off your computer, anything in RAM is lost. However, anything is ROM is non-volatile and anything flashed there will remain even if power is lost.

GPIO modules are on the same address and data buses as the ROM and RAM. Each periperphal has a set of control registers, including direction and data registers. Each register has a unique memory address. If you want to write something to the data register, get its address by looking at the data sheet, and write there. Reading is similar. By writing a one or zero into the direction register, you can configure the GPIO as either input or output.

To turn on an LED, for instance, we need to know what port and pin it's connected to. We then consult the data sheet to get the addresses of the registers for that pin and port, and then we just need to write some value to the registers.

Although the GPIO pins are not really memory, the CPU treats them like memory. That's why this method is called memory-mapped I/O.

Memory is accessed at a minimum granularity of one byte, but typically 4.

```
char* controlRegisterPtr = 0x50000000;
*controlRegisterPtr = 1;
```

Interrupts

A key concept in embedded systems is the need to process external stimuli, like a button press, or when a sensor detects a change in its environment. However, the microcontroller may already be busy executing another long running task - maybe it's waiting on a second sensor, or its busy updating a large display. The most efficient way to handle this issue is with *interrupts*.

An interrupt is a signal that is generated by the hardware or software when an event occurs that needs immediate attention. Once it's fired from an interrupt source, say a rising edge on a GPIO pin, the interrupt signal arrives at the CPU, which - if the conditions are right

and the correct bits are set - saves what it's doing, and executes a special function called an interrupt service routine (ISR), also called an interrupt handler.

An interrupt is a special type of exception. Others relevant to MCUs are faults, traps, and resets. Peripherals can raise interrupts. The CPU can be interrupted by more than one event, each of which has its own ISR stored in the interrupt vector table. Interrupts can be enabled or disabled. Interrupts may have different priorities, and higher priorities can preempt lower priorities. This means that when the CPU is servicing one interrupt, an interrupt with a higher priority can interrupt and make the CPU service it instead.

The flow of an interrupt is thus:

1. A peripheral raises an interrupt.
2. The CPU checks if N is enabled.
3. If N is enabled, the CPU marks N as pending.
4. The CPU checks the priority of N versus the current priority level, as it might be serving a higher priority interrupt.
5. If the priority of N is greater than the current priority level, then the CPU updates the current priority level to the level of the new interrupt and pushes the CPU state to the stack. It then puts the Nth entry in the vector table into the program counter, a special register that keeps track of where the CPU is in the code. The ISR is now running.

A common exam question is "how many times was the interrupt handler interrupted?" or "how many times was the CPU state restored?".

Exceptions

We mentioned several other types of exceptions. While interrupts are usually handled at the end of instructions and the CPU resumes at the next instruction, faults happen in the middle of an instruction and execution resumes in the same instruction. Just as with interrupts, higher-priority exceptions may preempt lower-priority exceptions.

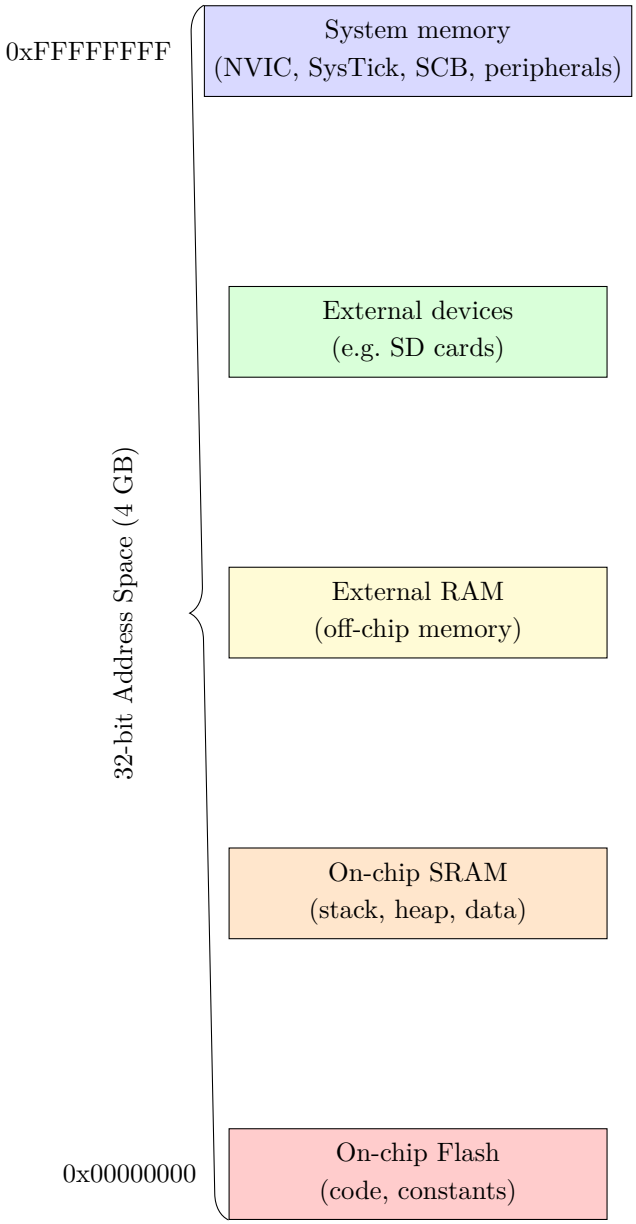


Figure 4: MCU Memory Layout

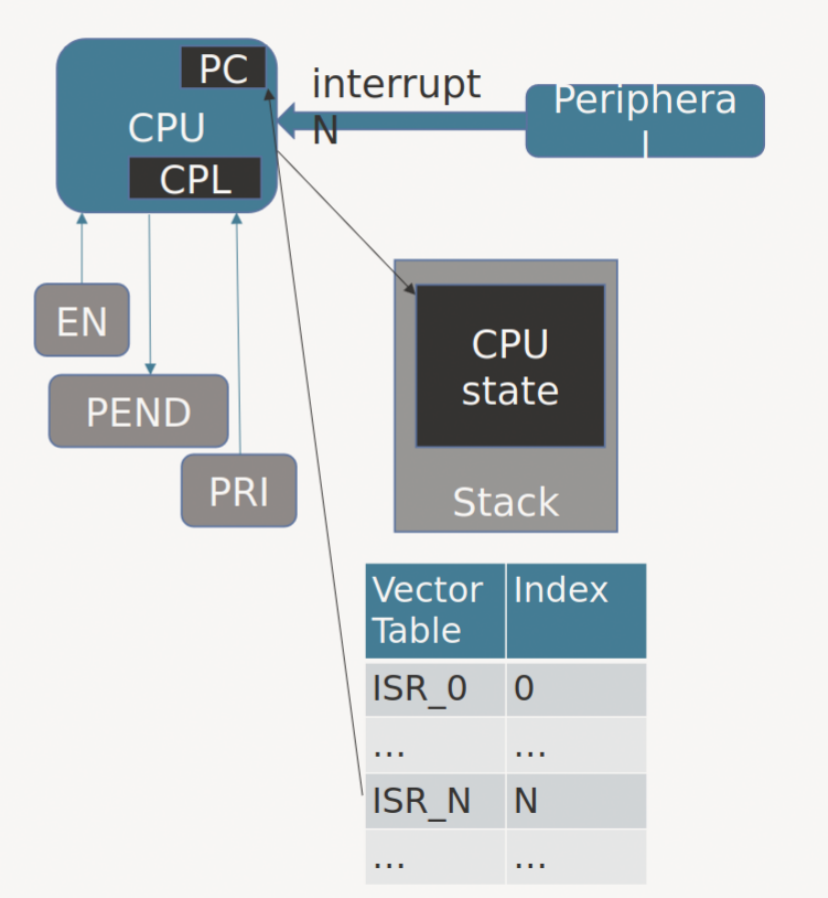


Figure 5: Interrupt Flow

Timers

In MCUs, we often want a way to periodically do something or many somethings. Every useful MCU has a hardware timer system. A *system ticker* (systick) is a special timer reserved for OS operations, but there is always a timer subsystem outside the CPU for general use.

The systick timer is a piece of hardware inside the MCU that generates an systick interrupt signal at fixed intervals. Every interrupt has a dedicated ISR to handle it, and in the case of systick the ISR is `SysTick_Handler`.

In the ARM Cortex-M, the system timer is built into the hardware of the CPU. Every time an IRQ is raised, the Nested Vectored Interrupt Controller (NVIC) hardware will determine whether or not to handle the systick interrupt.

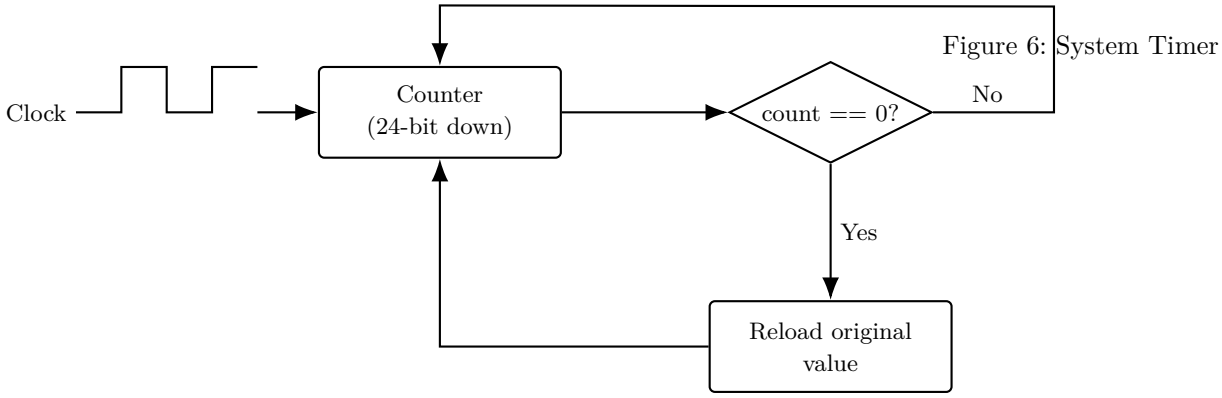


Figure 6: System Timer

When the counter hits zero, `COUNTFLAG` is set to 1. The choice of clock input can vary, as most MCUs have multiple. The clock input is ANDed with a flag to enable and disable.

For a timer, the desired interval T is

$$T = \frac{(\text{Prescalar} + 1) \times (\text{Reload} + 1)}{f} \tag{2}$$

Let's do an example. Suppose the clock tick frequency f is 80MHz and the goal is a systick interval s of 10ms. What must the reload value R be?

$$\begin{aligned} R &= sf - 1 && (3)\\ &= 10ms \times 80MHz - 1 && (4)\\ &= 800000 - 1 && (5)\\ &= 799999 && (6) \end{aligned}$$

Timers can do far more than just triggering interrupts. They can also

- Capture external events
- Drive certain peripherals like GPIO
- Output precisely timed signals

In output mode, hardware constantly compares the counter with a value stored in the *compare and capture register* (CCR), as in figure 7.

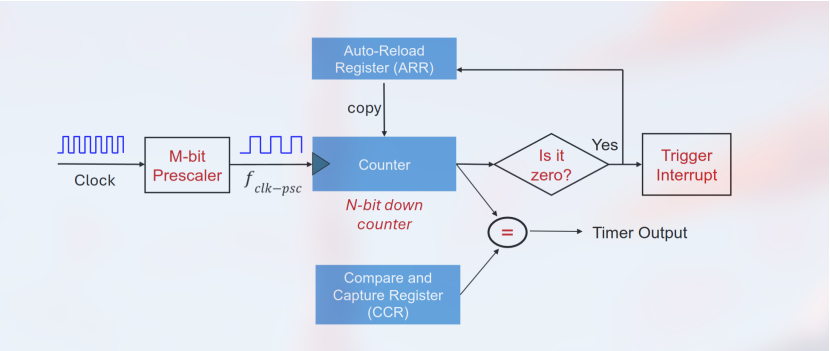


Figure 7: Timer Output Mode

In general purposes timers, you can count down, up, or up and down.

The output mode determines what happens when the CCR and counter are equal. One output mode is active high. When the timer value equals the CCR, then the timer output is set to high and stays there. Another output mode is toggle, where the output switches whenever the timer reaches the CCR value.

Pulse width modulation (PWM) is a convenient way to generate variable duty cycles. You just toggle the output each time it reaches a given value of the CCR.

$$\text{Duty Cycle} = \frac{\text{CCR}}{\text{ARR} + 1} \tag{7}$$

$$\text{Period} = (1 + \text{ARR}) \times \text{Clock Period} \tag{8}$$

PWM outputs a digital waveform (i.e. either 0 or 1), with a specific frequency and duty cycle.

We'd like to be able to have a DAC with a strong drive strength like we have with GPIO. We don't just want on/off states. We want multiple voltage levels. Consider a square wave with a varying duty cycle: We average the area under the curve. The duty cycle of the wave is then the analog level. We can average using a low-pass filter. As long as the wave frequency is much higher than the signal we want to model, no one will notice the averaging. We can accomplish averaging with low-pass filters. Each timer has a PWM mode where the output starts high, and goes low when CNT == CCRx. The higher the duty cycle, the higher the average. If the pulse frequency is much higher than that of the desired signal, the noise will be imperceptible.

Debouncing

When a mechanical button is pressed, there is a period where the signal is neither high nor low, because of vibrations in the conducting plate it connects or other non-ideal physical effects. This is known as *bouncing*, and it's rectified by adding a Schmitt trigger and RC circuit to smooth out the signal.

However, in real world systems we rarely have just one button. We often have a matrix input, like a keypad. Our Schmitt trigger setup doesn't work here, but luckily a software solution exists. Set up the CPU connected to the keypad to scan each key for being pressed. It doesn't need to check constantly, just often enough that it will catch a button press.

The work of scanning can be done incrementally using a timer interrupt. On each interrupt, the ISR will:

- read all the columns
- put the value read for each key of the current row into its own history byte
- turn off the voltage for the row
- turn on the voltage for the next row (for the next ISR invocation)
- return

The keys on a keypad can still bounce. Pressing and letting go of a button may look something like `00000001001011111...1111101000000`. In this example, the button bounces on press and release. However, the history byte eventually stabilizes and is full of entirely `1`s or `0`s. To detect a press or release, we search all the history bytes that represent the keys. The first time we detect a change, like in `00000001`, we say that is the start of a press or release. We say the press or release is done when the history byte queue is back to only one number.

We want to scan the keys faster than they can be pressed or released, but slower than the total bounce time for any key. Say a button can bounce for 10ms, and we scan one row of the 4-row keypad every 1ms. Then when you press the button, the instance you read a one, you do not know if the input is stabilized. It could still bounce, since it has been 0ms. The second time you read a bit, it could still be bouncing, since 4ms is within the 10ms bounce time. The third time you read a bit, it could still be bouncing, since it has only been 8ms. By the fourth bit you read, you are certain it is a one.

Multiplexing

Multiplexing is when you rotate through a task fast enough that it gives the appearance of being simultaneous. Key scanning is a specific example of input multiplexing and encoding. Another example is driving displays. Turn on one SSD at a time. Rotate through them rapidly enough that your persistence of vision makes it appear they are all on simultaneously and displaying different digits.

Direct Memory Access

Moving memory from one location to another makes poor use of your CPU. Moving data is so common that we build a co-processor called a *Direct Memory Access* (DMA) controller to do this without our CPU.

DMA is also useful for allowing peripherals to use memory independent of the CPU

The workflow for reading data from a peripheral with a CPU is:

1. Copy data from peripheral data register to buffer in memory
2. Copy data from buffer in memory to CPU

A DMA can just copy data directly from the register to the buffer. There are two kinds:

- Flow-through: DMA controller is used as an intermediate buffer (useful if we need to change the size of data type)
- Fly-by: DMA controller only sets up the bus between the source and destination

A DMA sits on the bus, stores a list of source/destination pairs being serviced, and is triggered by software, peripherals, timers, or other interrupts.

Digital-to-analog Converter

A digital-to-analog converter (DAC) converts digital data into a voltage signal.

$$\text{DAC}_{\text{output}} = V_{\text{ref}} \frac{\text{Digital Value}}{2^N - 1} \tag{9}$$

N is the *resolution* of the DAC, and 2^N is the number of voltage levels the DAC can generate.

Applications of DACs include digital audio, video display, and waveform generation.

A 5-bit DAC looks like Figure 8.

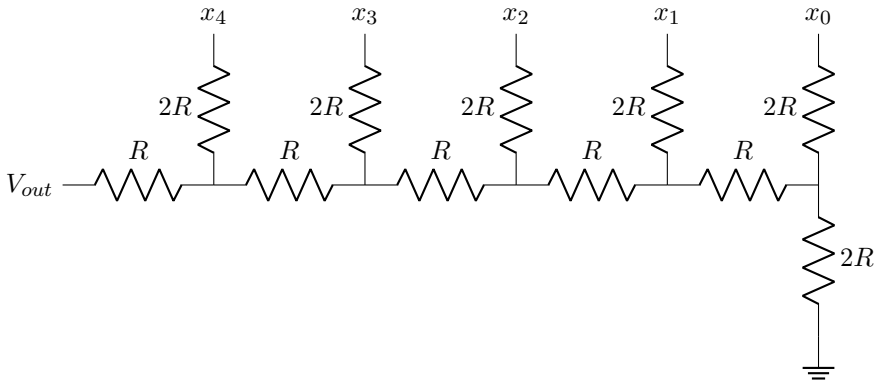


Figure 8: 5-bit DAC

The DAC conversion is nearly instantaneous. Some settling time is required due to capacitance and inductance in the circuit. No iteration or convergence is needed. However, resistor tolerance is a limitation. The higher N of the DAC, the more precise the resistors must be.

In place of a resistor ladder, a 2^N-to-1 mux can be used to select one voltage level instead.

Analog-to-digital Converter

As the name implies, an *analog-to-digital converter* (ADC) converts analog sound to a digital signal, like a microphone. The *resolution* N of an ADC is the number of binary bits in the ADC output.

An analog signal is continuous time and continuous amplitude, while a digital signal is discrete time and discrete amplitude

$$\text{ADC Output} = round\left((2^N - 1)\frac{V_{\text{input}}}{V_{\text{ref}}}\right) \tag{10}$$

V_{ref} is the maximum input voltage that can be converted by the ADC.

The least significant bit (LSB) holds the lowest value of the encoded analog signal. The maximum error due to quantization is

$$\pm\frac{1}{2}LSB = \pm\frac{1}{2}\frac{V_{\text{ref}}}{2^N - 1} \tag{11}$$

An ADC does not have infinitely fast conversion. The *sampling rate* is the number of output samples available per unit time.

The sampling frequency needed to prevent aliasing, which causes distortion, is given by the Nyquist-Shannon Sampling Theorem,

$$f_{\text{sampling}} \geq 2f_{\text{signal}} \tag{12}$$

This class focuses on *successive-approximation* (SAR) ADCs, as in Figure 9.

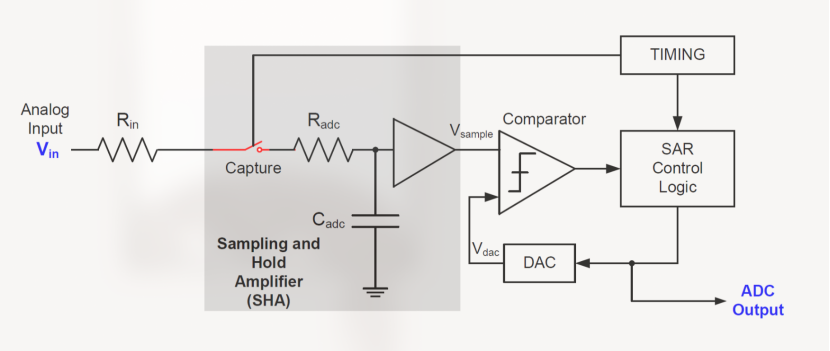


Figure 9: SAR ADC

Jitter is small rapid variations in a waveform resulting from fluctuations in the voltage supply or mechanical vibrations or other sources. Jitter is noise from the surroundings, from the microcontroller, and from the ADC itself. One of the most reliable ways of defeating noise is taking more samples than needed and averaging them out.

One type of this averaging out is called "moving window" sampling, so named because one averages N samples in a small section (window) of a signal, then moves the window by one and repeats the process.

```
#define HISTSIZE 128
```

```
int his1[HISTSIZE] = { 0 };
int sum1 = 0;
int pos1 = 0;

int his2[HISTSIZE] = { 0 };
int sum2 = 0;
int pos2 = 0;

...

reading = ADC1->DR;
sum1 -= hist1[pos1];
sum1 += hist1[pos1] = reading;
pos1 = (pos1 + 1) % HISTSIZE;
val = sum1/HISTSIZE;
sprintf(line, "%2.3f", val * 3 / 4095.0);
display1(line);

ADC1->CHSELR = 0;
ADC1->CHSELR |= 1 << 5;
while(!(ADC1->ISR & ADC_ISR_ADRDY));
ADC1->CR |= ADC_CR_ADSTART;
while(!(ADC1->ISR & ADC_ISR_EOC));

reading = ADC1->DR;
sum2 -= hist2[pos2];
sum2 += hist2[pos2] = reading;
pos2 = (pos2 + 1) & (HISTSIZE - 1);
val = sum2 >> 7;
// division is painfully slow
// if HISTSIZE is a power of 2
// replace divisions with a
// bitwise AND operation and
// right shift
sprintf(line, "%2.3f", val * 3 / 4095.0);
display2(line);
```

Serial Peripheral Interface

MCUs often need to communicate with another MCU or peripheral, on the same board or across boards. Parallel interfaces can send and receive multiple bits at the same time. Serial interfaces, however, send and receive one bit at a time.

It may seem like parallel is the obvious choice, but of course we wouldn't introduce serial interfaces if they didn't have some advantages. Parallel interfaces require more pins, wires and larger PCB footprint. Wires next to one another leads to crosstalk, electromagnetic interference. In practice systems tend to use serial interfaces. Since such interfaces send and receive only one bit at a time, there can either be one wire for both directions or one wire in each direction.

Cross-talk can be mitigated with twisted wires, but not eliminated

There are two kinds of peripherals, *synchronous* and *asynchronous*. Synchronous protocols have a clock that dictates the send/receive rate. Examples include Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I2C). Asynchronous protocols have no clock signal. The most famous example of an asynchronous protocol is Universal Asynchronous Receiver and Transmitter.

SPI is one of the most common serial interfaces. It operates in Master/Slave mode. There must be at least one master, while there can be multiple slaves. The master defines the clock and delivers a clock signal to each data recipient. It operates by shifting in bits each clock cycle.

SPI devices are designated as either masters or slaves. A master device initiates all data transfer operations. It drives the clock pin and data pin. It also drives slave select pins if slaves are selectively enabled. A slave device responds to transfer operations. Signals are named according to the devices' standpoint. E.g. MOSI: master out, slave in is the data transmitted by a master device and received by a slave device. The MISO input pin should never be left floating. It's fine to leave the MISO output floating, since it's an output and not an input.

What does this look like in practice? Say we wish to send `0xB1`. Each rising clock edge latches one bit into the slave. First, the master selects the slave by asserting the slave select signal. We send 8 bits, so there are 8 clock pulses. Finally, each clock pulse, one bit is transferred.

In this class, we assume the MSB is sent first.

The master-slave terminology is being phased out in industry. Updated datasheets might refer to SPI pins with terminology such as

The SPI clock doesn't need to be perfectly periodic and continuous. As long as it's not too fast, it can pause as needed since the slave takes the data bits only at clock pulses anyway.

SPI supports multiple slaves, which can share MOSI, MISO, and

Legacy Pin Name	Updated Pin Name
MOSI (Master Out, Slave In)	SDO (Serial Data Out), TX/TXD (Transmission)
MISO (Master In, Slave Out)	SDI (Serial Data In), RX/RXD (Reception)
SCK (Serial Clock)	SCK (Serial Clock)
NSS (Negative Slave Select)	CS (Chip Select)

Table 1: SPI Pin Names

clock lines. However, each slave needs its own select signal line. The slave selected drives the MISO line. No more than one slave may be selected at a time. SPI also supports more than one master. It is an error for multiple master devices to assert CS at the same time. Some coordination is needed to ensure that one master device reads/writes at any time. In the common multi-drop configuration, all MISO, MOSI, and SCK singals are shared, while the NSS is used to select which slave is active.

SPI is supported by LCD displays, OLED displays, SD card interfaces, and hundreds of other devices. Even if there isn't a purpose-built SPI peripheral on a board, it can be implemented with a shift register and GPIO pin.

In SPI, when sending N bits, you will receive N bits. This is unique to SPI.

Universal Asynchronous Receiver-Transmitter

The *Universal Asynchronous Receiver-Transmitter* protocol, or UART for short, is a simple, two-wire protocol for exchanging serial data.

- Universal: programmable for different use cases
- Asynchronous: sender provides no clock signal to receivers

UART can be used for communication between multiple MCUs, and MCU and its peripherals, or an MCU and PC. The FTD232 is a converter chip that converts UART to a standard USB interface. UART has a start bit and stop bit for each "frame" of data it sends, to help with asynchronous transfer. It also has a parity bit to help compensate for noise. If using odd parity, all the `1`s (including the parity bit) must add up to an odd number. Vice versa for even parity. More advanced error correction codes like Hamming or Reed-Solomon can detect or fix multiple bit flips, but these are outside the scope of this course.

The *baud* rate is the maximum rate of signal transitions per second for a serial communication system. The Baud rate is not the rate at which data is transmitted. It conveys how fast the signal changes, but not all signal changes convey meaningful data. For instance, if we have a start and stop bit, along with a parity bit, and five bits of data, then the data rate is only $\frac{5}{8}$ of the Baud rate.

RS-232 sets the recommended standard for UART. In the standard UART protocol, we make sure that the data line is kept high when idle, and use 0 as the start bit. Data bits are sent LSB first. Parity can be odd, even, or neither. The stop bit goes high. In UART, there is always a start bit. Word size ranges from 7 to 9 bits. Parity bit is option and can be even or odd. There is always a stop signal, but it can be 0.5, 1, 1.5, or 2 bit widths long. Almost everyone in the world uses 8N1, which is eight bits, no parity, one stop bit. UART has a sending pin (Tx) and a receiving pin (Rx). Both Tx and Rx can be active at the same time and independently, as seen in Figure 10.

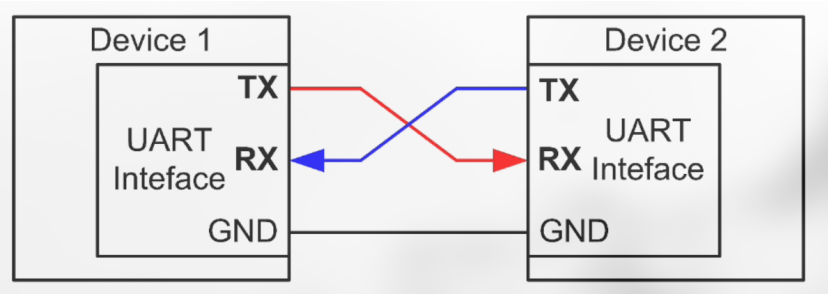


Figure 10: UART

No two clocks are the same. In UART, clocks sync on the falling edge of the start bit. There's a 1.5-bit duration wait to read the data and each new bit is read one bit-duration later. The clocks are resynced on the next start bit. The Baud rate of the receiver is allowed to differ from that of the transmitter by up to 5%. If each bit is shifted by 5%, then each successive bit will be a little later or a little earlier. This has a cumulative effect. $10.5^8 = 1.477$, still not beyond halfway at the 8th bit, but $1.05^9 = 1.551$, just over halfway at the 9th bit.

Inter-Integrated Circuit Protocol

Inter-Integrated Circuit (I2C), like SPI, is a serial communication protocol. Designed for low-cost, medium data rate applications, I2C is the de facto standard for 2-wire communications. I2C is serial, byte-oriented, multi-master, and multi-slave. It has a serial data line (SDA) and serial clock line (SCL), which need to be pulled up with resistors. It supports up to 100 kilobits a second in standard mode and 400 kilobits a second in fast mode.

SDA and SCL must be open-drain, connected to positive if the output is 1 and in high impedance state if the output is 0. The internal pull-up resistor is too weak, so we require external pull-up resistors on SDA and SCL (2kΩ for fast mode and 10kΩ for standard mode).

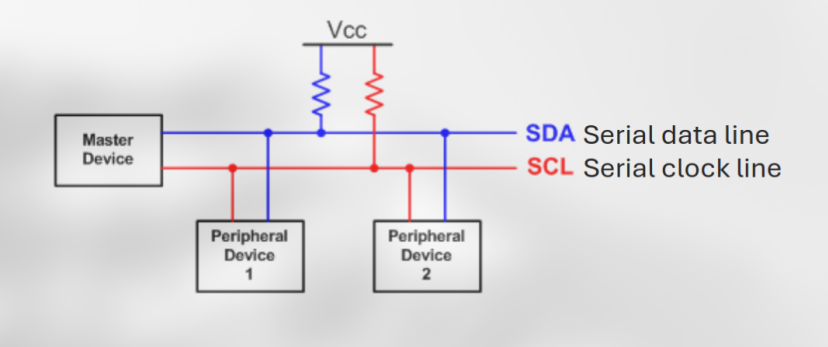


Figure 11: I2C

Normal digital logic outputs are connected to push-pull drivers. I2C signals are connected to open-drain drivers. They cannot pull up for a logic high. Instead, a pull-up resistor is responsible for keeping each signal high unless it is pulled low by any I2C driver.

While SPI has faster speed and supports full duplex, I2C is simpler and adding a new slave is easy. We prefer I2C when it's convenient to connect multiple devices using only two wires, and when speed is not critical.

I2C's slower speed is due partly to its open-drain design, which must wait for the line to be pulled high by the pull-up resistors for logic high, partly by bus capacitance, and partly by the length of the network. I2C has multiple modes that allow for faster communication:

- Standard mode: 0-100kb/s
- Fast mode: 100-400kb/s
- Fast mode plus: 400-1000kb/s
- High speed mode: 1000kb/s-3.4mbit/s

The effective data rate is less than half of the clock rate due to addressing, acknowledgements etc.

Despite (or perhaps because of) its physical simplicity, the I2C protocol is more complex than SPI. I2C does not have slave select lines; it uses either 7-bit or less commonly 10-bit slave addressing. An I2C device starts a transaction with a START (S) bit. It then sends a 7 or 10-bit address, followed by a 1-bit intent. 0 for write, 1 for read. It listens for an ACK/NACK sent by the receiver, and then sends a STOP (P) bit. The ACK is a logic low, and NACK is logic high. If no device on the I2C bus will respond to the particular address that was sent, then nothing will acknowledge. If nothing acknowledges, there is nothing to pull the line low. This indicates failure. Both addresses and data are sent most-significant-bit first. The I2C protocol makes no definitions for the contents of the data fields.

START and STOP bits are the only times that SDA changes when SCL is held high.

When a master device writes a command and single data byte to a slave device, it looks like Figure 12

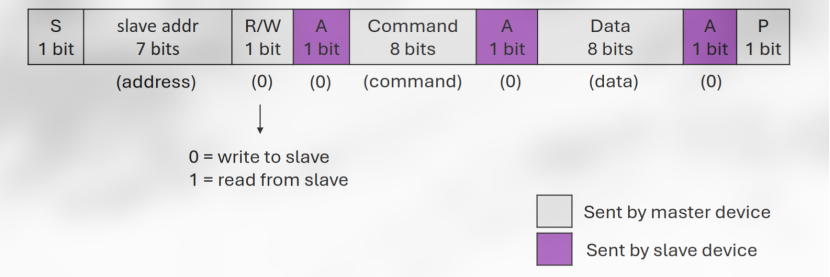


Figure 12: I2C Command Write

When a master device reads two data bytes from a slave device, it looks like Figure 13.

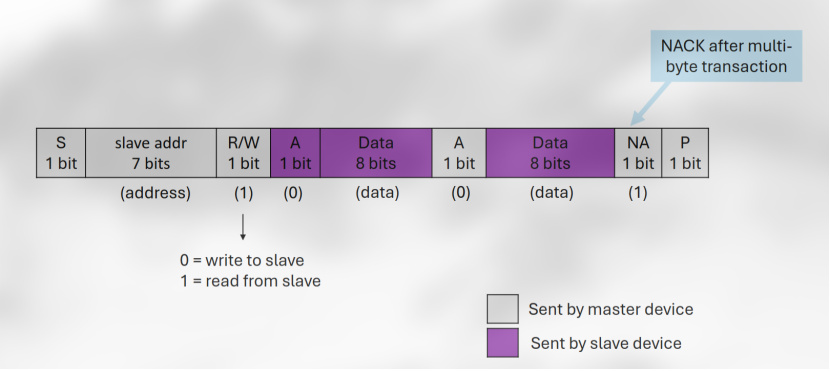


Figure 13: I2C Double Read

Both the data and clock lines of an I2C bus are open-drain. When clock stretching is enabled, any device on the I2C bus can lengthen the low time of a clock. The master drives the clock, but a slave device

may not be able to keep up. Any slave device may lengthen the clock only after the ACK bit and before the MSB of the next byte.

Computer Design

As good computer engineers, we ought to study computer design so we can design new computers and understand the ones that already exist.

In this class, we generally represent numbers in binary, or more commonly, hex. Converting from hex to binary is simple: take each hex digit, convert it to its corresponding binary representation, and insert that in the corresponding space in the binary number. For example, `0xeca8` becomes `1110 1100 1010 1000`.

There are two broad categories of microcontrollers, general purpose and embedded. General purpose are programmable and flexible. Embedded are extremely optimized, whole-system, non-programmable application specific devices.

The life of a silicon wafer is shown in Figure 14. The faulty chips

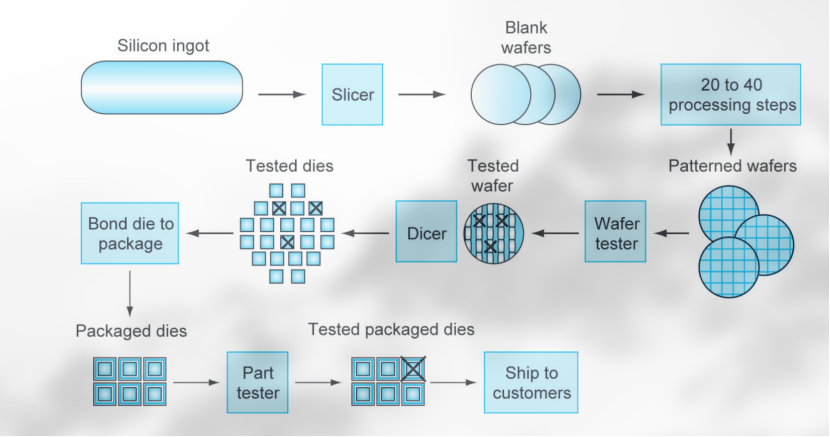


Figure 14: Silicon Lifecycle

that fail to meet speed benchmarks are sold as lower-grade chips in a process termed *binning*.

Even the simplest modern computers take thousands of engineers to design. It isn't feasible to understand everything about a computer at every level, so we must abstract and think of the system in terms of black boxes with inputs and outputs. This applies to electrical engineering, computer engineering, and programming. A high-level language like Python is an abstraction of C, which is an abstraction of assembly language, which is an abstraction of binary machine language.

```
// Consider the C statement
f = (g + h) - (i + j);
// The corresponding
// assembly could be
add t0, g, h   // t0 = g + h
add t1, i, j   // t1 = j + j
```

```
sub f, t0, t1 // f = t0 - t1
// The corresponding machine
// instruction would be 0s
// and 1s
```

The assembly code corresponding to a given statement in C depends on the device. Devices may run x86, ARM, or RISC-V, which are known as *instruction set architectures* (ISAs). There are three different types of ISAs:

- CISC (Complex Instruction Set Computer)
- RISC (Reduced Instruction Set Computer)
- DSPs (Digital Signal Processors)

In this class we focus on RISC, specifically RISC-V.

The general instruction cycle for what a computer is doing under the hood is:

1. Fetch: retrieve the next instruction from memory.
2. Decode: figure out what the instruction needs to do by determining what operations and additional operands are required for execution and sends respective signals to respective components within the CPU, such as the ALU or FPU, to prepare for the execution of the instruction.
3. Execute: the stage where the actual operation specified by the instruction is carried out by the relevant functional units of the CPU. Logical or arithmetic operations may be run by the ALU, data may be read from or written to memory, and the results are stored in registers or memory as required by the instruction. Based on output from the ALU, the PC might branch.
4. Update PC: point the program counter at the next instruction.

All CPUs use a pipeline, which could have as few steps as these four steps or as many as twelve.

Instruction Set Architecture

All computers share similar constraints in designing instructions. The specifications for a machine's instructions and functional operation are called an *instruction set architecture* (ISA). The fundamentals of one flavor of ISA extrapolate to another. In this class, we study the fundamental subset of RISC-V that is common to all ISAs.

Register Arithmetic

Recall the example of converting C to assembly.

```
f = (g + h) - (i + j);
// add t0, g, h
// add t1, i, j
// sub f, t0, t1
```

`g`, `h`, `i`, `j`, `t0`, and `t1` are the operands and are stored in *registers*. Registers are word-sized (typically, 8- to 64-bit) storage elements inside the core itself. The 32 registers in 32-bit architecture are fast locations for storing and accessing data. In RISC-V, data must be in registers to perform arithmetic. Register `x0` always equals zero. In addition to the registers, there are also 2^{30} memory words accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers.

The RISC-V 32I ISA has 32-bit registers: `x0-x31`.

The RISC-V assembly language has instructions like in Table 2.

Instruction	Example	Meaning
Add	`add x5, x6, x7`	`x5 = x6 + x7`
Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`
Add immediate (for constants)	`addi x5, x6, 20`	`x5 = x6 + 20`

Table 2: RISC-V Assembly Arithmetic

```
main:
    li x10, 1 // li = load immediate
    li x11, 2 // manually initialize
    li x12, 4
    li x13, 5

    add x10, x10, x11 // now, x10 = x10 + x11
    add x12, x12, x13 // now, x12 = x12 + x13
    add x10, x10, x12 // now, x10 = (x10 + x11) + (x12 + x13)
```

There are general purpose and special registers, as tabulated in Table 3.

Register	ABI Name	Description	Saver
x0	zero	Hardwired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5	t0	Temporary register	Caller
x6	t1	Temporary register	Caller
x7	t2	Temporary register	Caller
x8	s0/fp	Saved register / Frame pointer	Callee
x9	s1	Saved register	Callee
x10	a0	Function argument / Return value	Caller
x11	a1	Function argument / Return value	Caller
x12	a2	Function argument	Caller
x13	a3	Function argument	Caller
x14	a4	Function argument	Caller
x15	a5	Function argument	Caller
x16	a6	Function argument	Caller
x17	a7	Function argument	Caller
x18	s2	Saved register	Callee
x19	s3	Saved register	Callee
x20	s4	Saved register	Callee
x21	s5	Saved register	Callee
x22	s6	Saved register	Callee
x23	s7	Saved register	Callee
x24	s8	Saved register	Callee
x25	s9	Saved register	Callee
x26	s10	Saved register	Callee
x27	s11	Saved register	Callee
x28	t3	Temporary register	Caller
x29	t4	Temporary register	Caller
x30	t5	Temporary register	Caller
x31	t6	Temporary register	Caller

Table 3: RISC-V Registers

The operands for arithmetic instructions always come from registers. If the values of an operand is known ahead of time, then `addi` can be used. This is useful for adding constants to operands.

An *addressing mode* defines how an operand gets its values. All ISAs have a limited number of registers. The more you add, the slower they get and the more circuitry required to access. That means virtually all machines have memory, which is byte-addresses. Address 0 is byte 0, address 1 is byte 1, etc. An ISA defines many things, including the interaction between processor and memory. For instance, two

different ISAs may compute "Z = X + Y" differently. One might use the accumulator method consisting of

```
LoadA X
AddA Y
StoreA Z
```

while most others use the reg/reg method of

```
Load R1, X
Load R2, Y
Add  R3, R1, R2
Store R3, Z
```

Basic memory instructions in RISC-V are

	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
h!	Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory

Memory in RISC-V is byte-addressed. 4B, 2B, and 1B memory operations are all available. All registers on the RISC-V machine are 32 bits (4B). When you load something less than 4 bytes, the upper bits are generally zeroed or sign extended.

An addressing mode defines how an operand gets its values.

- Register addressing: read the value from a register
- Immediate addressing: read the value from a constant burned into the instruction.
- Offset addressing: read the value from memory. The address is computed by adding an immediate to the value in the register.

Consider a 4 byte word storage example with the address of `a` at `0x400`. Are the bytes of the word located at `0x400, 0x401, 0x402, 0x403` or `0x403, 0x402, 0x401, 0x400`? It depends on the ISA. In *big-endian* architectures, the word is stored as `0x400, 0x401, 0x402, 0x403`. These kind of systems store the most significant byte of a word at the smallest memory address and the least significant byte at the largest. In *little-endian*, the least significant byte is stored at the smallest address, like `0x400, 0x401, 0x402, 0x403`. RISC-V can be either, but in this class we'll stick with little-endian.

The location of the sign bit is always at the MSB for either little or big endian.

Consider the number `0xffffffff`. The decimal number this represents depends on if we interpret it as signed or unsigned. If it is a signed integer, it is -1. If it is an unsigned integer, it is 4294967295. Recall for unsigned integers, n bits gives 2^n combinations. We can treat unsigned integers as normal binary numbers, so the maximum is $2^32 - 1$ and the minimum is 0. Signed integers are slightly more

complex. Because one bit is taken as the sign bit, their range is -2^{31} to 2^{31-1}. To negate a 2's compliment signed number, flip all bits and add 1. To load a 16-bit representation of a signed number into a 32-bit register, replicate the MSB into the empty space.

All instructions in RISC-V are just 32 bits. Fields of the instructions are kept as consistent as possible.

R-type instructions have the form in Figure **??**.

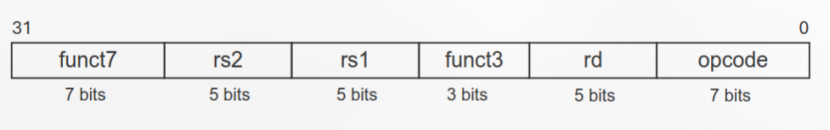


Figure 15: R-type Instructions

- `opcode`: partially specifies instruction
- `funct7 + funct3`: combined with opcode, specifies what operation
- `rs1`: first operand (source register 1)
- `rs2`: second operand (source register 2)
- `rd`: destination register (receives the result of the computation)

For example, the operation `x9, x20, x21` is `0x015A04B3`.

I-type instructions are immediate arithmetic and load instructions, and have the form given in Figure 16.

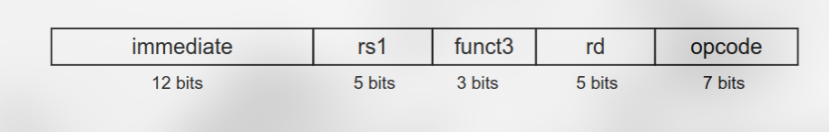


Figure 16: I-type Instructions

- `rs1`: source or base address register number.
- `immediate`: constant operand, or offset added to base address.

S-type instructions have the form in Figure 17

Figure 17: S-type Instructions

- `rs1`: base address register number
- `rs2`: source operand register number

- `immediate`: offset added to base address

What separates a computer from a calculator is its ability to take different actions based on a comparison, e.g. `if ... else`.

In RISC-V, two conditional branch instructions are

- `beq`: compare two values and branch if equal. What happens behind the scenes is that the instruction reads the values in the registers and if they are equal, jumps program to label
- `bne`: compare and branch if not equal.

In both cases, if the comparison is not true, the program goes to the next instruction.

Listing 1: Conditional Example

```
// The below assembly is
// equivalent to the snippet
// if (i == j) f = g + h; else f = g - h;
// in C, if f, g, h, i, j
// are stored in x19-x23

bne x22, x23, Else       // go to Else if i != j
add x19, x20, x12        // f  g + h (skip if i!= j)
beq x0, x0, Exit         // if 0 == 0, go to Exit
Else: sub x19, x20, x21 // f = g - h (skip if i == j)
Exit:
```

Listing 2: Loop Example

```
// The below assembly is
// equivalent to the snippet
// while (save[i] == k)
//   i += 1;
// in C

// a0 = i
// a1 = k
// s1 = base address of 'save'

// Basic block 1
loop: slli t0, a0, 2 // tmp reg t0 = i * 4
      add t0, t0, s1 // t0 = address save[i]
      lw t2, 0(t0)   // tmp reg t2 = save[i]
// Basic block 2
      bne t2, a1, exit
```

```
        addi a0, a0, 1
// Basic block 3
        beq x0, x0, loop
exit:
```

A *basic block* is a sequence of code without a branch or a target, except at the beginning or end.

Testing for equality is simple and intuitive. We want to test other stuff too, like greater than, less than, greater than or equal to, or less than or equal to. In RISC-V, the available instructions are

- `beq`: branch equal
- `bge`: branch greater than or equal
- `bgeu`: unsigned of the above
- `blt`: branch less than
- `bgtu`: unsigned of the above
- `bne`: branch not equal

The remaining cases can be constructed from the provided instructions.

Procedures

Programs more complicated than simple arithmetic need support for higher-level programming structures. A universal concept in programming is *procedures*, aka routines, methods, or function calls.

To implement a function call in assembly, we need to be able to pass arguments, jump to the function, return from the function, put the result somewhere the caller can see it, and least obviously, split/fill registers.

Every ISA has a convention for passing parameters to a function. Typically, the first N arguments are stored in registers. In RISC-V `x10-x17` (alias `a0-a7`) hold the first 8 function arguments. That means, before you call any function, the code that calls the function puts the function arguments in the right order in the registers `x10-x17`.

To get to a function, we need to branch to a labelled address. To get back, we need to remember where we came from. Almost all ISAs have a special jump-and-link (`jal`) instruction which unconditionally moves to the label and stores the location after the jump in a register.

For a complete example of what's needed in RISC-V to call a function, see Listing 3.

Listing 3: Arguments in Registers

```
li x10, 5 // 5 is the first argument
```

```
li a1, 7  // 7 is the second argument (a1 is aliased to x11)
// Call foo, put the return address in x1
jal x1, foo

foo:
    add a0, a0, a1
    // Return to the caller
    jalr x0, 0(x1)
```

The temporary registers `x5-x7` and `x28-x31`, aliased as `t0-t6` don't have to be preserved by the called function. However, the saved registers `x8-x9`, aliased as `s0-s11`, must be preserved by the callee.

A non-leaf function is one that calls other functions. Before it calls another function, it must preserve its return address and the arguments and temporaries needed after the call on the stack. When the called function returns, the calling function must restore the values it had saved to the stack.

Stack

Curious readers may wonder what happens when a function requires more than 8 parameters, or, in general, what happens when a program needs more registers to store data than are available. All computers have a stack data structure to handle copying register value from register to memory (spills) and getting the register value back from memory to register (fills). The stack pointer register `sp` points to the memory address of the most recently allocated entry of the stack. Pushing moves the stack pointer to allocate space, then puts register data on the stack. Popping takes data off the stack, puts it in a register, then moves the stack pointer to deallocate space.

The stack historically grows down in memory. The stack pointer may start at some high address like `0xffff` and be decreased to put new entries on (spilling). Once those entries are moved back into registers (filling), the stack pointer increases again. The stack pointer always points to the top of the stack.

Program Counter

The program counter is a fundamental concept. Every computer has one. The program counter stores the address of the current instruction in the special register `pc`.

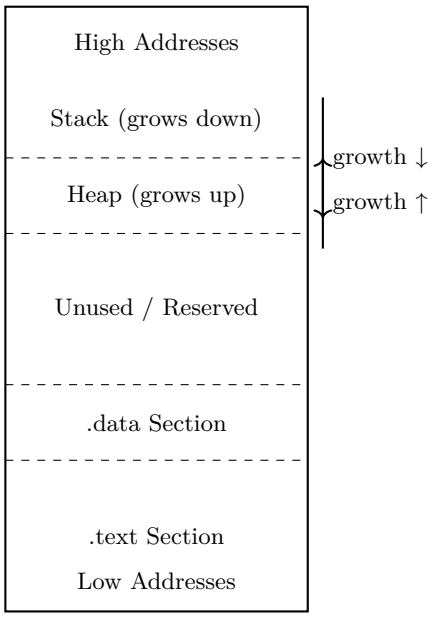
Another special register, `ra`, holds the return address for after a function call completes. When a function is called, the address immediately after the current line is pushed into `ra`. Once the function call completes (when the `jalr` instruction is reached), the program counter is set to the value in `ra` and the program resumes execution where

it left off. The next time a function call is reached, the value in `ra` is changed to the address of the line right after that function call, and so on.

Functions may have local data that also goes on the stack, such as a local array. It's useful to know where the frame for each function's data begins. Some computers will use the register `fp` to store this frame pointer.

Memory Layout

The memory in a running program is divided into several regions, each serving a specific purpose. The figure below illustrates a typical layout in RISC-V systems. The *stack* grows downward from high memory addresses, while the *heap* grows upward from low addresses. Between them lies a large unused region reserved for future allocations. The `.text` section contains the compiled program instructions, and the `.data` section contains global or static variables.



The `.text` Section The `.text` section of a program holds the compiled machine instructions that make up the executable code. This region is typically marked as read-only to prevent accidental modification of instructions at runtime.

The `.data` Section The `.data` section stores global and static variables that have been initialized before the program starts running. Variables declared with initial values (like `int x = 5;`) reside here, while uninitialized globals are placed in a separate `.bss` section.

Application Binary Interface

The Application Binary Interface (ABI) defines how compiled code interacts at the binary level. It specifies calling conventions, register usage, stack organization, and how data types are represented in memory. The ABI ensures that separately compiled modules, such as libraries and user code, can work together correctly. In RISC-V, the ABI defines which registers hold function arguments and return values, which must be preserved across calls, and how the stack frame is structured.

Name	**Register Number**	**Usage**	**Preserved on Call?**
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5–x7	5–7	Temporaries	no
x8–x9	8–9	Saved	yes
x10–x17	10–17	Arguments/results	no
x18–x27	18–27	Saved	yes
x28–x31	28–31	Temporaries	no

Table 4: Register Conventions