Contents

1 Control instructions

label

Operation Create a label

Format

label
name

Operand Stack No change

Description Creates a label at the given location that can be identified by *name*. Jump instructions must jump to named label. Jumps made to *name* will resume execution on the proceeding instruction.

jmp

Operation Unconditional jump to a label

Format

jmp
name

Operand Stack No change

Description Unconditional jump to a label with *name*.

Execution resumes at the label marked with *name*.

invokefunction

Operation Invoke a function

Format

invokefunction
name

Operand Stack *..., [arg1, [arg2, ...]] –>*
...

Description Invokes the function identified as *name*. The operand stack must contain the arguments, where the number, type, and order of the values must be consistent with the descriptor of the function.

The argument values are popped from the operand stack. A new frame is created on the stack for the method being invoked. The argument values are consecutively made the values of local variables of the new frame, with arg1 in local variable 0 and so on. The new frame is then made current, and the program counter is set to the first instruction of the function to be invoked. Execution continues with the first instruction of the function.

invokemethod

Operation	Invoke a method
Format	TODO
Operand Stack	TODO
Description	TODO: How to call methods?

return

Operation Return from a void function

Format *return*

Operand Stack ..., –>
[empty]

Description The current function must have no return type.

The interpreter returns control to the caller of the function, restoring the frame of the caller.

2 Load instructions

i64load

Operation Load an int64 from a local variable onto the operand stack

Format

i64load
index

Operand Stack *..., –>*
..., value

Description Accesses the entry of the local variable table at index *index* and pushes the value onto the operand stack. The entry at that index must be of type int64.

f64load

Operation Load a float64 from a local variable onto the operand stack

Format

f64load
index

Operand Stack *..., –>*
..., value

Description Accesses the entry of the local variable table at index *index* and pushes the value onto the operand stack. The entry at that index must be of type float64.

3 Store instructions

i64store

Operation Store an int64 from the operand stack into the local variable table

Format

i64store
index

Operand Stack *..., value–>*
...,

Description Pops the value off of the operand stack and stores it in the local variable table at *index*. The value must be of type int64.

f64store

Operation Store a float64 from the operand stack into the local variable table

Format

f64store
index

Operand Stack *..., value–>*
...,

Description Pops the value off of the operand stack and stores it in the local variable table at *index*. The value must be of type float64.

4 int64

- i64add
- i64sub
- i64mul
- i64div
- i64rem

- i64and
- i64or
- i64xor
- i64clear
- i64shl (requires unsigned right operand)
- i64shr (required unisgned right operand)

- i64neg (negation)
- i64comp (bitwise complement)

- if_i64eq
- if_i64ne
- if_i64lt
- if_i64le
- if_i64gt
- if_i64ge

- i64tof64
- i64toa

i64add

Operation Add two int64s

Format *i64add*

Operand Stack *..., value1, value2 –>*
..., result

Description Both *value1* and *value2* must be of type int64. The values are popped off the operand stack. The int64 *result* is pushed onto the stack.

The result is the 64 low-order bits in twos-complement format. Overflow may occur as a result of this operation. Despite the fact that overflow can occur, *i64add* never throws a runtime exception.

i64sub

Operation Subtract two int64s

Format *i64sub*

Operand Stack *..., value1, value2 –>*
..., result

Description Both *value1* and *value2* must be of type int64. The values are popped off the operand stack. The int64 *result* is *value1* - *value2*. *result* is pushed onto the stack.

The result is the 64 low-order bits int twos-complement format. Overflow may occur as a result of this operation. Despite the fact that overflow can occur, *i64sub* never throws a runtime exception.

i64mul

Operation Multiply two int64s

Format *i64mul*

Operand Stack *..., value1, value2 –>*
..., result

Description Both *value1* and *value2* must be of type int64. The values are popped off the operand stack. The int64 *result* is *value1* * *value2*. *result* is pushed onto the stack.

The result is the 64 low-order bits int twos-complement format. Overflow may occur as a result of this operation. Despite the fact that overflow can occur, *i64mul* never throws a runtime exception.

i64div

Operation Divide two int64s

Format *i64div*

Operand Stack *..., value1, value2 –>*
..., result

Description Both *value1* and *value2* must be of type int64. The values are popped off the operand stack. The int64 *result* is *value1 / value2*. *result* is pushed onto the stack.

i64div performs integer division, with *result* truncated towards 0.

If the dividend x is the most negative value for int64, the quotient q = x / -1 is equal to x.

Exception A run-time exception occurs when the divisor, *value2*, is 0.

i64rem

Operation Remainder after division of two int64s

Format *i64rem*

Operand Stack *..., value1, value2 –>*
..., result

Description Both *value1* and *value2* must be of type int64. The values are popped off the operand stack. The int64 *result* is *value1* % *value2*. *result* is pushed onto the stack.

For two int64 values x and y, the integer quotient $q = x/y$ and remainder $r = x\%y$ satisfy the following: $x = q * y + r$ and $|r| < |y|$

Exception A run-time exception occurs when the divisor, *value2*, is 0.

i64and

Operation Bitwise AND on two int64s

Format *i64and*

Operand Stack *..., value1, value2 –>*
..., result

Description Both *value1* and *value2* must be of type int64. The values are popped off the operand stack. The int64 *result* is calculated by taking the bitwise AND of *value1* and *value2*. *result* is pushed onto the stack.

i64or

Operation Bitwise OR on two int64s

Format *i64or*

Operand Stack *..., value1, value2 –>*
..., result

Description Both *value1* and *value2* must be of type int64. The values are popped off the operand stack. The int64 *result* is calculated by taking the bitwise OR of *value1* and *value2*. *result* is pushed onto the stack.

i64xor

Operation Bitwise XOR on two int64s

Format *i64xor*

Operand Stack *..., value1, value2 –>*
..., result

Description Both *value1* and *value2* must be of type int64. The values are popped off the operand stack. The int64 *result* is calculated by taking the bitwise XOR of *value1* and *value2*. *result* is pushed onto the stack.

i64clear

Operation Bitwise clear (AND NOT) on two int64s

Format *i64clear*

Operand Stack *..., value1, value2 –>*
..., result

Description Both *value1* and *value2* must be of type int64. The values are popped off the operand stack. The int64 *result* is calculated by taking the bit-complement of *value2*, then applying *value2* AND *value2*. *result* is pushed onto the stack.

Bitwise clear is equivalent to (x & (ˆy)), where ˆis the unary complement operator and & is the bitwise AND operator.

i64shl

Operation	Left shift on an int64
Format	*i64shl*
Operand Stack	TODO
Description	TODO: Left shift in Golang requires the right operand to be an unsigned integer

i64shr

Operation	Right shift on an int64
Format	*i64shr*
Operand Stack	TODO
Description	TODO: Right shift in Golang requires the right operand to be an unsigned integer

i64neg

Operation Negatation on an int64

Format *i64neg*

Operand Stack *..., value –>*
..., result

Description *value* must be of type int64. *value* is popped off the operand stack. *result* is the arithmetic negation of *value*, -*value*. *result* is pushed onto the stack.

This operation is equivalent to 0 - *value*.

An exception to this rule is if *value* is the most negative integer that can be represented. Since signed integers are represented in two's complement, the positive value of the most negative number is not included. If *value* is the most negative number possible for int64, then *result* = *value*.

i64comp

Operation Bitwise complement on int64

Format *i64comp*

Operand Stack *..., value –>*
..., result

Description *value* must be of type int64. *value* is popped off the operand stack. *result* is the bitwise complement of all bits in *result*. *result* is pushed onto the stack.

if_i64eq

Operation Check if two int64s are equal

Format

if_i64eq
name

Operand Stack *..., value1, value2 –>*
...,

Description Both *value1* and *value2* must be of type int64. Both values are popped off the operand stacked and compared.

If the two values are equal, then execution resumes at the label marked with *name*. Otherwise, execution resumes on the proceeding instruction.

if_i64ne

Operation Check if two int64s are not equal

Format

if_i64ne
name

Operand Stack *..., value1, value2 –>*
...,

Description Both *value1* and *value2* must be of type int64. Both values are popped off the operand stacked and compared.

If the two values are not equal, then execution resumes at the label marked with *name*. Otherwise, execution resumes on the proceeding instruction.

if_i64lt

Operation Check if int64 is less than other int64

Format

if_i64lt
name

Operand Stack *..., value1, value2 –>*
...,

Description Both *value1* and *value2* must be of type int64. Both values are popped off the operand stacked and compared.

If *value1* is less than *value2*, then execution resumes at the label marked with *name*. Otherwise, execution resumes on the proceeding instruction.

if_i64le

Operation Check if int64 is less than or equal to other int64

Format

if_i64le
name

Operand Stack *..., value1, value2 –>*
...,

Description Both *value1* and *value2* must be of type int64. Both values are popped off the operand stacked and compared.

If *value1* is less than or equal to *value2*, then execution resumes at the label marked with *name*. Otherwise, execution resumes on the proceeding instruction.

if_i64gt

Operation Check if int64 is greater than other int64

Format

if_i64gt
name

Operand Stack *..., value1, value2 –>*
...,

Description Both *value1* and *value2* must be of type int64. Both values are popped off the operand stacked and compared.

If *value1* is greater than *value2*, then execution resumes at the label marked with *name*. Otherwise, execution resumes on the proceeding instruction.

if_i64ge

Operation Check if int64 is greater than or equal to other int64

Format

if_i64ge
name

Operand Stack *..., value1, value2 –>*
...,

Description Both *value1* and *value2* must be of type int64. Both values are popped off the operand stacked and compared.

If *value1* is greater than or equal to *value2*, then execution resumes at the label marked with *name*. Otherwise, execution resumes on the proceeding instruction.

i64tof64

Operation Convert int64 to float64

Format *i64tof64*

Operand Stack *..., value* –>
..., result

Description *value* must be of type int64. *value* is popped off the operand stack. *result* is the result of converting *value* from an int64 to a float64. *result* is pushed onto the stack.

The conversion may result in a loss of precision in *result* due to the limited number of bits in the mantissa, as per the IEEE 754 standard.

i64toa

Operation Convert int64 to string

Format *i64toa*

Operand Stack *..., value –>*
..., result

Description *value* must be of type int64. *value* is popped off the operand stack. *result* is the result of converting *value* from an int64 to a string. *result* is pushed onto the stack.

5 float64

- f64add
- f64sub
- f64mul
- f64div

- if_f64eq
- if_f64ne
- if_f64lt
- if_f64le
- if_f64gt
- if_f64ge

- f64toi64

f64add

Operation Add two float64s

Format *f64add*

Operand Stack *..., value1, value2 –>*
..., result

Description Both *value1* and *value2* must be of type float64. The values are popped off the operand stack. The float64 *result* is calculated as the value of *value1* + *value2*. *result* is pushed onto the stack.

f64sub

Operation Subtract two float64s

Format *f64sub*

Operand Stack *..., value1, value2 –>*
..., result

Description Both *value1* and *value2* must be of type float64. The values are popped off the operand stack. The float64 *result* is calculated as the value of *value1* - *value2*. *result* is pushed onto the stack.

f64mul

Operation Multiply two float64s

Format *f64mul*

Operand Stack *..., value1, value2 –>*
..., result

Description Both *value1* and *value2* must be of type float64. The values are popped off the operand stack. The float64 *result* is calculated as the value of *value1* * *value2*. *result* is pushed onto the stack.

f64div

Operation Divide two float64s

Format *f64div*

Operand Stack *..., value1, value2 –>*
..., result

Description Both *value1* and *value2* must be of type float64. The values are popped off the operand stack. The float64 *result* is calculated as the value of *value1 / value2*. *result* is pushed onto the stack.

Notes As of Golang 1.7, floating-point division by 0 does not raise a runtime exception. It returns positive infinity, +Inf.

if_f64eq

Operation Check if two float64s are equal

Format

if_f64eq
name

Operand Stack *..., value1, value2 –>*
...,

Description Both *value1* and *value2* must be of type float64. Both values are popped off the operand stacked and compared.

If the two values are equal, then execution resumes at the label marked with *name*. Otherwise, execution resumes on the proceeding instruction.

if_f64ne

Operation Check if two float64s are not equal

Format

if_f64ne
name

Operand Stack *..., value1, value2 –>*
...,

Description Both *value1* and *value2* must be of type float64. Both values are popped off the operand stacked and compared.

If the two values are not equal, then execution resumes at the label marked with *name*. Otherwise, execution resumes on the proceeding instruction.

if_f64lt

Operation Check if a float64 is less than other float64

Format

if_f64lt
name

Operand Stack *..., value1, value2 –>*
...,

Description Both *value1* and *value2* must be of type float64. Both values are popped off the operand stacked and compared.

If *value1* is less than *value2*, then execution resumes at the label marked with *name*. Otherwise, execution resumes on the proceeding instruction.

if_f64le

Operation Check if a float64 is less than or equal to other float64

Format

if_f64le
name

Operand Stack *..., value1, value2 –>*
...,

Description Both *value1* and *value2* must be of type float64. Both values are popped off the operand stacked and compared.

If *value1* is less than or equal to *value2*, then execution resumes at the label marked with *name*. Otherwise, execution resumes on the proceeding instruction.

if_f64gt

Operation Check if a float64 is greater than other float64

Format

if_f64gt
name

Operand Stack *..., value1, value2 –>*
...,

Description Both *value1* and *value2* must be of type float64. Both values are popped off the operand stacked and compared.

If *value1* is greater than *value2*, then execution resumes at the label marked with *name*. Otherwise, execution resumes on the proceeding instruction.

if_f64ge

Operation Check if a float64 is greater than or equal other float64

Format

if_f64ge
name

Operand Stack *..., value1, value2 –>*
...,

Description Both *value1* and *value2* must be of type float64. Both values are popped off the operand stacked and compared.

If *value1* is greater than or equal to *value2*, then execution resumes at the label marked with *name*. Otherwise, execution resumes on the proceeding instruction.

f64toi64

Operation Convert float64 to int64

Format *f64toi64*

Operand Stack *..., value –>*
..., result

Description *value* must be of type float64. *value* is popped off the operand stack. *result* is the result of converting *value* from an float64 to a int64. *result* is pushed onto the stack.

This conversion truncates towards 0 (fraction is discarded).

6 string

- strconcat
- ...comparisons...

TODO: How will strings be stored in bytecode?
Most string conversions must be done with the `strconv` package. See https://golang.org/pkg/strconv/.

strlit

Operation Create a string literal

Format

strlit
name
len
literal

Operand Stack No change

Description *strlit* creates a string literal and creates a reference to it named *name*. *len* specifies the number of bytes that comprise the literal.

Future uses for the string can be accessed through the reference *name*.

Instruction Size *strlit* - 2 bytes
name - 2 bytes
len - 4 bytes
literal - *len* bytes

Example `strlit s 5 68 65 6c 6c 6f` creates the string which corresponds to "hello".

pushstr

Operation Push a string reference onto the operand stack

Format

pushstr
name

Operand Stack *... –>*
..., reference

Description The string referenced by *name* must exist. A reference to the string is pushed onto the stack, not the string itself.