Addressing mode

From Wikipedia, the free encyclopedia
(Redirected from Addressing modes)

Addressing modes are an aspect of the instruction set architecture in most central processing unit (CPU) designs. The various addressing modes that are defined in a given instruction set architecture define how machine language instructions in that architecture identify the operand (or operands) of each instruction. An addressing mode specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction or elsewhere.

In computer programming, addressing modes are primarily of interest to compiler writers and to those who write code directly in assembly language.

Contents

Caveats [edit]

Note that there is no generally accepted way of naming the various addressing modes. In particular, different authors and computer manufacturers may give different names to the same addressing mode, or the same names to different addressing modes. Furthermore, an addressing mode which, in one given architecture, is treated as a single addressing mode may represent functionality that, in another architecture, is covered by two or more addressing modes. For example, some complex instruction set computer (CISC) computer architectures, such as the Digital Equipment Corporation (DEC) VAX, treat registers and literal/immediate constants as just another addressing mode. Others, such as the IBM System/390 and most reduced instruction set computer (RISC) designs, encode this information within the instruction. Thus, the latter machines have three distinct instruction codes for copying one register to another, copying a literal constant into a register, and copying the contents of a memory location into a register, while the VAX has only a single "MOV" instruction.

The term "addressing mode" is itself subject to different interpretations: either "memory address calculation mode" or "operand accessing mode". Under the first interpretation instructions that do not read from memory or write to memory (such as "add literal to register") are considered not to have an "addressing mode". The second interpretation allows for machines such as VAX which use operand mode bits to allow for a literal operand. Only the first interpretation applies to instructions such as "load effective address".

The addressing modes listed below are divided into code addressing and data addressing. Most computer architectures maintain this distinction, but there are, or have been, some architectures which allow (almost) all addressing modes to be used in any context.

The instructions shown below are purely representative in order to illustrate the addressing modes, and do not necessarily reflect the mnemonics used by any particular computer.

How many addressing modes? [edit]

Different computer architectures vary greatly as to the number of addressing modes they provide. At the cost of a few extra instructions, and perhaps an extra register, it is normally possible to use the simpler addressing modes instead of the more complicated modes. It has proven[*citation needed*] much easier to design pipelined CPUs if the only addressing modes available are simple ones.

Most RISC machines have only about five simple addressing modes, while CISC machines such as the DEC VAX supermini have over a dozen addressing modes, some of which are quite complicated. The IBM System/360 mainframe had only three addressing modes; a few more have been added for the System/390.

When there are only a few addressing modes, the particular addressing mode required is usually encoded within the instruction code (e.g. IBM System/390, most RISC). But when there are lots of addressing modes, a specific field is often set aside in the instruction to specify the addressing mode. The DEC VAX allowed multiple memory operands for almost all instructions, and so reserved the first few bits of each operand specifier to indicate the addressing mode for that particular operand.

Even on a computer with many addressing modes, measurements of actual programs[*citation needed*] indicate that the simple addressing modes listed below account for some 90% or more of all addressing modes used. Since most such measurements are based on code generated from high-level languages by compilers, this reflects to some extent the limitations of the compilers being used.

Useful side effect [edit]

Some processors, such as Intel x86 and the IBM/390, have a **Load effective address** instruction. This performs a calculation of the effective operand address, but instead of acting on that memory location, it loads the address that would have been accessed into a register. This can be useful when passing the address of an array element to a subroutine. It may also be a slightly sneaky way of doing more calculation than normal in one instruction; for example, using such an instruction with the addressing mode "base+index+offset" allows one to add two registers and a constant together in one instruction.

Simple addressing modes for code [edit]

Absolute [edit]

```
+----+--------------------------------+
|jump|           address              |
+----+--------------------------------+

(Effective PC address = address)
```

The effective address for an absolute instruction address is the address parameter itself with no modifications.

PC-relative [edit]

```
+------+-----+-----+----------------+
|jumpEQ| reg1| reg2|         offset |    jump relative if reg1=reg2
+------+-----+-----+----------------+

(Effective PC address = next instruction address + offset, offset may be negative)
```

The effective address for a PC-relative instruction address is the offset parameter added to the address of the next instruction. This offset is usually signed to allow reference to code both before and after the instruction.

This is particularly useful in connection with conditional jumps, because typical jumps are to nearby instructions (in a high-level language most **if** or **while** statements are reasonably short). Measurements of actual programs suggest that an 8 or 10 bit offset is large enough for some 90% of conditional jumps[*citation needed*].

Another advantage of program-relative addressing is that the code may be position-independent, i.e. it can be loaded anywhere in memory without the need to adjust any addresses.

Some versions of this addressing mode only refer to one register ("jump unless reg1==0") or no registers, implicitly referring to some previously-set bit in the status register.

Register indirect [edit]

```
+-------+-----+
|jumpVia| reg |
+-------+-----+

(Effective PC address = contents of register 'reg')
```

The effective address for a Register indirect instruction is the address in the specified register. For example, (A7) to access the content of address register A7.

The effect is to transfer control to the instruction whose address is in the specified register.

Many RISC machines have a subroutine call instruction that places the return address in an address register – the register indirect addressing mode is used to return from that subroutine call.

Sequential addressing modes [edit]

sequential execution [edit]

```
+------+
| nop  |              execute the following instruction
+------+

(Effective PC address = next instruction address)
```

The CPU, after executing a sequential instruction, immediately executes the following instruction.

Sequential execution is not considered to be an addressing mode on some computers.

Most instructions on most CPU architectures are sequential instructions. Because most instructions are sequential instructions, CPU designers often add features that deliberately sacrifice performance on the other instructions – branch instructions – in order to make these sequential instructions run faster.

Conditional branches load the PC with one of 2 possible results, depending on the condition – most CPU architectures use some other addressing mode for the "taken" branch, and sequential execution for the "not taken" branch.

Many features in modern CPUs – instruction prefetch and more complex pipelineing, Out-of-order execution, etc. – maintain the illusion that each instruction finishes before the next one begins, giving the same final results, even though that's not exactly what happens internally.

Each "basic block" of such sequential instructions exhibits both temporal and spatial locality of reference.

CPUs that do not use sequential execution are extremely rare – they include some drum memory computers and the RTX 32P, which has no program counter.[1]

Conditional execution [edit]

Some computer architectures (e.g. ARM) have conditional instructions which can in some cases obviate the need for conditional branches and avoid flushing the instruction pipeline. An instruction such as a 'compare' is used to set a condition code, and subsequent instructions include a test on that condition code to see whether they are obeyed or ignored.

skip [edit]

```
+------+-----+-----+
|skipEQ| reg1| reg2|            skip the following instruction if reg1=reg2
+------+-----+-----+

(Effective PC address = next instruction address + 1)
```

Skip addressing may be considered a special kind of PC-relative addressing mode with a fixed "+1" offset. Like PC-relative addressing, some CPUs have versions of this addressing mode that only refer to one register ("skip if reg1==0") or no registers, implicitly referring to some previously-set bit in the status register. Other CPUs have a version that selects a specific bit in a specific byte to test ("skip if bit 7 of reg12 is 0").

Unlike all other conditional branches, a "skip" instruction never needs to flush the instruction pipeline.

Simple addressing modes for data [edit]

Register [edit]

```
+------+-----+-----+-----+
| mul  | reg1| reg2| reg3|      reg1 := reg2 * reg3;
+------+-----+-----+-----+
```

This "addressing mode" does not have an effective address and is not considered to be an addressing mode on some computers.

In this example, all the operands are in registers, and the result is placed in a register.

Base plus offset, and variations [edit]

```
+------+-----+-----+----------------+
| load | reg | base|     offset     |  reg := RAM[base + offset]
+------+-----+-----+----------------+

(Effective address = offset + contents of specified base register)
```

The offset is usually a signed 16-bit value (though the 80386 expanded it to 32 bits).

If the offset is zero, this becomes an example of *register indirect* addressing; the effective address is just the value in the base register.

On many RISC machines, register 0 is fixed at the value zero. If register 0 is used as the base register, this becomes an example of *absolute addressing*. However, only a small portion of memory can be accessed (64 kilobytes, if the offset is 16 bits).

The 16-bit offset may seem very small in relation to the size of current computer memories (which is why the 80386 expanded it to 32-bit). It could be worse: IBM System/360 mainframes only have an unsigned 12-bit offset. However, the principle of locality of reference applies: over a short time span, most of the data items a program wants to access are fairly close to each other.

This addressing mode is closely related to the indexed absolute addressing mode.

Example 1: Within a subroutine a programmer will mainly be interested in the parameters and the local variables, which will rarely exceed 64 KB, for which one base register (the frame pointer) suffices. If this routine is a class method in an object-oriented language, then a second base register is needed which points at the attributes for the current object (**this** or **self** in some high level languages).

Example 2: If the base register contains the address of a composite type (a record or structure), the offset can be used to select a field from that record (most records/structures are less than 32 kB in size).

Immediate/literal [edit]

```
+------+-----+-----+----------------+
| add  | reg1| reg2|    constant    |    reg1 := reg2 + constant;
+------+-----+-----+----------------+
```

This "addressing mode" does not have an effective address, and is not considered to be an addressing mode on some computers.

The constant might be signed or unsigned. For example move.l #$FEEDABBA, D0 to move the immediate hex value of "FEEDABBA" into register D0.

Instead of using an operand from memory, the value of the operand is held within the instruction itself. On the DEC VAX machine, the literal operand sizes could be 6, 8, 16, or 32 bits long.

Andrew Tanenbaum showed that 98% of all the constants in a program would fit in 13 bits (see RISC design philosophy).

Implicit [edit]

```
+-----------------+
| clear carry bit |
+-----------------+
```

The implied addressing mode[1], also called the implicit addressing mode[2], does not explicitly specify an effective address for either the source or the destination (or sometimes both).

Either the source (if any) or destination effective address (or sometimes both) is implied by the opcode.

Implied addressing was quite common on older computers (up to mid-1970s). Such computers typically had only a single register in which arithmetic could be performed – the accumulator. Such accumulator machines implicitly reference that accumulator in almost every instruction. For example, the operation <a := b + c;> can be done using the sequence <load b; add c; store a;> – the destination (the accumulator) is implied in every "load" and "add" instruction; the source (the accumulator) is implied in every "store" instruction.

Later computers generally had more than one general purpose register or RAM location which could be the source or destination or both for arithmetic – and so later computers need some other addressing mode to specify the source and destination of arithmetic.

Many computers (such as x86 and AVR) have one special-purpose register called the stack pointer which is implicitly incremented or decremented when pushing or popping data from the stack, and the source or destination effective address is (implicitly) the address stored in that stack pointer.

Most 32-bit computers (such as ARM and PowerPC) have more than one register which could be used as a stack pointer – and so use the "register autoincrement indirect" addressing mode to specify which of those registers should be used when pushing or popping data from a stack.

Some current computer architectures (e.g. IBM/390 and Intel Pentium) contain some instructions with implicit operands in order to maintain backwards compatibility with earlier designs.

On many computers, instructions that flip the user/system mode bit, the interrupt-enable bit, etc. implicitly specify the special register that holds those bits. This simplifies the hardware necessary to trap those instructions in order to meet the Popek and Goldberg virtualization requirements – on such a system, the trap logic does not need to look at any operand (or at the final effective address), but only at the opcode.

A few CPUs have been designed where every operand is always implicitly specified in every instruction – zero-operand CPUs.

Other addressing modes for code or data [edit]

Absolute/Direct [edit]

```
+------+-----+--------------------------------------+
| load | reg |          address                     |
+------+-----+--------------------------------------+

(Effective address = address as given in instruction)
```

This requires space in an instruction for quite a large address. It is often available on CISC machines which have variable-length instructions, such as x86.

Some RISC machines have a special *Load Upper Literal* instruction which places a 16-bit constant in the top half of a register. An *OR literal* instruction can be used to insert a 16-bit constant in the lower half of that register, so that a full 32-bit address can then be used via the register-indirect addressing mode, which itself is provided as "base-plus-offset" with an offset of 0.

Indexed absolute [edit]

```
+------+-----+-----+---------------------------------+
| load | reg |index|          address                |
+------+-----+-----+---------------------------------+

(Effective address = address + contents of specified index register)
```

This also requires space in an instruction for quite a large address. The address could be the start of an array or vector, and the index could select the particular array element required. The processor may scale the index register to allow for the size of each array element.

Note that this is more or less the same as base-plus-offset addressing mode, except that the offset in this case is large enough to address any memory location.

Example 1: Within a subroutine, a programmer may define a string as a local constant or a static variable. The address of the string is stored in the literal address in the instruction. The offset – which character of the string to use on this iteration of a loop – is stored in the index register.

Example 2: A programmer may define several large arrays as globals or as class variables. The start of the array is stored in the literal address (perhaps modified at program-load time by a relocating loader) of the instruction that references it. The offset – which item from the array to use on this iteration of a loop – is stored in the index register. Often the instructions in a loop re-use the same register for the loop counter and the offsets of several arrays.

Base plus index [edit]

```
+------+-----+-----+-----+
| load | reg | base|index|
+------+-----+-----+-----+

(Effective address = contents of specified base register + contents of specified index register)
```

The base register could contain the start address of an array or vector, and the index could select the particular array element required. The processor may scale the index register to allow for the size of each array element. This could be used for accessing elements of an array passed as a parameter.

Base plus index plus offset [edit]

```
+------+-----+-----+-----+-----------------+
| load | reg | base|index|          offset |
+------+-----+-----+-----+-----------------+

(Effective address = offset + contents of specified base register + contents of specified index register
```

The base register could contain the start address of an array or vector of records, the index could select the particular record required, and the offset could select a field within that record. The processor may scale the index register to allow for the size of each array element.

Scaled [edit]

```
+------+-----+-----+-----+
| load | reg | base|index|
+------+-----+-----+-----+

(Effective address = contents of specified base register + scaled contents of specified index register)
```

The base register could contain the start address of an array or vector, and the index could contain the number of the particular array element required.

This addressing mode dynamically scales the value in the index register to allow for the size of each array element, e.g. if the array elements are double precision floating-point numbers occupying 8 bytes each then the value in the index register is multiplied by 8 before being used in the effective address calculation. The scale factor is normally restricted to being a power of two, so that shifting rather than multiplication can be used.

Register indirect [edit]

```
+------+-----+-----+
| load | reg | base|
+------+-----+-----+

(Effective address = contents of base register)
```

A few computers have this as a distinct addressing mode. Many computers just use *base plus offset* with an offset value of 0. For example, (A7)

Register autoincrement indirect [edit]

```
+------+-----+-------+
| load | reg | base  |
+------+-----+-------+

(Effective address = contents of base register)
```

After determining the effective address, the value in the base register is incremented by the size of the data item that is to be accessed. For example, (A7)+ would access the content of the address register A7, then increase the address pointer of A7 by 1 (usually 1 word).

Within a loop, this addressing mode can be used to step through all the elements of an array or vector. A stack can be implemented by using this mode in conjunction with the next addressing mode (autodecrement).

In high-level languages it is often thought to be a good idea that functions which return a result should not have side effects (lack of side effects makes program understanding and validation much easier). This addressing mode has a side effect in that the base register is altered. If the subsequent memory access causes an error (e.g. page fault, bus error, address error) leading to an interrupt, then restarting the instruction becomes much more problematic since one or more registers may need to be set back to the state they were in before the instruction originally started.

There have been at least two computer architectures which have had implementation problems with regard to recovery from interrupts when this addressing mode is used:

- Motorola 68000(address is represented in 24 bits). Could have one or two autoincrement register operands. The 68010+ resolved the problem by saving the processor's internal state on bus or address errors.
- DEC VAX. Could have up to 6 autoincrement register operands. Each operand access could cause two page faults (if operands happened to straddle a page boundary). Of course the instruction itself could be over 50 bytes long and might straddle a page boundary as well!

Autodecrement register indirect [edit]

```
+------+-----+-----+
| load | reg | base|
+------+-----+-----+

(Effective address = new contents of base register)
```

Before determining the effective address, the value in the base register is decremented by the size of the data item which is to be accessed.

Within a loop, this addressing mode can be used to step backwards through all the elements of an array or vector. A stack can be implemented by using this mode in conjunction with the previous addressing mode (autoincrement).

See the discussion of side-effects under the autoincrement addressing mode.

Memory indirect [edit]

Any of the addressing modes mentioned in this article could have an extra bit to indicate indirect addressing, i.e. the address calculated using some mode is in fact the address of a location (typically a complete word) which contains the actual effective address.

Indirect addressing may be used for code or data. It can make implementation of *pointers* or *references* or *handles* **much easier**, and can also make it easier to call subroutines which are not otherwise addressable. Indirect addressing does carry a performance penalty due to the extra memory access involved.

Some early minicomputers (e.g. DEC PDP-8, Data General Nova) had only a few registers and only a limited addressing range (8 bits). Hence the use of memory indirect addressing was almost the only way of referring to any significant amount of memory.

PC-relative [edit]

```
+------+------+---------+-----------------+
| load | reg1 | base=PC |     offset      |   reg1 := RAM[PC + offset]
+------+------+---------+-----------------+

(Effective address = PC + offset)
```

The PC-relative addressing mode is used to load a register from a "constant" stored in program memory a short distance away from the current instruction. It can be seen as a special case of the "base plus offset" addressing mode, one that selects the program counter (PC) as the "base register".

There are a few CPUs that support PC-relative data loads. Such CPUs include:

The x86-64 architecture supports "RIP-relative" addressing, which uses the 64-bit instruction pointer RIP as a base register. This encourages position-independent code.

The ARM architecture supports PC-relative addressing.

When this addressing mode is used, the compiler typically places the constants in a literal pool immediately before or immediately after the subroutine that uses them, to prevent accidentally executing those constants as instructions.

This addressing mode, which always modifies a data register and then sequentially falls through to execute the next instruction (the effective address points to data), should not be confused with "PC-relative branch" which does not modify any data register, but instead branches to some other instruction at the given offset (the effective address points to an executable instruction).

Obsolete addressing modes [edit]

The addressing modes listed here were used in the 1950–1980 time frame, but are no longer available on most current computers. This list is by no

means complete; there have been many other interesting and peculiar addressing modes used from time to time, e.g. absolute-plus-logical-OR of two or three index registers.[*citation needed*]

Multi-level memory indirect [edit]

If the word size is larger than the address then the word referenced for memory-indirect addressing could itself have an indirect flag set to indicate another memory indirect cycle. Care is needed to ensure that a chain of indirect addresses does not refer to itself; if it did, you could get an infinite loop while trying to resolve an address.

The DEC PDP-10 computer with 18-bit addresses and 36-bit words allowed multi-level indirect addressing with the possibility of using an index register at each stage as well.

Memory-mapped registers [edit]

On some computers, the registers were regarded as occupying the first 8 or 16 words of memory (e.g. ICL 1900, DEC PDP-10). This meant that there was no need for a separate "Add register to register" instruction — you could just use the "add memory to register" instruction.

In the case of early models of the PDP-10, which did not have any cache memory, you could actually load a tight inner loop into the first few words of memory (the fast registers in fact), and have it run much faster than if it would have in magnetic core memory.

Later models of the DEC PDP-11 series mapped the registers onto addresses in the input/output area, but this was primarily intended to allow remote diagnostics. Confusingly, the 16-bit registers were mapped onto consecutive 8-bit byte addresses.

Memory indirect, autoincrement [edit]

On some early minicomputers (e.g. DEC PDP-8, Data General Nova), there were typically 16 special memory locations.[*citation needed*] When accessed via memory indirect addressing, 8 would automatically increment after use and 8 would automatically decrement after use. This made it very easy to step through memory in loops without using any registers.

Zero page [edit]

The Motorola 6800 family and MOS Technology 6502 family of processors were a register poor series of CISC microprocessors. Arithmetic and logical instructions were mostly performed against values in memory as opposed to internal registers. As a result, instructions were generally required to include a two byte (16-bit) location to memory. Given that opcodes on these processors were only one byte (8-bit) in length, memory addresses could make up a significant part of code size.

Designers of these processors included a partial remedy known as "zero page" addressing. The initial 256 bytes of memory ($0000 - $00FF; a.k.a., page "0") could be accessed using a one byte absolute or indexed memory address. This reduced instruction execution time by one clock cycle and instruction length by one byte. By storing often used data in this region, programs could be made smaller and faster.

As a result, the zero page was used similar to a register file. On many systems, however, this resulted in high utilization of the zero page memory area by the operating system and user programs. This limited its use since free space was limited.

Direct page [edit]

The zero page address mode was enhanced in several descendants of the MOS Technology 6502, including the WDC 65816, the MOS Technology 65CE02, and the Motorola 6809. The new mode, known as "direct page" addressing, added the ability to move the 256 byte zero page memory window from the start of memory (offset address $0000) to a new location within the first 64KB of memory.

The MOS 65CE02 allowed the direct page to be moved to any 256 byte boundary within the first 64KB of memory by storing an 8-bit offset value in the new B (block) register, equivalent to the 8-bit 6809 DP (direct page) register. The WDC 65816 went a step further and allowed the direct page to be moved to any location within the first 64KB of memory by storing a 16-bit offset value in the new D (direct) register.

As a result, a greater number of programs were able to utilize the enhanced direct page addressing mode versus legacy processors that only included the zero page addressing mode.

Scaled index with bounds checking [edit]

This is similar to scaled index addressing, except that the instruction has two extra operands (typically constants), and the hardware would check that the index value was between these bounds.

Another variation uses vector descriptors to hold the bounds; this makes it easy to implement dynamically allocated arrays and still have full bounds checking.

Register indirect to byte within word [edit]

The DEC PDP-10 computer used 36-bit words. It had a special addressing mode which allowed memory to be treated as a sequence of bytes (bytes could be any size from 1 bit to 36 bits). A one-word sequence descriptor held the current word address within the sequence, a bit position within a word, and the size of each byte.

Instructions existed to load and store bytes via this descriptor, and to increment the descriptor to point at the next byte (bytes were not split across word boundaries). Much DEC software used five 7-bit bytes per word (plain ASCII characters), with 1 bit unused per word. Implementations of C had to use four 9-bit bytes per word, since C assumes that you can access every bit of memory by accessing consecutive bytes

Index next instruction [edit]

The Elliott 503, the Elliott 803, and the Apollo Guidance Computer only used absolute addressing, and did not have any index registers. Thus, indirect jumps, or jumps through registers, were not supported in the instruction set. Instead, it could be instructed to *add the contents of the current memory word to the next instruction*. Adding a small value to the next instruction to be executed could, for example, change a `JUMP 0` into a `JUMP 20`, thus creating the effect of an indirect jump via self-modifying code.[3][4]

External links [edit]

- Addressing modes in assembly language

- Addresssing modes

v · d · e	CPU technologies	
Architecture	ISA : CISC · EDGE · EPIC · MISC · OISC · RISC · VLIW · ZISC · Harvard architecture · Von Neumann architecture · 32-bit · 64-bit · 128-bit	
Parallelism	**Pipeline**	Instruction pipelining · In-Order & Out-of-Order execution · Register renaming · Speculative execution
	Level	Bit · Instruction · Superscalar · Data · Task
	Threads	Multithreading · Simultaneous multithreading · Hyperthreading · Superthreading
	Flynn's taxonomy	SISD · SIMD · MISD · MIMD
Types	Digital signal processor · Microcontroller · System-on-a-chip · Vector processor	
Components	Arithmetic logic unit (ALU) · Barrel shifter · Floating point unit (FPU) · Back side bus · Demultiplexer · Registers · Memory management unit (MMU) · Multiplexer · Translation lookaside buffer (TLB) · Cache	
Power management	APM · ACPI (states) · Dynamic frequency scaling · Dynamic voltage scaling · Clock gating	

Categories: Computer architecture | Machine code | Assembly languages