1

Coding and Addressing Modes

This chapter focuses on two important characteristics of Instruction Set Architecture (ISA) (*cf.* § V1-3.5), which are instruction encoding and addressing modes.

1.1. Encoding and formatting an instruction

The instruction[1] is represented in a computer using a binary word in the format i bits, a multiple of the format n of the data and, in general, a multiple of the byte. We use the expression *machine code* to mean all those binary words representing the instruction to be executed. Instruction encoding depends on the architecture of the target processor. It is formed at least of an instruction code and, potentially, of one or more operands as Figure 1.1 illustrates.

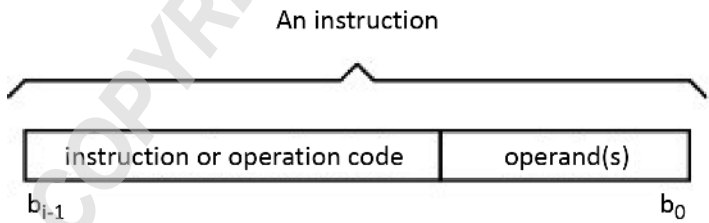

Figure 1.1. *Breakdown of an instruction*

1 In the context of a microprogrammed architecture (this will be covered in a future book by the author on microprocessors), it is sometimes called a macro-instruction to differentiate it from the micro-instruction, which is internal to the processor.

This instruction can be broken down into fields[2]. The instruction code, also called operation code (abridged to opcode), in format c, has one or more fields. The essential one is the function code. It defines the operation to be executed. Its format of f bits defines the maximum number of instructions F (= 2^f) in the instruction set[3]. Other fields can be added to this such as, for example, one that specifies the addressing mode (the addressing mode field) of the operands to the format as Figure 1.2 illustrates (VAX[4] approach from the Digital Equipment Corporation (DEC)). The processor therefore has 2^a addressing modes. Besides simplifying the encoding, one benefit is to separate the encoding of the function from that of the address, which makes it possible to make the instruction set symmetrical (*cf.* § 3.1.3). This instruction code generally takes the format of the data n of the processor to optimize access to primary memory. Since in our example n is fixed, the architect of the microprocessor or MPU (MicroProcessor Unit) must therefore compromise between the number of instructions and the number of addressing modes if the field exists. One field may be favored to the detriment of the other.

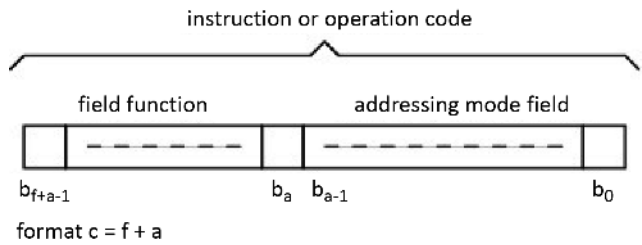

Figure 1.2. *An example of the structure of an operation code*

If the instruction requires, the operation code is followed by one or more operand fields (Figure 1.3), and their number is dependent on the operation (unary or binary) and the architecture. This operand field in the format o bits makes it possible to specify, depending on the addressing mode chosen, the value of the reference of the location of the operand needed for calculation or, potentially, the result. An operand's storage location, which is imposed by the programmer, compiler or linker or architecture, is a register or memory location. An instruction to one operand is

2 Although these fields exist, they cannot be documented or can only be documented partially, as for MC6800 from Motorola.

3 We can choose not to code the instruction (an uncoded instruction). This means that one bit is assigned to each of the possible operations. The gain lies in eliminating the logic of classic decoding and the corresponding stage in a pipelined architecture (this will be covered in a future book by the author on microprocessors). The immediate counterpart is an increase in its format.

4 VAX for Virtual Addressed eXtended.

called a “monadic”, and one with two operands, “dyadic”. When there are two operands, we speak of source and destination operands or sink operands or sometimes simply left and right operands. We cite the VAX mini-computer with a variable format as an example of encoding. The operation code included one to two bytes. It was eventually followed by no more than six operand specifiers, mainly address specifiers, making it possible to design the operand. The MPU MC6800 instruction format included one to three bytes, the first being an operation code indicating the addressing mode.

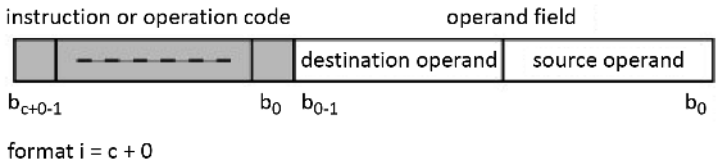

Figure 1.3. *Format of an instruction with two operands*

Table 1.1 shows the different address combinations for IA-32 instruction set (IA for Intel Architecture, also called i386). Combinations not indicated are not possible either due to the architecture or to their incoherence. We cite impossible memory (to) memory combinations in most architectures, as it is necessary to pass through a register and an immediate-register or immediate-memory, which cannot be done because of the impossibility of allocating a value to a constant.

Operands	
Destination	**Source**
Register	Immediate
Memory	Immediate
Register	Register
Memory	Register
Register	Memory

Table 1.1. *Possible address combinations in family IA-32*

The identification field (ID) of the operand(s) specifies the format and addressing mode (register or memory reference) as well as the direction of transfer (Figure 1.4). In a RISC microprocessor (Reduced Instruction Set Computer, this will be covered in a future book by the author on microprocessors), this field is included in the instruction’s code through simplification and in view of the reduced number of instructions and addressing modes.

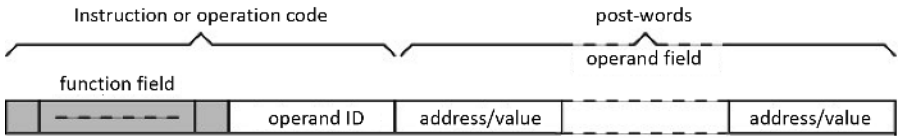

Figure 1.4. *An instruction with several operands*

By construction, the format of the instruction is fixed (fixed length), short or long, or variable (variable length). The value of a fixed format is a multiple of the byte in general. Its value will have a direct consequence for the incrementation value of the Program Counter (PC, *cf.* § V3-3.1.3). The benefit is that it will be possible to align the instructions (*cf.* § 3.1.2), thus accelerating memory reading or writing by reducing the number of memory accesses. The division of the instruction into sub-fields, for example, one for the instruction class (*cf.* Chapter 2), the second for the function, the third for the type of operands and the last for the operands and a unique format allowing simplification of the hardware, the counterpart being a larger format. A variable format, a multiple of the MPU data format, complicates the Control Unit (CU), and it has an impact on the number of machine cycles (*cf.* § V3-2.4.1) needed for decoding. During this phase, the decoder should determine the size of the instruction as quickly as possible. This information is needed, for example, for debugging, to determine the instruction boundaries or limits in the machine code (*interruptible "at instruction boundaries"*). On the other hand, it has the advantage of obtaining programs that take up less memory. In fact, a simple instruction such as `nop` (`no operation`, *cf.* § 2.8.5) will classically take up one byte compared to a word with several bytes with a fixed format. The format's variability makes it difficult to use a pipeline or a superscalar execution (this will be covered in a future book by the author on microprocessors). As an example of a fixed format, we cite the format n = 32 bits for MIPS Technologies microprocessors. Even if the format is fixed, the number of fields may vary as well as the format. Encoding uses three types, which are *Register (R-type), Immediate (I-type)* and *Jump (J-type) format* (Figure 1.5). The operation code, completed possibly by the function field, specifies the instruction. For the first type, the second field is a specifier of the source register (rs). The following specifies the target or destination register (rt or rd), which receives the result or branching condition. The last field is an immediate value, a jump or address displacement. For the J type, the operand is the jump address in a 26-bit format. For the last type, the third field is a destination register specifier (rd). The penultimate field indicates the value of a possible shift (0 = no shift). Note the conventions rt = rs + immediate and rd = rs + rt. This simple encoding should be compared with that of the Arm® family, which can show as many as 21 types (Arm 2000).

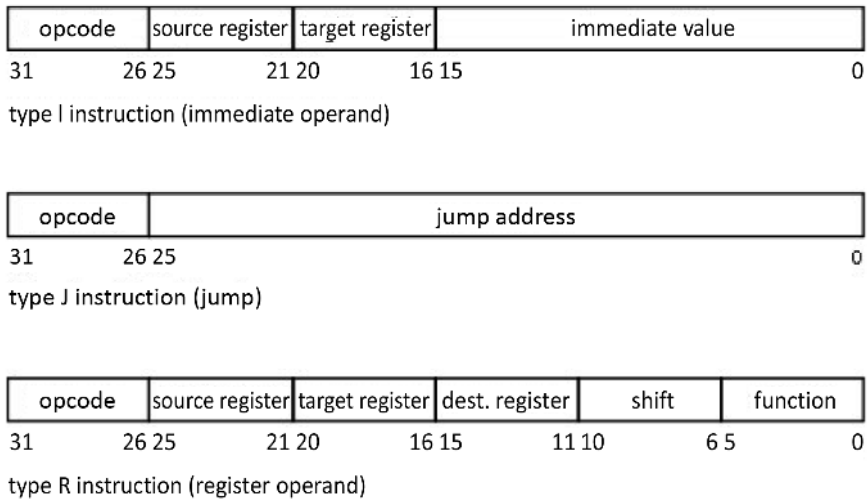

Figure 1.5. *Three fixed formats for MIPS instructions*

None of these different fields have been standardized and are dependent on the manufacturer and the MPU family. For example, for Bayliss *et al*. (1981), an instruction is formed of four fields, which are the function fields (opcode), reference fields, and format and class fields. The class specifies the number of operands and their types. The necessary format field if there is at least one operand indicates their location (memory, register or pile, for example). The reference field gives their location explicitly. Their operation code field specifies the operation to be executed.

Figure 1.6 shows the typical variable instruction of an existing microprocessor. The instruction code has a format of 6 bits. The direction bit D indicates the direction of transfer (0 = source specified by the field reg, 1 = destination specified by the field). The bit W specifies the transfer format (0 = byte, 1 = word of 16 bits). The 2rd byte is called a "post-byte". The mode field indicates whether the transfer involves only the registers or if the memory is involved, the two displacement fields therefore indicate the length of the latter. We recognize the Little Endian byte order (LE (Cohen 1981), *cf.* § 2.6.2 from Darche (2012)) typical of Intel architecture since the *Least Significant Byte* (LSB) is first stored in the memory, in the order of the increasing addresses. To finish, the R/M (Register/Memory) field, poorly named, specifies the addressing mode, that is, the method of calculating the effective address (*cf.* § 1.2). Another format exists where the instruction is coded on a single byte. Thus, the format of these instructions can vary from 1 to 6 bytes. It is

possible to add to these three types of prefix to modify the behavior of the instruction.

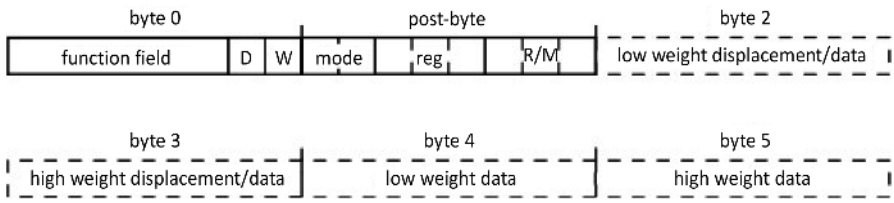

Figure 1.6. *Typical instruction format from 8086/88*

The architecture can also add a field, before or after the operation code to code the instruction class (called an extension of the operation code) or to specify a variable format. One example is the central IBM System/370 computer with its first 2 bits. The encoding of one instruction of the i486 by Intel is a typical example of the CISC approach (Complex Instruction Set Computer, this will be covered in a future book by the author on microprocessors). This type of instruction has a size ranging from 1 to 13 bytes. The word-code is therefore formed of one or two bytes for the operational code, a modify Register or Memory (mod R/M) byte, a Scale-Index-Base (SIB) byte, the bytes for displacement and the bytes for the immediate values. The reg/operation code field specifies a register or makes it possible to add information for the operation code. The R/M field specifies a register (2^3 at most) or, if it is combined with the mode field, makes it possible to specify a mode of address (24 maximum). The SIB byte makes it possible to specify the scale factor (0, 2, 4 or 8), an index register number and the base register number. In addition, one or more prefix bytes (in any order except for REX, see below) can change how the following instruction is interpreted. Figure 1.7 shows the instruction format for Intel IA-32 and Intel 64 architectures, which has changed with the evolution of MPUs. For example, the operation code for Pentium had a maximum size of two bytes. Today, the maximum length of an instruction is 15 bytes. The format for the instructions has not ceased growing.

Another example is Arm® architecture, which, to the left of the operation code, adds a condition field (Figure 2.23). Today, there are sets of instructions in multiple formats, a sort of compromise between fixed and variable formats with only two formats, for example, 32 bits and another value such as 16 bits with 19 different forms of encoding for Thumb® (Arm®) technology linked to the compression of these instruction codes (*cf.* § 1.1.1).

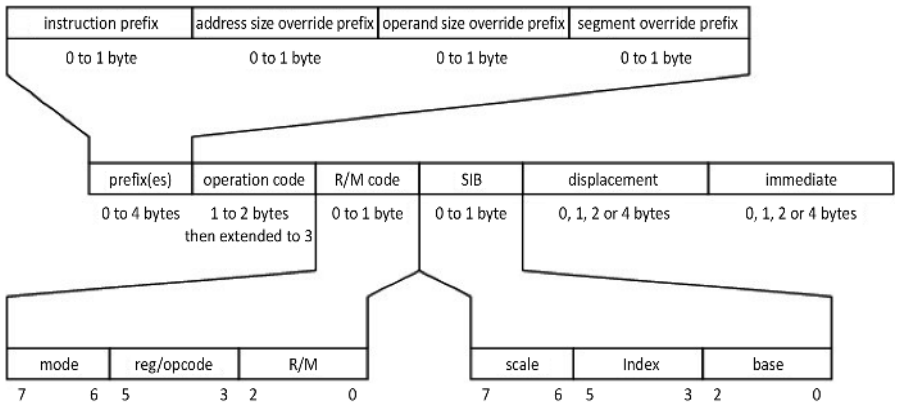

Figure 1.7. *Variable instruction format Intel IA-32 and Intel 64 (Intel 2016) architectures*

Several technical solutions exist for retaining ascending binary compatibility (*cf.* § 3.3.3). Intel has chosen the instruction prefix. It affects how the instruction is interpreted. For example, a REX (Register Extension) prefix in 64-bit mode that indicates that the instruction uses extended registers is a valid instruction (`inc` or `dec`) in IA-32 mode. This solution had already been used by Z80 with four non-assigned machine codes (hexadecimal values CB, DD, ED and FD as prefix) to expand its compatible instruction set with 8080. Another solution was to add a post-byte to distinguish between the sets of instructions. One recent example is the VEX prefix for Vector Extensions, which makes it possible to encode the AVX (Advanced Vector eXtensions, *cf.* § 2.7.1) extension from Intel.

The number of instructions, type of architecture (stack-based, register-based, etc.), the number of addressable registers, the number of internal busses and the type, format and location of the operands will have an influence on the format i of an instruction. For access to primary memory, the memory organization, in particular the exchange format (byte or word), byte order (remember the Endian story! *cf.* § V1-2.2.1) and the alignment (*cf.* § 2.6.1 from Darche (2012)), will have some influence. The ISA can be evaluated by the number of instructions F, their complexity, their format i and the memory space they occupy. The designer's choice will depend on the function of the desired performances (execution time, memory requirement, etc.), of the usage domains and the manufacturing cost. Complexity, if it is not material, could affect the software, in particular the compiler as in the RISC approach and in the programmer. The appendix shows the instruction encoding table for MPU 6809E from Motorola. For information, the aspect of decoding an instruction has been discussed in the previous volume.

1.1.1. *Code compression*

In order to limit the programs' memory footprint for reasons of cost, memory size, performance or, in particular, power saving, one solution is to compress the machine code at compilation and its decompression at execution, for example, when it is loaded in the MPU cache memory (Wolfe and Chanin 1992). One benefit lies in the fact that the compiler has not been modified. For implementation, the Huffman (1952) (de)compression algorithm can be used, for example. Because of its objectives, it is intended especially for embedded systems with an MPU/MCU[5] RISC. Two industrial examples are Thumb® and Thumb-2 for which the 16-bit instruction word is a compression of the classic version of Arm® processors, which have a 32-bit format. RISC-V (Waterman 2016) has a compressed version of its code suggested by (Waterman 2011). A comparison between MPUs can be made using a measurement of the code density.

The principle can quite clearly be applied to data and to buses (*cf.* V2) for the same aims.

1.2. Addressing modes

We recall that the address is a whole number that makes it possible to identify (we also say locate or spot) a place in the memory (*cf.* § V1-2.1). This, generated by an MPU, is termed "physical" (PA for Physical Address) since it is this that will be carried by the address bus. This physical address can be positive (i.e. natural integer) or also negative (i.e. relative integer) in the case of an address in Assembly Language (AL) or machine language, for example, for a displacement relative to the current value of the PC (Program Counter). Addressing is the mechanism for accessing information (data and instructions) stored in MPU registers or in other levels of the memory hierarchy (*cf.* § V1-2.3). The addressing or referencing mode specifies how to reach the instruction (code addressing mode) and its operands (operand addressing mode) during its execution. This distinction between the addressing code and its operands, which may moreover be an instruction classification (*cf.* § 2.1), may not exist (which is the most common scenario). One of the difficulties of using the concept is that its designation and its semantics vary depending on the architecture and on the designer of the CU (Control Unit). Thus, it involves sometimes only the memory address (memory address calculation mode) or it also covers the registers (operand addressing mode). The definition is taken in its widest sense. It does not therefore only involve access to the primary memory. The different addressing modes add to the wealth of a processor, and their number still varies depending on the architectures and designers. Addressing modes are one of

5 For MicroController Unit, i.e. a microcontroller (*cf.* § V3-5.3).

the ISA specification points (*cf.* § V1-3.5). For example, the IBM System/360 mainframe computer only has three (immediate, register and memory), but the Pentium microprocessor has nine. The more possibilities there are, the less the assembly language programmer will have to write the lines of code to carry out the desired operation. The argument refers today to the compiler designer, as assembly language is used less and less, except for teaching purposes or to meet a specific need in the use domain (*cf.* § V2-1.3). The other side of the coin is a more complex control unit and a longer execution for the instruction using it. We will see what the consequences of this will be covered in a future book by the author on microprocessors, which studies, among others, the RISC approach. If necessary, it specifies the means used to calculate the effective address (EA), also called the target address. This address is the result of the evaluation of an address according to its addressing mode. It will be applied on the address bus to reference the memory location if there is no virtual address mechanism at work (a mechanism that will be covered in a future book by the author on microprocessors). A synonym for EA (Effective Address) is "physical address". In the contrary scenario, the effective address is a logical address that should then be translated into a physical address in the case of the Virtual Memory (VM) mechanism. Depending on the manufacturers, the name may also be different or there may be other nuances. To finish, some microprocessors distinguish access to instructions and to their operands from access to Input–Output (I/O) registers with specialized instructions (I/O addressing mode), thus making it possible to address different Address Space (AS) (*cf.* § V3-2.1.1.1). One example is shown in § 2.8.2.

We define four modes of basic (i.e. simple) addressing, which are immediate addressing, implicit and explicit addressing and memory addressing. Memory addressing is broken down into direct, relative, indirect, indexed and based addressing. These modes indicate the way to fetch or store the operand. The storage of one value can only be done in a register or memory location. There can then exist combinations of these basic addressings, called complex addressings that can be replaced using a sequence of instructions with simple addressing. The other modes involve primary memory, the stack, the bit, the registers and those specific to a particular MPU family. To illustrate these, we have chosen some instructions that are representative of various MPUs. In these examples, all digital data will be expressed on base 10 (implicit base) with the exception of indications in the form of a character prefixing or post-fixing the value or of a number in subscript. To define the operand, the rules of syntax inspired by those of the MC6809 microprocessor will be the following:

#: immediate value

$: hexadecimal base

%: binary base

The registers will be the following:

PC: Program Counter

A: accumulator

The conventions for the pseudo-code will be the following:

← or =: assignment of the right-hand value (similar to an *rvalue*) in the left identifier (similar to an *lvalue*). The symbol used means "receives" or "takes the value". This left–right positional information avoids using parentheses, but it makes use of them for the right-hand value; they mean "contained in".

(): address access, of which the value is framed.

@: (calculation of) the two-point symbol address: concatenation

1.2.1. *Immediate addressing*

Immediate addressing mode, also called immediate data addressing mode, makes it possible to initialize a register or a memory location with a constant value d, which is specified after the instruction mnemonic (*cf.* § 2.1) (Figure 1.8), hence its other name "literal addressing mode". There is no effective address here since the memory is not addressed, but (DEC 1983) called it "PC immediate mode with auto-increment" as the PC (Program Counter) is used to address the value memorized immediately after the instruction code. One example is `LDA #%10101010` from MC6802 from Motorola, which means that the accumulator A receives the immediate binary value 10101010b (b for binary) in byte format.

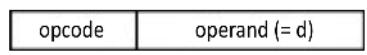

Figure 1.8. *Instruction with an operand field*

It is one of the fastest addressing modes since the value is included in the instruction and there is therefore no additional access to the main memory to fetch the operand accessed by another addressing. But this value is a constant. In addition, from the perspective of programming, the change of value means a modification in the program since the value field cannot be a destination. The extent of the values (in the sense of Chapter 2 of Darche (2000)) is limited by the number of bits

remaining after subtraction of those bits reserved for coding the operation itself (a similar limitation for the address for direct and relative addressing). In its extended or long version, the format is double that of a short format. The possibility of choosing makes it possible to decrease the number of clock cycles to fetch the operand. An alternative to this mode is register addressing, which contains a constant value, which is materially fixed. This is the current practice with RISC microprocessors (this will be covered in a future book by the author on microprocessors) such as Arm®, whose register r0 contains the null value (*cf.* § V3-3.1), which can serve for initialization and avoids time-consuming external access to the main memory.

1.2.2. *Register addressing*

The use of registers makes it possible not to slow the microprocessor down since the registers are integrated. An instruction that uses them in addressing mode will only require external access to fetch the instruction code. It is possible to address a register in two ways, explicitly and implicitly.

1.2.2.1. *Explicit addressing*

The operand field operand(s) R specifies the registers used for execution. It is sometimes called register (direct) addressing, the term "direct" indicating that the referencing in the register is found in the instruction coding, as for the direct memory address (Figure 1.9). These registers are accessible to the programmer. There is no effective address since the memory is not addressed, hence a fast execution of the instruction using it and a small instruction format. It is for this reason that RISC microprocessors prefer to use this mode. For other architectures, the number of registers accessible to the programmer is limited (order of size: about 20).

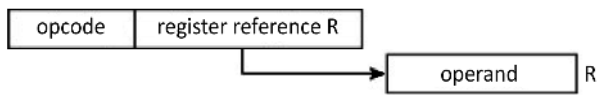

Figure 1.9. *Execution of an instruction using register addressing from one register*

The example below shows an addition in an Arm® microprocessor, which uses three registers: r0, r1 and r2:

```
ADD r0,r1,r2; r0 ← r1 + r2
```

Note, a distinction can be made between Data Register Direct Addressing and Address Register Direct Addressing as for MC68000 (*cf.* § V3-3.1.1).

1.2.2.2. *Implicit register addressing*

To simplify the programming, some instructions use one or more registers in an extended or implicit manner. In this addressing mode, also called implicit or implied addressing mode, no operand is specified after the instruction mnemonic (*cf.* § 2.1). Execution of the instruction involves the reference to operand that is not joined to the operating code. One synonym is implication (Brooks 1962). The instruction format is reduced by it. One example is the `dex` instruction from MCS6502, which decrements its index register X. The name of this appears in the mnemonic to facilitate programming. Sometimes when the accumulators are used, this mode is called "accumulator addressing". The example below applied to MC6809. The accumulator B specified by the last letter of the mnemonic receives a value expressed in hexadecimal base.

`LDB #$FA`; B ← FA_{16}

If the name of the registers does not appear in the mnemonic, then only a detailed reading of the technical documentation can specify the name of these registers. In the example below (MC6809), the instruction for multiplication `mul` (without operands) implicitly uses both implicit accumulator registers A and B and stores the concatenated result in these same registers, and the MSB (Most Significant Byte) is found in accumulator A, which in pseudo-code gives: A:B ← A × B.

Another example is the instruction from 8086 `mul bl`, which uses the implicit register A as source and destination operands in the case of multiplication in 8-bit format (ax ← bl × al for this example).

To generalize, an instruction lacking one or more operands found in a register (an accumulator for example) or in memory uses implicit addressing. We find this mode in machines with a single address called an accumulator or in the extreme case of zero-operand computers also called stack or pushdown-store machine (*cf.* § V1-2.7.1). By broadening the definition to registers that are not accessible to the programmer, any instruction for its execution uses the PC (Program Counter), which is therefore implicit.

1.2.3. *Memory addressing modes*

It is possible to address the memory in a direct, relative, indirect, indexed or based manner. Combinations of these modes are possible. Other specific modes are then presented.

1.2.3.1. *Direct addressing*

Direct or absolute addressing is without doubt the most natural. It can access a memory address location A defined (i.e. arranged immediately) after the instruction code in the operand field (Figure 1.10). It can therefore be considered a constant. The effective address EA is given by the following formula:

$$EA = A \qquad [1.1]$$

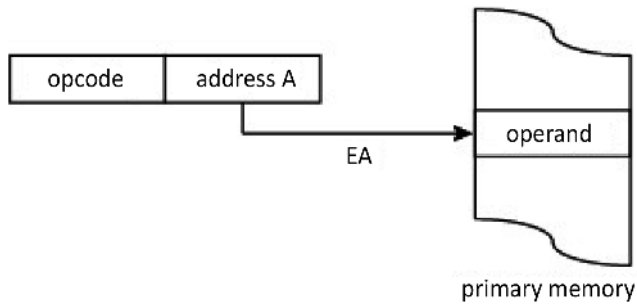

Figure 1.10. *Instruction with direct addressing*

It can be used by jump instruction to branch to a set location in the program. This mode is in fact an indirect mode with auto-incrementation using the PC (Program Counter) as an indirection register (*cf.* § 1.2.3.3 for indirection).

This mode allows for variations depending on the format of the address provided, the benefit lies in reducing the instruction's memory size. Some manufacturers thus distinguish the short mode from the extended mode, known as long mode, depending on the format of the address A, provided. In the short mode (absolute short, page zero, also known direct at Motorola, a base page (IEEE 1985)) illustrated in Figure 1.11, the address is expressed in a smaller format than that of a microprocessor. The address field may also be smaller than 3 bits, one example being the 8021 microcontroller from Intel or, classically, 8 bits in 8-bit MPUs. Page zero can be seen as a bank of registers (RF for Register File, *cf.* § V3-3.1.11.1). The MIPS firm speaks of pseudo-direct addressing. Aside from a smaller format, the second benefit lies in decreasing the number of memory accesses to fetch the instruction code and the operand address. It is equivalent to a basic addressing + displacement, as in the IBM System/370 architecture, with a null base address. One example is the MC6802 microprocessor where the address is in byte format, while the format of the MPU address bus is double. This then limits the address space to

the interval $[00, FF]_{16}$, hence the term “absolute short addressing” or “page zero”[6] (if the size of the memory page is 256 bytes). In the example below, the A register receives the content of memory location 00.

```
04F0   96 04 LDA  $00; A ← (00)
```

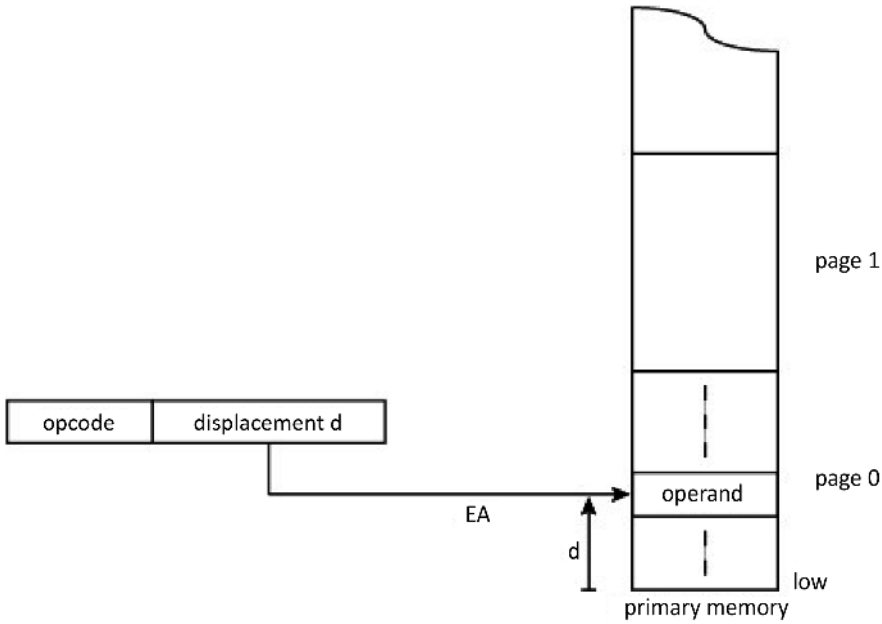

Figure 1.11. *Instruction with an address at page 0*

The concept of page zero addressing has been improved with direct page addressing. The direct page is now movable in a larger memory page. The start of the page is addressed by a specialized register (*cf.* § V3-3.1.1). We cite MC6809 (a page of 256 bytes in a space of 64 KiB, addressing capacity of the MPU itself, direct page (DP) register), the 65CE02 from Commodore Semiconductor Group or CSG (the same as before except that its addressing capacity is higher, base page register B) and the 65816 from the firm Western Digital Corporation (WDC) with an address over 16 bits in the direct page register D.

A direct addressing is limited in its extent for a given instruction format; there are bits reserved for coding the instruction, which should be subtracted from the bits

6 The mini-computer PDP-8 for Programmable Data Processor from DEC introduced in 1965 used this term.

reserved for the addressing. This limitation can be lifted if the instruction format is not limited (i.e. variable format). With extended addressing, the address belongs to the microprocessor's address space without restriction. The format is that of the address bus. It should be noted that the absolute address can be implemented with a basic address + displacement with a basic register with zero content base.

1.2.3.2. *Relative addressing*

Relative addressing, implied in PC (Program Counter-relative addressing), makes it possible to access a memory location relating to the current position of the program counter that, we recall, contains the address of the next instruction to be executed (Figure 1.12) after the decoding stage. This mode is in fact an indexed mode using the PC (*cf.* § 1.2.3.4 on indexing). With the following formula, we see that the effective address of the data or instruction relates to the PC by a value of d:

$$EA = PC_{\text{following instruction}} + d \qquad [1.2]$$

This is the favored mode for jump instructions, whether conditional or not (PC-relative branch). The relative displacement d is expressed in a signed integer representation, which is always the complement to 2^n (two's complement, *cf.* § II.2.5 from Darche (2000)). Depending on the size of the displacement, the extent of the jump will be limited to (-2^{n-1}, 2^{n-1} -1), with n being the format of the address field. Depending on the value of n of the relative address, we will call it a short or long jump. When the processor uses segmentation (this will be covered in a future book by the author on memories), jumps can be made within a single segment (intra-segment jump) or between two segments (extra-segment jump).

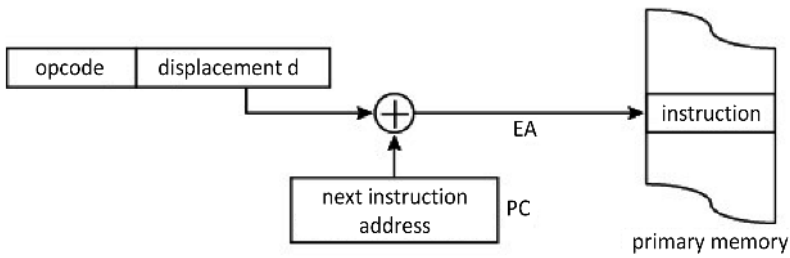

Figure 1.12. *Execution of an instruction in relative addressing*

The example below (x86) is a negative jump. The hexadecimal value F9 represents -7 in base 10. This means that the processor will connect 5 bytes higher than the instruction address, the difference of two bytes arising from the fact that the

PC has changed while the instruction was executed (incrementation of the size of this instruction, here, two bytes):

73 F9 `jnc` loop; PC ← PC + F916

Two particular cases should be cited: `jmp 00`, which jumps to the following instruction since the program counter has been incremented during the decoding phase of the execution cycle (*cf.* § V1-3.2 and V1-3.3.2) to direct the following instruction and `jump –n`, where n is the instruction format (in words) underway, which implements an infinite loop. This mode is linked to PC (PC with displacement or Program Counter with Displacement for MC68000). It can be seen as an indexed mode, the indexation register being the PC (*cf.* § 1.2.3.4).

This mode is useful for generating the independent code of implantation in memory (position-independent code). We also speak of a translatable code (relocatable code), a topic discussed in § 3.1.4. It is also at the root of implantation of classic control structures of high-level languages (if_<condition>_then_else, iterative structures (i.e. loops) such as while_<condition>_do, repeat_until_<condition>, for_<condition>_do, etc.) in assembly language.

This mode can even be used to address an operand (Figure 1.13). We cite x86 64-bit architectures with addressing called RIP (Instruction Pointer Register)-relative, ARMv8 with literal mode and MPU MC6809 with the program counter-relative mode.

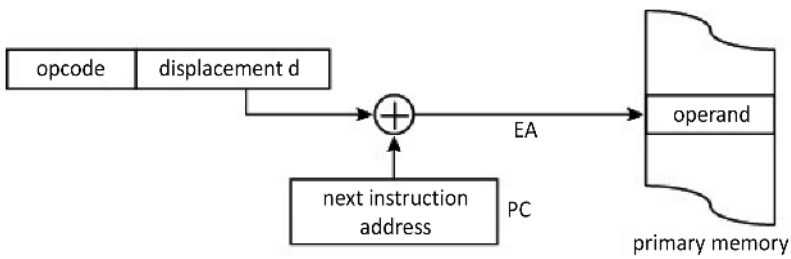

Figure 1.13. *Seeking an operand in relative addressing*

This mode can be seen as an indirect mode auto-incrementation using the PC (Program Counter) as an indirection register (*cf.* § 1.2.3.3).

1.2.3.3. *Indirect addressing*

It is useful to dissociate addressing of the operand from that of the instruction code. The address may thus vary without changing the reference indicated in the

instruction. This mode is used to implement the mechanism of the High-Level (programming) Language (HLL) pointer. In assembly language, the square brackets "[" and "]" are generally used to employ this mode. Some constructors use parentheses or the character @. A memory location or register contains the address of the operand. In indirect mode or register deferred mode (register indirect or register deferred addressing[7]) illustrated in Figure 1.14, the effective address EA is given by the following formula:

$$EA = R \qquad [1.3]$$

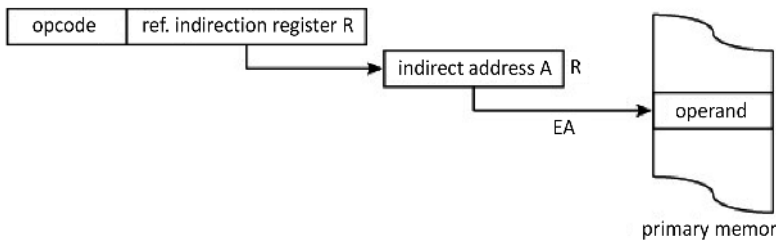

Figure 1.14. *Instruction with indirect register addressing*

In memory indirect addressing illustrated in Figure 1.15, the final effective address EA is given by formula [1.5]. Here, it is a double indirection:

$$EA' = A \qquad [1.4]$$

$$EA = (A) = A' \qquad [1.5]$$

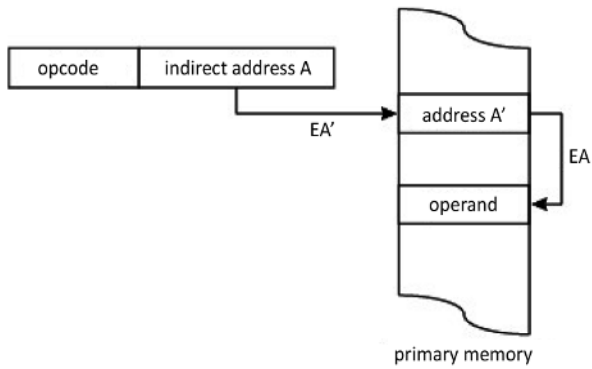

Figure 1.15. *Instruction with indirect memory addressing*

7 Vocabulary from DEC (1983).

This mode of addressing generally has a greater extent than direct addressing since the addressing format m is the same as that of the data format n. It was therefore useful for the first computers, which had a restricted addressing capacity (in the case of mini-PDP-8 computers from the DEC firm of the NOVA series from Data General, for example). Another advantage is the decrease in the instruction format, thus increasing the instruction throughput. For MC6809, the constructor speaks of "extended indirect addressing". The compiler will doubtlessly use this mode to implement the high-level language pointer mode such as C or Pascal by putting the value of the pointer (i.e. an address) in the indirection register.

An auto-increment or auto-decrement can be suggested, which can be done before (prefix "pre") or after (prefix "post") the instruction using it is executed. It makes it possible to implement operators directly, such as ++ and -- in the language C. This means that after execution of this operator, the value of the pointer that contains the address of the object pointed to is incremented or decremented by a value equal to the size of the pointed element. But in the MPU, the increment or decrement value is fixed at programming in low-level language. More generally, auto-increment or auto-decrement makes it possible to manage a memory index, which is useful, for example, in displacement in a data structure such as an array. Register indirect addressing with post- or pre-increment/decrement is adapted for digital signal processing to address samples.

This mode is in fact the one that makes it possible to implement absolute addressing mode using the PC (Program Counter) as an indirection register. It is for this reason that DEC (1983) with PDP series, which used the PC as a General-Purpose Register (GPR, *cf.* § V3-3.1), called it "PC absolute mode", equivalent to an immediate indirect addressing (immediate[8] deferred mode or auto-increment deferred mode). The term "immediate" means that the value immediately following the instruction code addressed by the PC will be used to fetch the address of the operand (EA = PC + 2 bytes in the case of the PDP-11 mini-computer) with, afterwards, an update to the PC. This same manufacturer proposed a relative deferred mode PC addressing, that is, indirect relative addressing, which uses the PC added to a displacement to fetch the operand's address (EA = ($PC_{instruction\ +\ 1}$ + displacement) in the case of PDP-11).

1.2.3.4. *Indexed and based addressing modes*

Indexed addressing is characterized by using an Index Register (IR) that contains a reference address, called a base or offset address, making it possible to access a

8 Here this means an immediate value following the instruction code that will serve as the address.

memory location. The content of this register, here R, is added to a displacement A specified with the instruction (Figure 1.16). The effective address EA is equal to:

$$EA = R + A \qquad [1.6]$$

Indexed addressing with null displacement is identical to register indirect addressing. This mode is equivalent to relative addressing if the index register is replaced by the PC (Program Counter). The index register may be implicit or designed explicitly as an operand. It can be dedicated specifically to this usage or it can be a GPR. In the former case, it is generally named X or Y (in the case of MCS6502). From the perspective of execution complexity, it adds an operation (addition) compared to the indirection. The @ symbol is generally used in assembly language to indicate this mode.

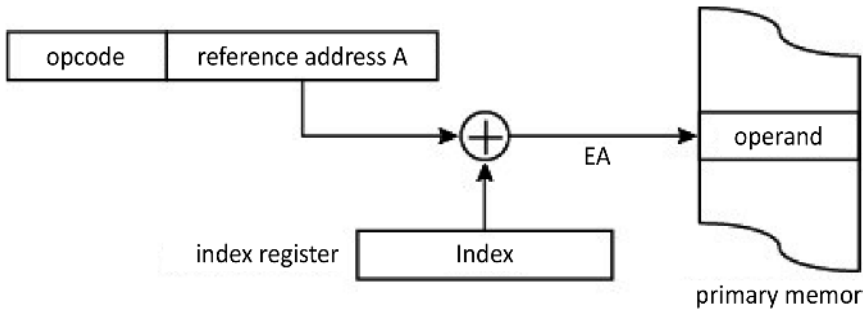

Figure 1.16. *Execution of an instruction in indexed addressing with displacement (indexation "true")*

Cushman (1975) speaks of "true" and "false" indexing. Indexing is called "true" when the index address is the operand, the case in Figure 1.16 and MPUs MCS6502 and 2650 (Signetics). In the second case, the index address is in the dedicated register and the operand is the index, one example being the MC6802/MC6809 (Figure 1.17). The second field of the instruction word, called a "modifier" in Simpson and Terrell (1987) has an 8-bit format, while the index register format has 16 bits. Some manufacturers such as Motorola consider the relative address as an indexed mode, the indirection register being the PC (Program Counter, *cf.* § 1.2.3.3).

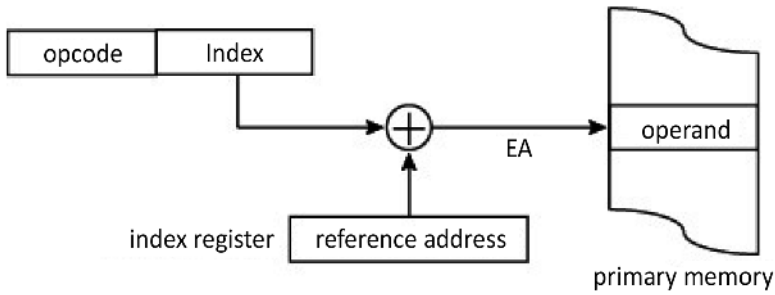

Figure 1.17. *Execution of an instruction in indexed addressing with displacement (false indexing)*

As for indirection with auto-increment or auto-decrement, auto-indexing can be suggested with the addition of an integer A to the value of the register R. The designer of M68HC12 speaks of pre-decrement and post-decrement indexed. At each execution, we will have:

$$EA = R + A \qquad [1.7]$$

$$R = R + 1$$

Relative addressing is similar to an indexed addressing by the PC (Program Counter). It is for this reason that DEC (1983) called it "PC-relative addressing mode".

Scaled indexed addressing mode makes it possible to multiply the content of the index register by a constant 1, 2, 4 or 8, for example, for 80386. This facilitates management of data structures in high-level languages as an array, a structure or record.

Base (plus) offset addressing arises from the principle above except that the index register is replaced by a base register (Figure 1.18), hence its other name: base register addressing. Intel uses the BX and BP (Base Pointer) for x86, the first addresses the data segment and the second addresses the stack. The IBM z System mainframe computer uses 16 General-Purpose Registers (GPR) in 64-bit format as a base register and the displacement is specific to the 12-bit format. At its origin, this mode made it possible to extend the address space. Today, this is no longer necessary.

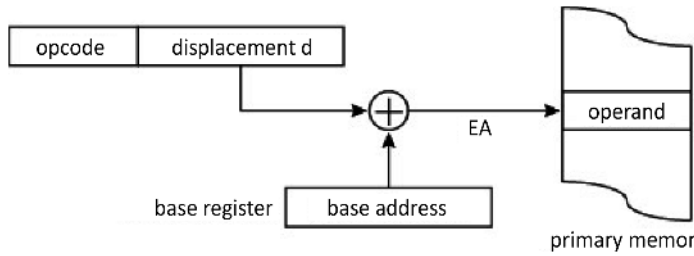

Figure 1.18. *Execution of an instruction in base addressing with displacement*

The difference between these two modes is more semantic than applicable to calculating the effective address. The index varies starting from a given index address with the instruction, while the base address is constant (hence its name) and an offset is provided with the instruction. Moreover, Intel uses the terms "base" and "indexed" for base addressing. Moreover, if no offset is specified with the instruction, Intel (1989) names the 8086 base and indexed addressing without offset "indirect register addressing". Often, in RISC microprocessors such as Arm®, the r0 register contains the constant 0, thus avoiding an immediate addressing using a main memory access that takes a great deal of time. If it is used as a base register, the addressing becomes absolute. The base mode is similar to segmented addressing (this will be covered in a future book by the author on memories). Another means of differentiating these two addressings is that there is no auto-increment with base addressing.

Calculation of the effective addressing depends on the storage order or endianness (*cf.* § V1-2.2.1) of the address' bytes. Thus, MCS6502 with a little-endian order is favored because the addition is carried out starting from the LSBs.

1.2.3.5. *Combinations of addressing modes*

It is possible to combine the addressing modes above. Some processors offer indirect addressing with indexing. The associated terms "pre-indexing" and "post-indexing" will qualify at what step of the address calculation the indexing will apply. Pre-indexing means that indexing is carried out on the indirection address (*pre-indexed indirect addressing mode*), hence the second name, "indexed indirect addressing mode".

We will have:

$$EA' = A + R \qquad [1.8]$$

$$EA = (EA')$$

Figure 1.19 shows the mechanism. One example was MCS6502, which included two registers called “index registers X and Y” even though X has already served for indirection. Its designer calls this mode (indirect,X), which is justified by the relationship [1.8]. It was also suggested by MC6809. DEC used the term “index deferred addressing mode”.

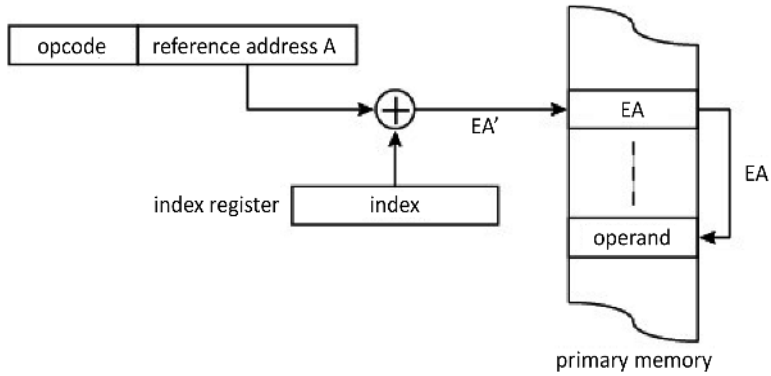

Figure 1.19. *Indirect indexed addressing or pre-indexing*

Post-indexed indirect addressing mode or indirect indexed addressing mode applies indexing after indirection, as illustrated in Figure 1.20. We will have:

$$EA' = A \qquad [1.9]$$

$$EA = (A) + R \qquad [1.10]$$

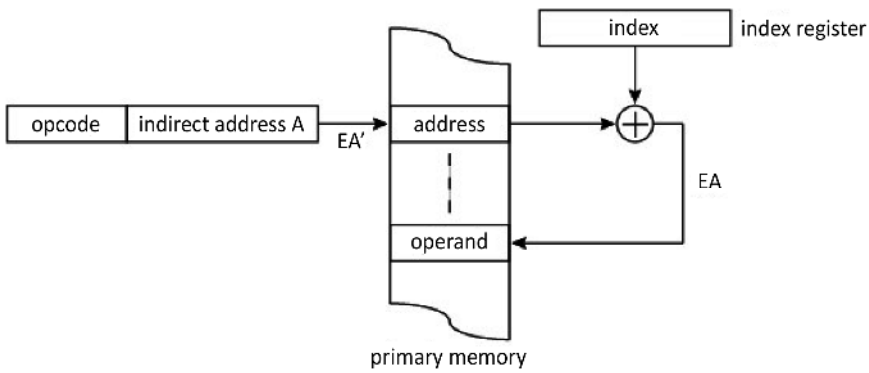

Figure 1.20. *Indirect indexed addressing or post-indexing*

The peculiarity of MCS6502 is that it used zero-page addressing as the address field was limited to 8 bits and the indexing occurred only on the lower part of the address (Figure 1.21). Its designer calls this mode (indirect),Y, which is justified by the relationship [1.10].

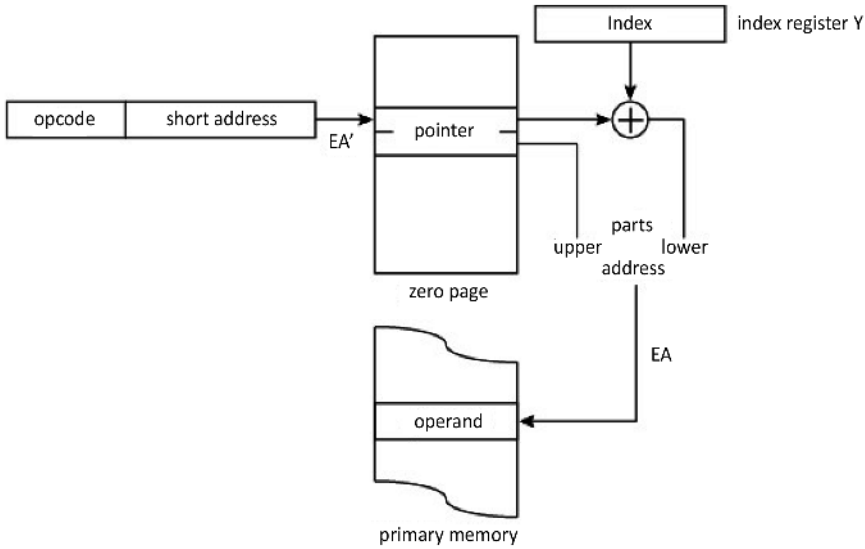

Figure 1.21. *Indirect indexed zero-page addressing of MCS6502*

A representative, penultimate example is MC6809, which offers 18 variations in mode, combining indexed and indirect addressings with the possibility of automatic post-increment or pre-decrement. This post-increment or pre-decrement is useful for managing a stack's pointer. Table 1.2 summarizes the possible combinations. R represents one of the four registers that can be used for indexing, the classics X and Y and the stack pointers user U and material S. Note the addressings using the program counter at the end. The offset is expressed in complement to 2^n representation.

Indexed and based addressings with or without offset (based indexed plus displacement addressing mode) can be combined, thus offering, for example, 17 possible variations in the case of microprocessor x86. One example of this use is addressing an array of records, of a vector or of a structure, the base pointing the start of the array and index, an element of the array and the displacement, a field of the element.

MC6809 assembly language notation	Description
,R	Zero-offset indexed
[,R]	Zero-offset indexed indirect
,R+	Zero-offset indexed post-increment of 1 (auto-increment R)
,R++	Zero-offset indexed post-increment of 2 (auto-increment R)
[,R++]	Zero-offset indexed post-increment of 2 indirect (auto-increment R)
,-R	Zero-offset indexed pre-decrement of 1 (auto-decrement R)
,--R	Zero-offset indexed pre-decrement of 2 (auto-decrement R)
[,--R]	Zero-offset indexed pre-decrement of 2 indirect (auto-decrement R)
n,R	Constant signed offset indexed (5, 8 or 16 bits offset from R)
[n,R]	Constant signed offset indexed indirect (5, 8 or 16 bits offset from R)
A,R	Accumulator A signed offset from R indexed
[A,R]	Accumulator A signed offset from R indexed indirect
B,R	Accumulator B signed offset from R indexed
[B,R]	Accumulator B signed offset from R indexed indirect
D,R	Accumulator D signed offset from R indexed
[D,R]	Accumulator D signed offset from R indexed indirect
n,PCR	Constant signed offset from PC indexed (8 or 16 bits)
[n,PCR]	Constant signed offset from PC indexed indirect (8 or 16 bits offset)
[n]	Extended indirect

Table 1.2. *Combined MC6809 addressing modes*

1.2.4. *Other addressing modes*

Other modes have been introduced to provide a high-level functionality or to adapt to a specific domain such as digital signal processing (*cf.* § V3-5.2), to a specific mechanism of a processor or to a component such as an I/O controller

(*cf.* Chapter 3 of Darche (2003)) or a microcontroller (*cf.* § V3-5.3). Moreover, other modes belong to high-level languages. To finish, some obsolete modes are presented.

1.2.4.1. *Memory-to-memory addressing*

The memory-to-memory transfer functionality is possible in a von Neumann-inspired MPU, but it should be seen as exceptional. This is the continuity of the tendency of CISC processors to implement high-level functionalities in the material. Intel calls this mode "string addressing" for its 8086. It involves addressing the characters of a string, that is, of an array of characters by indirection using both its pointer registers SI (Source Index) and DI (Destination Index). It makes it possible, among other things, to read or write a character and, whether the repeat prefix is conditional or not, to make a copy of it in the main memory. The search function in a string is also available.

1.2.4.2. *(Implicit) stack addressing*

Operands are found implicitly (i.e. they are not named) on the stack which is, we recall (*cf.* § 4.1), access to LIFO (Last-In/First-Out, push-in/pop-out or push-down/pop-up memory) and implemented in primary memory in modern MPU. The two primitives (i.e. functions) to access it are stacking() and unstacking(), translated into instructions respectively by `push()` and `pop()`, for example, in x86 architecture. These instructions implement, internally, an indirect addressing mechanism with the Stack Pointer register (SP), which memorizes the address at the top of the stack. The stack is implemented in main memory, but it can be implemented in the processor. The stacked element is specified with the operand. There are also specific instructions to a register, such as `pha/pla` (push/pull accumulator onto/from stack) from MC6800, which makes it possible to stack/destack this MPU's accumulator. MCS6502 uses `php/plp` (push/pull processor status on/from stack) this time for the MPU's context. By extension, stack computers do not explicitly name the operands (zero-operand, one-operand or two-operand addressing). For reading on this subject, see Koopman (1989).

The NS3200 (Hunter 1987) from National Semiconductor (NS) has broadened access to the stack by offering a mode called top-of-stack, literally "stack top", which makes it possible to access the data of the so-called summit, since modification of the pointer is not systematic (i.e. dependent on the operation). To finish with this topic, MPUs such as the families Arm®, PowerPC or MC68000 make it possible to use General-Purpose Registers (GPR) as stack pointers. The addressing mode is of indirect type with auto-increment/decrement.

1.2.4.3. *Bit addressing*

The first GPPs (for General-Purpose Processor, *cf.* § V3-1.1) did not have specialized instructions to manipulate (set at one/zero or extraction) or to test individually the bits of an operand by conditional branching (*cf.* § 2.4.1). It is generally microcontrollers that possess them as they have to read or modify binary information in memory or at the input–output ports (*cf.* § 3.1 from Darche (2003)). Thus, the microcontroller 68HC12 from the MC6800 family from Motorola has the instructions `bclr` (bit clear) and `bset` (bit set) that initialize respectively at 0 or at 1 one position bit specified with the help of a binary mask (see exercise E2.4) in an address word A. These instructions use this mode associated with pre-studied conventional addressing modes. It should be noted that the addressing space is limited compared to other modes. The example in Figure 1.22 shows a reset at 0 for the MSb (Most Significant bit) of an I/O port in byte format implanted at the address $0F_{16}$.

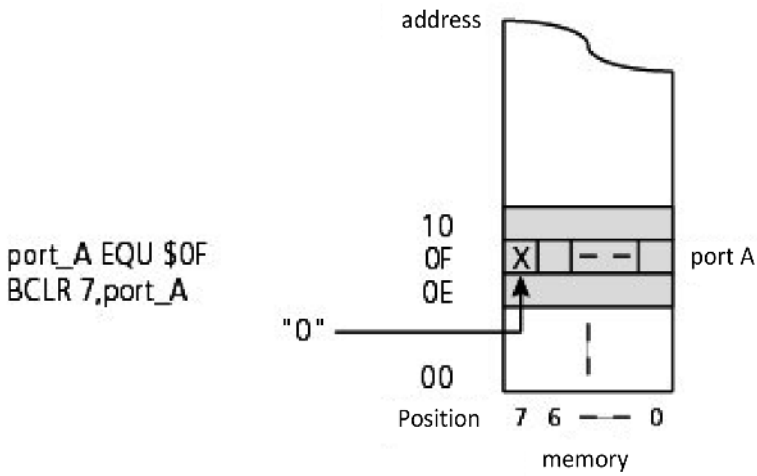

Figure 1.22. *Execution of an instruction in bit addressing*

1.2.4.4. *MMR addressing*

One possibility is to manipulate I/O registers as conventional addresses (MMR for Memory-Mapped Register, literally, registers projected into memory, *cf.* § V3-3.1.1 and V3-2.1.1.1) in reduced format (page zero addressing) with fast specialized access instructions. One example is the Digital Signal Processor (DSP), reference C5000 from Texas Instruments (TI).

1.2.4.5. *Addressing modes specific to the digital signal processor*

Other than indirect register addressing with post- or pre-increment/decrement, two other modes are particularly adapted to digital signal processing, which justifies their implementation in DSPs. This is circular addressing and (address) bit-reversed addressing.

1.2.4.5.1. Circular addressing

Digital signal processing consists of digitizing samples x_i ($i \in [0, \infty]$) of the signal that are stored in memory, then carrying out a mathematical processing such as filtering on them to then reconstruct the analog signal. To simplify the discourse, memorization of coefficients needed for the calculation is not attempted. The sample flow is of infinite length, and the calculation is only made on a limited number of consecutive samples on the sampled sequence. This set is called a "window". Linear addressing of the buffer FIFO (First In, First Out) illustrated in Figure 1.23a is not well adapted as it is necessary to test whether the pointers have reached the end. Moreover, the size of the buffer is necessarily high. The circular buffer (ring or cyclic buffer, circular queue), that Figure 1.23b shows, is a much better solution as it makes it possible to decrease its size to that of the window of samples needed for the calculation.

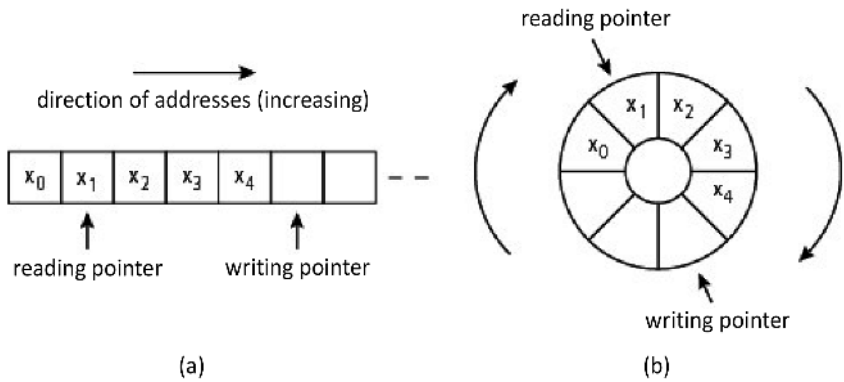

Figure 1.23. *Window of five samples*

Circular or modulo addressing makes it possible to implement a circular buffer in a Random Access Memory (RAM). As shown in Figure 1.24, it is necessary to have four pieces of information that are the size of the circular buffer L, the address of the base of buffer B, the index pointer of the buffer I and increment (relative integer) M. This addressing uses modular arithmetic where the extent of the values is finite to calculate the pointer addresses. The benefit of using it lies in the fact that

a block of L contiguous memory words is addressed by a pointer that uses a modulo addressing L. This means that once a pointer arrives at the end of a buffer, it is reinitialized to point the other end (more precisely, modulo addressing is the capacity to memorize the buffer).

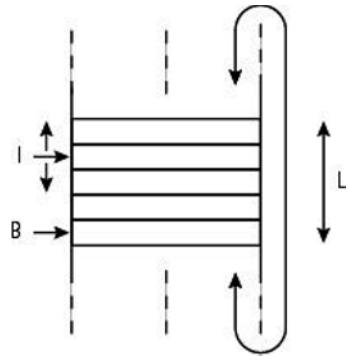

Figure 1.24. *Circular buffer*

This is conveyed in algorithmic form by:

$0 < |M| \leq L$

$I \leftarrow I + M$

if $M > 0$

then

if $I \geq B + L$

then $I \leftarrow I - L$; buffer overflow or overflow from above

end_if

otherwise

if $I < B$

then $I \leftarrow I + L$; buffer overflow or overflow from below

end_if

end_if

Management logic detects a buffer overflow when there is a wraparound. It then generates an interrupt request (see Chapter 5) to warn the handler. This automatic management avoids a costly rearrangement of data by shifting them (Figure 1.25(a))

and a permanent monitoring of the pointer value to know whether it has reached an end of the buffer in order to reinitialize it. It frees useful calculating power for processing. For example, as soon as the top of the buffer is reached, the following sample is stored at its start (Figure 1.25(b)).

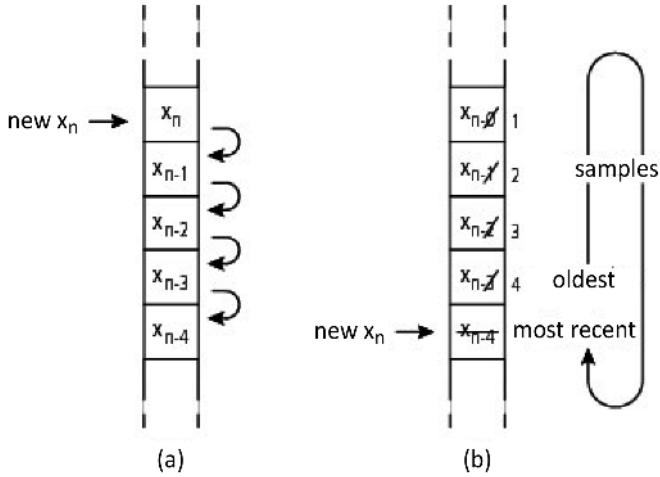

Figure 1.25. *Comparison between linear and circular addressings (from Rao (2001))*

The use domain is digital signal filtering carried out by a DSP where digital values, the results of a quantification of an analog signal, are stored in a delay line that can be implemented with a circular buffer in place of carrying out costly temporal shifts. The DSP ADSP-210xx family from Analog Devices uses this mode. One example of use is implementation of a Finite Impulse Response (FIR) described in § V3-5.2.

1.2.4.5.2. Reverse bit order addressing

Bit-reversed addressing makes it possible to manipulate materially the address without changing the source address. When the processor is set in this specific mode by the positioning of a flag (*cf.* § V3-3.1.5) in a control register, the address generator (AGU for Address Generation Unit, also called DAG for Data Address Generator or ACU for Address Computation Unit) generates bit-reversed addressing. This means that the LSbs (Least Significant bits) and MSb are exchanged, position 1 and m-2 bits are exchanged and so on (change from little-endian order to big-endian order or vice versa). This mode is used in implementation of the Fast Fourier Transform (FFT) algorithm (Cooley and Tukey 1965), an effective method for calculating a Discrete Fourier Transform (DFT), used for

filtering or spectral analysis. Remember that the FFT makes it possible to change the time domain to the frequency domain and vice versa. The problem is that the result output order differs from that of the input or vice versa. This mode makes it possible to preserve the initial order of the data by choosing out-of-order input samples to keep the output order of the data results identical to that of the input. Figure 1.26 shows the details of the calculation of a DIT (Decimal-In-Time) FFT, which is characterized by the inversion placed at the start, compared to calculation of a DIF (Decimal-In-Frequency) FFT, where the inverter is at the end. Each node represents a complex addition (in an imaginary sense). Without going into detail, note the value of the sample indices before and after inverting the order of their binary digits. W_K^{nk} is the twiddle factor, also called a Fourier coefficient or an nth root of unity. The dsPIC® microcontroller family from Microchip, DSP32xx from AT&T and DSPs from the SHARC® (DSP-21xxx) family from Analog Devices with the instruction `bitrev` that reverse the content of a register are examples of components offering it. The `mac` instruction was introduced into DSPs for this type of calculation (*cf.* § 2.8.4.2).

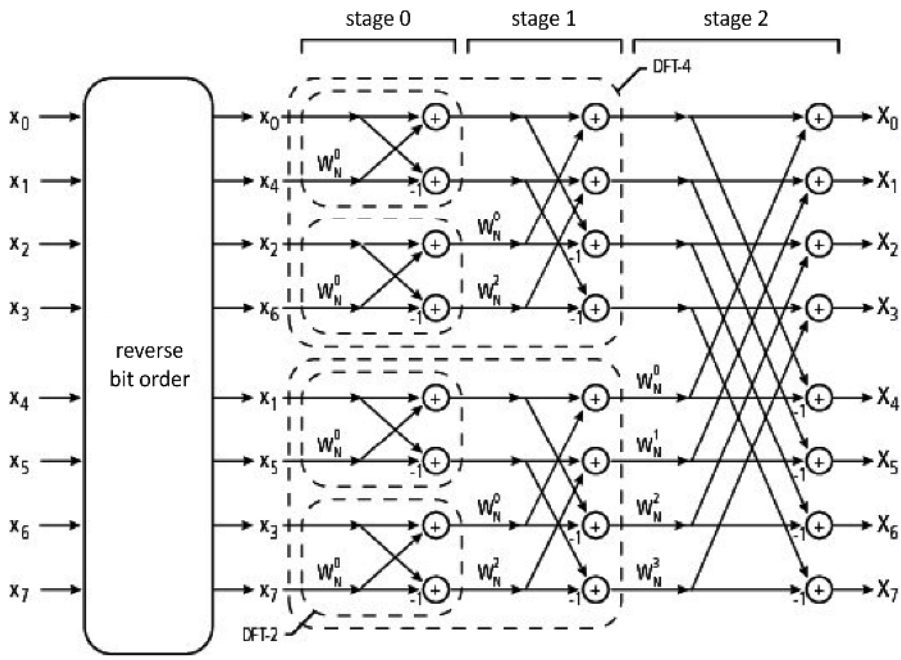

Figure 1.26. *Flow diagram of the algorithm of an 8-point FFT DIT in base 2*

To carry out this inversion of the address bit order, Reverse-Carry Arithmetic (RCA) is used. The sub-set managing the address or AGU (*cf.* § V3-3.4.4) reverses the direction of the bits retained when an increment is added to the value of an address register. Two processors that implement it are DSP32xx from AT&T and DSP56000 (Motorola 1992). The AGU also implements linear and modulo arithmetic.

1.2.4.5.3. Linear addressing

The DSP56000 uses a – perhaps poorly named – address modifier. It makes it possible to jump address at each access with a stored constant memorization in a register. The benefit is easy access to the elements of a complex data structure.

1.2.4.6. *Modes specific to the assembler*

The assembler can offer addressing modes that do not exist in the MPU. Each instruction using them will be replaced by an equivalent logical sequence. One example is symbolic addressing, which facilitates programming of a jump to a specific location in the code marked by a symbolic name called a label (*cf.* § V5-1.3.3). This mode belongs to assembly language (*cf.* § V5-1.3), unlike those seen previously in this chapter which belong to machine language. It is used to make a jump to a precise place in the code marked by this symbolic name. One example is MPU MIPS R2000/R3000 (Kane 1988).

1.2.4.7. *Obsolete modes*

The modes studied so far are those that are currently available. Some modes have been abandoned because they are complex or not useful. For example, page-zero and direct paged modes (microprocessor IM6100 from Intersil) with current memory sizes are no longer required. We also mention truncation, which consists of deleting the most significant address bits to adapt to addressing capacity in the storage hierarchy considered (Brooks 1962).

1.2.4.8. *Note*

Sequential execution of instructions in von Neumann architecture (*cf.* § V1-3.2.2) can be seen as a sequential addressing mode (source: Wikipedia).

1.2.5. *Summary on addressing*

Addressing modes have evolved to meet needs in the software industry to improve efficiency of programs and facilitate implementing functionalities of high-level languages as their control structures. It is useful to class addressing modes depending on their content, code or data. Simple code addressing modes are

Program Counter (PC)-relative absolute addressings and indirect register addressings. Sequential execution by `nop` instruction can be seen as an addressing mode. Sample data addressing modes are immediate, (direct) register, implicit and base plus offset modes. Mixed (code/data) modes are direct absolute and indexed, base plus index modes with or without offset (base plus index plus offset), scaled indexed modes, register indirect modes, indirect register modes with auto-increment, indirect memory and PC-relative modes.

Making the programmer accessible to registers that are not conventional, such as PC and SP, makes it possible to enrich addressing modes. Thus, some modes can be implemented using others, such as, for example, absolute and relative modes with respectively indirect and indexed modes.

The trend has been towards multiplying addressing modes, making it possible to adapt to complex data structures such as those of high-level languages or application domains such as digital signal processing with its operations such as convolution or correlation. This wealth of modes facilitates the life of the assembly language programmer and makes it possible for the code to be compact during compilation. The counterpart is the complexity of the CU (Control Unit), one of the defects of the CISC approach (this will be covered in a future book by the author on microprocessors). The number of possibilities of machine codes depends on the number of instructions and associated addressing modes. Therefore, MC6809 had 59 instructions and 1,464 machine codes (Motorola 1981, 1983). A reverse tendency was that of reduced instruction set architectures (RISC, this will be covered in a future book by the author on microprocessors).

1.3. Conclusion

The following chapter focuses on the instruction set for a generic microprocessor by presenting the different instruction families and extensions in this set.