# CPE 323 INSTRUCTION SET ARCHITECTURE: AN INTRODUCTION

**Aleksandar Milenkovic**

Computers cannot execute high-level language constructs like ones found in C. Rather they execute a relatively small set of machine instructions, such as addition, subtraction, Boolean operations, and data transfers. The statements from high-level language are translated into sequences of machine code instructions by a compiler. A machine code instruction is represented by a binary string which is hard to be read by humans. A human-readable form of the machine code is assembly language. However, assembly language may also include some constructs that serve only to help programmers write a better and more efficient code, faster. These constructs may translate into a sequence of machine code instructions.

Instruction set architecture, or ISA for short, refers to a portion of a computer that is visible to low-level programmers, such as assembly language programmers or compiler writers. It is an abstract view of the computer describing what it does rather than how it does. Note: computer organization describes how the computer achieves the specified functionality is out of scope in this course.

The ISA aspects include (a) class of ISA, (b) memory model, (c) addressing modes, (d) types and sizes of operands, (e) data processing and control flow operations supported by machine instructions, and (f) instruction encoding.

***Class of ISA.*** Virtually all recent instruction set architectures have a set of general-purpose registers visible to programmers. These architectures are known as *general-purpose register architectures*. Machine instructions in these architectures specify all operands in memory or general-purpose registers explicitly. In older architectures, machine instructions specified one or more operands implicitly on the stack – so-called *stack architectures*, or in the accumulator – so-called *accumulator architectures*. There are many reasons why general-purpose register architectures dominate in today's computers. Allocating frequently used variables, pointers, and intermediate results of calculations in *registers* reduces memory traffic; improves processor performance since registers are much faster than memory; and reduces code size since naming registers requires fewer bits than naming memory locations directly. A general trend in recent architectures is to increase the number of general-purpose registers.

General-purpose register architectures can be classified into *register-memory* and *load-store architectures*, depending on the location of operands used in typical arithmetic and logical instructions. In register-memory architectures arithmetic and logical machine instructions can have one or more operands in memory. In load-store architectures only load and store instructions can access memory, and common arithmetic and logical instructions are performed on operands in registers. Depending on the number of operands that can be specified by an

instruction, ISAs can be classified into *2-operand* or *3-operand architectures*. With 2-operand architectures, typical arithmetic and logical instructions specify one operand that is both a source and the destination for the operation result, and another operand is a source. For example, the arithmetic instruction *ADD R1, R2* adds the operands from the registers *R1* and *R2* and writes the result back to the register *R2*. With 3-operand architectures, instructions can specify two source operands and the result operand. For example, the arithmetic instruction *ADD R1, R2, R3* adds the operands from the registers *R1* and *R2* and writes the result to the register *R3*.

***Memory.*** Program instructions and data are stored in memory during program execution. Programmers see memory as a linear array of addressable locations as shown in Figure 1. In nearly all memory systems the smallest addressable location in memory is a single byte (8 bits). The range of memory that can be addressed by the processor is called an *address space*. For example, any program running on a 32-bit processor can address up to 4 GB ($2^{32}$ bytes) of the address space. Though the smallest addressable object is a byte, bytes are generally grouped into multi-byte objects. For example, in a 32-bit architecture we refer to 2-byte objects as half words, 4-byte objects as words, and 8-byte objects as double words. Machine instructions can directly reference and operate on words, half-words, or bytes. When referencing a multi-byte object in memory, its given address is the address of its first byte. For example, a half word located in memory at the address 8 will occupy two byte addresses 8 and 9.

Many instruction set architectures require multi-byte objects to be *aligned* to their natural boundaries. For example, if we assume a 4-byte wide memory (Figure 1), half words must begin at even addresses, while words and double words must begin at addresses divisible by 4. This kind of alignment requirement is often referred to as *hard alignment*. It should be noted that hard alignment is not an architectural requirement; rather it makes hardware implementation more practical. Even architectures that do not require hard alignment may benefit from having multi-byte objects aligned. Access to unaligned objects may require multiple accesses to memory, resulting in performance penalty. Another important issue related to memory is ordering the bytes within a larger object. There are two different conventions for byte ordering: *little-endian* and *big-endian* (Figure 2). With little-endian byte ordering, the least significant byte in a word is located at the lowest byte address, and with big-endian, the most significant byte in a word is located at the lowest byte address. For example, let us consider a 32-bit integer variable with a hexadecimal value of 0x1234ABCD stored in memory at word address 0x8. For both big-endian and little-endian byte ordering the most significant byte of the variable is 0x12 and the least significant byte is 0xCD. However, with the big-endian scheme, the most significant byte is at address 8, whereas with the little-endian scheme, the most significant byte is at address 11 (Figure 2).

***Types and Sizes of Operands***. Machine instructions operate on operands of certain types. Common types supported by ISAs include character (e.g., 8-bit ASCII or 16-bit Unicode), signed and unsigned integers, and single- and double-precision floating-point numbers. ISAs typically support several sizes for integer numbers. For example, a 32-bit architecture may include arithmetic instructions that operate on 8-bit integers, 16-bit integers (short integers), and 32-bit integers. Signed integers are represented using two's complement binary representation, while floating-point numbers rely on IEEE standard 754. Some ISAs support less frequently used data types, such as character strings, packed decimal or binary-coded decimal numbers (a decimal digit requires 4 bits, and two decimal digits are packed into a byte).

***Instructions.*** Machine instructions can be broadly classified into data processing and control flow instructions. Data processing instructions manipulate operands in registers and memory locations.

Common data processing instructions support integer arithmetic operations (e.g., add, subtract, compare, multiply, divide), logic operations (e.g., bitwise and, or, xor, nor, not); shift operations (e.g., shift to the right or left, rotate), and data transfer operations (*load* that moves a specified operand from memory to a register, *store* that moves a specified operand from register to a memory location, and *move* that transfers data between registers). If a computer is intended for applications that extensively use floating-point numbers, the ISA may support floating-point arithmetic (e.g., floating-point add, subtract, compare, multiply, divide). Several older ISAs support instructions that manipulate decimal operands and character string operands. In media and signal processing architectures we may encounter instructions that operate on more complex data types (e.g., pixels).

Machine instructions are fetched from memory and executed sequentially. Control-flow or branch instructions allow us to make decisions and change the execution flow to an instruction other than the next one in sequence. These instructions can be classified into conditional (often referred to as branches) and unconditional (often referred to as jumps), procedure calls, and procedure returns. A conditional branch instruction is defined by its outcome that determines whether the branch is taken or not taken; and by its target address that specifies the address of the following instruction in sequence to be executed, if the branch is taken. A jump instruction is defined by its target address only. Branch target addresses can be known at compile time (direct branches) or determined during program execution (indirect branches).

***Binary encoding of instructions.*** Instruction encoding defines binary representation of machine instructions. Exact encoding depends on many parameters, such as architecture type, the number of operands, the number and type of instructions, the number of general-purpose registers, operand types, and the size of address space. This representation affects not only the size of the program, but also the processor implementation. The operation and possibly the number of operands are typically specified by one instruction field called the *opcode*. For each operand the machine instruction includes an *addressing mode specifier* – a field that tells what addressing mode is used to access the operand, and one or more *address fields* that specify the operand address. Figure 3 shows a generalized instruction format for a 2-operand instruction. This approach to instruction encoding is often referred to as *variable length* – each operation can work with virtually all addressing modes that are supported by the ISA. An alternative approach is *fixed length* instruction encoding where the *opcode* is combined with addressing mode specifiers. Typically a single size is used for all instructions and this approach is used when there are a few addressing modes and operations. A third approach called *hybrid* is somewhere in between. It reduces variability in instruction encoding, but allows multiple instruction lengths. In load/store architectures all instructions except loads and stores find their operands in general-purpose registers, hence the addressing mode specifiers are not needed. Here we will assume that information about the number of operands and the corresponding addressing mode specifiers are all merged with the opcode field. Fixed length instruction formats require less complex decoding logic, resulting in faster decoding, but tend to increase the number of bits needed to encode an instruction, resulting in poor code density. Code density is an important characteristic of an instruction set, and it can be measured by the size of a program needed to complete a particular task.

***Addressing modes.*** A machine instruction can find its operand in one of three places: (a) as a part of the instruction, (b) in a general-purpose register, and (c) in memory. Operands in registers and memory can be specified directly or indirectly. Consequently, addressing modes can be broadly classified into (a) *direct* – the address field specifies the operand address and (b) *indirect* – the address field specifies a location that contains the operand address. A wide variety of

addressing modes is used in instruction set architectures, such as *immediate*, *register direct*, *register indirect*, *register indirect with displacement*, *memory direct*, and *memory indirect*, to name just a few. **Table 1** gives a list of the most common addressing modes with examples and usage. Each addressing mode is illustrated by a LOAD instruction that moves the specified operand into a general-purpose register. **Figure 4** gives a graphical illustration of these addressing modes.

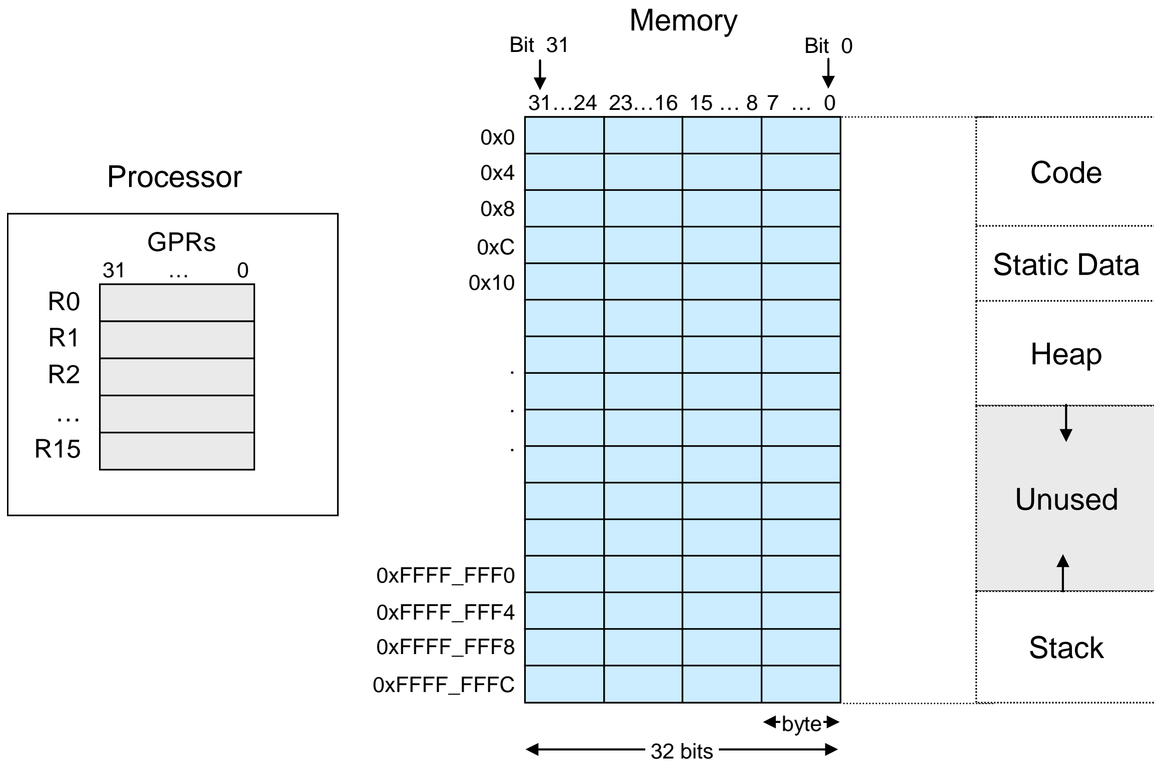



**Figure 1. Programming model: general-purpose registers (GPRs) and memory.**

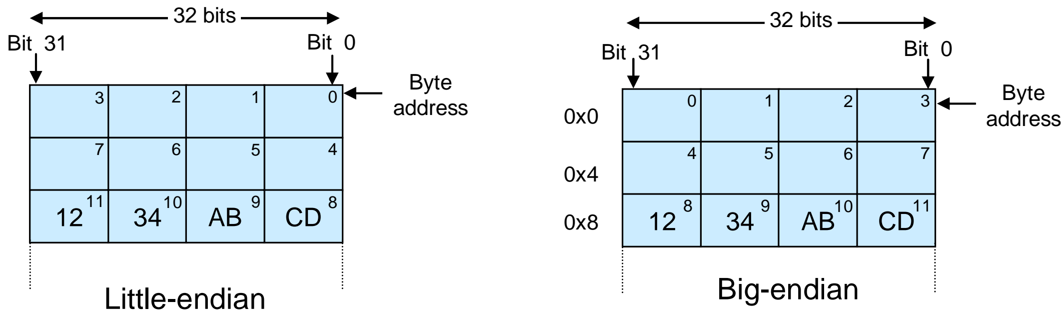



**Figure 2. Little-endian and big-endian byte ordering.**

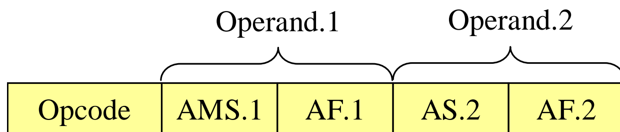
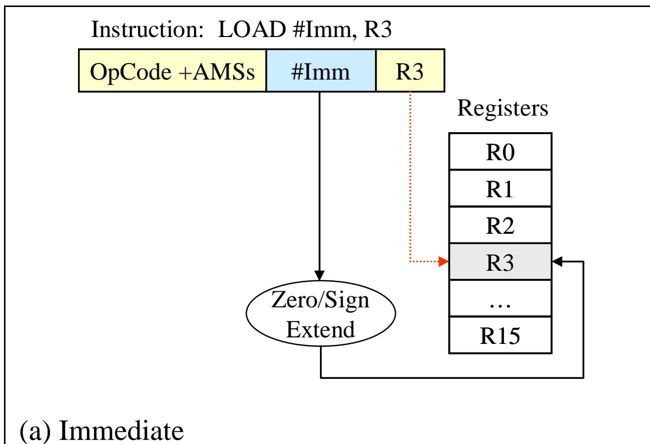
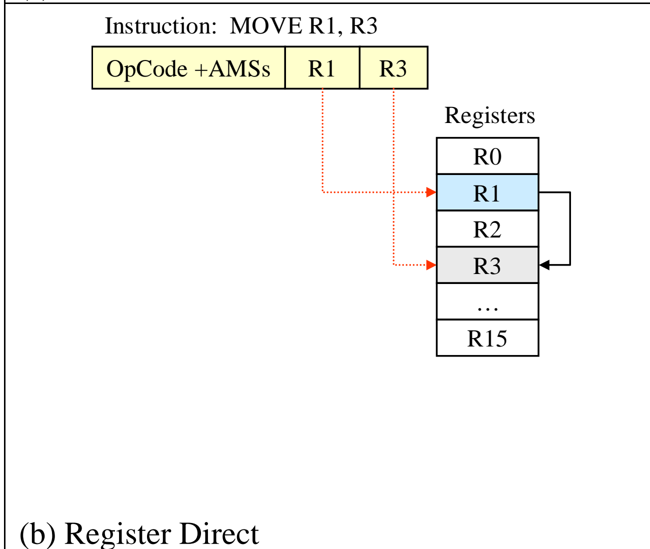


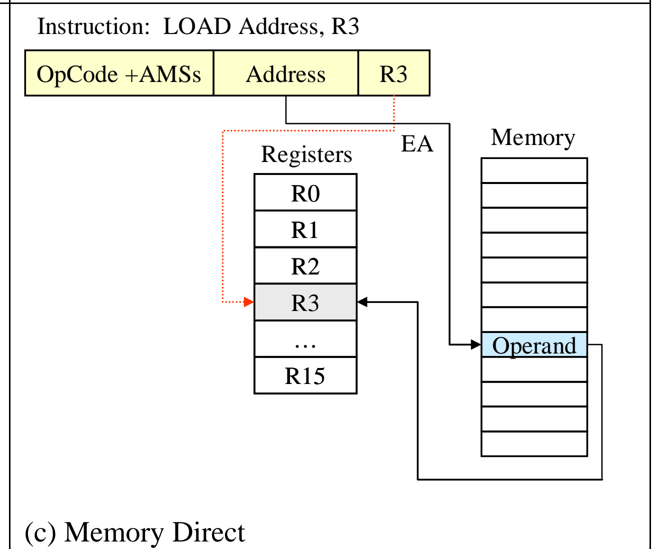

**Figure 3. A generalized 2-operand instruction format (AMS – Address Mode Specifier, AF – Address Field).**



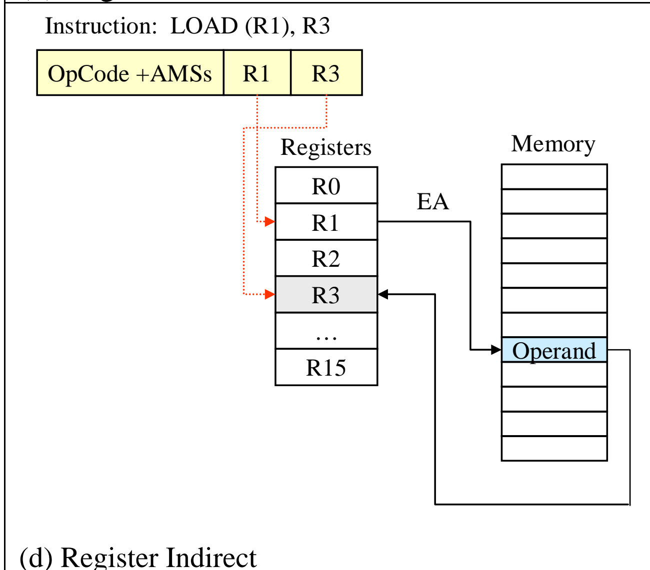
(a) Immediate



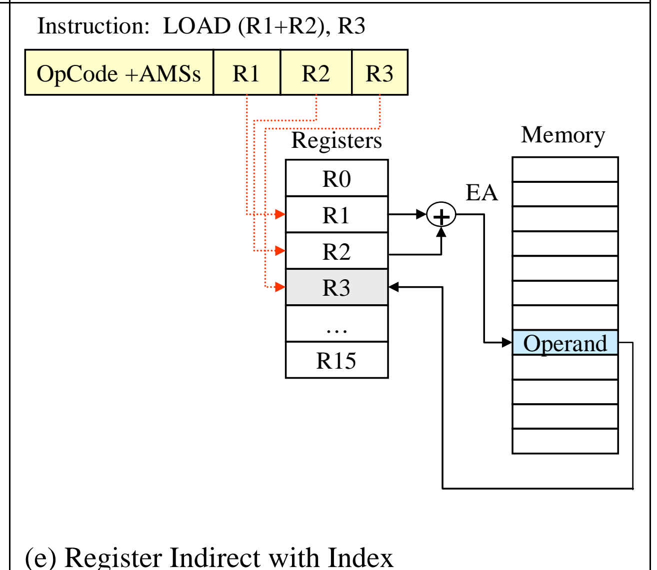
(b) Register Direct



(c) Memory Direct



(d) Register Indirect



(e) Register Indirect with Index

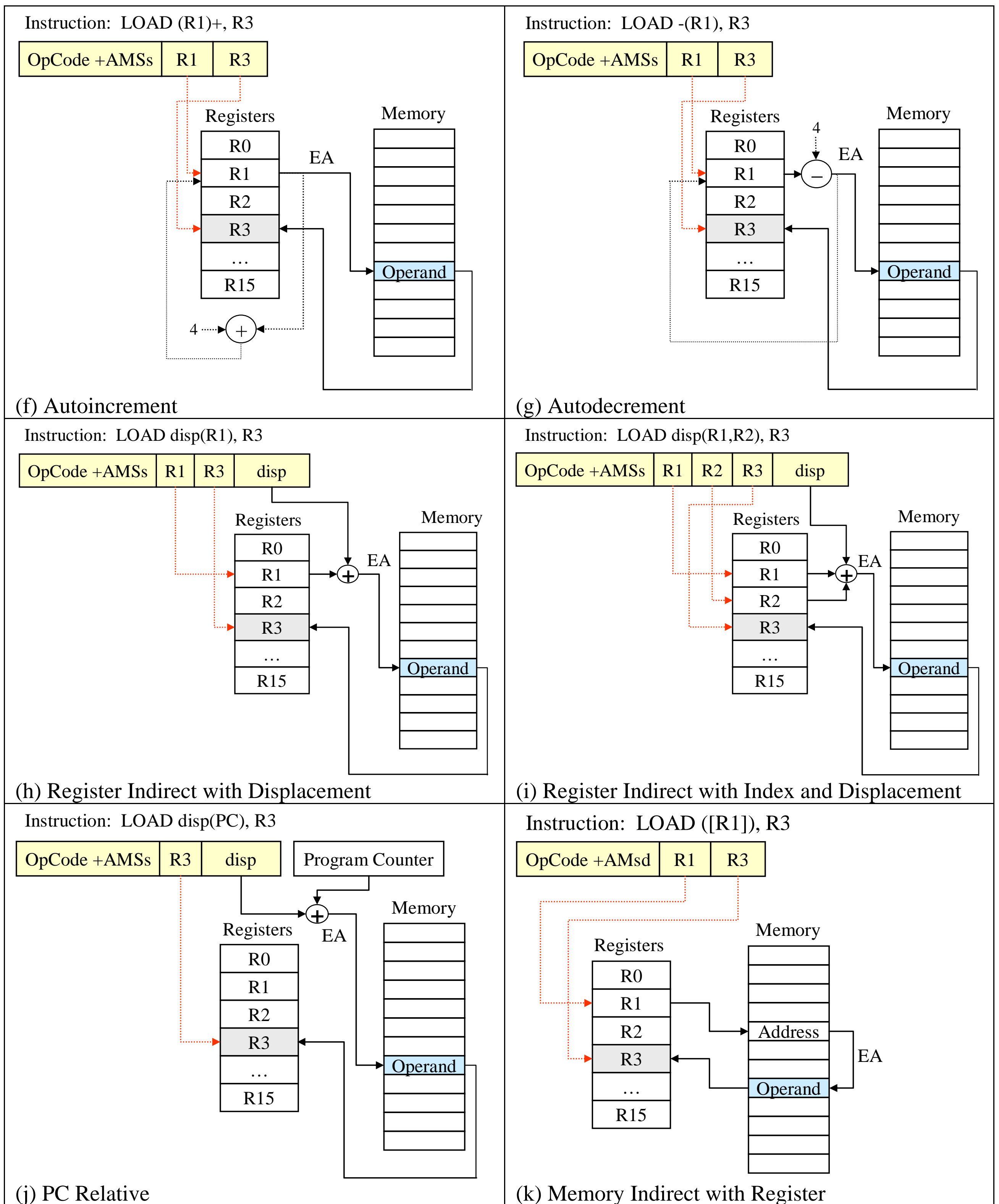



**Figure 4. Illustration of addressing modes**

**Table 1. Data addressing modes, example instructions, description, and typical use.** Legend: Register transfer language (RTL) is used to describe data transfers and operations. Square brackets [] indicate content of registers and memory locations, and backward arrows indicate data transfers from the source specified on the right-hand side of the expression to the destination specified on the left-hand side of the expression.

| Addressing Mode | Example Instruction | RTL Description | Typical Use |
|---|---|---|---|
| Immediate | `LOAD #3, R3` | `[R3] ← 0x00000003` | For constants |
| Register-direct | `LOAD R1, R3` | `[R3] ← [R1]` | When a value is in a register |
| Memory direct or Absolute | `LOAD $8000, R3` | `EA ← $00008000`<br>`[R3] ← [Mem(EA)]` | Access to static variables in memory |
| Register indirect | `LOAD (R1), R3` | `EA ← [R1]`<br>`[R3] ← [Mem(EA)]` | Access to variables in memory<br>using a pointer |
| Register indirect with index | `LOAD (R1+R2), R3` | `EA ← [R1] + [R2]`<br>`[R3] ← [Mem(EA)]` | Access to elements in an array of complex data structures (R1 points to the base, R2 is stride) |
| Register indirect with scaled index | `LOAD (R1+R2*4),R3` | `EA ← [R1] + [R2]*4`<br>`[R3] ← [Mem(EA)]` | Access to elements in an array of complex data structures (R3 points to the base, R2 is index) |
| Autoincrement | `LOAD (R1)+, R3` | `EA ← [R1]; [R1] ← [R1] + 4`<br>`[R3] ← [Mem(EA)]` | Access to elements of an array in a loop;<br>Access to stack (push/pop) |
| Autodecrement | `LOAD -(R1), R3` | `[R1] ← [R1] - 4; EA ← [R1]`<br>`[R3] ← [Mem(EA)]` | Access to elements of an array in a loop;<br>Access to stack (push/pop) |
| Register indirect with displacement | `LOAD 0x100(R1), R3` | `EA ← [R1] + 0x0100`<br>`[R3] ← [Mem(EA)]` | Access to local variables |
| Register indirect with scaled index and displacement | `LOAD 0x100(R1+R2*4), R3` | `EA ← 0x0100+[R1]+[R2]*4`<br>`[R3] ← [Mem(EA)]` | Access to arrays allocated on the stack |
| PC relative | `LOAD 0x100(PC), R3` | `EA ← 0x0100+[PC]`<br>`[R3] ← [Mem(EA)]` | Branches, Jumps, Procedure calls<br>Static data |
| PC relative with index | `LOAD (PC+R2),R3` | `EA ← [PC]+[R2]`<br>`[R3] ← [Mem(EA)]` | Branches, Jumps, Procedure calls<br>Static data |
| PC relative with scaled index and displacement | `LOAD 0x100(PC+R2*4), R3` | `EA ← 0x0100+[PC]+[R2]*4`<br>`[R3] ← [Mem(EA)]` | Branches, Jumps, Procedure calls<br>Static data |