

Microprocessors and Digital Systems

Semester 2, 2022

Dr Mark Broadmeadow

Tarang Janawalkar

This work is licensed under a Creative Commons
“Attribution-NonCommercial-ShareAlike 4.0 International” license.



Contents

Contents	1
1 Microcontroller Fundamentals	3
1.1 Architecture of a Computer	3
1.2 Microprocessors & Microcontrollers	3
1.3 ATtiny1626 Microcontroller	3
1.3.1 Flash Memory	3
1.3.2 SRAM	4
1.3.3 EEPROM	4
1.4 AVR Core	4
1.5 Programme Execution	4
1.6 Instructions	5
1.7 Interacting with memory and peripherals	6
1.8 Memory map	7
1.9 Assembly code	7
2 Digital Representations and Operations	7
2.1 Digital Systems	7
2.2 Representation	8
2.2.1 Binary Representation	8
2.2.2 Hexadecimal Representation	8
2.2.3 Numeric Literals	9
2.3 Unsigned Integers	9
2.4 Signed Integers	10
2.4.1 Sign-Magnitude	10
2.4.2 One's Complement	10
2.4.3 Two's Complement	11
2.5 Logical Operators	11
2.5.1 Boolean Functions	11
2.5.2 Negation	11
2.5.3 Conjunction	12
2.5.4 Disjunction	12
2.5.5 Exclusive Disjunction	12
2.5.6 Bitwise Operations	13
2.6 Bit Manipulation	13
2.6.1 Setting Bits	13
2.6.2 Clearing Bits	13
2.6.3 Toggling Bits	14
2.6.4 One's Complement	14
2.6.5 Two's Complement	15
2.6.6 Shifts	15
2.6.7 Rotations	16
2.7 Arithmetic Operations	16
2.7.1 Addition	16

2.7.2	Overflows	17
2.7.3	Subtraction	18
2.7.4	Multiplication	18
2.7.5	Division	19
3	Microcontroller Interfacing	19
3.1	Logic Levels	19
3.1.1	Discretisation	19
3.1.2	Logic Levels	19
3.1.3	Hysteresis	20
3.2	Electrical Quantities	20
3.2.1	Voltage	20
3.2.2	Current	21
3.2.3	Power	21
3.2.4	Resistance	21
3.3	Electrical Components	21
3.3.1	Resistors	21
3.3.2	Switches	22
3.3.3	Diodes	23
3.3.4	Integrated Circuit	23
3.4	Digital Outputs	24
3.4.1	Push-Pull Outputs	24
3.4.2	High-Impedance Outputs	25
3.4.3	Pull-up and Pull-down Resistors	25
3.4.4	Open-Drain Outputs	26
3.5	Microcontroller Pins	27
3.5.1	Configuring an Output in Assembly	28
3.5.2	Configuring an Input in Assembly	28
3.5.3	Peripheral Multiplexing	29
3.6	Interfacing to Simple IO	29
3.6.1	Driving LEDs	29
3.6.2	Interfacing to LEDs	29
3.6.3	Switches as Digital Inputs	30
3.6.4	Interfacing to Switches	31
3.6.5	Interfacing to Integrated Circuits	31

1 Microcontroller Fundamentals

1.1 Architecture of a Computer

Definition 1.1 (Computer). A computer is a digital electronic machine that can be programmed to carry out sequences of arithmetic or logical operations (computation) automatically.

Definition 1.2 (Control unit). The control unit interprets the instructions and decides what actions to take.

Definition 1.3 (Arithmetic logic unit). The arithmetic logic unit (ALU) performs computations required by the control unit.

1.2 Microprocessors & Microcontrollers

While a microcontroller puts the CPU and all peripherals onto the same chip, a microprocessor houses a more powerful CPU on a single chip that connects to external peripherals. The peripherals include memory, I/O, and control units.

The QUTy uses a microcontroller called ATtiny1626, that are within a family of microcontrollers called AVR.

1.3 ATtiny1626 Microcontroller

The ATtiny1626 microcontroller has the following features:

- CPU: AVR Core (AVRxt variant)
- Memory:
 - Flash memory (16KB) used to store program instructions in memory
 - SRAM (2KB) used to store data in memory
 - EEPROM (256B)
- Peripherals: Implemented in hardware (part of the chip) in order to offload complexity

1.3.1 Flash Memory

- Non-volatile — memory is not lost when power is removed
- Inexpensive
- Slower than SRAM
- Can only erase in large chunks
- Typically used to store programme data
- Generally read-only. Programmed via an external tool, which is loaded once and remains static during the lifetime of the program
- Writing is slow

1.3.2 SRAM

- Volatile — memory is lost when power is removed
- Expensive
- Faster than flash memory and is used to store variables and temporary data
- Can access individual bytes (large chunk erases are not required)

1.3.3 EEPROM

- Older technology
- Expensive
- Non-volatile
- Can erase individual bytes

1.4 AVR Core

Definition 1.4 (Computer programme). A computer programme is a sequence or set of instructions in a programming language for a computer to execute.

The main function of the AVR Core Central Processing Unit (CPU) is to ensure correct program execution. The CPU must, therefore, be able to access memory, perform calculations, control peripherals, and handle interrupts. Some key characteristics of the AVR Core are:

- 8-bit Reduced Instruction Set Computer (RISC)
- 32 working registers (R0 to R31)
- Program Counter (PC) — location in memory where the program is stored
- Status Register (SREG) — stores key information from calculations performed by the ALU (i.e., whether a result is negative)
- Stack Pointer — temporary data that doesn't fit into the registers
- 8-bit core — all data, registers, and operations, operate within 8-bits

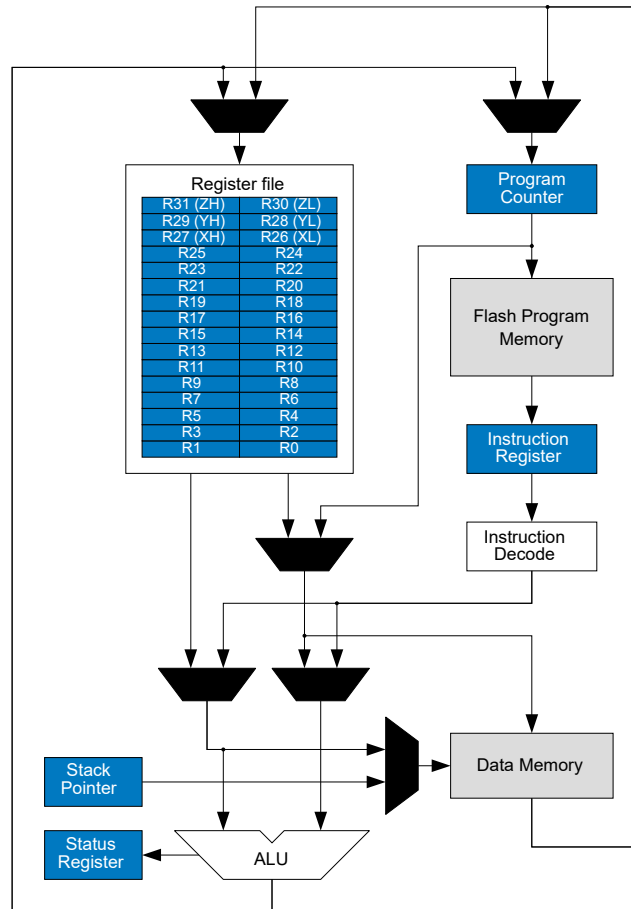
1.5 Programme Execution

At the time of reset, $PC = 0$. The following steps are then performed:

1. Fetch instruction (from memory)
2. Decode instruction (decode binary instruction)
3. Execute instruction:
 - Execute an operation

- Store data in data memory, the ALU, a register, or update the stack pointer
4. Store result
 5. Update PC (move to next instruction or if instruction is longer than 1 word, increment twice. The program can also move to another point in the program that has an address k , through jumps.)

This is illustrated in the following figure:



1.6 Instructions

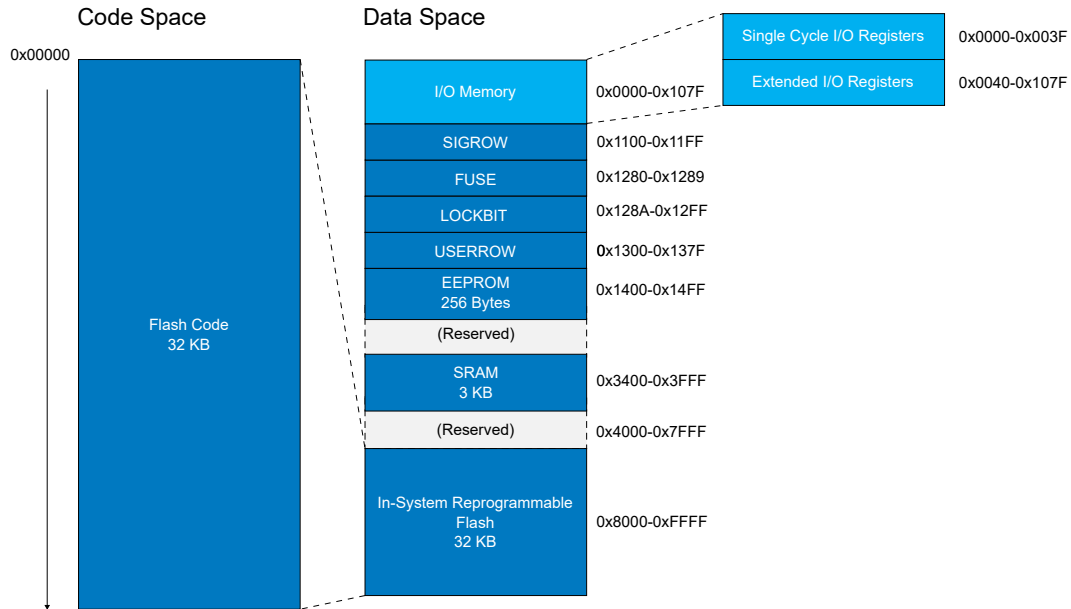
- The CPU understands and can execute a limited set of instructions — ~88 unique instructions for the ATtiny1626
- Instructions are encoded in programme memory as opcodes. Most instructions are two bytes long, but some instructions are four bytes long

- The AVR Instruction Set Manual describes all of the available instructions, and how they are translated into opcodes
- Instructions fall loosely into five categories:
 - Arithmetic and logic — ALU
 - Change of flow — jumping to different sections of the code or making decisions
 - Data transfer — moving data in/out of registers, into the data space, or into RAM
 - Bit and bit-test — looking at data in registers (which bits are set or not set)
 - Control — changing what the CPU is doing

1.7 Interacting with memory and peripherals

- The CPU interacts with both memory and peripherals via the data space
- From the perspective of the CPU, the data space is single large array of locations that can be read from, or written to, using an address
- We control peripherals by reading from, and writing to, their registers
- Each peripheral register is assigned a unique address in the data space
- When a peripheral is accessed in this manner we refer to it as being memory mapped, as we access them as if they were normal memory
- Different devices, peripherals and memory can be included in a memory map (and sometimes a device can be accessed at multiple different addresses)

1.8 Memory map



1.9 Assembly code

- The opcodes placed into programme memory are called machine code (i.e., code the machine operates on directly)
- We don't write machine code directly as it is:
 - Not human readable
 - Prone to errors (swapping a single bit can completely change the operation)
- Instead we can write instructions directly in assembly code
- We use instruction mnemonics to specify each instruction: **ldi**, **add**, **sts**, **jmp**, ...
- An assembler takes assembly code and translates it into opcodes that can be loaded into programme memory

2 Digital Representations and Operations

2.1 Digital Systems

A **bit**¹ is the most basic unit of information in a digital system. A bit encodes a logical state with one of two possible values (i.e., binary). These states are often referred to as:

¹The term *bit* comes from **b**inary **d**igit.

- true, false
- high, low (voltage states)
- on, off (logical states)
- set, reset
- 1, 0

A sequence of *eight* bits is known as a **byte**, and it is the most common representation of data in digital systems. A sequence of *four* bits is known as a **nibble**.

A sequence of n bits can represent up to 2^n states.

2.2 Representation

2.2.1 Binary Representation

The **binary system** is a base-2 system that uses a sequence of bits to represent a number. Bits are written left-to-right from **most significant** to **least significant** bit.

The first bit is the “most significant” bit because it is associated with the highest value in the sequence (coefficient of the highest power of two).

- The **least significant bit** (LSB) is at bit index 0.
- The **most significant bit** (MSB) is at bit index $n - 1$ in an n -bit sequence.

$0000_2 = 0$	$0100_2 = 4$	$1000_2 = 8$	$1100_2 = 12$
$0001_2 = 1$	$0101_2 = 5$	$1001_2 = 9$	$1101_2 = 13$
$0010_2 = 2$	$0110_2 = 6$	$1010_2 = 10$	$1110_2 = 14$
$0011_2 = 3$	$0111_2 = 7$	$1011_2 = 11$	$1111_2 = 15$

The subscript 2 indicates that the number is represented using a base-2 system.

2.2.2 Hexadecimal Representation

The **hexadecimal system** (hex) is a base-16 system. As we need 16 digits in this system, we use the letters A to F to represent digits 10 to 15.

Hex is a convenient notation when working with digital systems as each hex digit maps to a nibble.

$0_{16} = 0000_2$	$4_{16} = 0100_2$	$8_{16} = 1000_2$	$C_{16} = 1100_2$
$1_{16} = 0001_2$	$5_{16} = 0101_2$	$9_{16} = 1001_2$	$D_{16} = 1101_2$
$2_{16} = 0010_2$	$6_{16} = 0110_2$	$A_{16} = 1010_2$	$E_{16} = 1110_2$
$3_{16} = 0011_2$	$7_{16} = 0111_2$	$B_{16} = 1011_2$	$F_{16} = 1111_2$

2.2.3 Numeric Literals

When a fixed value is declared directly in a program, it is referred to as a **literal**. Generally, numeric literals can be expressed as either binary, decimal, or hexadecimal, so we use prefixes to denote various bases. Typically,

- **Binary** notation requires the prefix `0b`
- **Decimal** notation does not require prefixes
- **Hexadecimal** notation requires the prefix `0x`

For example, `0x80 = 0b10000000 = 128`.

2.3 Unsigned Integers

The **unsigned integers** represent the set of counting (natural) numbers, starting at 0. In the **decimal system** (base-10), the unsigned integers are encoded using a sequence of decimal digits (0–9).

The decimal system is a **positional numeral system**, where the contribution of each digit is determined by its position. For example,

$$\begin{array}{rcl}
 278_{10} & = 2 \times 10^2 & + 7 \times 10^1 & + 8 \times 10^0 \\
 & = 2 \times 100 & + 7 \times 10 & + 8 \times 1 \\
 & = 200 & + 70 & + 8
 \end{array}$$

In the **binary system** (base-2) the unsigned integers are encoded using a sequence of binary digits (0–1) in the same manner. For example,

$$\begin{array}{rclclcl}
 10101_2 & = 1 \times 2^4 & + 0 \times 2^3 & + 1 \times 2^2 & + 0 \times 2^1 & + 1 \times 2^0 \\
 & = 1 \times 16 & + 0 \times 8 & + 1 \times 4 & + 0 \times 2 & + 1 \times 1 \\
 & = 16 & + 0 & + 4 & + 0 & + 1 \\
 & = 21_{10}
 \end{array}$$

The range of values an n -bit binary number can hold when encoding an unsigned integer is 0 to $2^n - 1$.

No. of Bits	Range
8	0–255
16	0–65 535
32	0–4 294 967 295
64	0–18 446 744 073 709 551 615

Table 1: Range of available values in binary representations.

2.4 Signed Integers

Signed integers are used to represent integers that can be positive or negative. The following representations allow us to encode negative integers using a sequence of binary bits:

- Sign-magnitude
- One's complement
- Two's complement (most common)

2.4.1 Sign-Magnitude

In sign-magnitude representation, the most significant bit encodes the sign of the integer. In an 8-bit sequence, the remaining 7-bits are used to encode the value of the bit.

- If the sign bit is 0, the remaining bits represent a positive value,
- If the sign bit is 1, the remaining bits represent a negative value.

As the sign bit consumes one bit from the sequence, the range of values that can be represented by an n -bit sign-magnitude encoded bit sequence is:

$$-(2^{n-1} - 1) \text{ to } 2^{n-1} - 1$$

For 8-bit sequences, this range is: -127 to 127 .

However, there are some issues with this representation.

1. There are two ways to represent zero: $0b10000000 = 0$, or $0b00000000 = -0$.
2. Arithmetic and comparison requires inspecting the sign bit
3. The range is reduced by 1 (due to the redundant zero representation)

2.4.2 One's Complement

In one's complement representation, a negative number is represented by inverting the bits of a positive number (i.e., $0 \rightarrow 1$ and $1 \rightarrow 0$).

The range of values are still the same:

$$-(2^{n-1} - 1) \text{ to } 2^{n-1} - 1$$

however, this representation tackles the second problem in the previous representation as addition is performed via standard binary addition with *end-around carry* (carry bit is added onto result).

$$a - b = a + (\sim b) + C.$$

2.4.3 Two's Complement

In two's complement representation, the most significant bit encodes a negative weighting of -2^{n-1} . For example, in 8-bit sequences, index-7 represents a value of -128 .

The two's complement is calculated by adding 1 to the one's complement.

The range of values are:

$$-2^{n-1} \text{ to } 2^{n-1} - 1$$

This representation is more efficient than the previous because 0 has a single representation and subtraction is performed by adding the two's complement of the subtrahend.

$$a - b = a + (\sim b + 1).$$

2.5 Logical Operators

2.5.1 Boolean Functions

A Boolean function is a function whose arguments and results assume values from a two-element set, (usually $\{0, 1\}$ or $\{\text{false}, \text{true}\}$).

These functions are also referred to as *logical functions* when they operate on bits. The most common logical functions available to microprocessors and most programming languages are:

- Negation: **NOT**
- Conjunction: **AND**
- Disjunction: **OR**
- Exclusive disjunction: **XOR**

By convention, we map a bit value of 0 to **false**, and a bit value of 1 to **true**.

2.5.2 Negation

NOT is a unary operator that is used to **invert** a bit. It is typically expressed as:

- **NOT** a
- $\sim a$
- \bar{a}

Truth table:

a	NOT a
0	1
1	0

2.5.3 Conjunction

AND is a binary operator whose output is true if **both** inputs are **true**. It is typically expressed as:

- $a \text{ AND } b$
- $a \& b$
- $a \cdot b$
- $a \wedge b$

Truth table:

a	b	$a \text{ AND } b$
0	0	0
0	1	0
1	0	0
1	1	1

2.5.4 Disjunction

OR is a binary operator whose output is true if **either** input is **true**. It is typically expressed as:

- $a \text{ OR } b$
- $a | b$
- $a + b$
- $a \vee b$

Truth table:

a	b	$a \text{ OR } b$
0	0	0
0	1	1
1	0	1
1	1	1

2.5.5 Exclusive Disjunction

XOR (Exclusive **OR**) is a binary operator whose output is true if **only one** input is **true**. It is typically expressed as:

- $a \text{ XOR } b$
- $a \sim b$
- $a \oplus b$

Truth table:

a	b	$a \text{ XOR } b$
0	0	0
0	1	1
1	0	1
1	1	0

2.5.6 Bitwise Operations

When applying logical operators to a sequence of bits, the operation is performed in a **bitwise** manner. The result of each operation is stored in the corresponding bit index also.

2.6 Bit Manipulation

Often we need to modify individual bits within a byte, **without** modifying other bits. This is accomplished by performing a bitwise operation on the byte using a **bit mask** or **bit field**.

These operations can:

- **Set** specific bits (change value to 1)
- **Clear** specific bits (change value to 0)
- **Toggle** specific bits (change values from 0 \rightarrow 1, or 1 \rightarrow 0)

2.6.1 Setting Bits

To **set** a bit, we take the bitwise **OR** of the byte, with a bit mask that has a **1** in each position where the bit should be set.

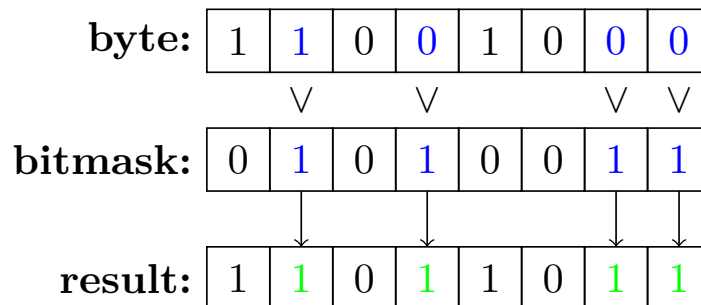


Figure 1: Setting bits using the logical or.

2.6.2 Clearing Bits

To **clear** a bit, we take the bitwise **AND** of the byte, with a bit mask that has a **0** in each position where the bit should be cleared.

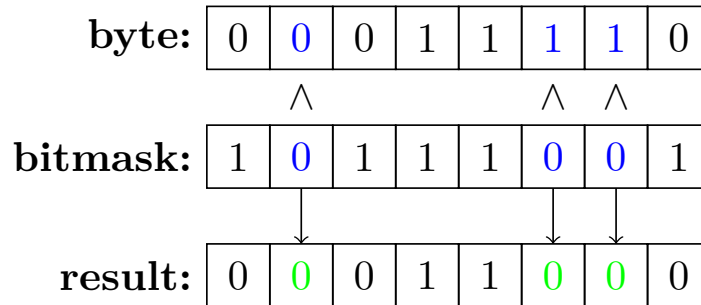


Figure 2: Clearing bits using the logical and.

2.6.3 Toggling Bits

To **toggle** a bit, we take the bitwise **XOR** of the byte, with a bit mask that has a **1** in each position where the bit should be toggled.

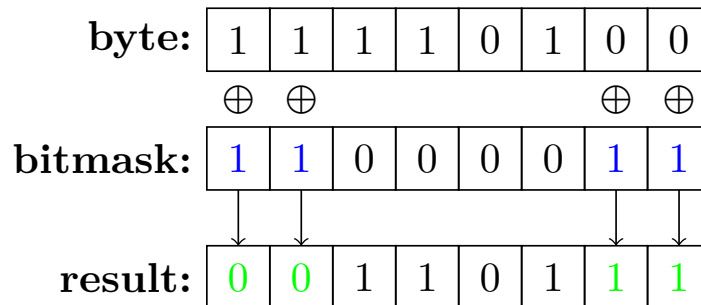


Figure 3: Toggling bits using the logical exclusive or.

Other bitwise operations act on the entire byte.

- One's complement (bitwise **NOT**)
- Two's complement (bitwise **NOT** + 1)
- Shifts
 - Logical
 - Arithmetic (for signed integers)
- Rotations

2.6.4 One's Complement

The one's complement of a byte inverts every bit in the operand. This is done by taking the bitwise **NOT** of the byte.

Similarly, we can subtract the byte from 0xFF to get the one's complement.

2.6.5 Two's Complement

The two's complement of a byte is the one's complement of the byte plus one. Therefore, we can apply take the bitwise **NOT** of the byte, and then add one to it.

2.6.6 Shifts

Shifts are used to move bits within a byte. In many programming languages this is represented by two greater than >> or two less than << characters.

$$a \gg s$$

shifts the bits in a by s places to the right while adding 0's to the MSB.

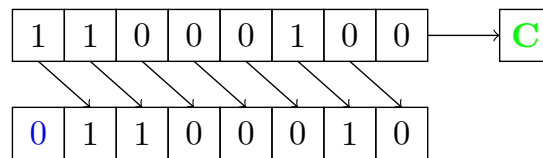


Figure 4: Right shift using **lsr** in AVR Assembly.

Similarly

$$a \ll s$$

shifts the bits in a by s places to the left while adding 0's to the LSB.

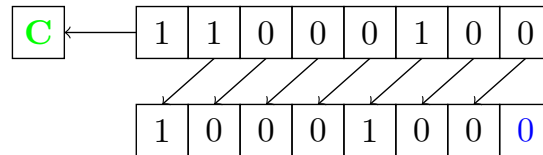


Figure 5: Left shift using **lsl** in AVR Assembly.

When using signed integers, the arithmetic shift is used to preserve the value of the sign bit when shifting.

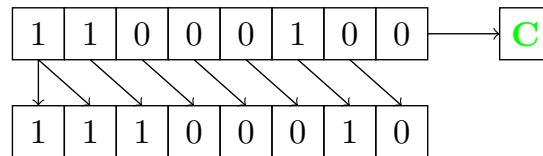


Figure 6: Arithmetic right shift using **asr** in AVR Assembly.

Left shifts are used to multiply numbers by 2, whereas right shifts are used to divide numbers by 2 (with truncation).

2.6.7 Rotations

Rotations are used to shift bits with a carry from the previous instruction.

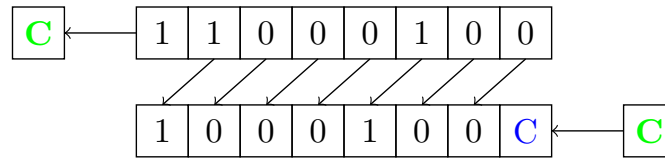


Figure 7: Rotate left using `rol` in AVR Assembly.

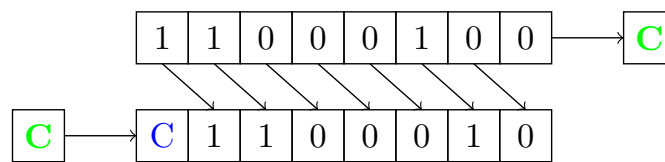


Figure 8: Rotate right using `ror` in AVR Assembly.

Here the blue bit is carried from the previous instruction, and the carry bit is updated to the value of the bit that was shifted out. Rotations are used to perform multi-byte shifts and arithmetic operations.

2.7 Arithmetic Operations

2.7.1 Addition

Addition is performed using the same process as decimal addition except we only use two digits, 0 and 1.

1. $0b0 + 0b0 = 0b0$
2. $0b0 + 0b1 = 0b1$
3. $0b1 + 0b1 = 0b10$

When adding two 1's, we carry the result into the next bit position as we would with a 10 in decimal addition. In AVR Assembly, we can use the `add` instruction to add two bytes. The following example adds two bytes.

```
; Accumulator
ldi r16, 0

; First number
ldi r17, 29
add r16, r17 ; R16 <- R16 + R17 = 0 + 29 = 29

; Second number
```

```
ldi r17, 118
add r16, r17 ; R16 <- R16 + R17 = 29 + 118 = 147
```

Below is a graphical illustration of the above code.

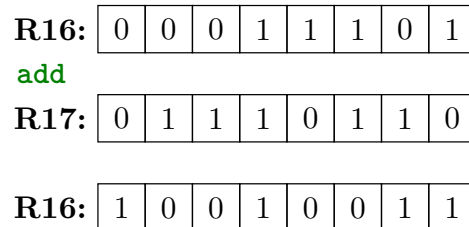


Figure 9: Overflow addition using **add** in AVR Assembly.

2.7.2 Overflows

When the sum of two 8-bit numbers is greater than 8-bit (255), an **overflow** occurs. Here we must utilise a second register to store the high byte so that the result is represented as a 16-bit number. To avoid loss of information, a **carry bit** is used to indicate when an overflow has occurred. This carry bit can be added to the high byte in the event that an overflow occurs. This is because the carry bit is 0 when the sum is less than 256, and 1 when the sum is greater than 255.

The following example shows how to use the **adc** instruction to carry the carry bit when an overflow occurs.

```
; Low byte
ldi r30, 0
; High byte
ldi r31, 0

; Empty byte for adding carry bit
ldi r29, 0

; First number
ldi r16, 0b11111111
; Add to low byte
add r30, r16 ; R30 <- R30 + R16 = 0 + 255 = 255, C <- 0
; Add to high byte
adc r31, r29 ; R31 <- R31 + R29 + C = 0 + 0 + 0 = 0

; Second number
ldi r16, 0b00000001
; Add to low byte
add r30, r16 ; R30 <- R30 + R16 = 255 + 1 = 0, C <- 1
; Add to high byte
adc r31, r29 ; R31 <- R31 + R29 + C = 0 + 0 + 1 = 1
```

Therefore the final result is: $R31:R30 = 0b00000001:0b00000001 = 256$. Below is a graphical representation of the above code.

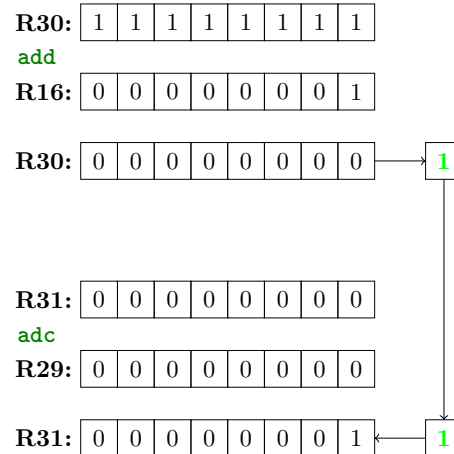


Figure 10: Overflow addition using `adc` in AVR Assembly.

2.7.3 Subtraction

Subtraction is performed using the same process as binary addition, with the subtrahend in two's complement form. In the case of overflows, the carry bit is discarded.

2.7.4 Multiplication

Multiplication is understood as the sum of a set of partial products, similar to the process used in decimal multiplication. Here each digit of the multiplier is multiplied to the multiplicand and each partial product is added to the result.

Given an m -bit and an n -bit number, the product is at most $(m + n)$ -bits wide.

$$\begin{aligned}
 13 \times 43 &= 00001101_2 \times 00101011_2 \\
 &= \begin{array}{r}
 00001101_2 \times 1_2 \\
 + 00001101_2 \times 10_2 \\
 + 00001101_2 \times 1000_2 \\
 + 00001101_2 \times 100000_2 \\
 = 00001101_2 \\
 + 00011010_2 \\
 + 01101000_2 \\
 + 110100000_2 \\
 = 1000101111
 \end{array}
 \end{aligned}$$

Using AVR assembly, we can use the `mul` instruction to perform multiplication.

```

; First number
ldi r16, 13
; Second number
ldi r17, 43

; Multiply
mul r16, r17 ; R1:R0 <- 0b00000010:0b00101111 = 2:47

```

The result is stored in the register pair R1:R0.

2.7.5 Division

Division, square roots and many other functions are very expensive to implement in hardware, and thus are typically not found in conventional ALUs, but rather implemented in software.

However, there are other techniques that can be used to implement division in hardware. By representing the divisor in reciprocal form, we can try to represent the number as the sum of powers of 2.

For example, the divisor 6.4 can be represented as:

$$\frac{1}{6.4} = \frac{10}{64} = 10 \times 2^{-6}$$

so that dividing an integer n by 6.4 is approximately equivalent to:

$$\frac{n}{6.4} \approx (n \times 10) \gg 6$$

When the divisor is not exactly representable as a power of 2 we can use fractional exponents to represent the divisor, however this requires a floating point system implementation which is not provided on the AVR.

3 Microcontroller Interfacing

3.1 Logic Levels

3.1.1 Discretisation

The process of discretisation translates a continuous signal into a discrete signal (bits). As an example, we can translate **voltage levels** on microcontroller pins into digital **logic levels**.

3.1.2 Logic Levels

For digital input/output (IO), conventionally:

- The voltage level of the positive power supply represents a **logical 1**, or the **high state**, and
- 0 V (ground) represents a **logical 0**, or the **low state**.

The QUTy is supplied 3.3 V so that when a digital output is high, the voltage present on the corresponding pin will be around 3.3 V. Because voltage is a continuous quantity, we must discretise the full range of voltages into logical levels using **thresholds**.

- A voltage **above** the input **high threshold** t_H is considered **high**.
- A voltage **below** the input **low threshold** t_L is considered **low**.

The interpretation of a voltage between these states is determined by **hysteresis**.

3.1.3 Hysteresis

Hysteresis refers to the property of a system whose state is **dependent** on its **history**. In electronic circuits, this avoids ambiguity in determining the state of an input as it switches between voltage levels.

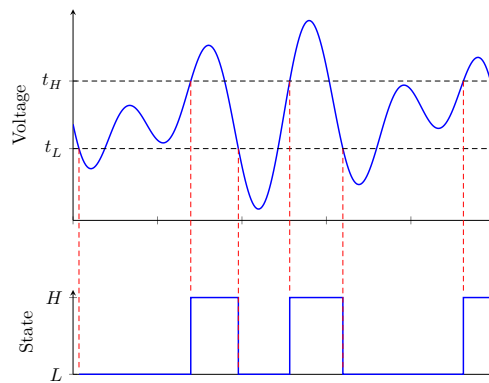


Figure 11: Example of hysteresis.

Given a transition:

- If an input is currently in the **low state**, it has not transitioned to the **high state** until the voltage crosses the **high input voltage** threshold.
- If an input is currently in the **high state**, it has not transitioned to the **low state** until the voltage crosses the **low input voltage** threshold.

It is therefore always preferable to drive a digital input to an unambiguous voltage level.

3.2 Electrical Quantities

3.2.1 Voltage

Voltage v measures the electrical *potential difference* between two points in a circuit, measured in **Volts (V)**.

- Voltage is measured across a circuit element, or between two points in a circuit, most commonly with respect to a 0 V reference (ground).
- It represents the **potential** of the electrical system to do **work**.

3.2.2 Current

Current i measures the *rate of flow of electrical charge* through a circuit, measured in **Amperes** (A).

- Current is measured through a circuit element.

3.2.3 Power

Power p is the rate of energy transferred per unit time, measured in **Watts** (W). Power can be determined through the equation

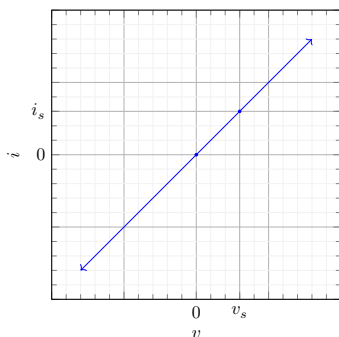
$$p = vi.$$

3.2.4 Resistance

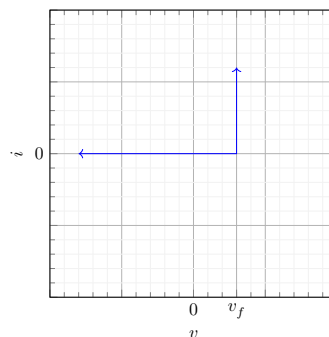
Resistance R is a property of a material to *resist the flow of current*, measured in **Ohms** (Ω). Ohm's law states that the voltage across a component is proportional to the current that flows through it:

$$v = iR.$$

Note that not all circuit elements are resistive (or Ohmic), such that they do not follow Ohm's law, this can be seen in diodes.



(a) VI curve for Ohmic components.



(b) VI curve for diodes.

Figure 12: Voltage-current characteristic curves for various components.

Although the wires used to connect a circuit are resistive, we usually assume that they are ideal, that is, they have zero resistance.

3.3 Electrical Components

3.3.1 Resistors

A **resistor** is a circuit element that is designed to have a specific resistance R .

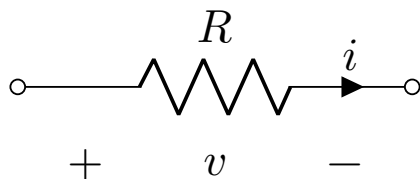


Figure 13: Resistor circuit symbol.

3.3.2 Switches

A **switch** is used to connect and disconnect different elements in a circuit. It can be **open** or **closed**.

- In the **open** state, the switch **will not conduct**² current
- In the **closed** state, the switch **will conduct** current

Switches can take a variety of forms:

- **Poles** — the number of circuits the switch can control.
- **Throw** — the number of output connections each pole can connect its input to.
- Momentary or toggle action
- Different form factors, e.g., push button, slide, toggle, etc.

Switches are typically for user input.

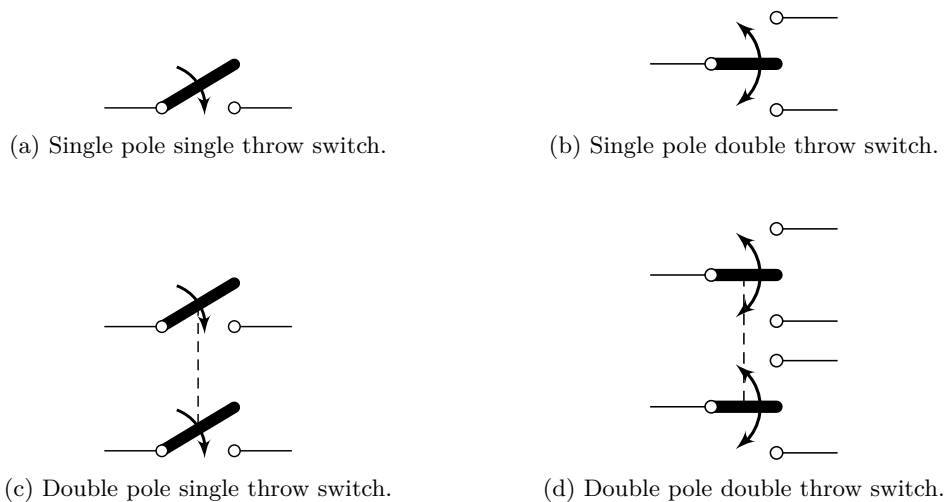


Figure 14: Various types of switches.

²Conductance is a measure of the ability for electric charge to flow in a certain path.

3.3.3 Diodes

A **diode** is a semiconductor device that conducts current in only one direction: from the **anode** to the **cathode**.

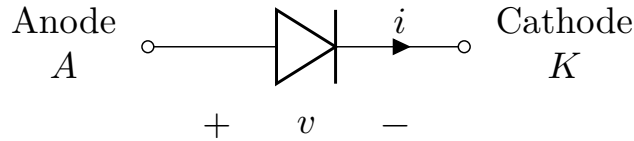


Figure 15: Diode symbol.

Diodes are a non-Ohmic device:

- When **forward biased**, a diode **does** conduct current, and the anode-cathode voltage is equal to the diodes **forward voltage**.
- When **reverse biased**, a diode **does not** conduct current, and the cathode-anode voltage is equal to the **applied voltage**.

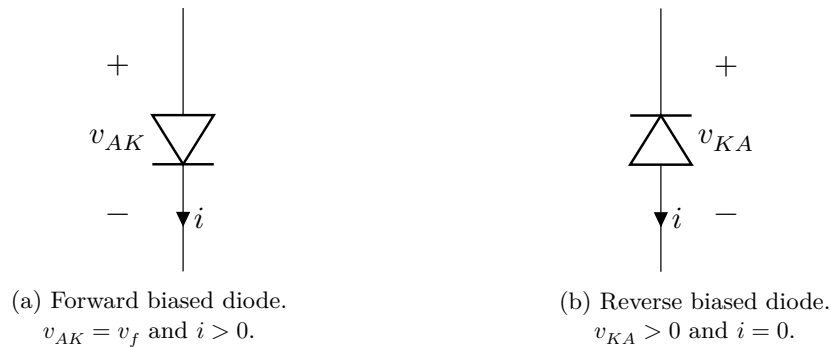


Figure 16: Diodes in forward and reverse bias.

A diode is only forward biased when the applied anode-cathode voltage **exceeds** the forward voltage v_f . A typical forward voltage v_f for a silicon diode is in the range 0.6 V to 0.7 V, whereas for Light Emitting Diodes (LEDs), v_f ranges between 2 V to 3 V.

3.3.4 Integrated Circuit

An **integrated circuit** (IC) is a set of electronic circuits (typically) implemented on a single piece of semiconductor material, usually silicon. ICs comprise of hundreds to many thousands of transistors, resistors and capacitors; all implemented on silicon. ICs are **packaged**, and connections to the internal circuitry are exposed via **pins**.

In general, the specific implementation of the IC is not important, but rather the **function of the device** and how it **interfaces** with the rest of the circuit. Hence ICs can be treated as a functional **black box**.

For digital ICs:

- **Input pins** are typically **high-impedance**, and they appear as an open circuit.
- **Output pins** are typically **low-impedance**, and will actively drive the voltage on a pin and any connected circuitry to a **high** or **low** state. They can also drive connected loads.

3.4 Digital Outputs

Digital output interfaces are designed to be able to drive connected circuitry to one of states, high, or low, however, the appropriate technique is **context specific**. When referring to digital outputs, we will refer to the states of a net. A **net** is defined as the common point of connection of multiple circuit components.

In this section we will consider:

- What kind of load the output drives?
- Could more than one device be attempting to actively drive the net to a specific logic level?

3.4.1 Push-Pull Outputs

A push-pull digital output is the most common form of output used in digital outputs. The **output driver** A *drives* the **output state** Y to:

- **HIGH** by connecting the output net to the supply voltage $+V$.
- **LOW** by connecting the output net to the ground voltage GND (0 V).

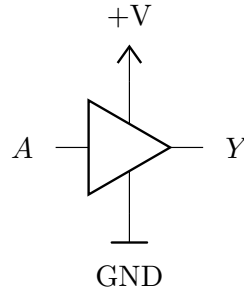


Figure 17: Push-pull output.

Hence the output state Y is determined by the logic level of the output driver A .

$$Y = A.$$

A	Y
LOW	LOW
HIGH	HIGH

Table 2: Truth table for a push-pull digital output.

The push-pull output Y can both source and sink current from the connected net.

3.4.2 High-Impedance Outputs

In many instances, a digital output is required to be placed in a high-impedance (HiZ) state. This is accomplished by using an **output enable** (OE) signal.

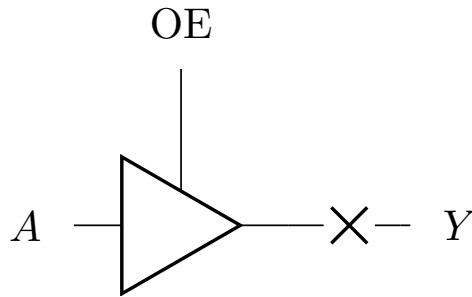


Figure 18: High-impedance output.

- When the OE signal is **HIGH**, the output state Y is determined by the output driver A .
- When the OE signal is **LOW**, the output state Y is in a **high-impedance** state.

A	OE	Y
LOW	LOW	HiZ
HIGH	LOW	HiZ
LOW	HIGH	LOW
HIGH	HIGH	HIGH

Table 3: Truth table for a push-pull digital output.

When the output is in **HiZ state**:

- The output is an effective **open circuit**, meaning it has **no effect** on the rest of the circuit.
- The voltage on the output net is determined by the **other circuitry** connected to the net.

HiZ outputs are typically used when multiple need to signal over the same wire(s).

3.4.3 Pull-up and Pull-down Resistors

When **no devices** are actively driving a net (e.g., all connected outputs are in the HiZ state), the state of the net is not well-defined. Hence we can use a **pull-up** or **pull-down** resistor to ensure that the state of the pin is always **well-defined**.

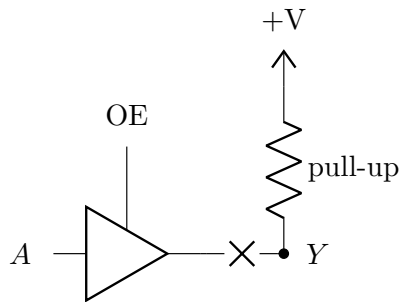


Figure 19: Pull-up resistor.

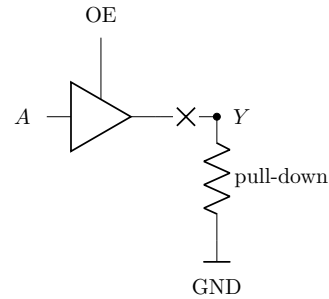


Figure 20: Pull-down resistor.

- When **no circuitry** is actively driving the net, the resistor will passively pull the voltage to either the voltage supply, or ground.
- When **another device** actively drives the net, the active device defines the voltage of the net. Hence the current from the resistor is simply sourced or sunk by the **active device**.

The resistors used as pull-up and pull-down resistors are typically in the $k\Omega$ range.

3.4.4 Open-Drain Outputs

Multiple push-pull outputs should never be connected to the same net as when one output is driven HIGH and another is driven LOW, an effective short circuit is created and one or more devices may be damaged. While push-pull outputs with an output enable may be used, the timing must be carefully managed.

Hence a more robust solution is to use open-drain outputs.

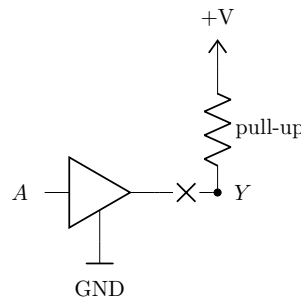


Figure 21: Open-drain output.

An open-drain output is either:

- In the **high-impedance** state, where the pull-up resistor is used to pull the net to the **high state** when the net is **not driven low**.
- **Connected to ground**, when the net is **driven low**.

3.5 Microcontroller Pins

Microcontrollers are interfaced via their exposed pins. These pins are the only means to access inputs and outputs, and they are used to interface with other electronic circuits in order to achieve a required functionality. Pins can be used for:

- General purpose input and output (GPIO) — pin represents a digital state
- Peripheral functions
- Other functions (power supply, reset input, clock input, etc.)

Pins are typically organised into groups of related IO banks, referred to as **ports** on the AVR microcontroller.

These ports and pins are assigned an alphanumeric identifier, (e.g., PB7 for pin 7 on port B).

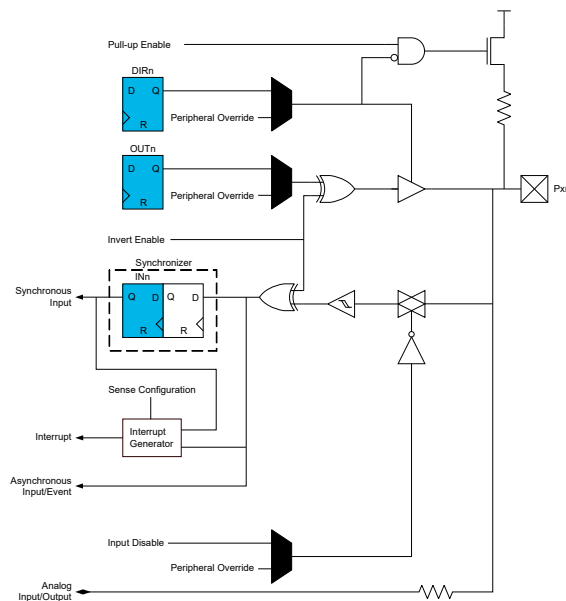


Figure 22: ATtiny1626 PORT block diagram.

To summarise this diagram:

- The data direction register (DIR) controls the push-pull output enable.
- The output driver register (OUT) drives the output state.
- The input register (IN) reads the output state.
- The internal pull-up register enabled through software.
- The physical voltage on the pin can be routed to an analogue to digital converter (ADC)
- Other peripheral functions can override port pin configurations and the output state.

3.5.1 Configuring an Output in Assembly

1. Place the port pin in a **safe initial state** by writing to the OUT register (HIGH or LOW depending on the context).
2. Configure the port pin as an output by **setting** the corresponding bits in the DIR register.
3. Set the desired pin state by writing to the OUT register.

```
; Load macros for easy access to port data space addresses.
#include <avr/io.h>

; Bitmask for pin 5
ldi r16, PIN5_bm

; Set initial safe state
sts PORTB_OUTCLR, r16 ; LOW if active HIGH
sts PORTB_OUTSET, r16 ; HIGH if active LOW

; Enable output
sts PORTB_DIRSET, r16 ; Enable output on PB5

; Set output state to desired value
sts PORTB_OUTSET, r16 ; Set state of PB5 to HIGH
```

3.5.2 Configuring an Input in Assembly

1. If required, enable the internal pull-up resistor by **setting** the PULLUPEN bit in the corresponding PINnCTRL register.
2. Read the IN register to get the current state of the pin.
3. Isolate the relevant pin using the AND operator.

```
; Load macros for easy access to port data space addresses.
#include <avr/io.h>

; Bitmask for pin 5
ldi r16, PIN5_bm

; Enable internal pull-up resistor if required
sts PORTB_PIN5CTRL, r16

; Read output state from data space
lds r17, PORTA_IN
; Read output state using virtual PORT
in r17, VPORTA_IN

; Isolate desired pin
andi r17, r16
```

3.5.3 Peripheral Multiplexing

Pins can be used to connect internal peripheral functions to external devices. As microcontrollers have more peripheral functions than available pins, peripheral functions are typically multiplexed onto pins.

Definition 3.1 (Multiplexing). Multiplexing is a method by which **multiple peripheral functions** are mapped to the **same pin**. In this scenario, only one function can be enabled at a time, and the pin cannot be used for GPIO.

- Peripheral functions can be mapped to different **sets of pins** to provide flexibility and to avoid clashes when multiple peripherals are used in an application.
- When enabled, peripheral functions **override** standard port functions.
- The **Port Multiplexer** (PORTMUX) is used to select which **pin set** should be used by a peripheral.
- Certain peripherals can have their inputs/outputs mapped to different **sets of pins** through the PORTMUX.

Note that we cannot re-map a single peripheral function to another pin, but must consider the entire set.

3.6 Interfacing to Simple IO

3.6.1 Driving LEDs

The **brightness** of an LED is proportional to the **current** passing through it. As LEDs are non-Ohmic, we cannot drive them directly with a voltage as this would result in an uncontrolled flow of current that may damage the LED or driver.

Instead, LEDs are paired with a **series resistor** to limit the flow of current. The appropriate current is dependent on the specific LED that is used and the capability of the driver device (microcontroller). A typical indicator LED requires a current of 1 mA to 2 mA.

3.6.2 Interfacing to LEDs

An LED can be driven in two different configurations from a microcontroller pin:

- **active high**; in which case the LED is **lit** when the pin is **HIGH**.
- **active low**; in which case the LED is **lit** when the pin is **LOW**.

Both of these configurations have their benefits, and the best configuration depends entirely on the context.

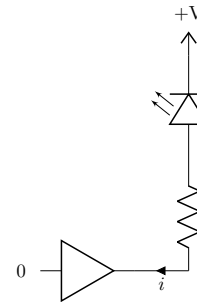
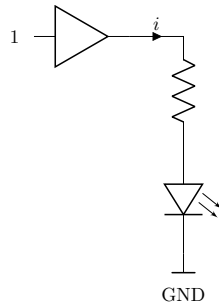


Figure 23: LED in an active high configuration. Figure 24: LED in an active low configuration.

On the QUTy, the LED display is driven in the **active low** configuration. This has a number of advantages:

- If the internal pull-up resistors are mistakenly enabled, no current will flow into the LEDs.
- The microcontroller pins can sink higher currents than they can source, allowing us to drive the display to a higher brightness.
- The display used on the QUTy has a common anode configuration, hence we must use an active low configuration to drive the display segments independently.

An LED is an example of a simple **digital output**, as we can map **logical states** to **LED states** (lit or unlit) for a digital output.

3.6.3 Switches as Digital Inputs

The state of a switch can be used to **set** the state of a pin. As the switch has two states (open or closed), these can be mapped directly to **logical states**.

This can be done by connecting the switch between the pin and voltage source representing one of the logic levels (ground or a positive supply).

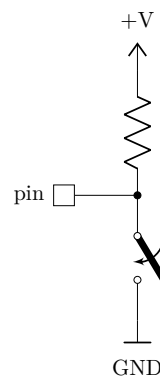
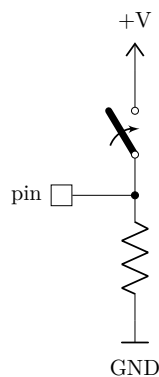


Figure 25: Switch in an active high configuration.

Figure 26: Switch in an active low configuration.

- When the switch is **open**, the pull-up/pull-down resistor is used to define the state of the switch.
- When the switch is **closed**, the state of the pin is defined by the voltage connected to via the switch.

3.6.4 Interfacing to Switches

As with LEDs, we can interface switches to microcontroller pins in two different configurations:

- **active high**; in which case the pin is **HIGH** when the switch is **closed**.
- **active low**; in which case the pin is **LOW** when the switch is **closed**.

An **active low** configuration is usually preferred as:

- it allows for the utilisation of an **internal pull-up resistor** that is commonly implemented in microcontrollers.
- It eliminates the risk of unsafe voltages being applied to the pin from the power supply in an active high configuration.
- It is easier to access a ground reference on a circuit board.

3.6.5 Interfacing to Integrated Circuits

For digital ICs,

- **Inputs** are typically **high impedance**
- **Outputs** are typically **push-pull**

This generally means that we can interface an IC by connecting its pins directly to the pins of a microcontroller.

- For **IC inputs**, the microcontroller pin is configured as an **output**, and the **microcontroller sets** the logic level of the net.
- For **IC outputs**, the microcontroller pin is configured as an **input**, and the **IC sets** the logic level of the net.

As microcontroller pins are typically configured as **inputs on reset**, a pull-up/pull-down resistor may be required if it is important for an IC input to have a **known state** prior to the configuration of the relevant microcontroller pins as outputs.