Postfix Reference Guide

*From Wiki**3*

Contents

The Postfix reference guide contains information about the structure and operations of the stack machine.

The original stack machine was created by Santos (2004). Is was composed by a set of macros to be used with printf functions. Each macro would “take” as arguments, either a number or a string. This was a simple and effective approach but was limited in its expressiveness.

The current postfix code generator class maintains the stack machine abstraction, but does not rely on macros. Instead, it defines an interface to be used by semantic analysers, as defined by a strategy pattern (Gamma et al., 1995). Specific implementations provide the realization of the postfix commands for a particular target machine. Since it is written in C++, it's very easy to extend to new needs and implementations (new target machines).

Like the original postfix code generator, the current abstraction uses an architecture based on a stack machine, hence the name "postfix", and three registers.

1. IP -- the instruction pointer -- indicates the position of the next instruction to be executed;
2. SP -- the stack pointer -- indicates the position of the element currently at the stack top;
3. FP -- the frame pointer -- indicates the position of the activation register of the function currently being executed.

In the following tables, the "stack" columns present the results of the actions on the values at the top of the stack. Note that only elements relevant in a given context, i.e., that of the postfix instruction being executed, are shown. The notation **#length** represents a set of *length* consecutive bytes in the stack, i.e., a vector.

OPERATION	stack before	stack after	Description of actions

Consider the following fictitious example:

FAKE	$ a #8 b	$ a b	This is a fake operation

In this example, before the FAKE operation, the stack had at its top **b**, followed by eight bytes, followed by **a**. After executing the FAKE operation (which used those elements in some way), the stack has at its top **b**, followed by **a**. The symbol **$** is used to denote the point in the stack not affected by the current operation (this could be the top if the stack were empty).

The following groups of operations are available in the Postfix interface:

Segments, Values, and Labels

Segment selection

These operations select various segments. They do not affect the stack.

BSS			Specifies/selects the data segment for uninitialized values
DATA			Specifies/selects the data segment for initialized values
RODATA			Specifies/selects the data segment for initialized constant values
TEXT			Specifies/selects the text (code) segment (default)
TEXT name			Specifies/selects the text (code) segment with name *name*
TEXT number			Specifies/selects the text (code) segment with name *number*

Values (declaration in segments)

These operations declare values directly in various segments. They do not affect the stack.

SALLOC size			Declares an uninitialized vector with length **size** (in bytes)
SSHORT value			Declares a static 16-bit integer **value**
SBYTE value			Declares a static 8-bit character **value**
SINT value			Declares a static 32-bit integer **value**
SDOUBLE value			Declares a static double precision (64-bit) floating point **value**
SFLOAT value			Declares a static simple precision (32-bit) floating point **value**
SADDR name			Declares a name for an address (i.e., declares the address associated with **name**)
SSTRING string			Declares a static NULL-terminated character **string** (C-like) (may contain special characters)

Labels

These operations handle symbols and their definitions within some segment. They do not affect the stack.

ALIGN			Forces the alignment of code or data
LABEL name			Generates a new label **name**
EXTERN name			Declares **name** as a symbol externally defined, i.e., defined in another compilation module
GLOBAL name, type			Declares a **name** with a given **type** (see below) -- the declaration of a name must preceed its definition

In a declaration common to several modules, any number of modules may contain common or external declarations, but only one of them may contain an initialized declaration. A declaration does not need to be specified in a specific segment.

Global names may be of different types. These labels are to be used to generate the types needed for the second argument of **GLOBAL**.

- NONE - Unknown type
- FUNC - Name/label corresponds to a function
- OBJ - Name/label corresponds to an object (data)

Addressing, Loading and Storing

Absolute addressing uses addresses based on named labels. Local addressing is used in function frames and uses offsets relative to the frame pointer to load data: negative addresses correspond to local variables, offset zero contains the previous (saved) value of the frame pointer, offset 4 (32 bits) contains the previous (saved) value of the instruction pointer, and, after offset 8, reside the function arguments.

Adressing operations

ADDR name	$	$ name	Absolute addressing: load address of **name**
ADDRA name	$ value	$	Absolute addressing: store **value** to **name**
ADDRV name	$	$ [name]	Absolute addressing: load value at **name**
LOCAL offset	$	$ fp+offset	Local addressing: load address of **offset**
LOCA offset	$ a	$	Local addressing: writes **a** to **offset**
LOCV offset	$	$ [fp+offset]	Local addressing: load value at **offset**

ADDRA, ADDRV, LOCA, LOCV are functionally equivalent to ADDR+STINT, ADDR+LDINT, LOCAL+STINT, LOCAL+LDINT, but the generated code is more efficient. They are compound operations (i.e., they contain not only the addressing part, but also the load/store part as well). Note that the postfix_writer visitor is, in general, incapable of generating these instructions.

Load operations

The load instructions assume that the top of the stack contains an address pointing to the data to be read. Each load instruction will replace the address at the top of the stack with the contents of the position it points to. Load operations differ only in what they load.

LDINT	$ addr	$ [addr]	Loads 4 bytes (int)
LDFLOAT	$ addr	$ [addr]	Loads 4 bytes (float)
LDDOUBLE	$ addr	$ [addr]	Loads 8 bytes (double)
LDBYTE	$ addr	$ [addr]	Loads 1 byte (char)
LDSHORT	$ addr	$ [addr]	Loads 2 bytes (short)

Store operations

Store instructions assume the stack contains at the top the address where data is to be stored. That data is in the stack, immediately after the address. Store instructions differ only in what they store.

STINT	$ val addr	$	Stores 4 bytes (int)
STFLOAT	$ val addr	$	Stores 4 bytes (float)
STDOUBLE	$ val addr	$	Stores 8 bytes (double)

STBYTE	$ val addr	$	Stores 1 byte (char)
STSHORT	$ val addr	$	Stores 2 bytes (short)

Simple Stack Operations

DUP32	$ a	$ a a	Duplicates the 32-bit value at the top of the stack
DUP64	$ a	$ a a	Duplicates the 64-bit value at the top of the stack
INT value	$	$ value	Pushes an integer **value**
FLOAT value	$	$ value	Pushes a 4-byte float **value** (single precision)
DOUBLE value	$	$ value	Pushes an 8-byte float **value** (double precision)
SP	$	$ sp	Pushes the value of the stack pointer
SWAP32	$ a b	$ b a	Swaps the two 32-bit values at the top of the stack
SWAP64	$ a b	$ b a	Swaps the two 64-bit values at the top of the stack
ALLOC	$ bytes	$ #bytes	Allocates in the stack an array with size **bytes**. Since this operation alters the meaning of offsets in the stack, care should be taken when local variables exist.

Arithmetic Operations

The arithmetic operations considered here apply to both signed and unsigned integer arguments, and to double precision floating point arguments.

Integer operations

NEG	$ a	$ -a	Negation (symmetric) of integer value
ADD	$ a b	$ a+b	Integer sum of two integer values
SUB	$ a b	$ a-b	Integer subtraction of two integer values
MUL	$ a b	$ a*b	Integer multiplication of two integer values
DIV	$ a b	$ a/b	Integer division of two integer values
MOD	$ a b	$ a%b	Remainder of the integer division of two integer values
UDIV	$ a b	$ a/b	Integer division of two natural (unsigned) integer values
UMOD	$ a b	$ a%b	Remainder of the integer division of two natural (unsigned) integer values.

Floating point operations

These operations take double precision floating point operands.

DNEG	$ a	$ -a	Negation (symmetric)
DADD	$ a b	$ a+b	Sum
DSUB	$ a b	$ a-b	Subtraction
DMUL	$ a b	$ a*b	Multiplication

DDIV	$ a b	$ a/b	Division

Increment and Decrement Operations

INCR delta	$ address	$ address	Adds **delta** to the value at the **address** at the top of the stack, i.e. *[address]* becomes *[address]+delta*
DECR delta	$ address	$ address	Subtracts **delta** to the value at the **address** at the top of the stack, i.e. *[address]* becomes *[address]-delta*

Type Conversion Operations

The following instructions perform type conversions. The conversions are from and to integers and simple and double precision floating point values.

D2F	$ d	$ f	Converts from double precision (64-bit) to single precision (32-bit) floating point
D2I	$ d	$ i	Converts from double precision (64-bit) floating point to integer (32-bit)
F2D	$ f	$ d	Converts from simple precision (32-bit) to double precision (64-bit) floating point
I2D	$ i	$ d	Converts from integer (32-bit) to double precision (64-bit) floating point

Comparison Operations

Integer comparison instructions

The comparison instructions are binary operations that leave at the top of the stack 0 (zero) or 1 (one), depending on the result of the comparison: respectively, **false** or **true**. The value may be directly used to perform conditional jumps (e.g., JZ, JNZ), that use the value of the top of the stack instead of relying on special processor registers ("flags").

EQ	$ a b	$ a≡b	*equal to*
NE	$ a b	$ a≠b	*not equal to*

GT	$ a b	$ a>b	*greater than*
GE	$ a b	$ a≥b	*greater than or equal to*
LE	$ a b	$ a≤b	*less than or equal to*
LT	$ a b	$ a<b	*less than*

The following consider unsigned operands:

UGT	$ a b	$ a>b	*greater than* for unsigned integers
UGE	$ a b	$ a≥b	*greater than or equal to* for unsigned integers
ULE	$ a b	$ a≤b	*less than or equal to* for unsigned integers
ULT	$ a b	$ a<b	*less than* for unsigned integers

Floating point comparison operator

This operator compares two double precision floating point numbers. The result is an integer value: less than 0, if the first operand is less than the second; 0, if they are equal; greater than 0, otherwise.

DCMP	$ a b	$ i	"compare" -- i<0, a<b; i≡0, a≡b; i>0, a>b

Bitwise Operations

NOT	$ a	$ ~a	Bitwise negation, i.e., one's complement
AND	$ a b	$ a∧b	Bitwise AND operation
OR	$ a b	$ a∨b	Bitwise OR operation
XOR	$ a b	$ a⊕b	Bitwise XOR (exclusive OR) operation

Rotation and Shift Operations

Shift and rotation operations have as maximum value the number of bits of the underlying processor register (32 bits in a ix86-family processor). Safe operation for values above that limit is not guaranteed.

ROTL	$ value nbits	$ value<rl>bits	Rotate **value nbits** to the left
ROTR	$ value nbits	$ value<rr>bits	Rotate **value nbits** to the right
SHTL	$ value nbits	$ value<<bits	Shift **value nbits** to the left
SHTRU	$ value nbits	$ value>>bits	Shift **value nbits** to the right (unsigned)
SHTRS	$ value nbits	$ value>>>bits	Shift **value nbits** to the right (signed)

Function Definition

The following sections cover defining and calling functions.

Starting a function

Each function must allocate space for its local variables. This is done immediately after being called and before any other processing. The relevant operations are ENTER (to specify a given memory amount) and START (no space is reserved for local variables. Note that these operations do more than manipulate the stack: they also create an activation register for the function, i.e., they update the frame pointer and define a new stack frame.

ENTER bytes	$	$ fp #bytes	Starts a function: pushes the frame pointer (activation register) to the stack and allocates space for local variables (**bytes**)
START	$	$ fp	Equivalent to "ENTER 0"

Leaving a function

STFVAL32 or STFVAL64 may be called to specify return values in accordance with C conventions. Only return values that fit in these registers need these operations. Other return values are passed by pointer.

Note that these operations make use of specific hardware registers (STFVAL32->eax, STFVAL64->st0).

STFVAL32	$ a	$	Removes a 32-bit integer value from the stack (to eax)
STFVAL64	$ d	$	Removes a double precision (64-bit) floating point value from the stack (to st0)

The stack frame is destroyed by the LEAVE operation. This action must be performed immediately before returning control to the caller (with RET).

LEAVE	$ fp ...	$	Ends a function: restores the frame pointer (activation register) and destroys the function-local stack data

After the function's stack frame is destroyed and the activation register is restored to the caller, control must also be returned to the caller (i.e., IP must be updated).

RET	$ addr	$	Returns from a function (the stack must contain the return address)
RETN bytes	$ #bytes addr	$	Returns from a function and removes **bytes** from the caller's stack after removing the return address. This is more or less the same as "RET+TRASH bytes". **Note that this is not compatible with the Cdecl calling conventions.**

Function Calls

In a stack machine the arguments for a function call are already in the stack. Thus, it is not necessary to put them there (it is enough not to remove them).

CALL name	$	$ return-address	Calls the **name**d function. The **return-address** is pushed to the stack.
BRANCH	$ address	$ return-address	Invokes a function at the **address** indicated at the top of the stack. The **return-address** is pushed to the stack.

When building functions that conform to the C calling convention, the arguments are destroyed by the caller, *after* the return of the callee, using TRASH and stating the total size (i.e., for all arguments).

TRASH bytes	$ #bytes	$	Removes **bytes** from the stack

To recover the returned value by the callee, the caller must call LDFVAL32, to put the value in eax in the stack. An analogous procedure is valid for LDFVAL64 (for double precision floating point return values -- value comes from st0).

LDFVAL32	$	$ value	Pushes the return **value** in the eax register to the stack
LDFVAL64	$	$ value	Pushes the return **value** in the st0 register to the stack

Basic Jump Operations

JMP label	$	$	Unconditional jump to **label** (does not affect or use the stack)
LEAP	$ address	$	Unconditional jump to the **address** at the top of the stack

Conditional Jump Operations

JZ label	$ value	$	Jump to **label** if the **value** at the top of the stack is 0 (zero)
JNZ label	$ value	$	Jump to **label** if the **value** at the top of the stack is non-zero

The following operations combine comparisons and jumps.

JEQ label	$ a b	$	Jump to **label** if a≡b
JNE label	$ a b	$	Jump to **label** if a≠b

JGT label	$ a b	$	Jump to **label** if a>b
JGE label	$ a b	$	Jump to **label** if a≥b
JLE label	$ a b	$	Jump to **label** if a≤b

JLT label	$ a b	$	Jump to **label** if a<b

The following are for the unsigned versions of the comparisons.

JUGT label	$ a b	$	Jump to **label** if a>b (unsigned)
JUGE label	$ a b	$	Jump to **label** if a≥b (unsigned)
JULE label	$ a b	$	Jump to **label** if a≤b (unsigned)
JULT label	$ a b	$	Jump to **label** if a<b (unsigned)

Other Operations

NIL			No action is performed
NOP			Generates a null operation (consumes time; does not change the processor's state)

Categories: **Compiladores** **Ensino**