STACKL Reference Manual

1 Introduction 5

2 stacklc: the stackl compiler 5

2.1 Special functions 5

2.1.1 Interrupt 5

2.1.2 system trap 6

2.1.3 startup 7

2.2 Generating assembly code 7

2.3 Pragmas 7

2.4 Command line arguments 8

3 slasm: The stackl assembler 8

3.1 File format 8

3.1.1 Comments 9

3.1.2 Labels 9

3.1.3 Dot commands 9

3.2 Pound commands 9

3.3 Command line options 10

4 The stackl machine 10

4.1 Execution Registers 10

4.1.1 Instruction Pointer (IP) 10

4.1.2 Stack Pointer (SP) 10

4.1.3 Frame Pointer (FP) 11

4.2 Flag Register (FLAG) 11

4.2.1 Interrupt Vector (IVEC) 11

4.3 Memory Management Registers 12

4.3.1 Base Pointer (BP) 12

4.3.2 Limit Pointer (LP) 12

4.4 Stack Frames 12

4.5 Input/Output 13

4.6 Loading programs 14

4.7 Command line arguments 15

4.8 Interrupts 15

5 Devices 16

5.1 Programmed IO Terminal (pio\_term) 16

5.1.1 Registers 17

5.2 Timer 17

5.3 Disk 18

5.3.1 Registers 18

6 Interpreter Architecture 19

6.1 Memory 19

6.2 Executable file format and the Loader 20

6.3 Adding IO devices to the interpreter 20

7 Instruction Reference 20

7.1 ADJSP <amount> 20

7.2 AND 21

7.3 BAND 21

7.4 BOR 21

7.5 BXOR 21

7.6 CALL <addr> 21

7.7 CALLI 21

7.8 CLID 22

7.9 COMP 22

7.10 DIVIDE 22

7.11 DUP 22

7.12 EQ 23

7.13 GE 23

7.14 GT 23

7.15 HALT 23

7.16 ILLEGAL 23

7.17 INP 23

7.18 JMPUSER <dest> 23

7.19 JUMP <dest> 24

7.20 JUMPE <dest> 24

7.21 LE 24

7.22 LT 24

7.23 MINUS 24

7.24 MOD 25

7.25 NE 25

7.26 NEG 25

7.27 NOP 25

7.28 NOT 25

7.29 OR 25

7.30 OUTS 26

7.31 PLUS 26

7.32 POP 26

7.33 POPCVAR <offset> 26

7.34 POPCVARIND 26

7.35 POPREG <reg num> 27

7.36 POPVAR <offset> 27

7.37 POPCVARIND 27

7.38 PUSH <value> 27

7.39 PUSHCVAR <offset> 28

7.40 PUSHCVARIND 28

7.41 PUSHFP 28

7.42 PUSHREG <reg num> 28

7.43 PUSHVAR <offset> 29

7.44 PUSHVARIND 29

7.45 RETURN 29

7.46 RETURNV 29

7.47 RTI 29

7.48 SEID 30

7.49 SHIFTL 30

7.50 SHIFTR 30

7.51 SWAP 30

7.52 TIMES 30

7.53 TRACE\_OFF 31

7.54 TRACE\_ON 31

7.55 TRAP 31

Appendix A stacklc grammar 32

# Introduction

The stackl environment is intended as a virtual environment that can be used for teaching operating systems classes. The environment allows students to program in C and be able to write interrupt service routines, syscalls, and other operating system code without the need for real hardware.

# stacklc: the stackl compiler

The stacklc compiler compiles a subset of the C programming language. The following are the current restrictions:

1. The switch statement is not implemented
2. static is not supported

The compiler will output a .sl file that contains the assembly version of the program. It will then invoke the assembler to create a .slb file that can be run by the interpreter.

## Special functions

The stackl interpreter requires knowledge of three specific functions:

1. Interrupt service routines
2. The system trap routine
3. The startup routine

These can use any name. They are identified to the compiler via pragmas or via the Interrupt Vector Table.

### Interrupt

An interrupt service routine is called when any hardware interrupt occurs. It takes no arguments and must not declare any local variables. If local variables are required, the interrupt routine can call another routine. The called routine does not have any restrictions.

The interrupt routine must return using the RTI instruction.

There are 16 interrupt vectors (not all of which are currently in use). The address of the interrupt service routine must be loaded into the interrupt service vector. See the section on interrupts for how to do this.

A sample interrupt routine is shown below:

int process\_interrupt()

{

int local\_variables\_are\_ok;

// process interrupt

return 0;

}

#pragma interrupt isr

int isr()

{

process\_interrupt();

asm("RTI");

}

### system trap

The system trap routine is called when the TRAP instruction executes. It must not declare any local variables. If local variables are required, the interrupt routine can call another routine. The called routine does not have any restrictions.

The system trap routine must return using the RTI instruction.

The arguments of the function that executed the TRAP will be available to the system trap function. Normally, the TRAP is contained within a syscall function. This setup facilitates passing information to the system trap function

A sample system trap routine is shown below:

int syscall(int op, char \*buffer)

{

asm("TRAP");

return 0;

}

#pragma systrap systrap

int systrap(int op, char \*buffer)

{

if (op == PRINT) asm("OUTS", buffer);

asm("RTI");

}

The system trap routine is called via interrupt vector 1. The address of the system trap routine must be loaded into this vector prior to executing the trap instruction. See the section on interrupts for an explanation of how to do this.

### startup

The startup routine is where the code begins executing. If an assembly unit does not have a pragma for the startup routine, the assembler will generate the equivalent of the following startup routine:

#pragma startup \_\_startup

int \_\_startup()

{

main();

asm("HALT");

}

## Generating assembly code

Assembly instructions can be generated using the asm and asm2 keywords. The asm and asm2 keywords behave similar to functions with the following prototype:

int asm(opcode, ...);

int asm2(opcode, opcode2, ...);

The optional parameters will be pushed onto the stack, then the opcode value(s) will be emitted to the instruction stream. The asm2 form is useful for two word instructions such as PUSH or JMPUSER.

If the asm keywords are used in an assignment statement, then the value at the top of the stack after the opcode executes will be popped and stored in the lval of the assignment.

Note that the asm instructions cannot be used in an expression. They can be used as a stand-alone statement, or as the only element on the right side of an assignment.

## Pragmas

The stackl compiler responds to the following pragmas:

#pragma once Used in include files to guarantee that a file is only included once per compilation unit.

#pragma interrupt <isr> Identifies <isr> as the name of the interrupt service routine for vector 0. The address of this routine is placed at address zero in memory.

#pragma feature <feature> The <feature> is passed to the interpreter to enable/disable particular features of the stackl machine. The currently recognized features include: pio\_term, dma\_term, disk, and inp.

#pragma library <lib> This pragma can be used in library header files to force the assembler to include <lib> without having to explicitly specify <lib> and the slasm command line.

#pragma stack\_size <size> Indicates the requested stack size for a program. This information can be used by an exec() function to set the LP for the upper bound on memory.

#pragma startup <start> Identifies <start> as the entry point for the executable that this file is part of.

#pragma systrap <trap> Identifies <trap> as the name of the trap handler. The address of this routine is placed at address four in memory.

Note: the interrupt and systrap pragmas can be used if no hardware generated interrupts are used. If any hardware interrupts are used, the interrupts must be placed in a separate interrupt vector table pointed to by the IVEC register.

## Command line arguments

The stackl compiler responds to the following command line arguments:

-ast The compiler will output the generated abstract syntax tree as an XML file.

-c Compile, but do not run the assembler.

-debug Causes the compiler to log information that might be useful for debugging the compiler. This option is depreciated and should not be used.

-help Prints a short help string and then exits

-version Prints the compiler version and then exits

-yydebug Causes the compiler to log information that might be useful for debugging the compiler. This option is depreciated and should not be used.

# slasm: The stackl assembler

The assembler assembles one or more .sl files into a .slb executable file. The assembler will be run automatically by the compiler unless the -c option is passed to stacklc.

## File format

In addition to valid instructions, the assembler recognizes the following:

### Comments

Comments begin with a semicolon (‘;’) and extend to the end of line

### Labels

Labels consist of valid C identifiers with an optional leading dollar sign (‘$’). Label definitions should be on a line by themselves followed by a colon (‘:’). Label references are preceded by an at-sign (‘@’). The following example illustrates this syntax:

JUMP @$LABEL\_2

$LABEL\_2:

### Dot commands

Lines that begin with a period (‘.’) control the behavior of the assembler. The following dot commands are recognized:

.block <size> Allocates <size> words of memory in the current memory segment

.codeseg The instructions that follow this directive will output information to the code portion of memory.

.data <value> Outputs the value to the current memory segment. if value begins with an at-sign (‘@’) it is assumed to be a label and the address that the label refers to will be output.

.dataseg The instructions that follow this directive will output information to the data portion of memory.

.function <name> Identifies the beginning of the named function. This information is used by the -dbg option to generate debugging information.

.source <file> <line> Identifies the source file and line that generated this assembly code. This information is used by the -dbg option to generate debugging information.

.string <str> Outputs the string in null-terminated form to the current memory segment.

## Pound commands

Lines that begin with a pound sign (‘#’) are used to pass pragma information from the compiler to the assembler. They must only appear at the beginning of the file. The assembler recognizes the following pound commands:

#interrupt <isr> The <isr> is the name of the interrupt service routine and the assembler will place the address of this routine in the interrupt vector.

#systrap <trap> <trap> is name of the the trap service routine and the assembler will place the address of this routine in the trap vector.

#startup <start> <start> is the name of the entry point for the executable. The assembler will generate a jump instruction to jump to this routine at the beginning of execution. If the assembler does not encounter a #startup command, it will generate code that calls main() and then halts the CPU.

Any other line beginning with a pound sign (‘#’) will be appended to the .slb header to be handled by the interpreter.

## Command line options

The assembler recognizes the following command line options:

-defs Output a .h file that defines constants for all the stackl machine instructions. This option is used internally by the compiler build.

-help Outputs a short help string then exits.

-list In addition to the .slb file, the assembler will output a listing file.

-version Outputs the assembler version, then exits.

# The stackl machine

The stackl machine is a stack based machine. As such, there are no general purpose registers. All operations make use of the stack. Typically, the operands of an opcode are popped off the stack and the result of the operation is pushed back onto the stack.

There are two categories of special purpose registers: execution and memory management. The execution registers are used for general instructions. The memory management registers are used to manage memory when multiple processes are running on the machine. The registers are described in the following sections.

## Execution Registers

### Instruction Pointer (IP)

The instruction pointer contains the address of the next instruction to be executed.

### Stack Pointer (SP)

The stack pointer contains the address of the first unused memory location at the top of the stack. The stack always contains word aligned data. The stack grows to increasing memory addresses.

Pseudocode for pushing an item onto the stack is as follows:

Memory[SP] = value;

SP++;

Pseudo-code for popping an item from the stack is as follows:

SP--;

value = Memory[SP];

### Frame Pointer (FP)

The frame pointer is used for function calls. It points to the location of the first local variable in a called function. The opcodes that include VAR in the opcode name reference the FP to locate the variable.

## Flag Register (FLAG)

The flag register contains bits that identify the mode the processor is in. The following bits are defined:

0 HALTED The CPU will stop and the interpreter will exit when this bit is set

1 USER\_MODE The CPU is in user mode when this bit is set. The CPU is in system mode when this bit is clear.

2 INT\_MODE The CPU is executing in interrupt mode when this bit is set.

3 INT\_DIS Interrupts are disabled when this bit is set.

4 VMEM The CPU is operating in virtual memory mode when this bit is set. This mode is not yet implemented.

16 I\_MACH A machine check is pending

17 I\_TRAP A trap instruction is pending

## Interrupt Vector (IVEC)

This register contains the absolute address of the interrupt vector table. The interrupt vector table should be large enough to contain 16 word-sized interrupt vectors. When the machine boots, the IVEC register is set to zero and only the first two interrupts (machine check and trap) should be used prior to setting the IVEC.

## Memory Management Registers

When operating in user mode, memory accesses are relative to the Base Pointer. When not in user mode, the Base Pointer and Limit Pointer are ignored, and all addresses are absolute.

### Base Pointer (BP)

The base pointer contains the address of the start of memory for the current process. All memory addresses are relative to the Base Pointer. In other words, if a process wants to access memory at address N, the actual address is BP+N. This is true whether the memory operation is fetching data or instructions.

### Limit Pointer (LP)

The limit pointer contains the value just above the highest memory that a process can access. In other words, legal memory access are those in the range of BP .. LP-1. Any access outside this range will cause a machine check.

## Stack Frames

The following diagram illustrates a stack frame. Higher addresses are higher in the diagram.



The caller must push the parameters onto the stack prior to executing the CALL instruction. Parameters must be pushed from right to left. In other words, a call to

foo(a,b,c)

must push c then b then a.

### Interrupt stack frames

The following diagram depicts the stack frame for an interrupt service routine. If the interrupt was caused by a TRAP instruction, the FP is not updated: it is left pointing to the local variables of the function that initiated the TRAP.



The RTI instruction restores state based on the SP, not the FP. As a result, interrupt functions must not have any local variables. If they did, the SP would be adjusted to make room for them and the RTI would not be able to find the interrupt frame.

## Input/Output

The CPU has two instructions for doing input/output the OUTS instruction prints the NULL terminated string whose address is on the top of the stack. This is a blocking instruction: the next instruction will not execute until the IO operation is complete.

The INP instruction provides a non-blocking means of doing input. The value on the top of the stack is assumed to be the address of an io\_blk\_t. The IO block has the following form:

typedef struct

{

int op;

int param1;

int param2;

} io\_blk\_t;

The op is used to specify the desired operation and indicates the completion of the operation. The two param values are operation dependent.

When the operation is finished the most significant bit of the op field is set (resulting in a negative value).

The following operations are defined:

PRINTS\_CALL 3 This operation will perform the equivalent of a C fputs() function to print the null terminated string pointed to by param1. The output will be delayed to approximately 10,000 characters per second to simulate the output speed of a serial device.

GETL\_CALL 6 This operation will perform a C gets() function. The address of the buffer is in param1. This function will read at most 255 characters plus a terminating null.

GETS\_CALL 5 This operation will perform the following C function:

scanf("%s", param1);

where param1 is the address of the buffer. There is no attempt to prevent buffer overflow.

GETI\_CALL 7 This operation will perform the following C function:

scanf("%d", param1);

where param1 is assumed to be the address of an integer.

EXEC\_CALL 8 This operation will call the loader to load a file into memory. The name of the file is in a string pointed to by param1. The high memory address of the load will be stored in param2 upon completion. The BP and LP registers should be set prior to initiating this operation with the BP indicating the starting address for the load. The word just past the high memory will contain the stack size requested by the process.

Since these are non-blocking operations, the initiator needs to poll the op field waiting for it to turn negative to determine when the operation is complete. If the operation fails, the instruction will set bit 30 (0x40000000) in the op field to indicate an error.

The INP instruction is not enabled by default. To enable the INP instruction use the following pragma:

#pragma feature inp

## Loading programs

Programs can be loaded into memory using the INP instruction with the EXEC\_CALL operation. The loader will load the code starting at the address indicated by the BP. The loader will adjust addresses so that they are relative to BP. After loading, the param2 of the io\_blk will contain the address of the first free word above the loaded file. This value can be used to set the SP and FP for executing the code. The contents of the first free word will contain the stack size requested by the program. This value can be used to set the LP.

The following procedure should be used to load code:

1. set the BP and LP to bound the memory available to the program
2. initiate the load using the INP instruction
3. wait for the load to complete
4. Clear the interrupt mode bit in the FLAG register (if set)
5. Set the FP and SP to the value in param2
6. Update the value of the LP using the value at Memory[SP] if desired.
7. Use the JMPUSER instruction to jump to address 8

Note that this is address 8 in the user’s code (relative to the BP), it is not absolute address 8.

The code for doing this is as follows:

// set BP to start\_addr

asm2("POPREG", BP\_REG, start\_addr);

// initiate and wait for INP instruction

io\_blk.op = EXEC\_CALL;

io\_blk.addr = "ostest/user.slb";

io\_blk.status = 0;

asm("INP", &io\_blk);

while((io\_blk.op & IO\_COMPLETE) == 0)

{

}

// Update the LP if desired

int \*stack\_size\_ptr;

stack\_size\_ptr = (int \*)io\_blk.status;

high\_mem = io\_blk.status + \*stack\_size;

asm2("POPREG", LP\_REG, high\_mem);

// start the user code

asm("DUP", io\_blk.param2);

asm2("POPREG", FP\_REG); // update FP

asm2("POPREG", SP\_REG); // update SP

asm2("JMPUSER", 8); // jump to 8 in user mode

Following the update to the FP, no VAR instructions should execute because they would reference the new FP. Following the update to the SP, stack operations will reference the new stack.

## Command line arguments

The stackl interpreter recognizes the following command line arguments:

-boot Enable the disk drive and load Block 0 into memory at address zero and then jump to Address 8.

-dbg Run the interpreter in debugger mode.

-dma\_term Enable the DMA terminal device. This devices is still experimental.

-help Print a brief help message and then exit.

-inp Enable the INP instruction.

-loader Print additional information while loading programs. This option is useful for debugging the interpreter.

-nopio\_term Disable the PIO Term device.

-opcodes Write an instruction trace to stderr.

-pio\_term Enable the PIO Term device

-version Print the version number of the interpreter and then exit.

## Interrupts

The stackl machine can respond to up to 16 different interrupts. Interrupt 0 is reserved for machine checks. Interrupt 1 is reserved for the TRAP instruction. The rest of the interrupts are available for hardware devices.

Interrupts are non-preemptive: an interrupt will not be serviced while the processor is already in interrupt mode.

Interrupts can be enabled/disabled with the CLID/SEID instructions (or by modifying bits in the FLAG register).

When a hardware device signals an interrupt, one of bits 16-31 are set in the FLAG register indicating which interrupt vector is to be used to service the interrupt. If any of these bits are set and interrupts are enabled and the processor is not in interrupt mode, then the highest priority pending interrupt will be serviced. The highest priority interrupt is the one indicated by bit 16, the lowest is the one indicated by bit 31.

The address of the interrupt service routine must be stored in the interrupt vector table prior to the occurrence of the interrupt. The address of the interrupt vector table is stored in the IVEC register. The following code illustrates how to initialize the interrupt vector table:

int Interrupt\_Vector[16];

int systrap(int size, int op, int parm1);

#pragma startup startup\_\_

int startup\_\_()

{

// Store the address of systrap in Vector 1

Interrupt\_Vector[1] = systrap;

// Load the IVEC reg

asm2("POPREG", IVEC\_REG, &Interrupt\_Vector);

...

}

For security reasons, the unused vectors should be loaded with the value 0x0001 to force a machine check if an “unused” interrupt vector actually gets used.

# Devices

This section discusses the IO devices available on the stackl machine.

## Programmed IO Terminal (pio\_term)

This device is used for doing programmed IO from/to the console. The device can operate in polled mode or in interrupt mode. This device uses interrupt vector 5.

The pio\_term must be enabled using the following pragma:

#pragma feature pio\_term

If this device is enabled, the INP instruction should not be used for performing input from the console.

Enabling this device changes the mode of the Linux console. Under most circumstances, the mode will be restored when the interpreter exits.

### Registers

The following registers are available for this device. These are byte-wide registers and should only be accessed using byte wide instructions. If a word sized instruction is used to read or write these registers, the behavior is undefined.

#### RDR – Read Data Register

This register is at address 0x0E000000. This is a read only register.

Data input from the console will appear in this register. Every time a new value appears in this register, the RECV bit in the IIR register will be set. The bit will be cleared when the data is read from the RDR.

#### XDR – Xmit Data Register

This register is at address 0x0E000000. This is a write only register.

Data written to this register will be sent to the console. Writing to this register will clear the XMIT bit in the IIR. The XMIT bit in the IIR will be set when the device is ready to transmit another byte.

#### IER – Interrupt Enable Register

This register is at address 0x0E000001. This is a read/write register.

This register controls what events will trigger an interrupt. Interrupts will be triggered if the bit corresponding to the event is set in this register. Bit 1 controls the READ interrupt, bit 2 controls the XMIT interrupt.

#### IIR – Interrupt Information Register

This register is at address 0x0E000002. This is a read only register. Reading the register will clear it.

This register indicates what caused the interrupt. The following bits are used:

0x01 ATTN: This device needs attention

0x02 RECV: The device received a new character

0x04 XMIT: The device finished transmitting a byte

## Timer

A timer can be configured to generate an interrupt every N instructions. This device uses interrupt vector 3. The timer has three word-sized registers. If the registers are accessed using byte instructions, the behavior is undefined.

The TIMER\_CSR register is at address 0x0C000000. Bit zero is the interrupt enable bit. Setting it to 1 will cause the timer to generate interrupts when the TIMER\_COUNT register is greater than or equal to the TIMER\_LIMIT register. Bit 31 is set when an interrupt is generated. Bit 31 is cleared by reading the TIMER\_CSR.

The TIMER\_COUNT register is at address 0x0C000004. It increments after each instruction executes and is reset to zero once it reaches the TIMER\_LIMIT.

The TIMER\_LIMIT register is at address 0x0C000008. It is used to specify how often an interrupt should be generated. NOTE: Be sure the limit is high enough so that the interrupt service routine can finish before another interrupt is signaled.

The TIMER\_TIME register is at address 0x0C00000C. It increments after each instruction executes, and never resets. This register gives the number of instructions executed since the Stackl machine booted. It will roll over once it reaches 0xFFFFFFFF. It should be considered an unsigned value.

## Disk

The stackl machine can access a simulated disk. The disk data is contained in a file named stackl.disk. Utilities are provided to initialize the drive and copy data to it. The disk feature is enabled with the following pragma:

#pragma feature disk

The makedisk utility can be used to create a blank disk. The copy2disk utility can be used to copy data onto the disk.

This device uses interrupt vector 2.

### Registers

The disk hardware is controlled by four registers. If byte instructions are used to access the registers, the behavior is undefined.

#### DISK\_STATUS

This register is at address 0x0D000000. The register is read-only and reading clears the register. The following bits are defined:

0 READ\_BUSY The disk is currently reading a block

1 READ\_DONE The disk has finished reading a block

2 READ\_ERROR The disk detected an error while reading

3 WRITE\_BUSY The disk is currently writing a block

4 WRITE\_DONE The disk has finished writing a block

5 WRITE\_ERROR The disk detected an error while writing

8 BAD\_BLOCK A bad block number was specified

31 ATTN The disk status has changed

#### DISK\_CMD

This register is at address 0xD000004. This register is used to control the disk drive operation. The register is write only. The following bits are defined:

0 INT\_ENA Interrupts are enabled when this bit is set

1 START\_READ A read operation will be started when this bit is set

2 START\_WRITE A write operation will be started when this bit is set

Only a single operation can be performed at a time. Software must guarantee that the previous operation is completed before starting a new operation.

#### DISK\_ADDR

This register is at address 0xD000008. It is used to specify the absolute address of the buffer to be read/written to the disk.

#### DISK\_BLOCK

This register is at address 0x0D00000C. It is used to specify the block number to be read or written.

## Generic IO Device

The Generic IO device is a multi-purpose device for doing input and output. It does not represent any device that would exist on real hardware. Instead, it provides a reasonable interface for performing a variety of functions.

The generic IO device can be enabled with the following pragma:

#pragma feature gen\_io

### Registers

#### Command/Status Register

The command and status register is at address 0x0B00000. The lower byte is for commands. The GEN\_IO device responds to the following commands:

# Interpreter Architecture

This section describes some of the implementation details of the interpreter. It does not define the behavior of the virtual machine, but rather how the virtual machine is implemented.

## Memory

In order to facilitate memory management, all memory references are made through the following four functions:

Get\_Word Returns a word-sized value. The address must be word aligned.

Set\_Word Sets a word-sized value. The address must be word aligned

Get\_Byte Returns a byte-sized value. There are no alignment requirements on the address.

Set\_Byte Sets a word-sized value. There are no alignment requirements on the address.

If the address specified to these functions is outside the range of main memory, then the address will be interpreted as the address of a memory mapped IO register.

There are two functions for managing the machine state. They get/set all the registers. These functions are Get\_Machine\_State and Set\_Machine\_State.

## Executable file format and the Loader

To be specified

## Adding IO devices to the interpreter

To be specified

# Instruction Reference

This section defines the opcodes for the stackl machine. All opcodes occupy a word.

For binary arithmetic operations, the left operand is always pushed onto the stack first.

Some opcodes take a parameter from the instruction stream (that is, they are two word instructions). These instructions are indicated by <param> in the opcode name where <param> is replaced by the name of the parameter. The parameter can be found at Memory[IP+1].

In the descriptions of the opcodes, memory offsets are given in word-sized increments. In other words, “IP+1” means the word following the IP, not the byte following the IP. Offsets relative to the IP assume the IP is pointing at the opcode.

In the descriptions of the effects of the opcodes, the operators have the meaning defined by the C language.

The numerical value of specific opcodes should not relied upon because they are subject to change in future versions of stackl.

## ADJSP <amount>

This opcode is used to adjust the stack pointer. This opcode has the following effect:

SP += <amount>

The <amount> can be a positive or negative number. The <amount> is in bytes, not words. If <amount> is not a multiple of the word size, the machine behavior will be undefined following this instruction.

## AND

This opcode performs a logical AND. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] && Memory[SP-1]

SP--

IP++

## BAND

This opcode performs a binary AND. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] & Memory[SP-1]

SP--

IP++

## BOR

This opcode performs a binary OR. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] | Memory[SP-1]

SP--

IP++

## BXOR

This opcode performs a binary XOR. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] ^ Memory[SP-1]

SP--

IP++

## CALL <addr>

This opcode is used for making function calls. This opcode has the following effect:

Memory[SP] = IP+1 // return address

Memory[SP+1] = FP

SP += 2

FP = SP

IP = <addr>

## CALLI

This opcode is used for making function calls. The address of the called function is taken off the stack. This opcode has the following effect:

temp = Memory[SP-1]

SP -= 1

Memory[SP] = IP+1 // return address

Memory[SP+1] = FP

SP += 2

FP = SP

IP = temp

## CLID

This opcode is used to clear the interrupt disable bit in the FLAG register. The current value of the flag is left on the top of the stack. This opcode has the following effect:

Memory[SP] = FLAG & FL\_INT\_DIS

FLAG &= ~FL\_INT\_DIS

SP++

IP++

Following this instruction, interrupts will be enabled (meaning that if an interrupt occurs, the interrupt service routine will be called).

## COMP

This opcode complements the value at the top of the stack. This opcode has the following effect:

Memory[SP-1] = ~Memory[SP-1]

IP++

Following this instruction, interrupts will be enabled (meaning that if an interrupt occurs, the interrupt service routine will be called).

## DIVIDE

This opcode performs a divide. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] / Memory[SP-1]

SP--

IP++

## DUP

This opcode duplicates the value at the top of the stack. This instruction has the following effect:

Memory[SP] = Memory[SP-1]

SP++

IP++

## EQ

This opcode performs an equivalence check. This instruction has the following effect:

Memory[SP-2] = (Memory[SP-2] == Memory[SP-1])

SP--

IP++

## GE

This opcode performs a greater than or equal to check. This instruction has the following effect:

Memory[SP-2] = (Memory[SP-2] >= Memory[SP-1])

SP--

IP++

## GT

This opcode performs a greater than check. This instruction has the following effect:

Memory[SP-2] = (Memory[SP-2] > Memory[SP-1])

SP--

IP++

## HALT

This opcode will cause the processor to stop executing instructions. The simulator will exit.

## ILLEGAL

This opcode will cause an illegal instruction exception and the simulator will exit.

## INP

This opcode will initiate an IO operation. The value at the top of the stack must be the address of an io\_blk. See the section on IO processing for a description of the IO blocks and their function.

Initiate IO Operation

SP--

IP++

## JMPUSER <dest>

This opcode causes the processor state to switch to user mode and then jump to an address in the user modes address space. This instruction has the following effect:

FL\_USER\_MODE bit is set in FLAG register

IP = <dest>

## JUMP <dest>

This is the jump instruction. It has the following effect:

IP = <dest>

## JUMPE <dest>

This is the conditional jump instruction. It will jump if the value on the top of the stack is equal to zero. It has the following effect:

SP--

val = Memory[SP]

if (val == 0)

IP = <dest>

else

IP += 2

## LE

This opcode performs a less than or equal to check. This instruction has the following effect:

Memory[SP-2] = (Memory[SP-2] <= Memory[SP-1])

SP--

IP++

## LT

This opcode performs a less than check. This instruction has the following effect:

Memory[SP-2] = (Memory[SP-2] < Memory[SP-1])

SP--

IP++

## MINUS

This opcode performs a subtraction. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] - Memory[SP-1]

SP--

IP++

## MOD

This opcode performs a modulus. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] % Memory[SP-1]

SP--

IP++

## NE

This opcode performs a not equal to check. This instruction has the following effect:

Memory[SP-2] = (Memory[SP-2] != Memory[SP-1])

SP--

IP++

## NEG

This opcode negates the value on the top of the stack. This instruction has the following effect:

Memory[SP-1] = - Memory[SP-1]

IP++

## NOP

This opcode performs no operation.

IP++

## NOT

This opcode performs a logical negation of the value on the top of the stack. This instruction has the following effect:

Memory[SP-1] = ! Memory[SP-1]

IP++

## OR

This opcode performs a logical OR. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] || Memory[SP-1]

SP--

IP++

## OUTS

This opcode sends output to the console. The value at the top of the stack is assumed to be the address of a NULL terminated string. The string at that address will be sent to the console. This is a blocking operation. The CPU will not execute another instruction until after the IO is complete.

## PLUS

This opcode performs an addition. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] + Memory[SP-1]

SP--

IP++

## POP

This opcode pops the stack. The value that is popped in not saved. This opcode has the following effect:

SP--

IP++

## POPARGS <size>

This opcode is used following the return from a function to remove the arguments passed to the function from the stack while preserving the return value of the function. The <size> can be a positive or negative number. The <amount> is in bytes, not words. If <size> is not a multiple of the word size, the machine behavior will be undefined following this instruction.

This opcode has the following effect:

SP--

Temp = Memory[SP]

SP -= size

Memory[SP] = Temp

SP++

IP++

## POPCVAR <offset>

This opcode pops a value off the stack and stores it in a byte variable. The offset to the variable is taken from the stack. Note that a word-sized value is popped, but the destination is a single byte.

The opcode has the following effect:

SP--

Memory[FP+offset] = Memory[SP]

IP++

## POPCVARIND

This opcode pops a value off the stack and stores it in a byte-sized variable. The offset to the variable is taken from the stack. Note that a word-sized value is popped, but the destination is a single byte. To use this opcode, the value should be pushed on the stack and then the offset. The opcode has the following effect:

Memory[Memory[SP-1]] = Memory[SP-2]

SP--

IP++

## POPREG <reg num>

This opcode pops the value from the top of the stack into the specified register. The registers are numbered as follows:

0 BP register

1 LP register

2 IP register

3 SP register

4 FP register

5 FLAG register

<register> = Memory[SP]

SP--

IP++

## POPVAR <offset>

This opcode pops a value off the stack and stores it in the word indicated by FP+<offset>. The opcode has the following effect:

SP--

Memory[FP+offset] = Memory[SP]

IP++

## POPVARIND

This opcode pops a value off the stack and stores it in a variable. The offset to the variable is taken from the stack. To use this opcode, the value should be pushed on the stack and then the offset. The opcode has the following effect:

Memory[Memory[SP-1]] = Memory[SP-2]

SP--

IP++

## PUSH <value>

This opcode pushes a value onto the stack. This opcode has the following effect:

Memory[SP] = <value>

SP++

IP++

## PUSHCVAR <offset>

This opcode pushes a byte onto the stack. The byte is located at FP+<offset>. To preserve stack alignment, the SP is increased by a word-size amount. The value of the unused bytes is unspecified.

This opcode has the following effect:

Memory[SP] = Memory[FP+<offset>]

SP++

IP++

## PUSHCVARIND

This opcode pushes a byte onto the stack. The offset to the byte comes from the stack. A single byte is pushed onto the stack but the SP is updated by a word-sized amount. The value of the unused three bytes is unspecified. To use this opcode, push the address of the value to be pushed onto the stack. This opcode has the following effect:

Memory[SP-1] = Memory[ Memory[SP-1] ]

IP++

## PUSHFP

This opcode pushes the frame pointer onto the stack. This opcode has the following effect:

Memory[SP] = FP

SP++

IP++

## PUSHREG <reg num>

This opcode pushes the value of the specified register onto the stack. The registers are numbered as follows:

0 BP register

1 LP register

2 IP register

3 SP register

4 FP register

5 FLAG register

Memory[SP] = <register>

SP++

IP++

## PUSHVAR <offset>

This opcode pushes a word onto the stack. The word is located at FP+<offset>. This opcode has the following effect:

Memory[SP] = Memory[FP+<offset>]

SP++

IP++

## PUSHVARIND

This opcode pushes a variable onto the stack. The offset to the variable comes from the stack. To use this opcode, push the address of the value to be pushed onto the stack. This opcode has the following effect:

Memory[SP-1] = Memory[ Memory[SP-1] ]

IP++

## RETURN

This opcode is used to return from a void function. This opcode has the following effect:

SP = FP-1

IP = Memory[FP-2]

FP = Memory[FP-1]

## RETURNV

This opcode is used to return from a function when the return function returns a value. The value to be returned is assumed to be at the top of the stack at the beginning of the instruction, and it is left on the top of the stack at the end of the function (after execution returns to the caller).

This opcode has the following effect:

temp = Memory[SP-1]

SP = FP-1

IP = Memory[FP-2]

FP = Memory[FP-1]

Memory[SP-1] = temp

## RTI

This opcode is used to return from interrupt mode. Both the interrupt and systrap functions should use this opcode to return. See the Interrupt section for more details

## SEID

This opcode is used to set the interrupt disable bit in the FLAG register. The current value of the flag is left on the top of the stack. This opcode has the following effect:

Memory[SP] = FLAG & FL\_INT\_DIS

FLAG |= FL\_INT\_DIS

SP++

IP++

Following this instruction, interrupts will be disabled.

## SHIFTL

This opcode performs a bit shift left. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] << Memory[SP-1]

SP--

IP++

## SHIFTR

This opcode performs a bit shift right. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] >> Memory[SP-1]

SP--

IP++

## SWAP

This opcode swaps the two items at the top of the stack.

temp = Memory[SP-2]

Memory[SP-2] = Memory[SP-1]

Memory[SP-1] = temp

IP++

## TIMES

This opcode performs a multiply. This instruction has the following effect:

Memory[SP-2] = Memory[SP-2] \* Memory[SP-1]

SP--

IP++

## TRACE\_OFF

This instruction turns of opcode trace mode (normally enabled with the -opcodes command line option).

## TRACE\_ON

This instruction turns on opcode trace mode (normally enabled with the -opcodes command line option).

## TRAP

This opcode performs a trap. It has the effect of calling the systrap() function. See the syscall section of this document for more details.

# stacklc grammar

The stacklc grammar is defined below:

program:

global\_decls

| /\* empty \*/

block:

stmts close

| '{' '}'

open:

'{'

close:

'}'

decls:

decls decl

| decl

decl:

var\_decl ';'

| var\_decl '=' expr ';'

| struct\_decl ';'

| typedef ';'

| error ';'

var\_decl:

type IDENTIFIER

| var\_decl '[' constant\_expression ']'

| func\_pointer

| STRUCT type IDENTIFIER

type:

type '\*'

| TYPE\_ID

struct\_decl:

struct\_header open decls close

struct\_header:

STRUCT IDENTIFIER

| STRUCT

global\_decls:

global\_decls global\_decl

| global\_decl

typedef:

TYPEDEF type IDENTIFIER

| TYPEDEF struct\_decl IDENTIFIER

global\_decl:

func\_decl

| STATIC func\_decl

| struct\_decl ';'

| typedef ';'

| CONST type IDENTIFIER '=' constant\_expression ';'

| DEFINE IDENTIFIER INT\_VAL

| DEFINE IDENTIFIER '-' INT\_VAL

| EXTERN var\_decl ';'

| STATIC var\_decl ';'

| var\_decl ';'

| PRAGMA ONCE { $$ = new cPragma("once", "");

| PRAGMA INTERRUPT IDENTIFIER

| PRAGMA SYSTRAP IDENTIFIER

| PRAGMA STARTUP IDENTIFIER

| PRAGMA FEATURE IDENTIFIER

| PRAGMA LIBRARY STRING\_LIT

| PRAGMA STACK\_SIZE INT\_VAL

| error ';'

func\_decl:

func\_header ';'

| func\_header '{' stmts '}'

| func\_header '{' '}'

func\_header:

func\_prefix paramspec\_list ')'

| func\_prefix ')'

func\_prefix:

type IDENTIFIER '('

func\_pointer:

type '(' '\*' IDENTIFIER ')' '(' paramspec\_list ')'

| type '(' '\*' IDENTIFIER ')' '(' ')'

paramspec\_list:

paramspec\_list',' paramspec

| paramspec

paramspec:

var\_decl

stmts:

stmts stmt

| stmt

stmt:

decl

| ';'

| IF '(' expr ')' stmt ELSE stmt

| IF '(' expr ')' stmt

| FOR '(' expr ';' expr ';' expr ')' stmt

| WHILE '(' expr ')' stmt

| DO stmt WHILE '(' expr ')' ';'

| expr ';'

| block

| RETURN expr ';'

| RETURN ';'

| asm\_stmt ';'

asm\_stmt:

ASM '(' string\_lit ')'

| ASM '(' string\_lit ',' params ')'

| ASM2 '(' string\_lit ',' constant\_expression ')'

| ASM2 '(' string\_lit ',' constant\_expression ',' params ')'

lval:

unary\_expression

params:

params',' assignment\_expression

| assignment\_expression

string\_lit:

STRING\_LIT

primary\_expression:

IDENTIFIER

| INT\_VAL

| string\_lit

| '(' expr ')'

postfix\_expression:

primary\_expression

| postfix\_expression '[' expr ']'

| postfix\_expression '(' params ')'

| postfix\_expression '(' ')'

| postfix\_expression '.' IDENTIFIER

| postfix\_expression PTR IDENTIFIER

| postfix\_expression INC

| postfix\_expression DEC

unary\_expression:

postfix\_expression

| INC unary\_expression

| DEC unary\_expression

| '+' cast\_expression

| '-' cast\_expression

| '~' cast\_expression

| '!' cast\_expression

| '\*' cast\_expression

| '&' cast\_expression

| SIZEOF unary\_expression

| SIZEOF '(' type ')'

cast\_expression:

unary\_expression

| '(' type ')' cast\_expression

multiplicative\_expression:

cast\_expression

| multiplicative\_expression '\*' cast\_expression

| multiplicative\_expression '/' cast\_expression

| multiplicative\_expression '%' cast\_expression

additive\_expression:

multiplicative\_expression

| additive\_expression '+' multiplicative\_expression

| additive\_expression '-' multiplicative\_expression

shift\_expression:

additive\_expression

| shift\_expression LEFT additive\_expression

| shift\_expression RIGHT additive\_expression

relational\_expression:

shift\_expression

| relational\_expression '<' shift\_expression

| relational\_expression '>' shift\_expression

| relational\_expression LE shift\_expression

| relational\_expression GE shift\_expression

equality\_expression:

relational\_expression

| equality\_expression EQ relational\_expression

| equality\_expression NE relational\_expression

and\_expression:

equality\_expression

| and\_expression '&' equality\_expression

exclusive\_or\_expression:

and\_expression

| exclusive\_or\_expression '^' and\_expression

inclusive\_or\_expression:

exclusive\_or\_expression

| inclusive\_or\_expression '|' exclusive\_or\_expression

logical\_and\_expression:

inclusive\_or\_expression

| logical\_and\_expression AND inclusive\_or\_expression

logical\_or\_expression:

logical\_and\_expression

| logical\_or\_expression OR logical\_and\_expression

conditional\_expression:

logical\_or\_expression

constant\_expression:

conditional\_expression

assignment\_expression:

conditional\_expression

| lval '=' expr

| lval '=' asm\_stmt

| lval PLUS\_EQ expr

| lval MINUS\_EQ expr

| lval TIMES\_EQ expr

| lval DIVIDE\_EQ expr

| lval MOD\_EQ expr

| lval OR\_EQ expr

| lval AND\_EQ expr

| lval XOR\_EQ expr

| lval LEFT\_EQ expr

| lval RIGHT\_EQ expr

expr:

assignment\_expression