Contents

1 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	*TODO*
Operand Stack	*..., value1, value2 –>* *...,*
Description	TODO: How to handle jumps? With labels, like x86, or byte numbers as with Java, or some other way?

if_i64ne

Operation	Check if two int64s are not equal
Format	*TODO*
Operand Stack	*..., value1, value2 –>* *...,*
Description	TODO: How to handle jumps? Same as if_i64eq

if_i64lt

Operation	Check if int64 is less than other int64
Format	*TODO*
Operand Stack	*..., value1, value2 –>* *...,*
Description	TODO: How to handle jumps? Same as if_i64eq

if_i64le

Operation	Check if int64 is less than or equal to other int64
Format	*TODO*
Operand Stack	*..., value1, value2 –>* *...,*
Description	TODO: How to handle jumps? Same as if_i64eq

if_i64gt

Operation	Check if int64 is greater than other int64
Format	*TODO*
Operand Stack	*..., value1, value2 –>* *...,*
Description	TODO: How to handle jumps? Same as if_i64eq

if_i64ge

Operation	Check if int64 is greater than or equal to other int64
Format	*TODO*
Operand Stack	*..., value1, value2 –>* *...,*
Description	TODO: How to handle jumps? Same as if_i64eq

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.

2 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	*TODO*
Operand Stack	*..., value1, value2 –>* *...,*
Description	TODO: How to handle jumps? With labels, like x86, or byte numbers as with Java, or some other way?

if_f64ne

Operation	Check if two float64s are not equal
Format	*TODO*
Operand Stack	*..., value1, value2 –>* *...,*
Description	TODO: How to handle jumps? Same as with if_f64eq

if_f64lt

Operation	Check if a float64 is less than other float64
Format	*TODO*
Operand Stack	*..., value1, value2 –>* *...,*
Description	TODO: How to handle jumps? Same as with if_f64eq

if_f64le

Operation	Check if a float64 is less than or equal to other float64
Format	*TODO*
Operand Stack	*..., value1, value2 –>* *...,*
Description	TODO: How to handle jumps? Same as with if_f64eq

if_f64gt

Operation	Check if a float64 is greater than other float64
Format	*TODO*
Operand Stack	*..., value1, value2 –>* *...,*
Description	TODO: How to handle jumps? Same as with if_f64eq

if_f64ge

Operation	Check if a float64 is greater than or equal other float64
Format	*TODO*
Operand Stack	*..., value1, value2 –>* *...,*
Description	TODO: How to handle jumps? Same as with if_f64eq

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).

3 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/.