Microprocessors and Digital Systems

Semester 2, 2022

Dr Mark Broadmeadow

Tarang Janawalkar

Contents

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 AVRs.

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

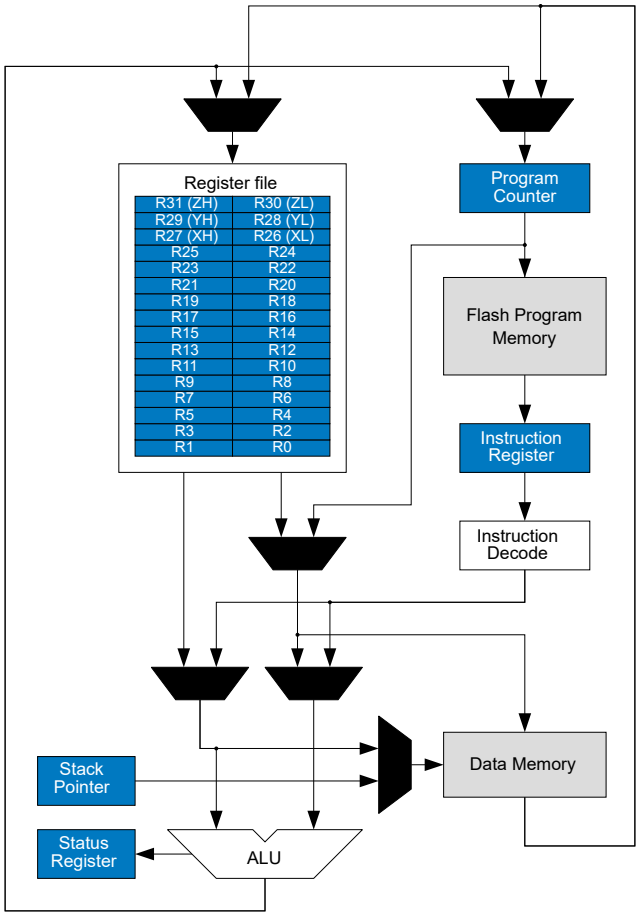
At the time of reset, PC = 0. The following steps are then performed:

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

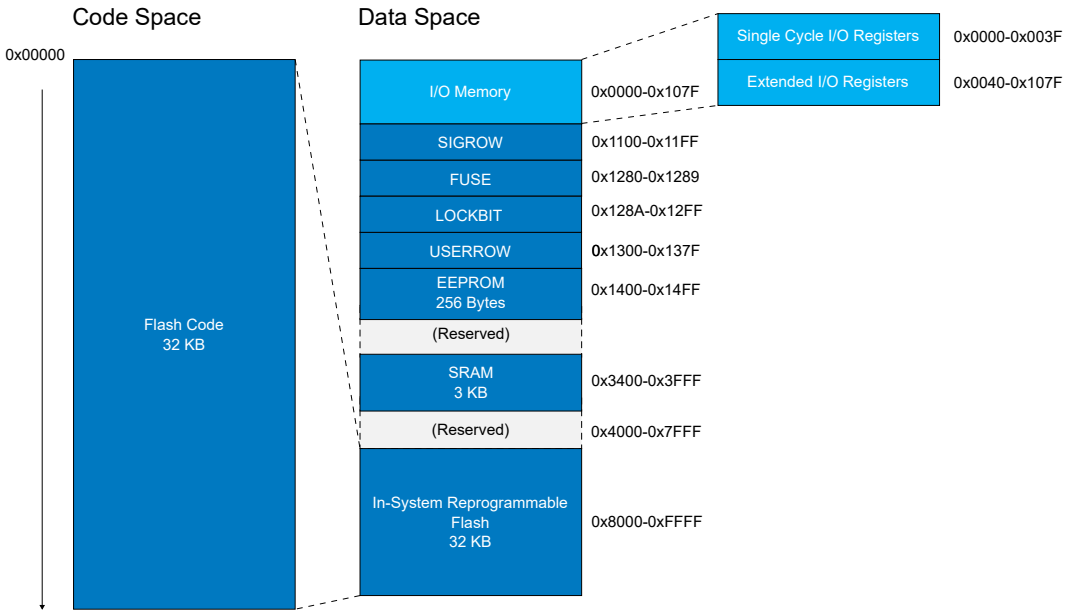
- The CPU understands and can execute a limited set of instructions — ~88 unique instructions for the ATtiny1626
- Instructions are encoded in programme memory as opcodes. Most instructions are two bytes long, but some instructions are four bytes long

- 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:

[1] The term *bit* comes from **b**inary dig**it**.

- 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,

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

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,

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

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:

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

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., $0 \to 1$ and $1 \to 0$).
The range of values are still the same:

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

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 `{false, 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
- $\sim a$
- $\overline{a}$

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 `AND` b
- a `&` b
- $a \cdot b$
- $a \wedge b$

Truth table:

a	b	a `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 `OR` b
- a `|` b
- $a + b$
- $a \vee b$

Truth table:

a	b	a `OR` b
0	0	0
0	1	0
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 `XOR` b
- a `^` b
- $a \oplus b$

Truth table:

a	b	a 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.

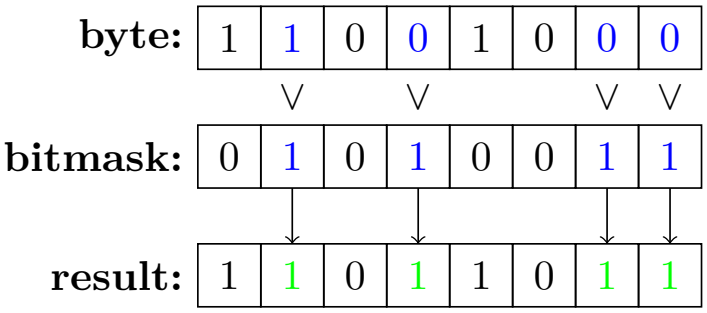


Figure 1: Setting bits using the logical or.

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.

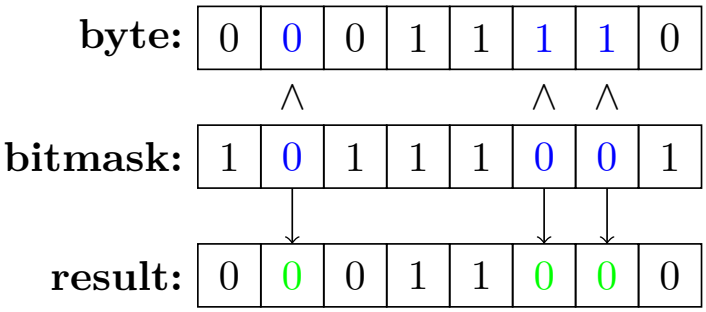


Figure 2: Clearing bits using the logical and.

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.

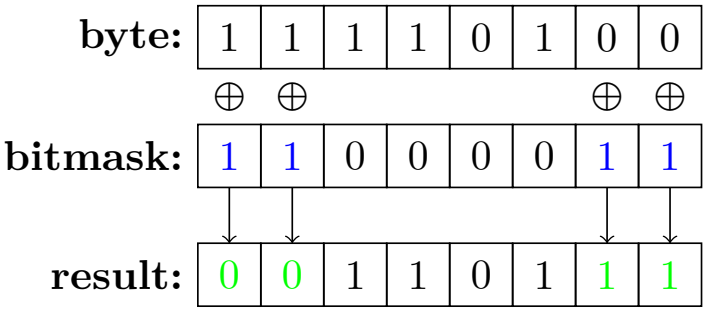


Figure 3: Toggling bits using the logical exclusive or.

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.

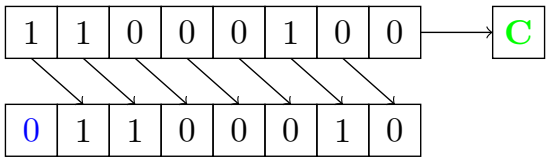


Figure 4: Right shift using `lsr` in AVR Assembly.

Similarly

$$a \ll s$$

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

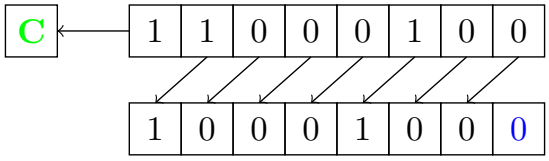


Figure 5: Left shift using `lsl` in AVR Assembly.

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

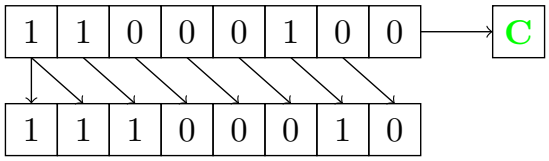


Figure 6: Arithmetic right shift using `asr` in AVR Assembly.

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

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

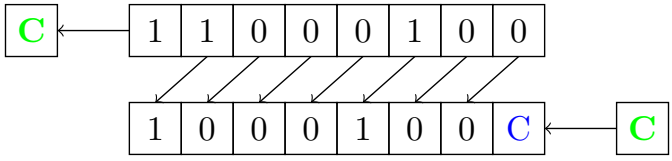


Figure 7: Rotate left using `rol` in AVR Assembly.

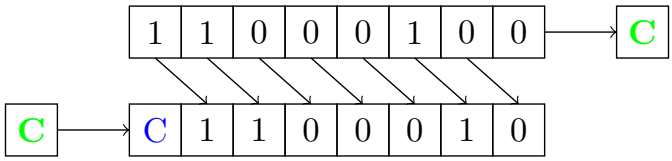


Figure 8: Rotate right using `ror` in AVR Assembly.

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, 17 ; R16 <- R16 + R17 = 0 + 29 = 29
```

```
; Second number
ldi r17, 118
add r16, 17 ; R16 <- R16 + R17 = 29 + 118 = 147
```

Below is a graphical illustration of the above code.

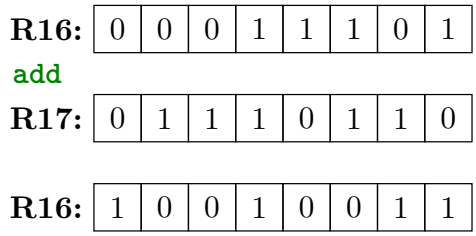


Figure 9: Overflow addition using `add` in AVR Assembly.

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.

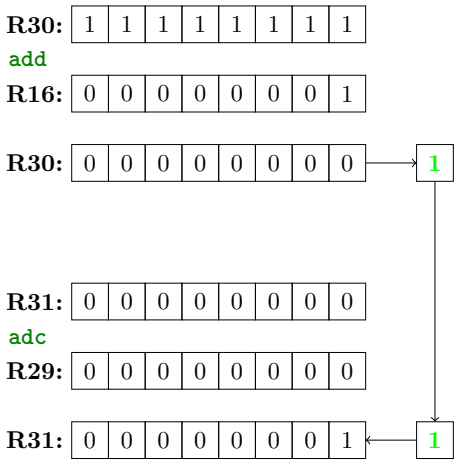


Figure 10: Overflow addition using `adc` in AVR Assembly.

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 multipicand 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.

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

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.