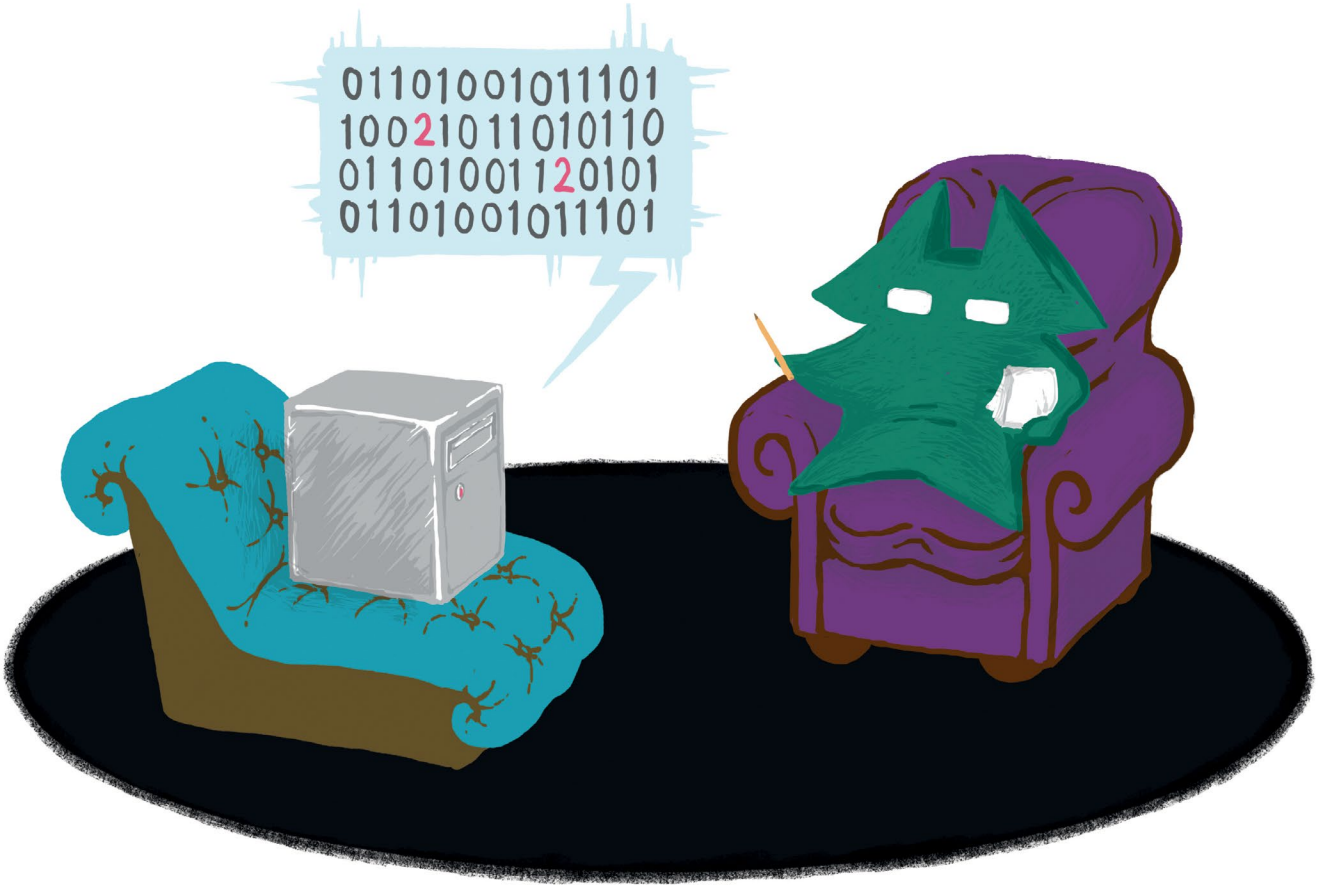


Machine code:
The gateway to the computer's soul

Computer hobbyists have always considered machine code to be something extraordinary – after all, it is the closest a programmer can get to the actual hardware. Although machine code is no longer the gateway to programming magic, understanding it will help in comprehending technology.

Story by Ville-Matias Heikkilä

Images by Mitol Meerna, Ville Matias Heikkilä, Visual6502.org, AMD

Earlier, knowledge of machine code was an almost required skill for game and demo programmers, for example, but nowadays it is mostly generated by high-level compilers. Being able to read machine code is still useful, nevertheless. You can evaluate the work of the compiler and examine and modify programs without their source code. Possessing this skill makes the computer and its software much more tangible. Machine code is still an important tool for people working with vintage hardware, microcontrollers and low-level security vulnerabilities.

Machines speak many languages

Not all machines can understand the same machine code. PC processors, for example, use x86 machine code and mobile devices use ARM machine code. A single machine code is also referred to as an instruction set or architecture.

For the sake of clarity, this article focuses on four instruction sets from the annals of computing history: 6502, x86, 68K and ARM. Since the design philosophies behind these instruction sets are also quite different, they will also provide an overall picture of the types of machine code that exist.

MOS Technology's 6502 is one of the most popular 8-bit processors. The 8-bit computers from Apple, Atari and Commodore and the Nintendo NES, for example, all use it or one of its clones. The 6502's traditional competitor was the Zilog Z80, based on the Intel 8080. AVR and PIC are newer 8-bit instruction sets that are mostly used in

6502
ADC #3
01101001 00000011
"Add this number to the accumulator with carry: 3!"

X86
ADD AX, 3
00000011 11000000 00000011
"Add this byte-wide number to the word-wide register AX*: 3!"
*EAX in 32- or 64-bit code

68K
ADDQ.L #3, D0
0101011001000000
"Add the small number 3 in full width to the data register 0!"

ARM
ADD R0, R0, #3
11100010100000000000000000000011
"Unconditionally add without flags!
Put the result in R0, the addends are R0 and the number 3 without extra zeros in the end."

AVR
ADIW R25:R24, 3
1001011000000011
"Add this number to the register pair R24-R25: 3!"

PDP-11
ADD #3, R0
0110010111000000 0000000000000011
"Full-width add the contents of the memory location pointed to by the program counter (3), autoincrementing the register, to the contents of the register R0!"

Instructions from different machine code dialects, broken down to bits.

The oldest parts of the register set for the current 64-bit x86 originate from the 1970s.

embedded systems.

The Intel x86 was made famous by the IBM PC compatibles. The original instruction set was 16-bit, but it has later been radically expanded and renewed – first to 32-bit for the 386 processor, then to 64-bit at the initiative of AMD. Despite the enhancements, the different historical sediments are still clearly visible in x86 machine code.

The Motorola MC68000 was used by most computers that competed with the IBM PC until the early 1990s: the Amiga, Atari ST and Macintosh as well as most UNIX workstations. It is based on the instruction sets of larger 1970s computers and is a pure CISC (Complex Instruction Set Computer) by design.

ARM is currently the most popular instruction set. It dominates the mobile platforms, in particular, but may even replace the x86. The instruction set was originally used on the Archimedes home computer, and it became popular since it offered a lot of power with a low amount of silicon. ARM is a RISC (Reduced Instruction Set Computer). Other RISCs include MIPS, SPARC, PowerPC and AVR, for example.

Following instructions

Machine code instructions are fairly dense strings of ones and zeros. The instruction presented on page 54 performs the same task in several different machine code variants. Each instruction adds the number 3 to one of the processor's internal registers, but the bit width varies, among other things: the 6502's `adc` uses 8-bit numbers, which means that the largest sum can amount to a few hundred, while the ARM can count into the billions with its 32-bit wide calculations. The number of bits in a processor or instruction set usually refers to the maximum bit width of basic calculations.

Strings of ones and zeros are difficult to read for humans. This is why people usually process machine code in symbolic form, known as assembly language. The assembly representation can also be used to guess what the instruction does even if the instruction set is not known; for example, `ad(d)` refers to addition. The same machine code may have several different assembly language syntaxes that are used by different assembly compilers or assemblers – such as the Intel and AT&T syntaxes for the x86.

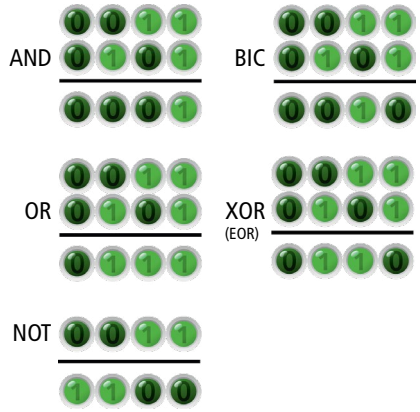
A machine code instruction usually consists of an opcode (operation code), the addressing mode and the operands. The opcode is the "verb" and it corresponds to the first word in an assembly statement, also known as a mnemonic; `add`, for example. The operands are the "nouns" that follow it: registers, numbers and memory addresses. Addressing modes can be compared to the forms of declension in human languages. They indicate how the operand part should be interpreted – whether it is a memory address or a number – and provide additional attributes; for example, the suffix `.b`, `.w` or `.l` on a 68K instruction indicates whether the operation is performed in 8, 16 or 32 bits.

Registers rotate data

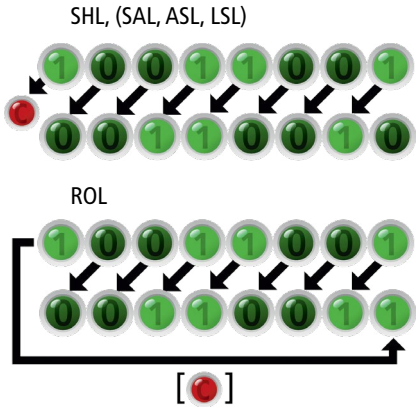
In most machine code dialects, the major part of the data processing occurs inside registers. They can be viewed as processor-internal fixed variables. The number of registers, their width and their manner of use differ substantially from one instruction set to another.

The 6502 has a very small register set and each register is tied to specific tasks. Most calculations will need to be performed in the accumulator register, A. The index registers X and Y are mostly suited for memory addressing and loop counting, which A cannot perform. In addition to these, the 6502 only has the stack pointer S, the status register P and the instruction pointer PC that indicates the memory address for the next instruction. PC is the only 16-bit register; the others are 8-bit. The limited register space is supplemented by the "zero page", the first 256 bytes of the memory, and many types of memory addressing can only be performed via the zero page.

The ARM and other RISCs, for their part, have a highly symmetrical and general-purpose register set. Theoretically, any register can be used for any purpose. The only exceptions are register R15, which is the instruction



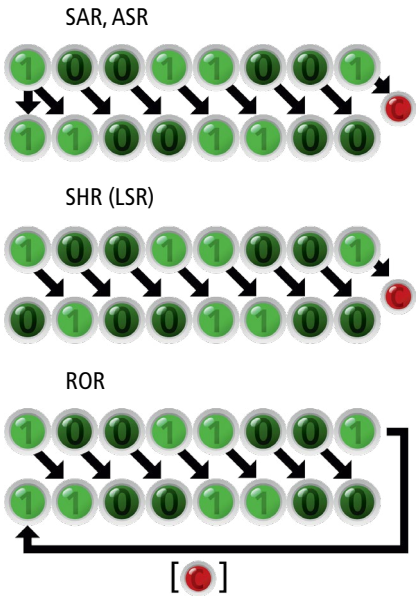
Bit operations from the instruction sets discussed in this article. BIC is used by ARM.



Operation of the bit shift instructions. Many instruction sets have different names for ROR and ROL that use the carry digit, such as RCR and RCL.

	Intel X86	68k	AT&T X86
Operand order	add destination,source	add.w source,destination	addw source,destination
Memory addressing	add ax,[1234]	add.w 1234,destination	addw 1234,%ax
Immediate	add ax,1234	add.w #1234,destination	addw $1234,%ax
Indexed address	[ebx+esi+8]	8(a0,d1.L)	8(%ebx,%esi)
Hexadecimal	1234h	$1234	0x1234
Location of the instruction	jmp $	jmp pc	jmp .
Data byte	db 123	ds.b 123	.byte 123

Assembly syntaxes are usually quite similar, but they may have some confusing differences. Here are a few examples.



8×	4×	2×	1×	Unsigned	Signed
0	0	0	0	0	+0
0	0	0	1	1	+1
0	0	1	0	2	+2
0	0	1	1	3	+3
0	1	0	0	4	+4
0	1	0	1	5	+5
0	1	1	0	6	+6
0	1	1	1	7	+7
1	0	0	0	8	-8
1	0	0	1	9	-7
1	0	1	0	10	-6
1	0	1	1	11	-5
1	1	0	0	12	-4
1	1	0	1	13	-3
1	1	1	0	14	-2
1	1	1	1	15	-1

Four-bit integers interpreted as unsigned and signed, using two's complement.

pointer, and a separate status register. The basic ARM has 16 32-bit registers, but most other RISCs have 32 or more basic registers.

The registers on the x86 were originally specialised; for example, only the registers BX, SI, DI and BP could be used for memory addressing. The 32-bit update removed some of these restrictions. Nevertheless, even the current 64-bit operation mode has some instructions that are bound to specific registers: for example, the single-byte command `stosb` saves the contents of the 8-bit AL (*accumulator low*) register to the memory location where the original DI (*destination index*) register's 64-bit extension RDI is pointing at.

The basic register set of the 68k is divided into eight data and address registers D0–D7 and A0–A7, of which A7 is used as a stack pointer. It also has a separate status register, CCR, and the instruction pointer, PC. The address registers were originally 24-bit, but they were expanded to 32 bits in the 68020. All registers can be used for calculations in a fairly general manner, but memory addressing must use the address registers.

Addressing modes modify the instructions

The simplest machine code instructions have no operands; this means that their operation is tied to specific registers. The instruction `stosb` on the x86 mentioned above is an example of this implicit form of addressing. Other examples include instructions for returning from a subroutine (`ret, rts`) and the instructions for setting and clearing flags (`sec, clc`).

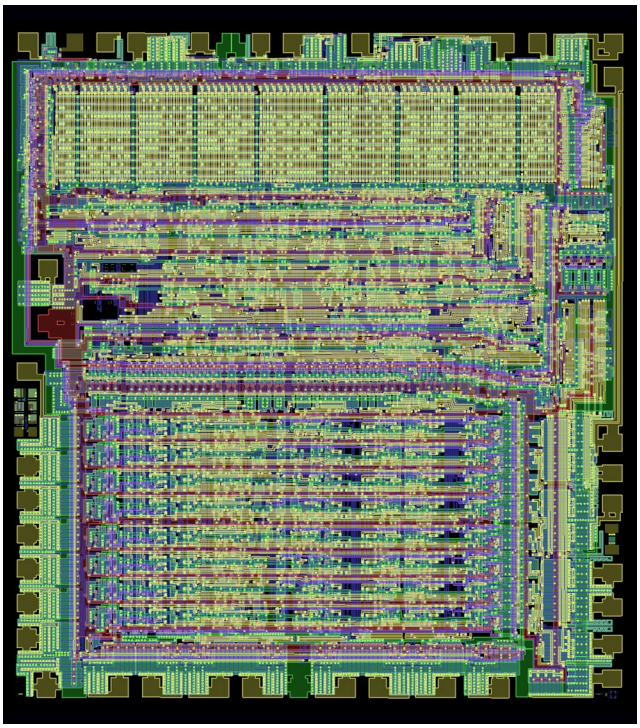
The typical number of operands in an instruction varies from one machine code to another. On the 6502, most instructions have one operand. This operand is usually a memory address, in which case the calculation occurs between the accumulator register and the memory location. The x86 and 68k have two operands: a source and

```
clc                 asl $FE
lda $FE             rol $FF
adc #$34            asl $FE
sta $FE             rol $FF
lda $FF             asl $FE
adc #$12            rol $FF
sta $FF
```

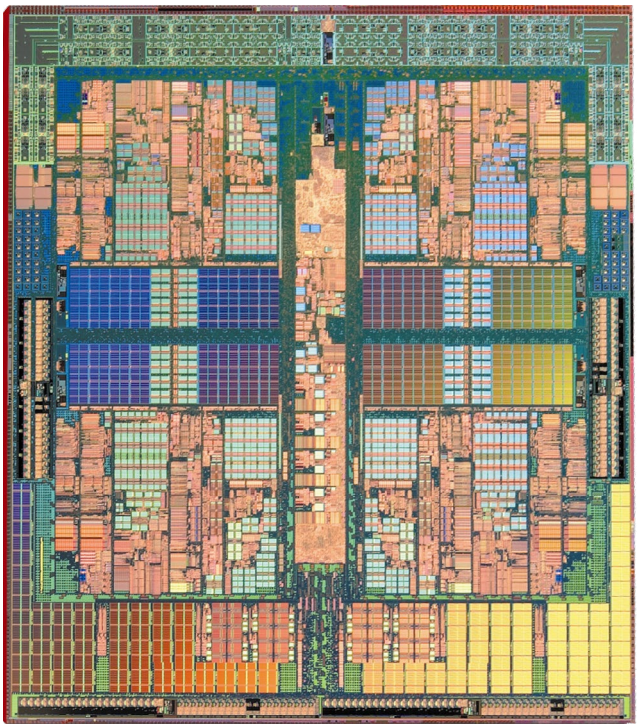
Handling 16-bit numbers with the 8-bit 6502. The example on the left adds the hexadecimal number $1234 to the value of the number saved at memory locations $FE and $FF, the one on the right multiplies it by eight by shifting the bits.

```
lp: cmp r0,r1
    subgt r0,r0,r1
    suble r1,r1,r0
    bne lp
```

A loop that calculates the largest common denominator on an ARM by using conditional execution. An Euclidean algorithm subtracts the smaller number from the larger one until the numbers are equal.



The internals of a 6502 processor. The lower half is dominated by an 8-band arithmetic and register unit, the top part has a microcode table that converts the instructions into execution steps. Between them you will find the rest of the operational logic, such as branch and flag handling.



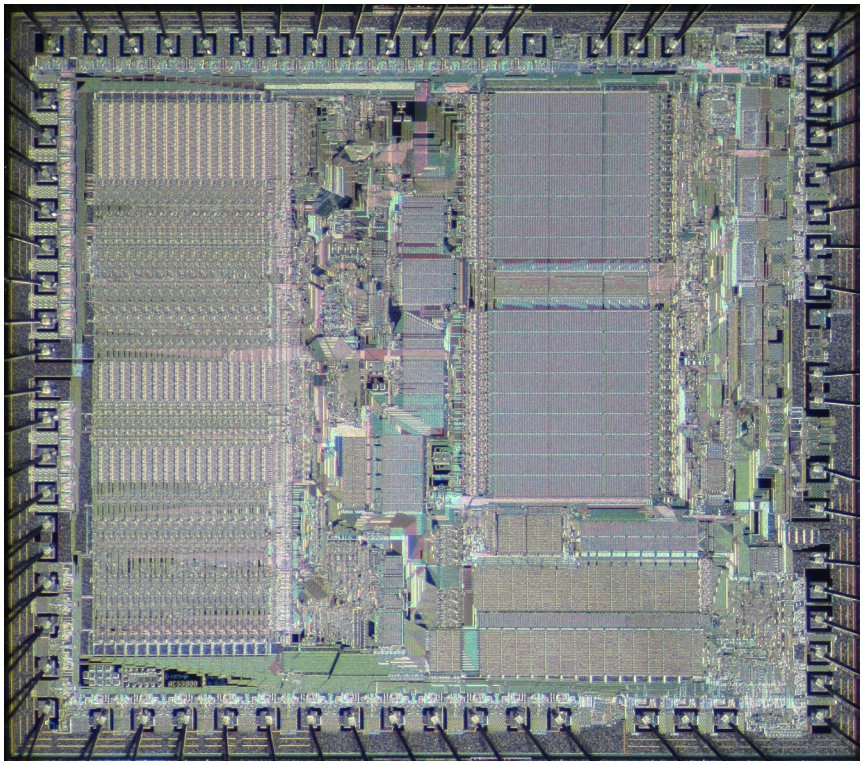
The internals of an AMD Phenom X4 processor. Most of the surface area of the four symmetrically positioned 64-bit cores is taken up by cache memory and instruction decoding and sequence logic.

destination operand for each instruction. A typical ARM instruction has three operands: two sources and one destination. Forth-style stack-based machine codes can be considered zero-operand variants.

For most processors, the main part of machine consists of operations between registers. However, *immediates* or different memory references can also be used as operands in addition to registers.

There are often limits to combining operands: on the x86, one of the operands must always be a register or an immediate; there is no direct command for "add value of memory location 2 to value of memory location 1". However, memory references can be very complex in accordance with the CISC philosophy. For example, the 32-bit x86 instruction `mov eax,[ebx+ecx*4+1256]` forms a memory address by adding together a constant and two registers, of which ECX has its bits shifted two steps to the left.

In ARM-type RISCs, most instructions can only receive registers or immediates as their operands. Memory handling must be arranged by means of dedicated load and store



The internals of a Motorola 68000. Can you find the arithmetic and register unit?

```
lp:  movem (a0)+,(d1-d7)          lp:  subcc r2,r2,#1
     movem (d1-d7),-(a1)               ldmia r0,(r3-r13)
     dbne d0,lp                        stmdb r1,(r3-r13)
                                       bne lp
```

A loop that copies the contents of a memory area in reverse order to another memory area by using the register set instructions. 68k on the left, ARM on the right.

instructions (`ld`, `st`, `mov`) that do not perform calculations.

Memory handling on the ARM and 68k is improved by addressing types where the contents of the register are incremented or decremented while the register is used for memory addressing. This is handy when scanning memory areas.

Instead of using direct addresses, it is often preferable to refer to memory locations by using the location of the instruction as a fixed point. The conditional jump instructions on the 6502 and x86 can be used to jump forward or backward by a maximum of 128 bytes; this means that the instruction only takes up two bytes. Program code that does not use direct memory addresses is called *position-independent*, since it can be executed as is from any location in memory.

Computers like to compute

Most processors use binary integers by default. The 6502, 68k and x86 also offer Binary Coded Decimals (BCD) where four bits correspond to each of the decimals 0–9. Floating point numbers, for their part, are processed with separate floating point units that have their own registers and instructions.

Negative integers are nearly always expressed as two's complements, where the sign is changed by flipping the bits around and adding one to the result. Therefore, a number that contains only ones has a value of -1, like a tape counter that goes from 000 to 999 when rewound. The same bit string can be interpreted as either signed or unsigned, and the differences become especially apparent during multiplication, division and comparison.

All machine codes offer addition and subtraction for integers (`add`, `sub`). The 8-bit machines usually lack multiplication and division (`mul`, `div`), which means that they must be implemented by means of subroutines or tables. RISCs usually only contain multiplication.

Bit operations include both logical bit operations (`and`, `or`, `eor`/`xor`) and *bit shifts* that come in many forms. The functionality of the bit operations is presented in the enclosed diagrams. The difference between an "arithmetic" and "logical" bit shifts is that in an arithmetic shift, the number is assumed to be signed and its top bit is kept in place.

EX, EXG, XCHG	exchange	Exchange the contents of the registers.
LD	load	Load from memory.
MOV, MOVE	move	Copy data from register or memory to register or memory.
POP, PL	pop, pull	Pick the topmost value in the stack.
PUSH, PH	push	Add to the top of the stack.
ST	store	Store in memory

Data transfer.

ADC, ADDX	add with carry/extend	Add with carry digit.
ADD	add	Add.
DEC	decrement	Decrement by one.
DIV	divide	Divide.
INC	increment	Increment by one.
MUL	multiply	Multiply.
NEG	negate	Switch the sign.
SBB, SBC, SUBX	subtract with borrow/carry/extend	Subtract with carry digit.
SUB	subtract	Subtract.

Basic arithmetic operations.

AND	and	AND operation by bit.
ASL, SAL	arithmetic shift left	Shift bits to the left.
ASR, LSR, SHR	[arithmetic/logical] shift right	Shift bits to the right, extending the topmost bit.
EOR, XOR	exclusive or	Exclusive OR by bit.
LSL, SHL	[logical] shift left	Shift bits to the right, extending with zero.
NOT	not	Reverse the bits.
OR	or	OR operation by bit.
ROL, RL, RCL	rotate [with carry] left	Rotate bits counterclockwise [through the C flag].
ROR, RR, RCR	rotate [with carry] right	Rotate bits clockwise [through the C flag].

Bit operations.

Sometimes, the result of the operation will not fit in the destination register. For example, the sum of two 8-bit numbers has 9 bits. The topmost bit is usually recorded in the *carry flag* (C). The carry digit is used for chaining the calculations: the instructions `adc`/`addx` and `sbc`/`sbb`/`subx` are additions and substractions that consider the carry digit from the previous calculation.

What ifs

A conditional jump is the typical machine code equivalent to the `if` clause in higher-level languages. For example, the instruction `beq`, `je` or `jz` will jump to the memory address provided as the operand if the result of the previous arithmetic operation was zero. Before the jump, it is common to use a comparison instruction, `cmp`/`cp`, which performs the subtraction without saving the result. The jump instructions are usually named from the point of view of comparison; if the result of the subtraction is zero, the numbers are equal (`e`/`eq`).

The information concerning the result is usually saved in status register bits that are known as flags. The carry flag mentioned above is one of them. Conditional jump instructions examine the status of the flags and jump if a condition is met. Typical flags include:

- The zero flag (Z) that indicates whether the result of a calculation is zero.
- The sign flag (S) or negative flag (N) that corresponds to the top bit of a result that fits in a register. For negative numbers, this is 1.
- The carry flag (C) that corresponds to the bit carried over from an arithmetic operation.
- The overflow flag (O or V) is set when the extension of the result does not fit in the carry flag.

On the 6502, x86 and 68k, each calculation instruction affects the flags.

BIT, BT, BTST, TEST	bit test	Test individual bits (AND without saving the result).
CLf	clear flag	Clear a flag (e.g. C).
CMP, CP	compare	Compare (subtract without saving the result).
Scc, SETcc	set on condition	Set the value of the register to the truth value (e.g. NE).
SEf, STf	set flag	Set a flag (e.g. C).

Comparison and flags.

Bcc, Jcc	branch/jump on condition	Jump if the condition (e.g. NE) is met.
BL, BAL	branch and link	Branch to subroutine, place return address in the link register.
DBcc, LOOP	decrement and branch, loop	Decrement the value of the register and branch if the condition is met.
JMP, JP, B, BRA	jump/branch	Branch to memory address.
JSR, JR, BSR	jump/branch to subroutine	Branch to subroutine, place return address in the stack.
RET, RTS	return from subroutine	Return from the subroutine to the main routine.
SWI, INT, TRAP, BRK, SYSCALL	software interrupt, trap, break, system call	Perform a software interrupt.

Jump instructions.

HLT	halt	Halt the processor (wait for interrupt).
NOP	no operation	Do nothing.

Other instructions.

CC, NC	no/clear carry	Carry digit = 0
CS, C	carry set	Carry digit = 1
EQ, E, Z	equal/zero	Numbers equal (zero flag set)
GT, G	greater [than]	First value > second value
LT, L	less [than]	First value < second value
NE, NZ	not equal/zero	Numbers not equal (zero flag cleared)
NS, PL	no sign, plus	Result not negative (sign = 0)
S, MI	sign, minus	Result negative (sign = 1)
VC, NO	no/clear overlow	Overflow flag cleared.
VS, O	overflow set	Overflow flag set.

Conditions (as part of instructions).

On the ARM, the effect on flags is expressed for each instruction with the suffix `cc` (*condition code*). ARM does not always require conditional jumps, since the execution of any instruction can be made conditional. For example, the instruction `addeq` operates like `add`, but it is only executed if the zero flag is set.

Stacking up other stuff

A normal unconditional jump instruction may be called `jmp`, `bra` or `b`, while a subroutine jump is called `jsr`, `bsr`, `call` or `bl`. Subroutine calls store the value of the instruction pointer. This allows the execution to resume from the place where the subroutine was called. The return instruction is typically called `ret` or `rts`.

Older instruction sets typically save the return address in a memory area known as the stack. Instead, RISCs use a register that the subroutine stacks by itself if it aims to call other subroutines. The linking jump instruction for ARM is called `bl` (*branch and link*). The link register is usually R14 and the instruction pointer is R15, so the instruction for returning from the subroutine is `mov r15,r14`.

The stack stores other things in addition to return addresses. Since the subroutines use the same registers as the main program, the values of the register values will commonly need to be stored in the stack. Stack space can also be reserved for local variables that do not fit inside the registers. The x86 and 6502 have push and pop/pull instructions that are bound to the stack pointer, whereas the ARM and 68k use regular memory handling instructions for stack handling. The ARM and 68k also have instructions for saving or loading a desired register set at once.

Calling conventions are used to keep larger programs in check. They define how parameters and return values are relayed between the main program and subroutine, and which registers the subroutine is allowed to modify.

The world is memory

From the processor's point of view, the entire outside world consists of memory. Memory is usually divided into memory cells that are the size of an 8-bit byte and have their own numeric address.

There are two main methods for storing numbers that consist of several bytes. The 68k uses *big-endian* byte order, which means that the most significant bits are stored in the first byte. The 6502 and x86 use *little-endian* byte order and store the lower bits first. ARM can operate with either byte order; little-endian is more common, however.

In simpler devices, the physical RAM, ROM and control chips have fixed areas within the memory space. In a VIC-20 program, for example, writing to address `$900F` will always affect the colour register of the video chip. More complex hardware allows for changing the memory structure visible to the program.

If the machine has more memory than the address space can hold, such as over 64 kilobytes in a 6502 based machine, *banking* is required. Banking means selecting which parts of the total memory are visible in specific areas of the memory space. Modern operating systems modify the visible structure of the memory in order to prevent different processes from accessing unauthorised memory areas. At the same time, the code is prevented from modifying the state of the processor by switching from *supervisor mode* to *user mode* during its execution.

Virtual memory means all memory visible to the program needs to correspond to physical memory. If the address space is large enough, the program may request the operating system to extend the virtual memory to the entire contents of the hard drive, for example. When the program tries to access a memory location that is not in physical memory, this causes an exception that the operating system handles by loading the desired location from the hard drive into physical memory. From the point of view of the program, the entire contents of the drive are permanently accessible in memory.

```
        !to "skrolli.prg",cbm
        *=$0801     ; Start address of the program.

; Obligatory BASIC portion: 10 SYS2061 + final zeroes:
!byte $0b,$08,$0a,$00,$9e,$32,$30,$36,$31,0,0,0

        ldx #0      ; Set counter (X) to zero.

loop0   txa         ; Copy X to A in order to
        and #15     ; calculate X AND 15.
        tay         ; Result to Y; then fetch
        lda msg,y   ; a byte from address msg+Y.

        sta $0400,x ; Copy it to the each
        sta $0500,x ; 256-byte block of the
        sta $0600,x ; screen memory at the
        sta $0700,x ; offset X.

        inx         ; Increment X.
        bne loop0   ; Repeat until rolls back to zero.

        rts         ; Return to BASIC interpreter.

msg     !scr "read skrolli!!! "
```

A 6502 example for the Commodore 64. The PRG file generated by the ACME cross-assembler can be started directly in the VICE emulator, for example.

```
        bits 16         ; Nasm to 16-bit mode.
        org 0x100       ; COM programs start at 0x100.

        mov ax,0xb800   ; Start address of screen memory
        mov es,ax       ; .. to the segment register ES.
        xor di,di       ; Set Destination Index to zero.

        mov ah,14       ; High byte of AX is the color.

loop1   mov si,msg      ; Source Index to start of text.
        mov cx,16       ; Set loop counter to 16.

loop0   lodsb           ; AL <- [DS*16+SI], SI incs.
        stosw           ; AH*256+AL -> [ES*16+DI], DI +2.
        loop loop0      ; CX decs, repeat until 0.

        cmp di,80*25*2  ; Gone through the whole screen?
        jne loop1       ; If not, continue the loop.

        ret             ; Return to the command shell.

msg     db "Read Skrolli!!! "
```

A 16-bit x86 example for MS-DOS. NASM will compile the code and create an executable COM file.

Memory speed is not a bottleneck for 1970s processors. On the 6502, for example, memory-resident tables and unrolled loops should be used in code that is critical in terms of speed, if you can fit them in memory. For modern processors, however, a calculation needs to be really complex in order to benefit from a pre-calculated table. Internal caches and smart pipelines mean that unrolling loops is more likely to slow down the code than make it faster.

Controlling devices

Computer equipment includes auxiliary chips that have their own control registers. On the 6502, 68k and ARM, these registers are visible in the memory space. However, the x86 uses separate I/O ports that are handled with the `in` and `out` instructions.

Interruptions were designed to relieve the processor from the burden of continuously polling the states of the different devices. A device can send out an *interrupt request* (IRQ) that causes the processor to stop what it is doing and move to the interrupt handling routine. In order to manage routine tasks, most operating systems execute a timer interrupt a few dozen times per second.

In its simplest form, an interrupt is no different than a subroutine call. The start address for the subroutine is fetched from a branch table according to the interrupt type and number. In modern operating systems, the interrupt also switches the processor into supervisor mode. Only an operating system that is running in supervisor mode can access external hardware, and applications perform a *non-maskable interrupt* (NMI) when they require assistance from the operating system.

Several instructions at once

The commonly used instruction sets go back several decades, but processor operation has changed significantly during this time. Parallelism has been increased, in particular.

Traditional CISC processors run only one instruction at a time. The execution of an instruction is divided into several consecutive stages that are coded in the processor's internal microcode table. On the 6502, executing an instruction consists of 2–8 stages, whereas division on the 8086 takes up over 100 clock cycles. On these processors, a programmer can calculate the execution time for their code simply by adding together the clock cycles required for the instructions and dividing the result by the clock frequency.

One of the key ideas of RISC architectures is that the execution of simple instructions may occur in parallel. The original ARM processor on the Archimedes has a three-stage pipeline: the processor saves the result from one arithmetic operation into a register while performing the next operation and reading the following instruction from memory.

Pipeline technology means that jumps are relatively costly. Executing a jump means discarding the execution stages of the instructions that follow it. There are several ways to prevent this issue. Conditional execution, used by ARM, is one of them: omitting one or two instructions is less costly than purging the entire pipeline. *Branch prediction* is a more advanced technique; the processor tries to guess whether the jump will occur and loads instructions into the pipeline accordingly. Speculative execution, on the other hand, executes both options and discards the effects of the one that did not occur.

Many processors have several parallel pipelines, allowing them to execute consecutive instructions in real time. However, consecutive instructions commonly depend on each other's results; this means that the programmer or processor should arrange the instructions in a manner where consecutive instructions do not use the same registers. In processor automation, these techniques are referred to as *out-of-order execution* and *register renaming*.

The x86 architecture has offered its fair share of challenges for processor designers. Since the 1990s, complex x86 instructions have been broken down into RISC style microinstructions that utilise the above techniques.

```
# Define the symbol _start that points the
# linker to the beginning of execution.

.globl _start
_start:

# Initialize the loop counter.

        movq $1024,%rbp     mov r8,#1024

# Execute the system call write(1,msg,15),
# where 1 is the standard output and 15 the length.
# The write call is number 1 in 64-bit Linux
# and 4 in 32-bit.

loop0:  movq $1,%rax        mov r7,#4
        movq %rax,%rdi      mov r0,#1
        movq $msg,%rsi      adr r1,msg
        movq $15,%rdx       mov r2,#15
        syscall             swi 0

# Decrement the counter, jump if not zero.

        decq %rbp           subcc r8,r8,#1
        jnz loop0           bne loop0

# Execute the system call exit(0)

        movq $4,%rax        mov r7,#1
        xorq %rdi,%rdi      mov r0,#0
        syscall             swi 0

# The string to be written:

msg:    .string "Read Skrolli!! "
```

A Linux example that uses kernel calls for 64-bit x86 (on left) and 32-bit ARM (on right). You can compile the program on the target system by using gcc -nostdlib program.s -o program or separately by calling the as assembler and ld linker.

Special instructions for special assignments

Although the basic instruction sets can more or less do everything, they often have special extensions that speed up the performance of specific tasks.

Floating point arithmetic has been a traditional requirement for scientific calculation. The idea is that numbers are presented using the mantissa and exponent, which offers a substantially larger value range than integers. PC processors started receiving integrated floating point units in the age of the 80486, but not all game consoles and mobile devices had floating point hardware even in the 2000s.

Digital signal processors (DSPs) were used to speed up the processing of image and sound data. Even basic processors started receiving DSP-type SIMD (*single instruction, multiple data*) instructions in the 1990s. Examples of SIMD extensions include MMX and SSE for the x86 and NEON for the ARM.

True to their name, SIMD instructions use several different data elements in parallel. For example, the MMX instruction `paddb mm0,mm1` interprets the values of the 64-bit multimedia registers MM0 and MM1 as rows of separate 8-bit bytes when adding them together. There are also instructions for rearranging data elements, for example.

The registers in the SSE and NEON are 128-bit, and the elements can also be floating point numbers. SSE also supports complex floating point operations such as square roots, and it has replaced the old x87 instructions on modern x86s.

In the mobile world, in particular, the same chip may contain an enormous amount of specialised arithmetic logic. For example, the Qualcomm Snapdragon 810 contains eight 64-bit ARM processor cores, each of which has three discrete pipelines and the NEON and floating point extensions. The chip also has a 288-core graphics processing unit, a 32-bit DSP and control chips that are specific to different radio protocols. Your pocket may be performing more simultaneous calculations than an old-age supercomputer.

Hack away!

The most natural, and often the most rewarding, machine code projects can be found in the field of simple information technology, such as old home computers, embedded systems and electronics platforms like the Arduino. They allow for studying the operation of the device at a precision of individual bit shifts and clock cycles, and for utilising the specific features of the processor in ways that higher-level languages do not allow. *Cross-assemblers* running on a different system are typically used when writing software for these small devices, and emulators can also be leveraged for assistance. You can easily find ready-made guides for your platform of choice.

On larger computers, high-level compilers offer the easiest route to machine code; for example, using the `-S` option in GCC creates an assembly source code file that you can examine and edit. Compilers also support inline assembly i.e. embedding assembly sections into high-level language. Optimising the speed of your code is no longer a viable motivator for learning the machine code of modern languages; instead, you can use it to write programs that are as short as possible.

Other, more direct tools are also available in addition to assembly compilers. Machine code monitors and debuggers are intended for on-the-fly editing of memory and memory-resident programs. Hex editors can be used to examine and modify program files, and many of them can display an assembly representation of the file contents.

This article was a very concise look into the essence of machine code. You can use the information contained herein to examine assembly code, but you should have detailed documents concerning the instruction set and processors available before going deeper. The best way to learn the secrets of machine code is to select a suitable project and start writing code.