YASM Specification

Brian Howard

Fall 2015, Version 1.1

The intermediate code for the YASL compiler is code for a virtual machine called YASM (Yet Another Stack Machine). The machine's primary storage is a stack of words, each of which holds a single integer (at least 16 bits wide, signed). There are also two word-sized registers, the stack pointer (SP) and the program counter (PC), plus a zero-indexed array of at least ten word-sized locations called the display. The program code is stored in a read-only section of memory, indexed by an integer; the first address of a loaded program is 0. The stack grows downward from a high address (STACKLIMIT $-$ 1); it may be stored in the same address space as the program, although it is assumed that the stack will never grow down far enough to interfere with the loaded program (an implementation may or may not check for this stack overflow condition; similarly, it is unspecified whether the implementation will check for stack underflow, where SP exceeds STACKLIMIT).

In the following, the action "pop an integer off the stack" means to retrieve the integer at location SP, then add one to SP; therefore, SP always points to the top entry on the stack (except when the stack is empty, where SP = STACKLIMIT). The action "push an integer on the stack" means to first subtract one from SP and then store the integer at the location given by the updated SP. When the machine starts, the PC is 0 and the SP is STACKLIMIT. The program is executed by repeatedly retrieving an instruction from location PC in the program code, adding one to PC, and then executing the instruction; this continues until the machine executes the `HALT` instruction or until there is no instruction at location PC (because it has passed the end of the loaded program code).

The following are the instructions of YASM, together with their effect on the stack and other storage locations. Each instruction is encoded within a single word of the program code; the details of the encoding are implementation-specific.

- `LABEL` *name*: Executing this instruction has no effect; its sole purpose is to serve as the target for branch instructions.
- `BRANCH` *name*: Set PC to the location of the first `LABEL` instruction with the given name; if the label does not exist, the program halts.
- `BRANCHZERO` *name*: Pop an integer off the stack; if it is zero, then perform the same action as `BRANCH` *name*.

- `BRANCHNEG` *name*: Pop an integer off the stack; if it is negative, then perform the same action as `BRANCH` *name*.
- `CALL` *name*: Push the PC on the stack, then perform the same action as `BRANCH` *name*.
- `RETURN`: Pop an integer off the stack and use it as the new value of PC.
- `RESERVE` n: Subtract the integer n from SP.
- `DROP` n: Add the integer n to SP.
- `ENTER` n: Push the value at offset n in the display, then store the value of SP into the display at offset n.
- `EXIT` n: Pop an integer off the stack and store it at offset n in the display.
- `ADDRESS` n, x: Push the value x plus the value at offset n in the display.
- `LOAD`: Pop an integer off the stack and use it as a location within the stack space; push the value found at that location on the stack.
- `STORE`: Pop an integer off the stack and use it as a location within the stack space; pop another integer off the stack and store it at that location.
- `CONSTANT` n: Push the value n on the stack.
- `ADD`: Pop two integers off the stack, add the first to the second, and push the result on the stack.
- `SUB`: Pop two integers off the stack, subtract the first from the second, and push the result on the stack.
- `MUL`: Pop two integers off the stack, multiply the first by the second, and push the result on the stack.
- `DIV`: Pop two integers off the stack, divide the first into the second, and push the integer quotient on the stack. The quotient is truncated toward zero, just as in Java.
- `MOD`: Pop two integers off the stack, divide the first into the second, and push the integer remainder on the stack. The remainder is the number that has to be added to the quotient times the divisor to get back to the original dividend; it will be negative if the dividend is negative and not evenly divisible by the divisor (again, this is the same as in Java).
- `READINT`: Read the next integer, followed by a newline, from the console and push it on the stack.

- `WRITEINT`: Pop an integer off the stack and print it to the console.
- `WRITECHAR`: Pop an integer off the stack and print the character with that ASCII code to the console (it is implementation-dependent whether codes beyond the ASCII range may be printed and what encoding they use if so).
- `READLINE`: Read a line of input (terminated by a newline) from the console.
- `WRITELINE`: Write a newline character to the console.
- `HALT`: Stop execution of the program.