# Three-address code

In computer science, **three-address code**[1] (often abbreviated to TAC or 3AC) is an intermediate code used by optimizing compilers to aid in the implementation of code-improving transformations. Each TAC instruction has at most three operands and is typically a combination of assignment and a binary operator. For example, `t1 := t2 + t3`. The name derives from the use of three operands in these statements even though instructions with fewer operands may occur.

Since three-address code is used as an intermediate language within compilers, the operands will most likely not be concrete memory addresses or processor registers, but rather symbolic addresses that will be translated into actual addresses during register allocation. It is also not uncommon that operand names are numbered sequentially since three-address code is typically generated by the compiler.

A refinement of three-address code is A-normal form (ANF).

## Examples

In three-address code, this would be broken down into several separate instructions. These instructions translate more easily to assembly language. It is also easier to detect common sub-expressions for shortening the code. In the following example, one calculation is composed of several smaller ones:

```
# Calculate one solution to the [[quadratic
equation]].
x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
```

```
t1 := b * b
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x := t9
```

Three-address code may have conditional and unconditional jumps and methods of accessing memory. It may also have methods of calling functions, or it may reduce these to jumps. In this way, three-address code may be useful in control-flow analysis. In the following C-like example, a loop stores the squares of the numbers between 0 and 9:

```
...

for (i = 0; i < 10; ++i) {
    b[i] = i*i;
}

...
```

```
    t1 := 0                 ; initialize i
L1: if t1 >= 10 goto L2     ; conditional jump
    t2 := t1 * t1           ; square of i
    t3 := t1 * 4            ; word-align address
    t4 := b + t3            ; address to store i*i
    *t4 := t2               ; store through pointer
    t1 := t1 + 1            ; increase i
    goto L1                 ; repeat loop
L2:
```

## See also

- Intermediate language
- Reduced instruction set computer
- Static single assignment form (SSA)

# References


1. V., Aho, Alfred (1986). *Compilers, principles, techniques, and tools* (https://www.worldcat.org/oclc/12285707) Sethi, Ravi., Ullman, Jeffrey D., 1942-. Reading, Mass.: Addison-Wesley Pub. Co. p. 466. ISBN 0201100886. OCLC 12285707 (https://www.worldcat.org/oclc/12285707)