Chapter 1

Cybernetics

Cybernetics is the science of communications and automatic control systems in machines and living things. For the desired degree of responsiveness and reliability in cyber-physical systems, the effective coordination of such machines generally requires a certain degree of **edge computing** (**decentralized**, calculated on the machines themselves). **Cloud computing** (**centralized** on large remote clusters of computers, aka servers), together with fast wired (§3.1.2) or wireless (§3.1.3) communication protocols, often complements edge computing for complex coordination tasks. We thus begin this study with a survey of the essential ideas and modern technologies that underlie the remarkable performance of both small computers and large servers today.

Contents

1.1 Bits, bytes, integers, floats, and parity

A brief review of digital logic, storage, and communication forms our initial focus.

1.1.1 CMOS vs TTL; binary & hexadecimal

The starting point for the **binary** (two-state) digital logic used in modern CPUs is the **binary digit** (**bit**), a signal voltage that is either **logical low** (somewhere near ground), **logical high** (somewhere near V_{DD}), or quickly transitioning from one of these binary states to the other (in between the clock pulses that coordinate the computations on the CPU). Different MCUs and peripherals, and indeed different regions within a single MCU, use different values for V_{DD}. Within CPU cores, low-voltage complementary metal oxide semiconductor (**CMOS**) technology is used, with a V_{DD} of one of the following: {3.3V, 2.5V, 1.8V, 1.5V, 1.2V, 1.0V, 0.9V, 0.8V, ...}, with signals in the range $(0, V_{DD}/3)$ interpreted as logical low, and $(2\,V_{DD}/3, V_{DD})$ interpreted as logical high. To improve the performance of the ever-decreasing transistor sizes within MCUs, the value of V_{DD} used within CPU cores has been gradually decreasing over the years. Note also that most modern MCUs incorporate one or more Low-dropout (LDO) regulators to provide very stable power at the precise voltage(s) necessary for the MCU to function properly.

Between the MCU and other components (elsewhere on the motherboard, on daughterboards, or on the electromechanical system itself), transistor-transistor logic (**TTL**) is often[1] used, with the range $(0, 0.8\text{V})$ interpreted as logical low, and $(2\text{V}, V_{CC})$ interpreted as logical high, where V_{CC} is either 3.3V or 5V. MCUs with 3.3V TTL inputs & outputs (**i/o**) can thus communicate seamlessly with 5V TTL peripherals; however (**warning!**) this only works if those pins set as inputs on the 3.3V TTL device are rated as **5V tolerant**, which must be checked. If they are not, a **level shifter** must be used between the two connected devices.

A collection of 4 bits is called a **nibble**, which represents a number between 0_{10} (a.k.a. 0000_2 or 0_{16}) and 15_{10} (a.k.a. 1111_2 or F_{16}), where the subscript indicates the base of the number system used, with 2 denoting binary, 10 denoting decimal, and 16 denoting hexadecimal notations. Similarly, a collection of 8 bits is called a **byte**, which represents a number between 0_{10} (a.k.a. $0000\,0000_2$ or 00_{16}) and 255_{10} (a.k.a. $1111\,1111_2$ or FF_{16}). Many different notations are used to indicate the representation of numbers with different bases, including, for example, the representation of 184 in decimal (which is commonly indicated with no ornamentation) as 0b10111000 or 1011 1000b in binary, and as 0xB8 or #B8 in hexadecimal.

A collection of 3 bits may be used to represent a number between 0_8 (a.k.a. 000_2) and 7_8 (a.k.a. 111_2), referred to as an **octal** (base-8) number. Three octal digits (that is, 9 bits, denoted rwxrwxrwx) are used by the linux chmod command (see §2.2.3.2) to set {read, write, execute} permissions on a file for the {owner, group, world}.

Ternary (three-state) logic is also sometimes used in creative ways in embedded systems, particularly with arrays of buttons and LEDs (see §6.13). That is, a single binary (logical 0 or 1) output signal on some pin can also be set as an input on that device, which effectively puts it into a third state Z, known as **high impedance**. Setting such a pin as a logical 0 output can, for example, drive an LED connected (through a resistor) to V_{CC}, whereas setting it as a logical 1 output can drive a different LED connected (through a resistor) to GND, and setting such a pin to the high impedance state Z (that is, setting it as an input) turns both connected LEDs off.

Using **multi-level cell** (MLC) flash memory technology, four-state [two-bit] logic is commonly used for each individual storage symbol, and both eight-state [three-bit, a.k.a. triple-level cell (TLC)] and even sixteen-state [four-bit, a.k.a. quadruple-level cell (QLC)] logic has been explored, as discussed further in §11.3.2.

[1] A notable exception is that daughterboards for the 96boards family of motherboards operate i/o at CMOS signal levels of 1.8V.

SI prefix name	kilo-	mega-	giga-	tera-	peta-	exa-	zetta-	yotta-
SI prefix symbol	k	M	G	T	P	E	Z	Y
decimal power	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}	10^{21}	10^{24}
binary prefix name	kibi-	mebi-	gibi-	tebi-	pebi-	exbi-	zebi-	yobi-
binary prefix symbol	Ki	Mi	Gi	Ti	Pi	Ei	Zi	Yi
binary power	2^{10}	2^{20}	2^{30}	2^{40}	2^{50}	2^{60}	2^{70}	2^{80}
ratio	1.024	1.0486	1.0737	1.0995	1.1259	1.1529	1.1806	1.2089

Table 1.1: Decimal powers, as used in SI, versus binary powers, as used in characterizing computer systems.

uint8_t	int8_t	uint16_t	int16_t	uint32_t	int32_t	uint64_t	int64_t
0 : 255	− 128 : 127	0 : 65,535	− 32,768 : 32,767	$0 : 2^{32} - 1$	$-2^{31} : 2^{31} - 1$	$0 : 2^{64} - 1$	$-2^{63} : 2^{63} - 1$

Table 1.2: Ranges covered by $N = 8$, 16, 32, and 64 bit binary representations of unsigned and signed integers.

A large number of bits (abbreviated with a lowercase b) or bytes (abbreviated with an uppercase B) is indicated using a prefix corresponding to a binary power that is close to, but not quite the same as, the corresponding decimal power used in the International System of Units (SI; see §11.1.1-11.1.2), as indicated in Table 1.1. Thus, unambiguously, a Kib is 1024 bits, a KiB is 1024 bytes, a MiB is 1,048,576 bytes, a GiB is 1,073,741,824 bytes, etc. Quite unfortunately, as of 2021, SI prefixes (representing decimal powers) are still used quite often for the nearby binary powers in the computer literature, commonly denoting 1024 bits as a kb (or Kb), 1024 bytes as a KB, 1,048,576 bytes as a MB, 1,073,741,824 bytes as a GB, etc. We eschew this (sloppy) dominant paradigm in this text, simply by inserting an "i" as the second character of each prefix when denoting storage capacities, communication speeds, etc, as the percentage uncertainty that is introduced by doing otherwise increases as you move to the right in the above table (which is certainly the trend when quantifying storage capacities and communication speeds as time goes forward!), and encourage hardware manufacturers, retailers, tech reporters, book/wikipedia authors, researchers, instructors, bloggers, and others to do the same.

1.1.2 Integer & fixed-point representations, and their (fast) arithmetic in ALUs

Integer arithmetic on MCUs is usually formed using binary representations of integers that are $N = 8$, 16, 32, or 64 bits long, and either unsigned or signed, covering the (decimal) integer ranges indicated in Table 1.2.

When storing or transmitting a multiple-byte **word** (containing one or more integers, fixed-point real numbers, or floating-point real numbers; see §1.2) in a computer, the individual bytes stored (or, transmitted over a communication channel) that make up such a word can be ordered using one of two different conventions:

- with the **big endian** convention, the "big end" (that is, the most significant byte, aka **MSB**, in the sequence) is stored first (at the lowest storage address), or transmitted first, whereas
- with **little endian** convention, the "little end" (the least significant byte, or **LSB**) is stored or transmitted first.

For example, the two bytes (16 bits) required for representing the integer $A2F3_{16}$ is stored as $A2_{16}$ at memory address a and $F3_{16}$ at memory address $a+1$ using the big-endian convention, and the same integer is stored as $F3_{16}$ at memory address a and $A2_{16}$ at memory address $a+1$ using the little-endian convention. Within a byte, the order of the bits is usually stored the same (most significant bit to least significant bit) in all computers, regardless of how the bytes are arranged; however, the terms big endian vs little endian may also be used to characterize the order in which individual bits are transmitted over a communication channel.

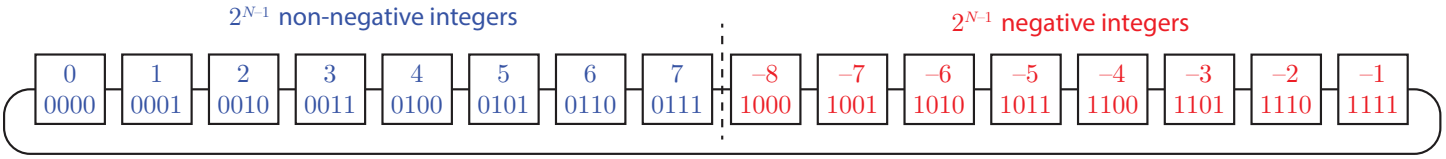

Figure 1.1: Periodic number line useful in visualizing two's complement notation.

Signed representations of negative integers are formed using the **two's complement** convention, given by simply inverting the N bits of the corresponding (unsigned) integer (in binary form) and adding one. This effectively scoots the set of all 2^{N-1} negative integers included in the representation to the right of the 2^{N-1} non-negative integers on a number line ordered by the raw (unsigned) binary number, as illustrated for the $N = 4$ case in Figure 1.1. Adding 1 (that is, 0...01) to any number on this periodic number line shifts it to the right by one, modulo 2^N (that is, moving off the right end of the line wraps back around on the left end). Similarly, adding -1 (that is, 1...11) to any number on this line corresponds to shifting it to the left by one, modulo 2^N (that is, moving off the left end of the line wraps back around on the right end). Thus, normal unsigned binary addition on this number line, ignoring binary overflow (that is, ignoring the process of wrapping around on the ends of the line) corresponds to the addition of positive and negative integers in this two's complement convention, as long as **integer overflow** (that is, exceeding the range indicated in Table 1.2, and thus crossing the halfway point on the number line, indicated by the vertical dashed line in Figure 1.1 in the $N = 4$ case) is not encountered (which can be checked for and flagged). All modern CPU cores include (fast) hardware implementations (by an **arithmetic logic unit**, or **ALU**) of the $\{+, -, \times\}$ operations on integers represented in such binary forms, which (remarkably) generally execute in a single clock cycle.

Binary representations of unsigned or signed integers, and the fast (ALU) implementations of $\{+, -, \times\}$ acting thereon, can be applied directly to real (rational) numbers with a fixed (specified in advance) number of binary digits after the (implied) decimal point. This representation of **fixed point** real numbers, using N bits, is referred to as **Q format**, and is commonly denoted UQ$m.n$ [a.k.a. UQn] for unsigned real numbers, and Q$m.n$ [a.k.a. Qn] for signed real numbers (in two's complement format), where n indicates the number of binary digits after the decimal point, and (optionally) m indicates the number of binary digits before the decimal point, with $m + n = N$. Addition and subtraction of two fixed-point real numbers [once aligned to the same Q format, so they have the same number of binary digits after the (implied) decimal point] is the same as integer addition and subtraction using binary representations; again, integer overflow must be checked for and flagged if it occurs. Multiplication of two fixed-point real numbers is, conceptually, the same as integer multiplication using binary representations. In addition, note that the product of a Q$m_1.n_1$ real number and a Q$m_2.n_2$ real number results in a Q$m.n$ real number with $m = m_1 + m_2$ and $n = n_1 + n_2$; the result must thus generally be both rounded (reducing the number of significant digits kept after the decimal point) and checked for overflow in order to fit it into another N bit Q format representation. As much as possible, scaling all fixed-point real variables in a problem (both before and after the necessary sums, differences, and products) to be $O(1)$ over the entire operational envelop of the electromechanical system under consideration is particularly convenient, using, e.g., the UQ1.7 (in the range $[0, 1.99219]$), Q1.7 (in the range $[-1, 0.99219]$), UQ1.15 (in the range $[0, 1.9999695]$), and Q1.15 (in the range $[-1, 0.9999695]$) formats[2]. Note that:

• To convert a real number r into Q$m.n$ format, multiply r by 2^n, round to the nearest integer, and convert this integer to two's complement binary form.

• To convert a number b in Q$m.n$ format back to a real number, consider b as a regular binary number (with no decimal point), convert this binary number (in two's complement form) to an integer, and divide by 2^n.

[2] In general, the range of a UQ$m.n$ number is $[0, 2^m - 1/2^n]$, and the range of a UQ$m.n$ number is $[-(2^{m-1}), 2^{m-1} - 1/2^n]$.

1.1.3 Floating-point representations, and their (fast) arithmetic in FPUs

It is, of course, significantly easier to program, especially at the prototyping stage, using **floating-point arithmetic** [that is, using real numbers represented with a **sign bit**, an **exponent**, and a **significand** (a.k.a. **mantissa**)], so that the scaling of the real numbers can be managed by the CPU core on the fly, and a very wide range of scalings can be encountered without loss of precision. Floating point real numbers, as defined by the ubiquitous IEEE 754 standard, are represented with:
• $N = 16$ bits ("half precision"), with 1 sign bit, 5 bits defining the exponent, and $k = 10$ bits defining the significand, representing numbers from $\pm 6.10 \times 10^{-5}$ to ± 65504 with $\log_{10} 2^{k+1} = 3.3$ decimal digits of precision,
• $N = 32$ bits ("single precision"), with 1 sign bit, 8 bits defining the exponent, and 23 bits defining the significand, representing numbers from $\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$ with 7.2 decimal digits of precision, or
• $N = 64$ bits ("double precision"), with 1 sign bit, 11 bits defining the exponent, and 52 bits defining the significand, representing numbers from $\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$ with 16 decimal digits of precision.

For the feedback control of electromechanical systems, single precision is more than enough, and in most cases half precision is sufficient (if the FPU implements it; as of 2021 most do not, though Armv8.1-M introduces hardware support for half-precision floats to the ARM Cortex M family, starting with the Cortex M55).

In addition to nonzero normal numbers (that is, floating-point numbers that can be represented in half, single, or double precision as defined above, without leading zeros in their significand), various special values are represented and handled correctly by FPUs implementing the IEEE 754 standard, specifically:
• signed zeros $\{+0, -0\}$ [with $(+0) = (-0)$ for the purpose of comparisons],
• signed infinities $\{+\infty, -\infty\}$ [e.g., $1/(+0) = (+\infty)$, $1/(-\infty) = (-0)$, $(+\infty) * (-2) = (-\infty), \ldots$],
• subnormal numbers [that is, smaller floating-point numbers that can be represented in half, single, or double precision at reduced precision, with leading zeros in their significand],
• Not a Numbers (NaNs), handling indeterminant forms [e.g., $(\pm\infty) \times (\pm 0)$, $(\pm 0)/(\pm 0)$, $(+\infty) + (-\infty), \ldots$], real operations with complex results [e.g., $\sqrt{-1}$], and operations involving one or more NaNs as arguments.

For example, taking s as the sign bit, e as the exponent, and f as the fractional part of an $N = 32$ bit binary representation of a floating-point number in single precision format as follows,

s	e (8 bits)								f (23 bits)																		
b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	b_{23}	b_{22}	b_{21}	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

and defining $e_{\max} = FF_{16} = 255_{10}$ and $e_{\text{off}} = 7F_{16} = 127_{10}$, the IEEE 754 standard interprets cases with:
• an exponent e of 01_{16} to $(e_{\max} - 1)$ as denoting a nonzero normal number given by $(-1)^s \times 2^{e - e_{\text{off}}} \times 1.f$
• an exponent e of 00_{16}, with $f \neq 0$, as denoting a subnormal number given by $(-1)^s \times 2^{-(e_{\text{off}} - 1)} \times 0.f$,
• an exponent e of 00_{16}, with $f = 0$, as denoting a signed zero, with sign given by $(-1)^s$,
• an exponent e of $e_{\max}$, with $f = 0$, as denoting a signed infinity, with sign given by $(-1)^s$, and
• an exponent e of $e_{\max}$, with $f \neq 0$, as denoting an NaN.

The half precision ($N = 16$ bit) format is analogous, with $e_{\max} = 1F_{16} = 31_{10}$ and $e_{\text{off}} = F_{16} = 15_{10}$; the double precision ($N = 64$ bit) format is also analogous, with $e_{\max} = 7FF_{16} = 2047_{10}$ and $e_{\text{off}} = 3FF_{16} = 1023_{10}$.

Interrogation of the individual bits of a floating-point representation might occasionally be useful to the embedded programmer, and in this setting the above explanation should suffice. The actual encoding of the fundamental operations $\{+, -, \times, \div\}$ on real numbers represented in floating-point notation is rather complex, and is taken care of remarkably quickly (again, in many cases, executing in a single clock cycle!) by the **floating point unit**s (**FPU**s) within modern CPU cores, and the MCUs which embed them.

Integer arithmetic (§1.1.2) is significantly simpler for a processor to execute than floating-point arithmetic. Thus, many auxiliary processing units (see §1.5.3-1.5.4), like FMACs and DSPs, and indeed many low-cost MCUs (like the ARM Cortex M0 and some implementations of the M3 and M4), do not include hardware FPUs, and thus floating-point arithmetic must instead be emulated in software on these processors, which is relatively

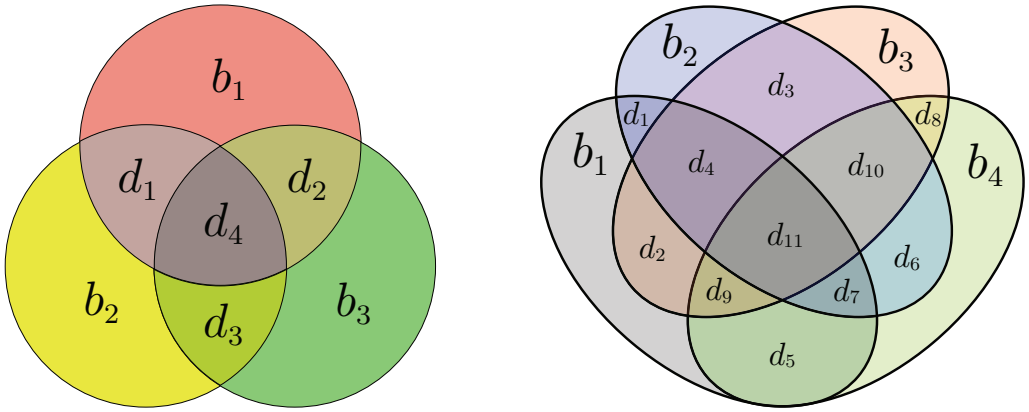

Figure 1.2: Venn diagram illustrations of (left) the [7,4] Hamming code and (right) the [15,11] Hamming code.

slow. In such settings, it is strongly preferred to use fixed point arithmetic instead, carefully scaling all real numbers in the problem to make full use of the fixed point binary representations used while never encountering overflow over the entire operational envelop of the electromechanical system under consideration (note that this usually takes considerable testing of the system to verify).

1.1.4 Parity checks, error detection, and error correction

When pushing certain subsystems (memory and communication in particular) to their physical limits (high speed, low power, small footprint, etc.), occasional bit errors may occur. There are a variety of simple and effective ways to identify such infrequent errors, and in certain cases even correct for them.

The simplest approach is to append a **single parity bit** to each set of k data bits that is stored in memory or sent over a communication channel; this parity bit is selected such that the sum (modulo 2) of all the data bits in the set, plus this parity bit, is 0 (if **even parity** is used) or 1 (if **odd parity** is used). When the entire set of $n = k + 1$ bits (data plus parity) is recalled from memory or received on the other end of the communication channel, this sum is again performed, and an error is flagged if it is of the wrong value. This approach is capable of **single error detection** (SED), with two or more errors in any set of n bits causing misinterpretation; note, however, that if single bit errors are random and infrequent, double bit errors will be extremely infrequent.

The idea of using parity bits to check for errors may be extended to facilitate stronger error detection, and even error correction. As shown in Figure 1.2, this is illustrated by the $[n, k]$ **linear binary codes** (LBCs) with:

- $r = 3$ parity bits $\{b_1, b_2, b_3\}$, $k = 4$ data bits $\{d_1, d_2, d_3, d_4\}$, and $n = r + k = 7$ total bits in a $[7, 4]$ LBC, or
- $r = 4$ parity bits $\{b_1, b_2, b_3, b_4\}$, $k = 11$ data bits $\{d_1, \ldots, d_{11}\}$, and $n = r + k = 15$ total bits in a $[15, 11]$ LBC.

In each of these example LBCs, an r set Venn diagram may be drawn with exactly one of the k data bits in each of the intersections. The r parity bits $\{b_1, \ldots, b_r\}$ are then selected such that parity (say, even parity) is achieved by summing the 2^{r-1} bits in each of the r sets in this Venn diagram. If a recalled/received set of n bits is assumed to be corrupted by at most one error, then during the subsequent parity checks of all r sets,

- if parity fails on just a single set, the corresponding parity bit b_i is itself identified as corrupted, whereas
- if parity fails on multiple sets, the data bit d_i corresponding uniquely to that set intersection is corrupted.

In either case, flipping the corrupted bit corrects the error, thus performing **single error correction** (SEC). This approach extends immediately to $[2^r - 1, 2^r - 1 - r]$ LBCs for $r \geq 2$, known as **binary Hamming codes**.

The general idea of storing or sending multiple redundant bits is extended significantly in Chapter 7, to develop LBCs (as well as linear codes over larger alphabets of symbols per digit) capable of multiple error detection and correction.

1.2 Central Processing Unit (CPU) cores

An essential defining characteristic of modern CPUs is the **word size**, which defines

(a) the number of bits in the **data bus** (the parallel wires carrying data within the CPU),
(b) the number of bits in the **memory addresses**, and
(c) the number of bits in the **instruction codes** enumerating the low-level executable commands in the CPU,

all of which are generally integer multiples of the word size, which on modern CPUs is 8, 16, 32, or 64 bits.

Doubling the width of the data bus doubles the amount of information that can be delivered from point A to point B within the CPU in a single clock cycle, but substantially increases the complexity of the circuitry; different tradeoffs are thus reached for the width of the data bus for different CPU designs.

Common memory configurations in modern MCUs include 16 address bits, facilitating the direct addressing of 64 KiB of memory, and 32 address bits, facilitating the direct addressing of 4 GiB of memory. Note that, in many CPUs, the number of physical address pins implemented can actually be less than or even (with a bit of additional logic) greater than the number of address bits. In particular, the 64 address bits of some modern 64-bit CPUs (that is, CPUs with a word size of 64 bits) facilitate the addressing of an absurdly large amount of memory (16 EiB); 64-bit CPUs thus generally implement only between 40 and 52 physical address pins, facilitating the direct addressing of 1 TiB to 4 PiB of memory (reminder: see §1.1.1 for definitions of binary powers).

There are two primary types of computer architectures (i.e., the set of rules that describe the organization of computer systems), the Harvard architecture, which strictly separates memory storage and signal busses for program instructions from those for data, and the von Neumann architecture, in which instructions and data share the same memory and busses. Modern implementations of the Harvard architecture usually relax the strict separation between instructions and data, allowing the instruction memory to actually be accessed as data, and are thus more accurately referred to as Modified Harvard architectures.

There are also two primary types of instruction set architectures (ISAs), RISC (**reduced instruction set computer**) and CISC (**complex instruction set computer**), in addition to a growing number of hybrid approaches that are increasingly blurring the lines between the two. The RISC ISA (pioneered by MIPS and perfected by ARM) has a small set of simplified (fixed-length) instructions operating on a large number of registers, and a streamlined instruction pipeline allowing a reduced number of clock cycles per instruction. In contrast, the CISC ISA (notably implemented and perpetuated by x86 CPUs) has a larger set of more complex (variable-length) instructions operating on a smaller number of registers, with each instruction executing a variable number of low-level operations (e.g., load something from memory, perform some arithmetic, store result back in memory). Note that the RISC ISA generally accesses memory through dedicated simple instructions, whereas the CISC ISA accesses memory as an integral part of its more complicated (multi-step) instructions.

Modern CPUs, and MCUs, appropriate for embedded applications include

- ARM Cortex A (32- and 64-bit), as implemented by Amlogic, Broadcomm, Rockchip, Samsung, TI Sitara, ...,
- ARM Cortex R (32- and 64-bit),
- ARM Cortex M (32-bit), as implemented by Cypress, Infineon, Microchip, Nuvoton, NXP LPC, STM32, ...,
- NVIDIA Carmel (64-bit),
- Qualcomm Kryo (64-bit),

- Intel 8051 (8-bit), as implemented by Cypress, Maxim, Silicon Labs, ...,
- Microchip AVR (including ATtiny and ATmega) and PIC (8-, 16-, and 32-bit),
- Tensilica Xtensa (64-bit),
- TI MSP430, MSP432, and C2000 (16- and 32-bit),

and many many others; most in this list (except the Intel 8051) are designed around RISC CPU cores.

1.3 Cache-based memory subsystems

The ALU and FPU of the CPU can approach their full speeds doing useful computations only if they can: (a) quickly access both the instructions to be performed next, as well as the data necessary to perform these instructions, and (b) quickly shift the results of these computations to somewhere secure for later use.

As a general rule, the smaller the data storage subsystem, the faster it can be made, but at a significant cost. Ordered from fastest/most expensive/largest footprint per byte on down, the primary storage technologies are:

- Static Random Access Memory (SRAM): 1-5 ns access time, volatile (data lost when powered down). Expensive!
- Dynamic Random Access Memory (DRAM): 5-25 ns access time, volatile, frequent refreshes (~ 1 Hz) required.
- Flash Memory / SD Cards / EEPROM[3]: 50-500 ns access time, non-volatile, limited write endurance.
- Solid State Drives (SSD[4]): 10-100 μs access time, non-volatile, hot swappable, limited write endurance.
- Hard Disk Drives (HDD): 5-20 ms access time, non-volatile, hot swappable, excellent write endurance. Cheap!

Significantly, as the size of a data storage subsystem grows, it generally becomes easier to download/upload increasingly large *blocks* of data, all at essentially the same time, at relatively little added cost (time and energy).

To reduce the mean access time & energy, and overall expense & physical size, required of the memory subsystem (all of which are important in embedded applications), the communication between the CPU and the main memory (DRAM or Flash) [and, to even slower "disk" storage[5]] is often assisted by a hierarchy of smaller/faster cache memory (SRAM & DRAM), together with a **memory management unit** (MMU) or **memory protection unit** (MPU) coordinating its use. Cache memory is often divided into multiple levels, including:

- **L1i**, for queueing up the instructions to be performed next, and
- **L1d**, **L2**, **L3**, and **L4** (or a subset thereof[6], with the smaller numbers enumerating the faster/smaller "lower" levels of the cache hierarchy), both for bringing data to the handful of **registers** holding the data actually used by the ALU and FPU, and for storing the results of the computations performed back in the main memory.

When using a cache-based memory system, small fixed-size **cache blocks** (aka **cache lines**) of contiguous memory are downloaded/uploaded whenever updating the lower levels of the cache hierarchy[7], and larger cache blocks are downloaded/uploaded whenever updating the higher levels of the cache hierarchy, or communicating between the highest level of cache (aka the **last level cache**) and the main memory itself.

The CPU also usually includes a translation lookaside buffer (**TLB**), which translates the virtual addresses used by each program to their corresponding physical addresses in the main memory, for both the instructions to be executed as well as the corresponding data storage[8].

The majority of the silicon area on most modern CPUs is in fact taken up by the MMU, the TLB, and the L1i, L1d, and (sometimes) L2 and L3 memory caches, thus indicating the critical importance of these subsystems to the overall CPU performance (higher levels of cache, if used, are often incorporated on separate ICs). The several components of a modern cache-based memory system usually interact quite efficiently with little if any intervention by you, the embedded programmer. However, a high-level understanding of how such systems behave can assist you in implementing certain programming directives that can make such systems run even better, and to streamline the data flow when the CPU stalls due to cache conflicts.

[3] Flash (see §11.3.2) comes in two types, NAND and NOR. Flash is a type of EEPROM designed for high speed and density, with large erase blocks ($\gtrsim 512$ bytes) and limited write endurance ($\sim 10^4$ write cycles). The term "EEPROM" is saved for non-volatile memory built with the same technology, but with small erase blocks (1 to 8 bytes) and better write endurance ($\sim 10^6$ write cycles).

[4] SSDs are self-contained subsystems using flash memory together with their own cache memory to both increase effective speed and improve endurance. Many of the concepts discussed in this section extend directly to the control of cache-based SSD systerms.

[5] "Disk" storage may refer to both filesystems and virtual memory on both SSDs and HDDs.

[6] How many levels of cache should be implemented for the best overall system performance generally depends on the total amount of main memory accessible by the system, and the ratio of the CPU speed to the main memory speed, which is often much slower.

[7] At any given level, a **cache entry** generally includes both the copied data as well as a **tag** indicating the corresponding range of addresses in the main memory.

[8] The TLB is often split into an Instruction TLB and Data TLB, and may be split into levels (e.g., L1 ITLB/DTLB, L2 ITLB/DTLB, ...).

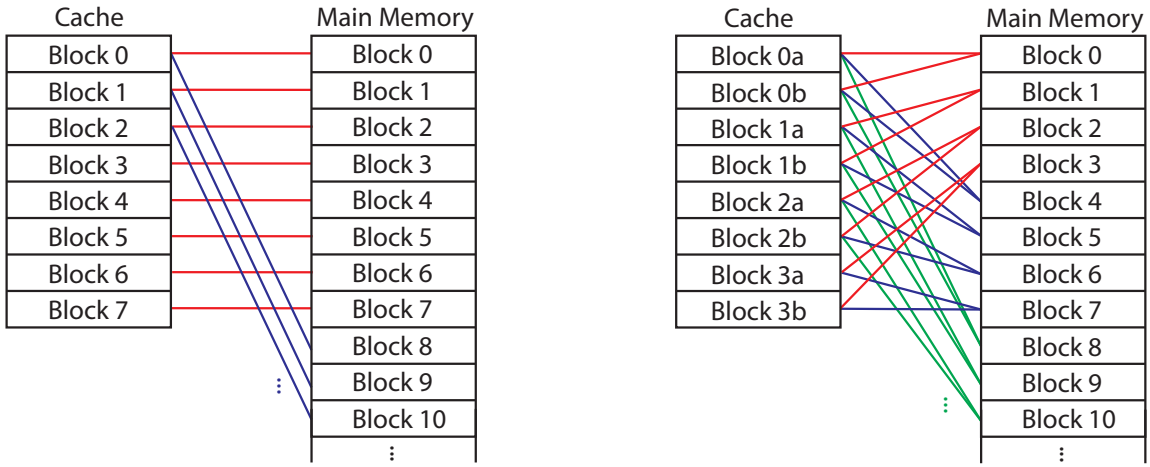

Figure 1.3: Illustrations of (left) a direct mapped cache, and (right) a two-way set associative cache.

When initiating a read or write to/from a particular memory location, the CPU first checks to see if a copy of that memory location is already represented in its L1 cache. If it is (constituting a **cache hit**), the CPU interfaces directly, and quite quickly, with this highest-speed cache. If it is not (a **cache miss**), the MMU must look in successively higher levels of (slower-speed) cache, all the way out to the main memory if necessary, to reach the relevant memory location. The MMU may also create a new cache block, at one or more levels of the cache, containing this memory location together with nearby entries of the main memory; to do this, it must generally **evict** one of the existing cache blocks at each affected level.

Where, exactly, such a new cache block may be placed within a cache is governed by the placement policy associated with that cache level, which may allow the new cache block to placed:

(a) at just a single location, based on the least significant bits of the corresponding memory address block, called a **direct mapped** cache (see Figure 1.3a);
(b) at any of N locations (typically, $N = 2$, 4, or 8), based on the least significant bits of the memory address and the replacement policy used (discussed below), called an N**-way set associative** cache (see Figure 1.3b);
(c) at either of 2 locations, following either the direct-mapped policy mentioned above or a hash function pointing somewhere else, called an **two-way skew associative** cache; or
(d) anywhere it wants, called a **fully associative** cache.

If the placement policy allows a choice to be made in the placement of the new cache block [see (b), (c), and (d) above], this decision is made by the replacement policy of the MMU. Amongst many possible such policies, one common choice is to evict the **least-recently used** cache block. The larger the number of choices in the placement policy, the more places that need to be searched in cache for the requested memory location, but the less likely a very recently loaded cache block (possibly containing useful information for impending calculations) will need to be evicted to make room for a new cache block.

When compiling code for cache-based memory systems, the general goal is to maximize the percentage of cache hits (aka the **hit rate**) in the lowest levels of cache. This goal is achieved with algorithms that are compiled with high degrees of locality of reference, including both **temporal locality**, in which certain variables are reused repeatedly, and **spatial locality**, in which the data needed for subsequent computations is generally stored physically close to each other in the main memory (and is thus likely already present in existing cache blocks, which are loaded when preparing for the preceding computations).

The MMU must implement a rather involved set of rules in order to achieve cache coherence; that is, to make the entire multi-level cache-based memory system appear, for the purpose of programming simplicity, as a single, unified, very fast memory system. The MMU achieves this by carefully coordinating both the *reading* of the main memory and the higher levels of cache by the lower levels of cache, as well as the *writing* of the

data generated by the CPU back to the various levels of cache and, ultimately, back to the main memory.

When reading saved data from the main memory into the various levels of cache, there are two types of approaches that the MMU may implement. With **inclusive** cache designs, which are the most common, smaller and smaller sub-blocks of the data stored in the higher levels of cache and the main memory are duplicated into each successively lower level of cache. This approach simplifies the connection between the various levels of cache (keeping the ankle bone connected to the leg bone, etc), thereby simplifying the problem of maintaining cache coherence, but increases the communication between the various cache levels. With **exclusive** cache designs, on the other hand, two caches never share the same data. This approach avoids repetition, shuns redundancy, eshews reiteration, and resists recapitulation, but leaves the placement policy of the MMU (and/or the embedded programmer, via compiler directives) with the question which data to put into which levels of cache.

When writing the new data generated by the CPU back out to the various levels of cache and, ultimately, all the way to the main memory, there are two types of **write policies** that may be implemented. When using a **write through** policy at a particular cache level, newly updated data at that cache level is copied back immediately to the corresponding section of the next higher level of cache or the main memory. This approach allows the cache block to be overwritten immediately with a different section of memory when necessary, but increases the amount of communication between cache levels. When using a **write back** policy at a particular cache level, on the other hand, the updating of the next higher level of cache or the main memory with the updated data at that cache level is deferred until the corresponding cache block soon needs to be evicted to make room for the caching of a different section of memory. This approach reduces the communication between the different cache levels as well as the number of data writes, which is more efficient, but introduces a possible delay between when the "eviction notice" is received by a particular cache block, and when that block is actually ready to cache a different section of memory. Note that it is particularly important to use a write back policy to the main memory and to SSD when either is implemented on flash, which has limited write endurance.

Whenever a cache contains updated data that has not yet been copied up to the next higher level of cache and the main memory, that section of cache is said to be **dirty**. Note also that, in multicore and multi-CPU systems, a typical cache implementation might be configured as follows:

- each core has a dedicated L1 cache,
- each CPU has a dedicated L2 cache, shared amongst its multiple cores, and
- the entire system has a single L3 cache, shared amongst its multiple CPUs.

Higher levels of cache and the main memory may thus be updated by other CPU cores, as well as by certain peripherals with **direct memory access** (DMA). Whenever a cache contains old data that has not yet been copied down from the next higher level of cache and the main memory, that section of cache is said to be **stale**. Substantial care must be taken by the MMU to keep track of both the dirty and the stale sections of cache at all levels, and to update them when appropriate, in order to keep the cache coherent.

Steps an embedded programmer can take to use cache-based memory systems more efficiency include:

1) structuring and ordering computations in the compiled code to maximize both temporal and spatial locality,
2) keeping certain memory locations, for variables that are reused repeatedly [e.g., indices $\{i, j, k, \ldots\}$, constants c_i, and temporary variables t_i], locked in cache,
3) implementing write through policies for the lower-level cache blocks used primarily for data input to the CPU, which need to quickly replaceable,
4) implementing write back policies for cache blocks used primarily for data storage to the main memory, to minimize unnecessary communication between cache levels,
5) bypassing the use of cache altogether for certain data that is only accessed occasionally, and
6) manually **flushing** (copying back to higher levels) cache blocks that will not be needed again soon.

1.4 Hardware for exploiting parallelism

Most numerical algorithms can be arranged such that the majority of the calculations performed do not actually depend upon the results of the immediately preceding calculations. Such situations allow for **parallel computing**, in which some calculations may be done simultaneously (or, nearly so) with others, allowing the entire algorithm to complete much more quickly. Parallelism within numerical algorithms is quantified by its **granularity**: problems with **fine-grained parallelism** have a relatively small number of calculations that may be performed independently before their results must be shared in order to continue, whereas problems with **coarse-grained parallelism** have a relatively large number of computations that may be performed independently before their results must be shared in order to continue. Problems with coarse-grained parallelism naturally evident at the outset of the problem formulation are sometimes said to be **embarrassingly parallel**.

The identification of techniques to expose and exploit parallelism is essential for two key reasons. First, of course, identifying parallelism allows the computer's operating system (see §2) to assign multiple compute resources to the problem at hand simultaneously [i.e., the various ALUs and FPUs within the different CPU cores in the system (see §1.4.3), together with certain other compute resources that may also be available, as surveyed in §1.5.3-1.5.4]. This enables significantly more computational work to be completed per clock cycle.

Equally important, at a lower level, identifying parallelism allows a **self-optimizing compiler** to make much more effective use of all available levels of high-speed cache memory (see §1.3) for each individual CPU core being used, by performing a delicate *regrouping* and *reordering* of the various computations to be performed, thus maximizing both the temporal and spatial locality of the data needed for each and every calculation to be performed along the way. This is best achieved by adhering to the following high-level guidelines:

(a) Write clean codes that clearly/simply reveal the problem structure at hand (e.g., if your computer language allows it, somehow writing A*B for matrix/matrix multiplication, or A\b for Gaussian elimination, instead of looping over all of the individual indices involved in such basic but time-consuming computations yourself).

(b) Use a modern self-optimizing compiler that calls the BLAS (basic linear algebra subprograms) and LAPACK (linear algebra package) software libraries extensively (or, if the programming language or compiler you are using doesn't do this for you, call these routines yourself from within your code, and consider changing to a different programming language or compiler!). These libraries are meticulously hand tuned by each CPU vendor to maximize hit rates in each level of cache for the fundamental linear algebra problems that your code will spend the bulk of its time solving at any given problem size. You are unlikely to do better on your own.

(c) If at all possible, define the problem size at *compile time*, via constants defined in the code header, rather than at run time, via data files that are read in (post compilation). This important (but, often-overlooked) third guideline helps the compiler to decide, at compile time, specifically how to reorder the various loops involved in order to achieve maximum performance from the cache. Indeed, for many (large) problems, the advantage here is so significant that recompiling the code in question immediately before any large run, once the size of the problems to be solved are identified and defined in the code header, can be quite beneficial.

Most numerical algorithms can actually be arranged (or, rearranged) to reveal a *hierarchy* of parallelism within, with some fine-grained parallelism embedded within its innermost loops, and successively coarser-grained parallelism evident in the loops that surround them. Modern CPUs and compilers can effectively exploit many of these different levels of parallelism simultaneously, in order to achieve remarkable degrees of computational efficiency with relatively little specific intervention by the embedded programmer.

It is important to understand the several ways that modern computers exploit parallelism to see other specific things the embedded programmer can do [besides points (a) through (c) above] to help facilitate the parallelization process. Note that the subsections that follow are ordered from techniques best suited to exploit the finest-grained parallelism available (in the innermost loops), to those that are better suited for exploiting successively coarser and coarser grained parallelism.

1.4.1 Instruction pipelining and branch prediction

Even relatively simple (i.e., RISC) instructions may themselves generally be divided up into a number of smaller steps; for example, (a) fetch the instruction, (b) fetch the operands, (c) do the instruction, (d) write the results. Instruction pipelining is a technique for implementing parallelism on a CPU over each of these smaller steps, thus effectively keeping each corresponding part of the ALU or FPU busy doing useful work at each timestep. For example, at a clock cycle when instruction k is just starting with step (a) above, instruction $k-1$ can (simultaneously) be executing step (b), instruction $k-2$ can be executing step (c), and instruction $k-3$ can be finishing up with step (d). For this to work correctly, the calculations must be ordered in such a manner that a fine degree of parallelism is available, such that later commands don't try to fetch the results of earlier commands until they are actually available, which can take a few timesteps.

Branch prediction is a technique used to keep such instruction pipelines full even during the execution of conditional (if-then-else) statements. This is achieved by guessing (based on previous executions of each conditional) which branch the code is most likely to take, and proceeding assuming that the conditional will actually take that direction this time. If it does, the instruction pipeline remains full right through the conditional statement. If it does not, however, the tentative results of each calculation after the conditional must be discarded, before they are written back to memory, and the pipeline re-initialized with the instructions on the other branch of the conditional. Branch prediction is especially valuable in CISC systems, with complex instructions and thus relatively long pipelines, and on codes that frequently encounter conditionals. [Note that the code for handling branch predictions is generally inserted by the compiler, if the appropriate flags are set, and thus need not be written by the embedded programmer.] The overall time penalties associated with incorrect branch predictions may be kept small by (a) minimizing the number of conditional statements that are encountered by the numerical algorithm (eliminating such conditionals altogether from all but the outermost loops of the numerical algorithms used), and (b) using RISC processors, which have relatively short instruction pipelines.

1.4.2 Vectorization (SIMD)

As discussed in §1.1.2 and 1.1.3, the fixed-point and floating-point representations of real numbers that are useful in embedded applications are typically only 16 or 32 bits long, whereas the word length of high-performance CPU cores is 32 or 64 bits, and data bus and register sizes of modern CPUs and DSPs (see §??) can be even larger (e.g., 128 bits or more). Such an organization facilitates, where useful, the grouping of real numbers together as a **vector**, and performing quickly the same arithmetic operations on all elements of the vector simultaneously (or, nearly simultaneously), leveraging the extensive fine-grained parallelism often present in the innermost loops of substantial numerical algorithms. This basic idea goes by several names; in the early days of computing on Cray supercomputers (including the Cray-1, Cray X-MP, Cray-2, & Cray Y-MP), this process was called **vectorization**, and operated on very large vectors (with, e.g., 64 double-precision floats). The idea of vectorization went dormant in the mid 90's, but was revived for desktop and embedded processors, using much shorter vectors, under the general name of **SIMD** (single-instruction, multiple data), with different implementations appearing under various trademark names including MMX/SSE (Intel), 3DNow! (AMD), Altivec (Freescale), VMX (IBM), Velocity Engine (Apple), and, more recently, Neon and Helium (ARM).

1.4.3 Shared-memory multiprocessing

At the next coarser level of granularity of parallelism in numerical algorithms, multiple substantial tasks can often be identified that can be run completely independently from each other for a while [say, computing $O(10^3)$ or more floating-point operations (FLOPS) before having to share results with those of other tasks in order to

continue]. Such independent tasks are often found in the outermost loops of a code, and do not actually need to contain the same set of commands in order for the compiler to be able to parse them out and organize how to compute them in parallel; this setting is thus occasionally referred to as MIMD, to distinguish it from the SIMD setting required to parallelize innermost loops via vectorization, as discussed in §1.4.2.

The most straightforward way to leverage such coarse-grained parallelism is **multithreading**; that is, the spawning and running of multiple independent "threads" by a single numerical algorithm, each of which may run for a while on a different CPU core (as coordinated by the scheduler, as discussed further in §2.1) before pausing from time to time to synchronize its results with the other threads, but all of which ultimately access the same main memory. This setting is referred to as **shared-memory multiprocessing**, and may be coordinated directly by an embedded programmer from within a numerical code using **OpenMP** compiler directives, or in many cases can be efficiently coordinated by a good self-optimizing compiler.

As discussed in detail in §1.3, the use of high-speed cache memory (often, at multiple levels) has become essential for modern CPUs to reach their full potential, as CPUs are now typically much faster than the main memory that they access, but wide data paths allow large blocks of data to be retrieved from main memory in relatively little additional time (as compared with the time required to retrieve a single byte). In multi-core systems, L1 cache is typically dedicated to each core, L2 cache is dedicated to each CPU (shared amongst all cores on that CPU), and (often) L3 cache is shared amongst all CPUs, providing the gateway to the main memory. The challenge of maintaining cache coherence in multicore settings complicates the execution of complex numerical algorithms using shared-memory multiprocessing, in which data must be shared frequently between the different running threads, though in most applications the problem of maintaining cache coherence is taken care of by the MMU, with relatively little intervention required by the embedded programmer.

Most modern computers with a handful of CPU cores for shared-memory multiprocessing implement some sort of **symmetric multiprocessing** (SMP[9]), in which all compute cores have equal access to all memory and peripherals (usually via some arrangement of a data bus, address bus, and control bus), and may thus be treated essentially equally by the scheduler (see §2.1) for all tasks (i.e., no specific tasks are restricted to certain processors). Following this approach, two specific design paradigms simplify the organization:

(a) **homogeneous computing**, in which only one kind of CPU core is used, and
(b) **uniform memory access** (UMA), in which all cores have equal access to all sections of main memory.

Demands on peak computational performance in embedded systems continue to increase steadily, following the celebrated "Moore's Law" (that is, the observed doubling of the IC density in leading CPUs, and thus their performance, about once every 2 years). At the same time, the maximum clock speeds that CPUs can support is only increasing gradually in recent years, with higher clock speeds requiring higher voltages as well as increased power consumption to operate the CPU. Thus, embedded computers are now tending to include more and more CPU cores. Further, demands on computational performance in most applications are found to vary substantially over time, and power efficiency during the quiescent times is often just as important as peak computational performance during the active times. One approach to achieving an improved balance between *maximizing peak computational performance* and *minimizing time-averaged power consumption* is thus to implement **dynamic voltage and frequency scaling**, automatically reducing both the effective CPU clock speed as well as the voltage driving the CPU, in real time, when the recent average computational load is found to be relatively light[10].

When designing computers to meet even stricter requirements, however, both of the simplifying paradigms (a) and (b) above eventually become limiting factors, and must be relaxed in order to build systems with even greater peak computational performance, and with even lower average power consumption. Thus:

[9] The abbreviation SMP usually denotes symmetric multiprocessing, but is occasionally used more generally for shared-memory multiprocessing, which may or may not be symmetric. We recommend the former, more restrictive use, which is more common.

[10] In this setting, a relevant performance metric is FLOPS per MHz, in addition to peak FLOPS.

• The **heterogeneous computing** paradigm is now quite common, in which the embedded computer includes more than one type of CPU core (one with higher peak performance, and one with lower average power consumption), which may be selectively turned on and off. There are many different ways in which this general idea may be implemented; examples include ARM's big.LITTLE and DynamIQ technologies.
• An emerging design paradigm for embedded computers is **nonuniform memory access** (NUMA), in which each CPU (or, CPU cluster) is closely associated with only a subset of the main memory, and it takes substantially more time to read from or write to memory that is more closely associated with the other CPUs in the system [though all of the main memory shares a single large address space]. This approach was perfected in the field of HPC by Silicon Graphics (SGI) under the brand name NUMAlink, and (as of 2021) is only beginning to emerge in computers designed for embedded applications.

Note finally that, akin to branch prediction (see §1.4.1), **speculative execution** of independent threads of a multithreaded code following a conditional statement, or for which there is potentially stale data input, may be performed in the setting of shared-memory multiprocessing if sufficient computational resources are available, with speculative locks used to delay the write-back (or, the deletion) of the results of the speculative section of code until the conditional itself is evaluated, or the potentially stale data input has been verified as correct.

1.4.4 Distributed-memory multiprocessing

To solve even bigger problems, leveraging the coarsest-grained parallelism that can be identified in a numerical algorithm, many independent computers, each with their own dedicated memory, may work together over a fast network operating as a computer cluster. When large centralized computer clusters, and the codes running on them, are particularly well tuned for the coordinated distributed computation of very large individual jobs[11], this setting is often referred to as high performance computing (HPC).

Cluster-based "cloud" computing in the HPC setting is a very natural complement to "edge" computing for many large-scale real-time problems addressed by embedded sensors. Examples of interest include:

• the forecasting of the evolution of the track and intensity of hurricanes or forest fires and, simultaneously, the uncertainty quantification (UQ) related to such forecasts,
• the development of a single detailed map of a region, based on the information gathered from several independent mobile robots, each moving through and exploring different overlapping subregions, and each independently executing their own simultaneous localization and mapping (SLAM) algorithms, etc.

In such problems, a large computation needs to be performed on the cluster, fusing the Big Data being gathered, in real time, from numerous (often, heterogenous) sources (e.g., mobile robots), often using complex physics-based models. At the same time, based on the UQ performed on the cluster, the mobile robots often need to be redispatched intelligently to different subregions, a setting referred to as adaptive observation.

In the HPC setting, distributed computing leverages a fast and reliable communication network (see §3.1), such as[12] Ethernet or Infiniband, between the independent computers making up the cluster. As opposed to shared-memory multiprocessing (§1.4.3), in which the MMU and a good self-optimizing compiler can often handle most if not all of the low-level details related to cache coherence and the coordination of distinct threads related to a certain job, in distributed-memory multiprocessing the necessary passing of data (aka messages) between the independent computers in the cluster must often be coordinated manually by the programmer from

[11] As opposed, for example, to the maintenance of transactional databases used for ticket sales, large-scale search, social media, etc., with the cluster interacting simultaneously, and essentially independently, with a very large number of users.

[12] HPC is a very small market indeed, as compared to consumer electronics (largely supporting web surfing, video games, office productivity applications, etc). HPC today advances mostly by repurposing cutting-edge commercial off-the-shelf (COTS) electronics technologies developed for consumer electronics. In this setting, the possible deployment of Thunderbolt as a potential new technology for networking in HPC clusters is quite interesting.

within the numerical code, which is a rather tedious process. Some variant of the **Message Passing Interface** (MPI)[13] is generally used for this process in the distributed-memory setting, effectively solving (by hand) similar problems as those solved by the MMU (automatically) for maintaining cache coherence in the shared-memory setting, passing messages and blocking new computations only when necessary. Primitive operations used to coordinate message passing and computations in MPI-1 include

- **point-to-point** message passing (from one specific node to another),
- **one-to-all** message passing (aka **broadcast**),
- **all-to-one** message passing, together with an operation like summing (aka **reduce**),
- **all-to-all** message passing, for rearranging the data on the cluster, etc.

Such commands can be either blocking (halting a thread's execution until the command is completed) or non-blocking, or follow a ready-send protocol in which a send request can only be made after the corresponding receive request has been delivered. MPI-2 introduces certain additional operations, including

- **one-sided** put (write to remote memory), get (read from remote memory), and accululate (reduce) operations,
- the ability of an existing MPI process to create a new MPI process,
- the ability of one MPI process to communicate with an MPI process created by a different MIP process, etc.

Note that FT-MPI is a remarkable extension (plug-in) that adds significant fault tolerance capabilities to MPI; Open MPI also includes significant fault tolerance capabilities.

In the field of robotics, the problem of distributed computation is referred to as **distributed control**. Distributed control systems generally implement several nested control loops on the individual mobile robots or machines (e.g., on an assembly line) involved. **Decentralized control systems** denote controllers that are primarily distributed on each robot or machine, with no central supervisory authority. **Centralized control systems**, in contrast, denote controllers that primarily operate on a central supervisory computer. Most practical control systems for multi-robot teams or multi-machine assembly line operations are some "hierarchical" hybrid between the two, with decentralized low-level/high-speed control feedback on the inner loops (e.g., coordinating the motion of an individual robot arm), coupled with centralized high-level coordination and fault management on the outer loops (adjusting the speed of the assembly line, etc). Mobile robots add the significant complication of very unreliable communication links, a challenge that requires significant care to address.

1.4.5 Summary: enabling the efficient parallel execution of codes

The reordering of the individual calculations within a numerical code, maximizing the temporal and spatial locality of the data needed for each calculation to be performed, and thus maximizing the effectiveness of all available levels of cache memory, is best achieved by using a modern self-optimizing compiler, with a high level of optimization selected, together with steps (a), (b), and (c) described in the introduction of §1.4.

Pipelining (with or without branch prediction) and SIMD vectorization, as discussed in §1.4.1 – 1.4.2, are both facilitated by the remarkable hardware of the modern CPU itself, together with the low-level opcodes used by good self-optimizing compilers to leverage this hardware. The use of both techniques can be activated by you, the embedded programmer, rather easily, simply by compiling your code with the appropriate compiler flags set to enable these features. With today's CPUs and compilers, it is generally not necessary for you to write code in assembler and deal with such low-level opcodes yourself, thus leaving you to attend to higher-level, more consequential issues. The efficient use of shared-memory multiprocessing (§1.4.3) sometimes takes a bit more work, leveraging OpenMP compiler directives to tune the default behavior generated by the compiler when necessary. The use of distributed-memory multiprocessing (§1.4.4) is, as of 2021, much harder, and must usually be coordinated manually by the user (often leveraging MPI), as introduced briefly above.

[13] Some HPC languages, like Coarray Fortran (which is implemented by G95), are beginning to implement coding constructs that that make higher-level parallel programming in the distributed memory setting significantly easier.

1.5 Microcontrollers (MCUs) and associated coprocessors

We now shift topics from the foundational ideas in modern computing, to the technologies that implement these ideas in embedded applications. At the heart of such an implementation is a **microcontroller** (MCU[14,15]), which is an integrated circuit (IC) that fuses one or more CPU(s) together with an interconnecting bus fabric, a (SRAM-, DRAM-, and/or Flash-based) memory subsystem, and a number of useful coprocessors, such as:

- arithmetic logic units (ALUs) for fast integer and fixed-point computations [§1.1.2],
- (on some MCUs) floating-point units (FPUs) for fast floating-point computations at specific precisions [§1.1.3],
- programmable interrupt controllers (PICs) to handle signals that trigger specific new actions [§1.5.2],
- general purpose timer/counter units [§1.5.5], and
- communication interfaces [§3] for connecting the MCU to a range of input/output (i/o) devices [§6],

as well as other application-specific integrated circuits (ASICs) useful for commonly-needed functions, such as:

a. dedicated hardware for transcendental function approximation [§1.5.3.1],
b. ring buffers for computing finite impulse response & infinite impulse response (FIR/IIR) filters [§1.5.3.2],
c. cyclic redundancy check (CRC) units, for performing fast detection of bit errors [§1.5.3.3],
d. random-number generators (RNGs) [§1.5.3.4], etc.

Some leading MCU families, and the CPUs that they embed, were surveyed briefly in §1.2. As indicated there, popular MCUs range from simple 8-bit devices, with just a few simple coprocessors, to remarkably efficient integrations of high-performance, low-power 32-bit or 64-bit CPUs with high-performance coprocessors (DSPs, GPUs, NPUs, FPGAs, CPLDs, PRUs, etc), together dozens of timers and other independent hardware communication subsystems (each functioning independently, in real time, *without* loading the main CPU core of the MCU, and often operating with direct memory access), including specifically:

e. quadrature encoder counters, for quantifying the (clockwise or counterclockwise) rotations of shafts,
f. pulse-width modulation (PWM) generators, for driving servos and ESCs,
g. UART, SPI, and I2C channels, for hooking up other ICs and (nearby) off-board peripherals,
h. CAN controllers, for longer-distance communication over differential pairs of wires,
i. USB controllers, for communicating with desktop/laptop/tablet computers and associated peripherals,
j. digital-to-analog and analog-to-digital converters (DACs and ADCs), for interfacing with analog devices,
k. inter-IC sound (I2S) channels and/or serial audio interfaces (SAIs), for audio channels,
l. on-board or off-board oscillators, coin cell power backup, and real-time clocks (RTCs), for scheduled wakeup,
m. integrated op amps, for building analog filter circuits (low-pass, band-pass, notch, PID, lead/lag, …)
n. memory controllers (e.g., FSMC and quad SPI channels), for hooking up additional memory, etc.

Loading the CPU, other serial comm protocols can be bit-banged using reconfigurable general-purpose input/outputs (GPIOs). An example modern MCU is the STM32G474, a block diagram of which is given in Figure 1.4. This MCU, built around an ARM Cortex M4 (a RISC CPU with 3-stage instruction pipeline, a modified Harvard architecture, and an FPU for single-precision floats), is implemented in the Beret family of boards introduced in §5, and integrates several coprocessors (indeed, in *all* 14 categories, a through n, mentioned above).

[14] In contrast (but, similar in many respects), a **microprocessor** (MPU) is an IC designed to form the heart of a desktop, laptop, or high-performance tablet computer, with hardware subsystems focused more on computational performance, graphics, and efficiently accessing a much larger memory and data storage footprint than a typical MCU.

[15] A third relevant category today is what is often called a **mobile processor** (MP), which is an IC that implements many of the same components as an MCU or MPU, but is tuned specifically for low-power operation, standby, and sleep. Modern MPs, which achieve remarkable flops/MHz, flops/mW, and (due to very large scale manufacturing, for use in smartphones) peak flops/$ ratios on real-world problems, are particularly well positioned for advanced high-level applications in embedded systems (performing vision-based feature recognition and SLAM, machine learning, etc.), as a complement to MCUs for handling the real-time low-level motor control functions. Note that the dividing lines between MPs, MPUs, and MCUs continues to blur, and emphasizing the distinction between them is not necessarily productive moving forward.

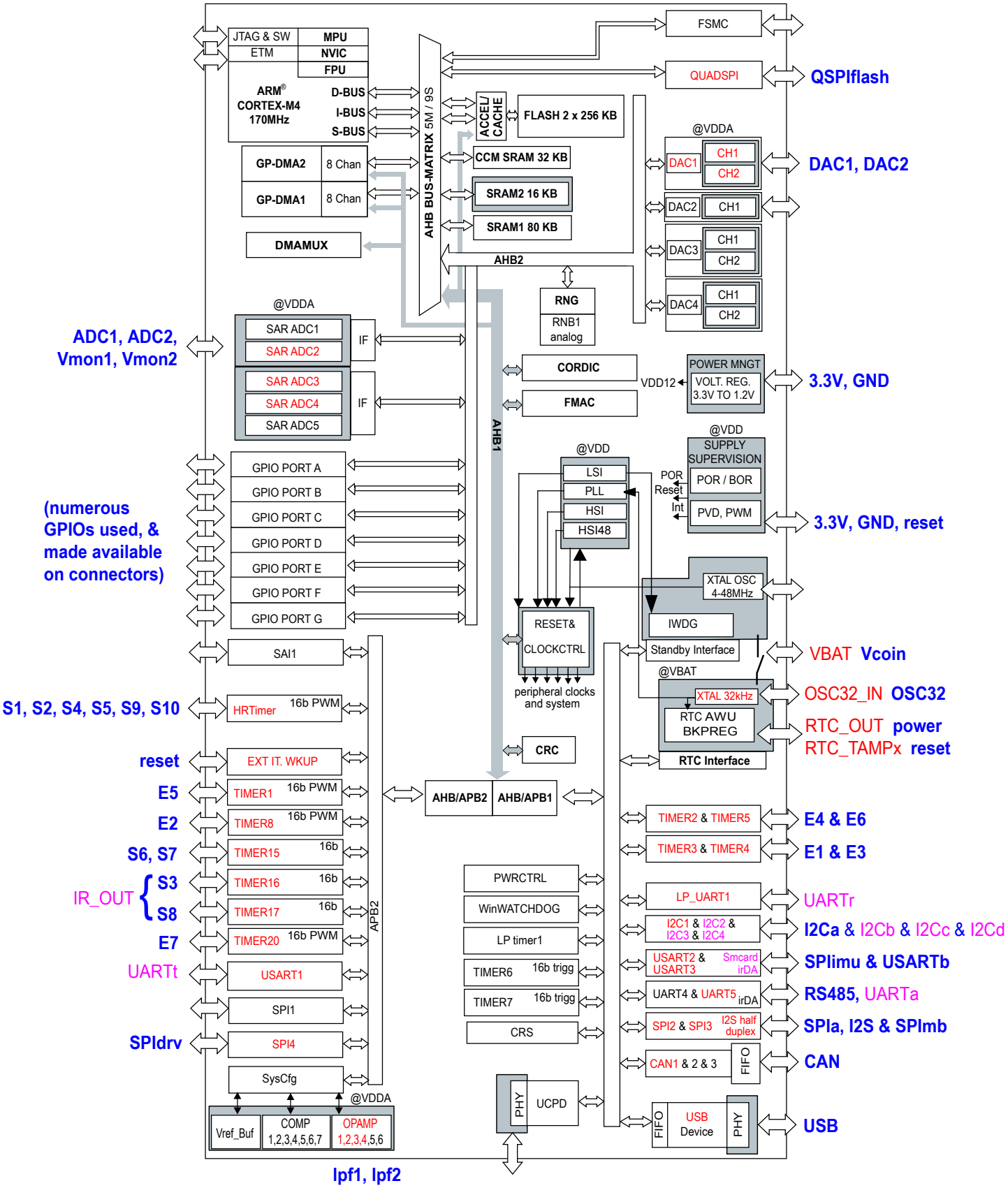

Figure 1.4: Block diagram of the STM32G474 MCU, with the hardware leveraged by the Berets (see §5) highlighted in color (see text). Image adapted from the STM32G474 datasheet, courtesy of STMicroelectronics.

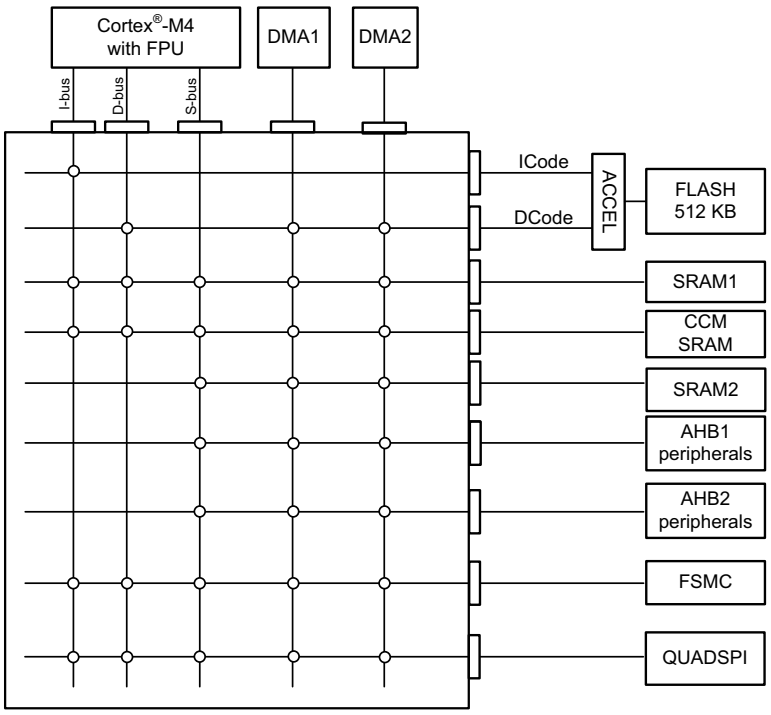

Figure 1.5: Bus matrix connections to/from the ARM Cortex M4 in the STM32G474; circles at intersections in the grid indicate an allowed connection from the corresponding master (at top) to the slave (at right). Image adapted from the STM32G474 datasheet, courtesy of STMicroelectronics; see also ST Application Note AN4031.

1.5.1 Busses, memory management, and direct memory access (DMA)

At the heart of a modern MCU is one or more CPU core(s). The complex fabric interconnecting these CPU core(s) within the MCU to auxiliary processing units, to the various memory subsystems, and to the connected peripherals, is organized into a number of distinct busses, each with specific privileges, as illustrated for the STM32G474[16] in Figure 1.5. Most modern processors follow ARM's open standard Advanced Microcontroller Bus Architecture (AMBA) protocol, which includes the Advanced High-performance Bus (AHB), which is responsible for both the sending of an address to memory as well as the subsequent writing or reading of data or instructions to/from that memory address (via busses ranging from 64 bits to 1024 bits in width), and the lower-complexity Advanced Peripheral Bus (APB), which coordinates lower-bandwidth register and memory access by system peripherals (via a 32-bit bus).

Another essential aspect of modern CPUs is direct memory access (DMA), a feature that allows coprocessors and peripherals to read or update memory locations directly, without tying up the CPU as a choke point. In some implementations, DMA can also be used, without bogging down the CPU, to copy or move data from multiple memory locations into a single communication data stream, or to take data from a single data stream and distribute it to the appropriate memory locations, common processes referred to as scatter/gather I/O.

[16] In the ARM Cortex M4 CPU implemented in the STM32G474 MCU, there are three main busses, the ICode (instruction) interface, the DCode (data) interface, and the System interface, denoted **I-BUS**, **D-BUS**, and **S-BUS** in Figures 1.4 and 1.5.

1.5.2 Programmable interrupt controllers (PICs)

The CPU of an MCU often needs to wait for a trigger (for example, a clock pulse, or a signal from an external peripheral) before beginning a specific new action or computation. The CPU also needs to be able to handle various **exceptions** that occur when something unexpected happens (divide by zero, etc.). Such an event is generically known as an interrupt request (IRQ). There are many possible sources of IRQs, and at times they can arrive at the MCU in rapid succession, and thus need to be carefully prioritized and dealt with by the CPU accordingly. IRQs are handled by a dedicated unit on the CPU called[17] a programmable interrupt controller (PIC). The PIC assigns a priority and a block of code, called an interrupt service routine (ISR), for the CPU to deal with any given IRQ, if/when one is detected.

IRQs are denoted as maskable or non-maskable, which essentially distinguishes whether or not they may be ignored (at least, for the time being) by the ISR that is associated with that IRQ. Interrupts that deal with non-recoverable hardware errors, system reset/shutdown, etc., are often flagged as non-maskable interrupts (NMIs). Common interrupts generated and handled by user code, however, should generally NOT be flagged as NMIs, since NMIs hinder other normal operations (stack management, debugging, ...). Common interrupts that are time critical should instead be flagged as high priority maskable interrupts, and if such IRQs are missed by the system during testing, the behavior of the scheduler (see §2.1) should be adjusted to make certain that such high priority maskable interrupts are set up to be dealt with in a timely fashion.

1.5.3 Application specific integrated circuits (ASICs)

Application specific integrated circuits (ASICs) are dedicated coprocessors that are hard-wired for narrowly-defined purposes. As introduced previously, representative examples include transcendental function generators, ring buffers, cyclic redundancy check calculation units, random-number generators, etc. To illustrate, this section discusses various characteristics of these four common ASICs. Note that the hardware implementing the timer / counter units discussed in §1.5.5, and the communication subsystems discussed in §1.5.6, may also be considered as ASICs.

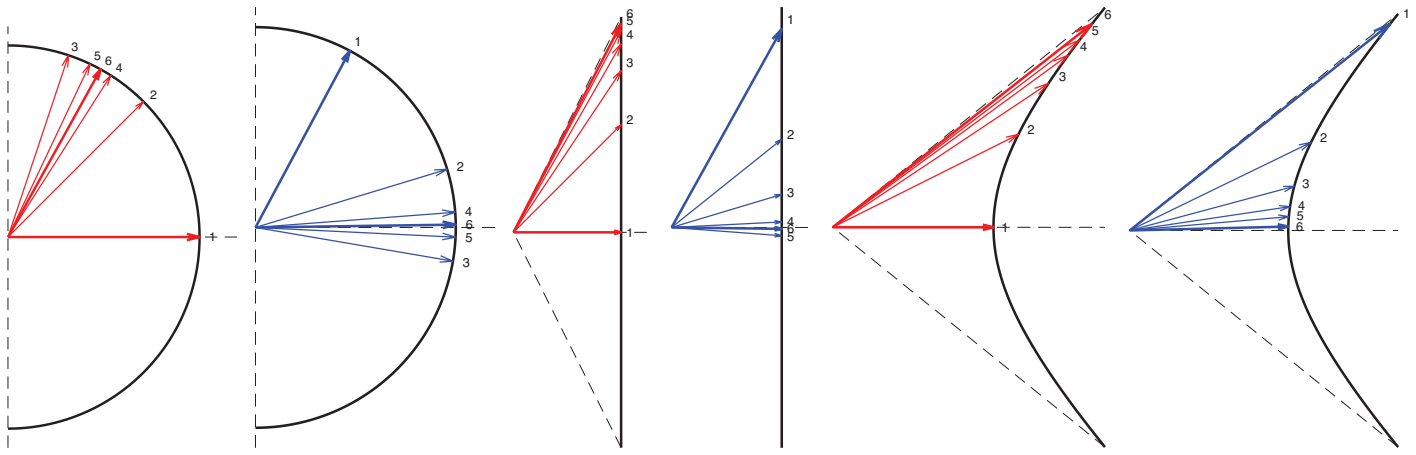

Figure 1.6: Geometric interpretation of the (successively smaller and smaller) rotations of the iterative CORDIC algorithm developed in §1.5.3.1, for (a, b) circular, (c, d) linear, and (e, f) hyperbolic rotations, illustrating both (a, c, e) "rotation" mode, which performs a generalized rotation of the vector (x_0, y_0) [illustrated here for $(x_0, y_0) = (1, 0)$] by the angle z_0, and (b, d, f) "vectoring" mode, which rotates the vector (x_0, y_0) to the positive x axis, while incrementing z_0 by the angle Δz required for such a rotation. Code at RR_cordic_viz.m.

[17] The PIC on the ARM Cortex M4, as depicted in Figure 1.4, is called a nested vectored interrupt controller (NVIC).

circular	$\mu = 1,$	$\alpha_i = \text{atan}\,(1/2^i),$	$f_i = \tan(\alpha_i) = 1/2^i,$	$\bar{K}_i = 1/\sqrt{1+1/2^{2i}}$
linear	$\mu = 0,$	$\alpha_i = 1/2^i,$	$f_i = 1/2^i,$	$\bar{K}_i = 1$
hyperbolic	$\mu = -1,$	$\alpha_i = \text{atanh}(1/2^i),$	$f_i = \tanh(\alpha_i) = 1/2^i,$	$\bar{K}_i = 1/\sqrt{1-1/2^{2i}}$

Table 1.3: Formulas for μ, α_i, f_i, and $\bar{K}_i$, for (top) circular, (middle) linear, and (bottom) hyperbolic CORDIC rotations. Defining $K_i = \bar{K}_1\,\bar{K}_2\,\cdots\bar{K}_i$, the first few values of $\{\alpha_i,\ f_i,\ K_i\}$ are reported in Table 1.4.

circular $i = 0, 1, 2, \ldots$	$\alpha_i =$	0.78540,	0.46365,	0.24498,	0.12435,	0.06242,	0.03124,	0.01562,	...
	$f_i =$	1,	1/2,	1/4,	1/8,	1/16,	1/32,	1/64,	...
	$K_i =$	0.70711,	0.63246,	0.61357,	0.60883,	0.60765,	0.60735,	0.60728,	...
linear $i = 0, 1, 2, \ldots$	$\alpha_i =$	1,	1/2,	1/4,	1/8,	1/16,	1/32,	1/64,	...
	$f_i =$	1,	1/2,	1/4,	1/8,	1/16,	1/32,	1/64,	...
	$K_i =$	1,	1,	1,	1,	1,	1,	1,	...
hyperbolic $i = 1, 2, 3 \ldots$	$\alpha_i =$	0.54931,	0.25541,	0.12566,	0.06258,	0.06258,	0.03126,	0.01563,	...
	$f_i =$	1/2,	1/4,	1/8,	1/16,	1/16,	1/32,	1/64,	...
	$K_i =$	1.15470,	1.19257,	1.20200,	1.20435,	1.20671,	1.20730,	1.20745,	...

Table 1.4: Angles α_i, rotation factors f_i, and cumulative scale factors K_i of the CORDIC algorithm for (top) circular, (middle) linear, and (bottom) hyperbolic rotations. Note there are two rotations in the hyperbolic case with $f = 1/16$ and $\alpha = \text{atanh}(1/16) = 0.06258$ (see text). The full table of coefficients needed to apply CORDIC to achieve single-precision floating-point accuracy in all cases is computed in RR_cordic_init.m.

1.5.3.1 CORDIC approximation of transcendental functions†

The efficient *software* approximation (to a selectable precision) of various transcendental functions is discussed in detail in §2.7. Specialized *hardware* suitable for approximating such functions even faster (again, to selectable precision), while offloading the CPU for other tasks, may also be implemented. The clever algorithm underlying such hardware is known as CORDIC (coordinate rotation digital computer), and is well suited for compact implementation on both ASICs and more general-purpose coprocessors (DSPs, FPGAs, etc).

We will discuss the CORDIC *algorithm* itself first, including its software and hardware implementations; *interpretation* [Figure 1.6] of the convergence of the CORDIC algorithm is deferred to the end of this section.

The operations on $\{x,\ y,\ z\}$ that underlie all six forms of CORDIC are given, at each iteration, by

$$\begin{pmatrix} x \\ y \end{pmatrix}_{i+1} = \bar{K}_i \begin{pmatrix} 1 & -\mu\,\sigma_i f_i \\ \sigma_i f_i & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}_i, \tag{1.1a}$$

$$z_{i+1} = z_i - \sigma_i \alpha_i. \tag{1.1b}$$

That is, at each iteration, a "scaled rotation" is performed on $(x,\ y)$, and z is incremented by $\pm\alpha_i$. The coefficients $\{\alpha_i,\ \bar{K}_i\}$ may be precalculated according to the various formula given in Table 1.3, the first few values of which are listed in Table 1.4. The variable μ is just a sign bit, and is set as 1 for circular rotations, 0 for linear rotations, and -1 for hyperbolic rotations. The variable σ_i is also a sign bit, and is selected so that each iteration drives either z (for "rotation" mode) or y (for "vectoring" mode) towards zero as the iterations proceed. The factor f_i (and, in the case of linear rotations, the corresponding angle α_i) is halved at each iteration. In the case of circular and hyperbolic rotations, the first several angles α_i may be stored in small look up tables on

†This section, like later sections of this text marked with a dagger (†), is a bit harder than those around it, and may be skimmed or skipped upon first read without significantly disrupting the continuity of the presentation.

Algorithm 1.1: Main iteration of the CORDIC algorithm; full code available at RR_cordic_core.m.

```
for j=1:n                                                    % perform n iterations
  % Compute sign of next rotation (mode=1 for "rotation", mode=2 for "vectoring")
  if mode==1, sigma=sign(v(3)); else, sigma=-sign(v(2)); end

  %%%% BELOW IS THE HEART OF THE CORDIC ALGORITHM %%%%
  v(1:2)=[1 -mu*sigma*f; sigma*f 1]*v(1:2);    % generalized rotation of v(1:2) by f
  v(3)  =v(3)-sigma*ang;                       % increment v(3)

  % update f (divide by 2) [factors {1/2^4, 1/2^13, 1/2^40} repeated in hyperbolic case]
  if mu>-1 || ((j~=4) && (j~=14) && (j~=42)), f=f/2; end
  % update ang from tables, or divide by 2
  if j+1<=cordic_tables.N && rot<3, ang=cordic_tables.ang(rot,j+1); else, ang=ang/2; end
end
% NOTE: the scaling of v by K, if necessary, is done in RR_cordic.m, not in this code.
```

the (hardware) CORDIC unit; once α_i becomes sufficiently small (at around iteration $i = 25$), the subsequent α_i are, again, simply halved at each iteration. Finally (important!), defining a *cumulative* scaling factor after n iterations such that $K_n = \bar{K}_1 \bar{K}_2 \cdots \bar{K}_n$, which may also be precalculated, multiplication by the individual scaling factors $\bar{K}_i$ in (1.1a) may be deferred, and the cumulative scaling factor K_n instead applied to (x, y) by the CORDIC preprocessing unit, either at the end, or at the beginning, of the n iterations performed.

A full floating-point implementation of the above algorithm is available at RR_cordic_core.m, with extensive preprocessors at RR_cordic.m and RR_cordic_derived.m; the core of this code is listed in Algorithm 1.1. Note that such a *software* implementation of CORDIC is actually not very efficient as compared with the software approximation of transcendental functions using Chebyshev expansions, as discussed in §2.7. Where CORDIC becomes particularly useful, however, is its realization in specialized *hardware*, including both ASICs and high-performance coprocessors like DSPs and FPGAs (see §1.5.4), using fixed-point binary representations (see §1.1.2) of the real numbers involved. In this setting, the halving operations in Algorithm 1.1 may be accomplished quickly, with single bit shifts (to the right) of the corresponding fixed-point numbers. Further, one can implement the logic of the sign bits (that is, σ and μ) essentially for free. In such hardware, the computational cost of most of the iterations of Algorithm 1.1 in the case of circular or hyperbolic rotations is thus:

- three integer additions, during the generalized rotation of `v(1,2)` and the increment of `v(3)`,
- one bit shift, during the update of `f`, and
- one table lookup [or, a second bit shift], to determine the next value of `ang` (that is, of α_i).

An efficient hardware implementation of CORDIC is discussed in ST AN5325, which establishes that hardware implementations of CORDIC can have a very small silicon footprint, and in many cases of interest (for various transcendental functions, at specified levels of precision) can be substantially faster than computing these same functions using precompiled software libraries (see §2.7). Note in particular (in Table 1.4) that the angles reduce by about a factor of two at each iteration; convergence (i.e., the additional accuracy achieved per iteration) of this algorithm is thus said to be **linear**. Other iterative algorithms we will encounter later have substantially faster convergence; the key to the success of CORDIC is its remarkably simplicity, as itemized above.

In the remainder of this section, we turn to the *interpretation* of what the CORDIC iterations defined above actually accomplish. As mentioned previously, there are $3 \cdot 2 = 6$ forms of the CORDIC algorithm, with:

- three different types of rotations: circular ($\mu = 1$), linear ($\mu = 0$), or hyperbolic ($\mu = -1$) [see Table 1.3], and
- two different modes for σ_i:

rotation mode, which takes $\sigma_i = \text{sign}(z_i)$, eventually driving $z_n \to 0$ upon convergence, or (1.2a)

vectoring mode, which takes $\sigma_i = -\text{sign}(y_i)$, eventually driving $y_n \to 0$ upon convergence. (1.2b)

Interpretation in the case of circular rotations.
For circular rotations ($\mu = 1$), noting that $\cos^2 x + \sin^2 x = 1$ and $\tan x = \sin x / \cos x$ and thus

$$\cos\alpha_i = 1/\sqrt{1+\tan^2\alpha_i} = 1/\sqrt{1+2^{-2i}} = \bar{K}_i \quad \text{and} \quad \sin\alpha_i = \tan\alpha_i/\sqrt{1+\tan^2\alpha_i} = \bar{K}_i\,\tan\alpha_i,$$

the rotation in (1.1a) may be written [note: the scaling by $\bar{K}_i$ is deferred in the code]

$$\begin{pmatrix} x \\ y \end{pmatrix}_{i+1} = G_i \begin{pmatrix} x \\ y \end{pmatrix}_i \quad \text{where} \quad G_i = \bar{K}_i \begin{pmatrix} 1 & -\sigma_i f_i \\ \sigma_i f_i & 1 \end{pmatrix} = \begin{pmatrix} \cos(\sigma_i\alpha_i) & -\sin(\sigma_i\alpha_i) \\ \sin(\sigma_i\alpha_i) & \cos(\sigma_i\alpha_i) \end{pmatrix}; \tag{1.3}$$

this is called a Givens rotation, and corresponds to an anticlockwise rotation of $(x, y)_i$ by the angle $(\sigma_i\alpha_i)$ at each iteration. Of course, successive Givens rotations accumulate; denoting $\phi_3 = \phi_2 + \phi_1$, $c_i = \cos(\phi_i)$, $s_i = \sin(\phi_i)$, and applying the identities $c_2c_1 - s_2s_1 = c_3$ and $s_2c_1 + c_2s_1 = s_3$, this may be verified as follows:

$$\begin{pmatrix} c_2 & -s_2 \\ s_2 & c_2 \end{pmatrix} \begin{pmatrix} c_1 & -s_1 \\ s_1 & c_1 \end{pmatrix} = \begin{pmatrix} c_2c_1 - s_2s_1 & -c_2s_1 - s_2c_1 \\ s_2c_1 + c_2s_1 & -s_2s_1 + c_2c_1 \end{pmatrix} = \begin{pmatrix} c_3 & -s_3 \\ s_3 & c_3 \end{pmatrix}. \tag{1.4}$$

Thus, successive applications of (1.3) result in a total Givens rotation of the original $(x_0,\, y_0)$ vector by $\alpha = \sum_{i=0}^{n} \sigma_i\alpha_i$. Note that the α_i are scaled by a factor of 0.5, or slightly larger, at each iteration; as a result, for large n and by appropriate selection of the σ_i, total rotations anywhere in the range $-\alpha_{\text{max}} \le \alpha \le \alpha_{\text{max}}$ are possible, where $\alpha_{\text{max}} = \sum_{i=0}^{n} |\sigma_i\alpha_i| = \sum_{i=0}^{n} \alpha_i = 1.743287$ (that is, a bit over $\pi/2$). Thus:

• Using rotation mode (1.2a), selecting $\sigma_i = \text{sign}(z_i)$ at each iteration so that $z_n \to 0$ for large n, a total Givens rotation of $-\alpha_{\text{max}} \le z_0 \le \alpha_{\text{max}}$ radians [and a cumulative scaling of K_n^{-1}] is applied to the $(x_0,\, y_0)$ vector [see Table 1.5]. As a special case, defining $(x_0,\, y_0) = (K_n,\, 0)$, we have $(x_n,\, y_n) \to (\cos z_0,\, \sin z_0)$ in this mode.

• Using vectoring mode (1.2b), selecting $\sigma_i = -\text{sign}(y_i)$ at each iteration, the original $(x_0,\, y_0)$ vector is rotated [if K_n is applied] along a curve of constant $x^2 + y^2$ (that is, along a curve of constant radius from the origin) such that $y_n \to 0$, while the increments of z_i in (1.1b) again keep track of the total rotation performed in the process of rotating the vector $(x_0,\, y_0)$ to $(x_n,\, 0)$, so that $(x_n,\, z_n) \to (K_n^{-1}\sqrt{x_0^2 + y_0^2},\, z_0 + \text{atan}\,(y_0/x_0))$.

Interpretation in the case of hyperbolic rotations.
For hyperbolic rotations ($\mu = -1$), noting that $\cosh^2 x - \sinh^2 x = 1$ and $\tanh x = \sinh x / \cosh x$ and thus

$$\cosh\alpha_i = 1/\sqrt{1-\tanh^2\alpha_i} = 1/\sqrt{1-2^{-2i}} = \bar{K}_i \quad \text{and} \quad \sinh\alpha_i = \tanh\alpha_i/\sqrt{1-\tanh^2\alpha_i} = \bar{K}_i\,\tanh\alpha_i,$$

the transformation in (1.1a) may be written [note: the scaling by $\bar{K}_i$ is deferred in the code]

$$\begin{pmatrix} x \\ y \end{pmatrix}_{i+1} = H_i \begin{pmatrix} x \\ y \end{pmatrix}_i \quad \text{where} \quad H_i = \bar{K}_i \begin{pmatrix} 1 & \sigma_i f_i \\ \sigma_i f_i & 1 \end{pmatrix} = \begin{pmatrix} \cosh(\sigma_i\alpha_i) & \sinh(\sigma_i\alpha_i) \\ \sinh(\sigma_i\alpha_i) & \cosh(\sigma_i\alpha_i) \end{pmatrix}. \tag{1.5}$$

This transformation is called a "hyperbolic rotation". Successive transformations by H_i also accumulate; denoting $\phi_3 = \phi_2 + \phi_1$, $C_i = \cosh(\phi_i)$, $S_i = \sinh(\phi_i)$, and applying the identities $C_2C_1 + S_2S_1 = C_3$ and $S_2C_1 + C_2S_1 = S_3$, this may be verified as follows [cf. (1.4)]:

$$\begin{pmatrix} C_2 & S_2 \\ S_2 & C_2 \end{pmatrix} \begin{pmatrix} C_1 & S_1 \\ S_1 & C_1 \end{pmatrix} = \begin{pmatrix} C_2C_1 + S_2S_1 & C_2S_1 + S_2C_1 \\ S_2C_1 + C_2S_1 & S_2S_1 + C_2C_1 \end{pmatrix} = \begin{pmatrix} C_3 & S_3 \\ S_3 & C_3 \end{pmatrix}. \tag{1.6}$$

Thus, successive applications of (1.5) result again in a total rotation of the $(x_0,\, y_0)$ vector by $\alpha = \sum_{i=0}^{n} \sigma_i\alpha_i$. In contrast with the circular case, the α_i in the hyperbolic case are scaled by a factor of 0.5, or slightly *smaller*, as

	rotation mode ($z_n \to 0$)	vectoring mode ($y_n \to 0$)
circular ($\mu = 1$)	$\begin{pmatrix} x_n \\ y_n \end{pmatrix} \to K_n^{-1} \begin{pmatrix} \cos z_0 & -\sin z_0 \\ \sin z_0 & \cos z_0 \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$	$\begin{pmatrix} x_n \\ z_n \end{pmatrix} \to \begin{pmatrix} K_n^{-1}\sqrt{x_0^2 + y_0^2} \\ z_0 + \text{atan}\,(y_0/x_0) \end{pmatrix}$
linear ($\mu = 0$)	$\begin{pmatrix} x_n \\ y_n \end{pmatrix} \to \begin{pmatrix} x_0 \\ y_0 + z_0\, x_0 \end{pmatrix}$	$\begin{pmatrix} x_n \\ z_n \end{pmatrix} \to \begin{pmatrix} x_0 \\ z_0 + y_0/x_0 \end{pmatrix}$
hyperbolic ($\mu = -1$)	$\begin{pmatrix} x_n \\ y_n \end{pmatrix} \to K_n^{-1} \begin{pmatrix} \cosh z_0 & \sinh z_0 \\ \sinh z_0 & \cosh z_0 \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$	$\begin{pmatrix} x_n \\ z_n \end{pmatrix} \to \begin{pmatrix} K_n^{-1}\sqrt{x_0^2 - y_0^2} \\ z_0 + \text{atanh}(y_0/x_0) \end{pmatrix}$

Table 1.5: Convergence of the CORDIC algorithm for large n. Leveraging various identities, several derived functions may also be determined, as implemented in RR_cordic.m and RR_cordic_derived.m.

i is increased. Thus, in order to assure that *all* angles over a continuous range can be reached by a set of successive rotations, the typical approach used is to do two rotations associated with the angles $\alpha = \text{atanh}(1/2^4)$, $\text{atanh}(1/2^{13})$, and $\text{atanh}(1/2^{40})$ [see, e.g., Table 1.4]. With three such double-rotations built into the algorithm, it may be shown that, for large n and by appropriate selection of the σ_i, total rotations anywhere in the range $-\alpha_{\max} \le \alpha \le \alpha_{\max}$ are possible, where now $\alpha_{\max} = \sum_{i=0}^{n} \alpha_i = 1.118173$. Thus:

• Using rotation mode (1.2a), selecting $\sigma_i = \text{sign}(z_i)$ at each iteration so that $z_n \to 0$ for large n, a total generalized rotation of $-\alpha_{\max} \le z_0 \le \alpha_{\max}$ radians [and a cumulative scaling of K_n^{-1}] is applied to the $(x_0,\, y_0)$ vector [see Table 1.5]. As a special case, defining $(x_0,\, y_0) = (K_n,\, 0)$, we have $(x_n,\, y_n) \to (\cosh z_0,\, \sinh z_0)$.

• Using vectoring mode (1.2b), selecting $\sigma_i = -\text{sign}(y_i)$ at each iteration, the original $(x_0,\, y_0)$ vector is rotated [if K_n is applied] along a curve of constant $x^2 - y^2$ such that $y_n \to 0$, while the increments of z_i in (1.1b) again keep track of the total rotation performed in the process of rotating the vector $(x_0,\, y_0)$ to $(x_n,\, 0)$, so that $(x_n,\, z_n) \to (K_n^{-1}\sqrt{x_0^2 - y_0^2},\, z_0 + \text{atanh}(y_0/x_0))$.

Interpretation in the case of linear rotations.
For linear rotations ($\mu = 0$), the transformation in (1.1a) may be written

$$\begin{pmatrix} x \\ y \end{pmatrix}_{i+1} = J \begin{pmatrix} x \\ y \end{pmatrix}_i \quad \text{where} \quad J = \begin{pmatrix} 1 & 0 \\ \sigma_i f_i & 1 \end{pmatrix}. \tag{1.7}$$

Again, successive transformations by J accumulate, which may be verified as follows:

$$\begin{pmatrix} 1 & 0 \\ f_2 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ f_1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ f_2 + f_1 & 1 \end{pmatrix} \tag{1.8}$$

Thus, successive applications of (1.7) result in a translation of y_0 by $\sum_{i=0}^{n} \sigma_i f_i x_0$ (note that the x_i remain constant, due to the first row of J). The f_i in the linear case are exactly halved at each iteration, assuring convergence, for large n and by appropriate selection of the σ_i, of total translations anywhere in the range $-\Delta y_{\max} \le \Delta y \le \Delta y_{\max}$, where $\Delta y_{\max} = \sum_{i=0}^{n} |\sigma_i f_i x_0| = c\,|x_0|$, where we have initialized $\alpha_0 = f_0 = 1$ (see Table 1.4), so that $c = 2$ (but other choices are certainly possible). Thus:

• Using rotation mode (1.2a), selecting $\sigma_i = \text{sign}(z_i)$ at each iteration so that $z_n \to 0$ for large n, a total translation of $\Delta y = z_0\, x_0$ is applied to y_0 [see Table 1.5]. As a special case, defining $y_0 = 0$, we have $y_n \to z_0\, x_0$.

• Using vectoring mode (1.2b), selecting $\sigma_i = -\text{sign}(y_i)$ at each iteration, the original $(x_0,\, y_0)$ vector is rotated along a line of constant x, such that $y_n \to 0$. Noting that $\Delta y = z_0 x_0$ in rotation mode, it is seen that vectoring mode is again its complement, with $\Delta z = y_0/x_0$ [see Table 1.5].

In practice, linear mode is useful for approximating multiply/accumulate and divide/accumulate operations on very simple hardware that is only capable of integer addition and bit shifts.

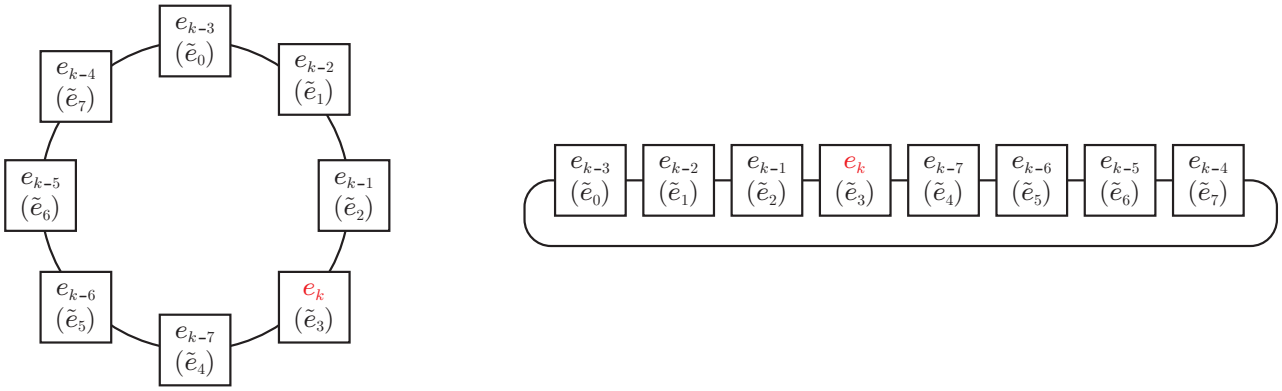

Figure 1.7: A ring buffer of the DT signal e_k and its 7 most recent tap delays at timestep $k = 11$: (left) as laid out conceptually, as a ring, and (right) as laid out in memory. At timestep $k = 12$, the next value, e_{12}, replaces the value in memory location $\tilde{e}_4$, the counter k is incremented by 1, and the other existing values of $\tilde{e}$ stay put.

1.5.3.2 Ring buffers

Many of the essential operations that an embedded controller needs to implement are linear discrete-time (DT) difference equations (see §??) that may be written as **finite impulse response** (FIR) filters of the form

$$u_k = b_0\, e_k + b_1\, e_{k-1} + \ldots + b_n\, e_{k-n}, \tag{1.9a}$$

or **infinite impulse response** (IIR) filters of the form

$$u_k = -a_1\, u_{k-1} - \ldots - a_m\, u_{k-m} + b_0\, e_k + b_1\, e_{k-1} + \ldots + b_n\, e_{k-n}. \tag{1.9b}$$

To perform such computations quickly, in addition to fast access (see §1.3) to the (fixed) a_i and b_i coefficients, fast access to current and recent values (aka **tap delays**) of the DT signals e_k and (in the case of IIR filters) u_k are needed. Instead of shifting all of these most recent values in memory at every timestep, a much faster approach is to use a ring buffer (aka circular buffer), such as that illustrated in Figure 1.7 (with $r = 8$ elements). With this approach, at each timestep k, the most recent value of e_k is stored in memory location $\tilde{e}_{\text{mod}(k,r)}$ [that is, within a ring buffer with $r \geq n$ memory locations allocated] using modular arithmetic, and u_k is given by:

$$\tilde{u}_{\text{mod}(k,r)} = b_0\, \tilde{e}_{\text{mod}(k,r)} + b_1\, \tilde{e}_{\text{mod}(k-1,r)} + \ldots + b_n\, \tilde{e}_{\text{mod}(k-n,r)} \qquad \text{or} \tag{1.10a}$$

$$\tilde{u}_{\text{mod}(k,r)} = -a_1\, \tilde{u}_{\text{mod}(k-1,r)} - \ldots - a_m\, \tilde{u}_{\text{mod}(k-m,r)} + b_0\, \tilde{e}_{\text{mod}(k,r)} + b_1\, \tilde{e}_{\text{mod}(k-1,r)} + \ldots + b_n\, \tilde{e}_{\text{mod}(k-n,r)}. \tag{1.10b}$$

This approach renders it unnecessary to shift each of the saved values of e and u in memory by one location at each timestep, instead simply shifting the index k used to reference these values in their (fixed, until replaced) locations in the ring buffers, and using this index (and reduced values of it, like $k - j$) in a modulo fashion.

FIR filters (1.10a) and IIR filters (1.10b) are needed so often in embedded computing that many modern CPU cores targeting applications in robotics and cyberphysical systems include specialized hardware or software implementations of both the ring buffers themselves (with the mod command on the indices handled automatically) together with the additional low-level multiply/add circuitry or code required to implement such filters remarkably quickly, without significantly burdening the available CPU core(s).

In many DT filters, dubbed **strictly causal** (see §??), $b_0 = 0$. In such problems, (1.10a) or (1.10b) can simply be calculated between timestep $k - 1$ and timestep k.

In the case of **semi-causal** filters, however, $b_0 \neq 0$. In such problems, the strictly causal part of the RHS of (1.10a) or (1.10b) [which may involve a substantial number computations if n or m is large] can still be calculated between timestep $k - 1$ and timestep k. As soon as the new value of e_k becomes available, the RHS can then be updated by adding $b_0 \cdot e_k$, and then the result may be applied directly on the output as u_k, thus applying the output u_k very very soon after the input e_k is received, though not quite instantaneously.

Ring buffers, as described above, can be implemented in dedicated hardware, such as ST's FMAC units (see ST AN5305), or in software, such as when using ARM Helium. Performing such computations in ASIC hardware on the MCU has the obvious advantage of offloading the ARM core; however, the size (and, the associated capabilities) of such ASICs needs to be decided upon when the (general purpose) MCU is designed, when the demands of the ultimate application are largely unknown. Performing such computations with streamlined software constructs on the ARM core of the MCU leads to a much more scalable solution (from small filters to large) to better fit the demands of the end application. Thus, ring buffers for MCUs targeting a narrow range of applications are most efficiently implemented on appropriately-sized ASICs; for general-purpose MCUs targeting a more broad range of applications, software-based solutions might be preferred.

1.5.3.3 Cyclic redundancy check (CRC) calculation units

As introduced in §1.1.4, and discussed in much greater detail in §??,

1.5.3.4 Random Number Generators (RNGs)

MCUs are useful because they are deterministic. (Joke: the definition of crazy...)

1.5.4 Coprocessors: DSPs, GPUs, NPUs, FPGAs, CPLDs, PRUs

More general-purpose coprocessors than ASICs, but with more specialized structure than CPUs, are sometimes called application-specific standard parts (**ASSPs**) or application-specific instruction-set processors (**ASIPs**). Many are perhaps best considered as some kind of System-On-Chip (**SoC**). Regardless of ambiguity in the literature on precisely what to call this general category of coprocessor, there are a handful of well-defined classes of coprocessors in this general category that are of principal importance in many modern MCUs, including:

DSP

GPU A GPU consists of multiple SIMD units with a large amount of associated memory.

NPU

FPGA

CPLD

PRU

1.5.5 Timer / counter units

PWM

Encoders

1.5.6 Other dedicated communication hardware

The major wired and wireless communication protocols available today for embedded systems include PWM, UART, I2C, SPI, CAN, RS485, USB, Ethernet, Wifi, and Bluetooth, among others, as discussed further in Chapter 3. Most MCUs implement dedicated hardware to support a number of these communication modes.

1.5.7 Pin multiplexing

1.6 Single Board Computers (SBCs)

1.6.1 Subsystem integration: SiPs, PoPs, SoCs, SoMs, and CoMs

Integration of ICs:

- System-In-Package (SiP),
- Package-On-Package (PoP),
- System-On-Chip (SoC),
- System On Module (SoM),
- Computer On Module (CoM)

acronyms

1.6.2 Power management

i. ultra-low standby and sleep modes for battery-based operation, with various cues available for wakeup, etc.,

1.6.2.1 Sleep/wake modes, real-time clocks

IoT and low-power modes
Clock speed regulation

1.6.2.2 Switching regulators

efficiency vs. ripple rejection & voltage stability/accuracy

1.6.2.3 Switching regulators

1.6.2.4 Low-dropout (LDO) regulators

LDO
Power
Internet of Things

1.6.3 Case study: Raspberry Pi

Daughterboards
A detailed case study of a powerful class of daughterboards, dubbed Berets, is provided in §5.

model	connectivity	MCU	specs	PCB
UNO rev3		ATmega328P[8]	1x 20 MHz AVR	69 x 53
Mega 2560 rev3		ATmega2560[8]	1x 16 MHz AVR	102 x 53
MKR WAN 1300	LoRa	SAMD21[32]	1x 48 MHz M0+	68 x 25

Table 1.6: Some popular Arduino boards, and the principal specs of their Microchip MCUs; $()^N$ denotes an N-bit MCU.

	popular models	Berets	connectivity	MCU	specs	size
96Boards	Qualcomm RB3	Green	W5, BT5, GigE	SDA845 SoC[64]	4x 2.8 GHz A75 + 1.8 GHz 4x A55, DSP	85 x 54
	Shiratech Stinger96	Green	LTE	STM32MP157[32]	2x 800 MHz A7, 1x 209 MHz M4, GPU	85 x 54
ASUS	Tinkerboard	Rasp	W4, B4, GigE	RK3288[32]	4x 1.8 GHz A17, GPU	86 x 54
Banana Pi*	Banana Pi M64	Rasp	W5, BT4, GigE	Allwinner A64[64]	4x 1.2 GHz A53, GPU	92 x 60
	Banana Pi M4	Rasp	W5, BT4.2, GigE	RTD395[64]	4x A53, GPU	92 x 60
BeagleBoard*	BeagleBone AI	Black	W5, BT4.2, GigE	AM5729[32]	2x 1.5 GHz A15, 2x DSP, GPU, PRU, 4x M4	86 x 53
	BeagleBone Blue		W4, BT4.1	OSD3358 SiP[32]	1x 1 GHz A8, PRU, 1x M3	86 x 53
	BB Black Wireless	Black	W4, BT4.1	OSD3358 SiP[32]	1x 1 GHz A8, PRU, 1x M3	86 x 53
Hardkernel	ODROID XU4	Rasp†	GigE	Exynos 5422[32]	4x 2.1 GHz A15, 4x 1.4 GHz A7, GPU	83 x 58
Khadas	VIM3	Rasp	module	Amlogic A311D[64]	4x 2.2 GHz A73, 2x 1.8 GHz A53, GPU, NPU	82 x 58
Libre*	Le Potato	Rasp	LAN	Amlogic S905X[64]	4x 1.512 GHz A53, GPU	85 x 56
NanoPi*	NEO2	Red	GigE	Allwinner H5[64]	4x 1.5 GHz A53	40 x 40
NVIDIA	Jetson Nano Dev Kit	Rasp	GigE	(custom)[64]	4x 1.43 GHz A57, 128x Maxwell GPU	100 x 80
	Jetson Xavier NX	Rasp	W5, BT5, GigE	(custom)[64]	6x Carmel, 384x Volta GPU + 48x Tensor	103 x 90
Orange Pi*	Orange Pi 4B	Rasp	W5, BT5, GigE	RK3399[64]	2x 1.8GHz A72, 4x 1.4GHz A53, GPU	91 x 56
	OPi Zero LTS	Red	W4, LAN	AllWinner H2+[32]	4x 1.296 GHz A7, GPU	48 x 46
Pine64*	RockPro64	Rasp	GigE, module	RK3399[64]	2x 1.8GHz A72, 4x 1.4GHz A53, GPU	133 x 80
Radxa*	Rock Pi 4	Rasp	W5, BT5, GigE	RK3399[64]	2x 1.8GHz A72, 4x 1.4GHz A53, GPU	85 x 54
	Rock Pi S	Red	W4, BT4, LAN	RK3308[64]	4x 1.3 GHz A35	38 x 38
Raspberry Pi	Raspberry Pi 4B	Rasp	W5, BT5, GigE	BCM2711[64]	4x 1.5 GHz A72	85 x 56
	Raspberry Pi 3A+	Rasp	W5, BT4.2	BCM2837B0[64]	4x 1.4 GHz A53	65 × 56
	RPi Zero W	Rasp	W4, BT4.1	BCM2835[32]	1x 1 GHz ARM11	65 x 30

Table 1.7: Some popular linux-based Single-Board Computers (SBCs), and their principal specs; $()^N$ denotes an N-bit MCU. *SBCs from Banana Pi, BeagleBoard, Libre, NanoPi, Orange Pi, Pine64, and Radxa are **??**, facilitating derivative designs. Any SBC compatible with the Raspberry Beret is also compatible with the Red Beret. †A shifter shield is required to connect a Raspberry or Red Beret to the ODROID XU4. In the above list, LAN implies 100Base-T ethernet (i.e., 10x slower than GigE).