

# Intermediate Code

$x$  and  $y$  can either be variables or constants, used as operands to instructions.  $t$  is a destination variable.

$t = x$	Simple Assignment
$t = \text{load from } x$ $\text{store } x \text{ to addr } y$	Load from and store to memory
$\text{declare var } t$ $\text{allocate } x \text{ bytes for var } y$	Declare a new variable $t$ . Allocate memory for $y$ (used for allocating aggregate data structures e.g. arrays).
$\text{reference var } x$	A non-executable instruction used internally to mark a variable as live at this program point.
$t = \&x$	Address-of operator
$t = \sim x$	Bitwise NOT
$t = !x$	Logical NOT
$t = x * y$	Multiplication
$t = x / y$	Division
$t = x \% y$	Modulus
$t = x + y$	Addition
$t = x - y$	Subtraction
$t = x \ll y$	Left-shift
$t = x \gg y$	Right-shift (signed-extending for signed $x$ , zero-filling for unsigned $x$ )
$t = x \& y$	Bitwise AND
$t = x   y$	Bitwise OR
$t = x \wedge y$	Bitwise XOR
$t = x \&\& y$	Logical AND
$t = x    y$	Logical OR
$t = x < y$	Less-than comparison
$t = x > y$	Greater-than comparison
$t = x \leq y$	Less-than or equal comparison
$t = x \geq y$	Greater-than or equal comparison
$t = x == y$	Equality comparison
$t = x != y$	Not equal comparison
$t = \text{call } f(p_1, p_2, \dots)$	Call function $f$ with parameters $p_i$ (either variables or constants)

<code>tail-call f(p<sub>1</sub>, p<sub>2</sub>, ...)</code>	Call function <code>f</code> and return the result from the current function
<code>return [x]</code>	Return from the current function. The return value <code>x</code> is optional.
<code>label &lt;l&gt;</code>	Attach a label to the current program point (immediately before the next instruction).
<code>br &lt;l&gt;</code>	Unconditional branch
<code>br &lt;l&gt; if x == y</code>	Conditional branch; executed if operands are equal.
<code>br &lt;l&gt; if x != y</code>	Conditional branch; executed if operands are not equal.
<code>t = &amp;&lt;sid&gt;</code>	Static address of the string literal with id <code>&lt;sid&gt;</code>
<code>t = (i8 → i16) x</code> <code>t = (i8 → u16) x</code> <code>t = (u8 → u16) x</code> <code>t = (u8 → u16) x</code>	Char promotions
<code>t = (i16 → i32) x</code> <code>t = (u16 → i32) x</code>	Promotions to signed integer
<code>t = (i16 → u32) x</code> <code>t = (u16 → u32) x</code> <code>t = (i32 → u32) x</code>	Promotions to unsigned integer
<code>t = (i32 → i64) x</code> <code>t = (u32 → i64) x</code>	Promotions to signed long
<code>t = (i32 → u64) x</code> <code>t = (u32 → u64) x</code> <code>t = (i64 → u64) x</code>	Promotions to unsigned long
<code>t = (u32 → f32) x</code> <code>t = (i32 → f32) x</code> <code>t = (u64 → f32) x</code> <code>t = (i64 → f32) x</code>	Integer to float conversions
<code>t = (u32 → f64) x</code> <code>t = (i32 → f64) x</code> <code>t = (u64 → f64) x</code> <code>t = (i64 → f64) x</code>	Integer to double conversions
<code>t = (f32 → f64) x</code>	Float to double promotion
<code>t = (f64 → i32) x</code>	Double to int conversion
<code>t = (i32 → i8) x</code> <code>t = (u32 → i8) x</code> <code>t = (i64 → i8) x</code> <code>t = (u64 → i8) x</code> <code>t = (i32 → u8) x</code> <code>t = (u32 → u8) x</code> <code>t = (i64 → u8) x</code>	Integer truncation

<code>t = (u64 → u8) x</code> <code>t = (i64 → i32) x</code> <code>t = (u64 → i32) x</code>	
<code>t = (u32 → *) x</code> <code>t = (i32 → *) x</code> <code>t = (* → i32) x</code>	Conversions between integer and pointer
<code>nop</code>	No-op
<code>break &lt;loop_block_id&gt;</code> <code>continue &lt;loop_block_id&gt;</code> <code>end handled &lt;multiple_block_id&gt;</code>	Control-flow instructions inserted by the Relooper algorithm as it processes branch instructions.
<code>if x == y {} else {}</code> <code>if x != y {} else {}</code>	Conditional control flow instructions with nested instructions for each branch. These are only inserted by the Relooper algorithm, to replace a conditional branch with conditionally setting the label variable and then branching.

---