1

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 = load from x` `store x to addr y`	Load from and store to memory
`declare var t` `allocate x bytes for var y`	Declare a new variable `t`. Allocate memory for `y` (used for allocating aggregate data structures e.g. arrays).
`reference var x`	A non-executable instruction used internally to mark a variable as live at this program point.

`t = &x`
`t = ~x`
`t = !x`
	Address-of operator
	Bitwise NOT
	Logical NOT

`t = x * y`
`t = x / y`
`t = x % y`
`t = x + y`
`t = x - y`
`t = x << y`
`t = x >> y`
`t = x & y`
`t = x | y`
`t = x ^ y`
`t = x && y`
`t = x || y`
`t = x < y`
`t = x > y`
`t = x <= y`
`t = x >= y`
`t = x == y`
`t = x != y`
	Multiplication
	Division
	Modulus
	Addition
	Subtraction
	Left-shift
	Right-shift (signed-extending for signed `x`, zero-filling for unsigned `x`)
	Bitwise AND
	Bitwise OR
	Bitwise XOR
	Logical AND
	Logical OR
	Less-than comparison
	Greater-than comparison
	Less-than or equal comparison
	Greater-than or equal comparison
	Equality comparison
	Not equal comparison

`t = call f(`p_1`, `p_2`, ...)`	Call function `f` with parameters p_i (either variables or constants)

`tail-call f(p1, p2, ...)`	Call function `f` and return the result from the current function
`return [x]`	Return from the current function. The return value `x` is optional.
`label <l>`	Attach a label to the current program point (immediately before the next instruction).
`br <l>`	Unconditional branch
`br <l> if x == y`	Conditional branch; executed if operands are equal.
`br <l> if x != y`	Conditional branch; executed if operands are not equal.
`t = &<sid>`	Static address of the string literal with id <sid>

`t = (i8 ⟶ i16) x`
`t = (i8 ⟶ u16) x`
`t = (u8 ⟶ u16) x`
`t = (u8 ⟶ u16) x`
	Char promotions

`t = (i16 ⟶ i32) x` `t = (u16 ⟶ i32) x`	Promotions to signed integer

`t = (i16 ⟶ u32) x`
`t = (u16 ⟶ u32) x`
`t = (i32 ⟶ u32) x`
	Promotions to unsigned integer

`t = (i32 ⟶ i64) x` `t = (u32 ⟶ i64) x`	Promotions to signed long

`t = (i32 ⟶ u64) x`
`t = (u32 ⟶ u64) x`
`t = (i64 ⟶ u64) x`
	Promotions to unsigned long

`t = (u32 ⟶ f32) x`
`t = (i32 ⟶ f32) x`
`t = (u64 ⟶ f32) x`
`t = (i64 ⟶ f32) x`
	Integer to float conversions

`t = (u32 ⟶ f64) x`
`t = (i32 ⟶ f64) x`
`t = (u64 ⟶ f64) x`
`t = (i64 ⟶ f64) x`
	Integer to double conversions

`t = (f32 ⟶ f64) x`	Float to double promotion
`t = (f64 ⟶ i32) x`	Double to int conversion

`t = (i32 ⟶ i8) x`
`t = (u32 ⟶ i8) x`
`t = (i64 ⟶ i8) x`
`t = (u64 ⟶ i8) x`
`t = (i32 ⟶ u8) x`
`t = (u32 ⟶ u8) x`
`t = (i64 ⟶ u8) x`
	Integer truncation

`t = (u64 ⟶ u8) x`
`t = (i64 ⟶ i32) x`
`t = (u64 ⟶ i32) x`	

`t = (u32 ⟶ *) x`
`t = (i32 ⟶ *) x`
`t = (* ⟶ i32) x`
	Conversions between integer and pointer

`nop`	No-op

`break <loop_block_id>`
`continue <loop_block_id>`
`end handled <multiple_block_id>`
	Control-flow instructions inserted by the Relooper algorithm as it processes branch instructions.

`if x == y {} else {}` `if x != y {} else {}`	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.