Assembly Language Manual for the STACK Computer

Assembly language programmers should read the hardware description of the STACK computer together with information about the effect of each instruction and the calling sequence prior to writing any programs. This document assumes familiarity with the STACK computer and its operations.

Introduction

Machine language programs are written in binary. The instructions must be placed in memory and the program counter set to the address of the first instruction. Data may also be loaded into memory. To facilitate this process a load module format is used in the machine program file. This is so named because the loader is used to read the file on disk and transfer it into memory.

Load Module Format:

address contents
address contents
...
address contents
address

The last line is just an address. This is the initial value of the program counter and hence the address of the first instruction to be executed. The “contents” can be instructions or data. Here is a load module for a program that adds the values in locations 0 and 1 and puts the result in location 2.

00000000 000000000011
00000001 100000000001
00000111 000100000000
00000100 000100000001
00000101 100000000010
00000110 001000000010
00000111 100000000001
00000011

Notice that we have initialized location 0 to the value 3 in binary and the location 1 is initialized to -1 (signed magnitude is used.) The program’s first instruction is found at memory location 3. Why?

Assembly language allows the program to write machine instructions in an English-like form. A translator called an assembler then will produce the load module. The load module above was produced from the following assembly language program.

; this is a comment
X WORD 3
Y WORD -1
Z ALLOC 1
ST PUSH X
PUSH Y
ADD
POP Z
HALT
START ST
Assembly language programming for the STACK computer assumes one assembly language instruction per machine language instruction and each instruction will be on a single line. In addition to instructions the assembly language program may contain assembler directives. Directives are used to allocate memory for data storage and to indicate which instruction is to be executed initially. These are instructions to the assembler itself. In our example we see the three directives: WORD, ALLOC, and START.

A fundamental principle of assembly language programming is that the programmer must be aware of how the machine is designed and how each instruction interacts with the machine’s hardware. The assembler will detect syntactical errors and undefined labels, but it will not prevent the programmer from doing stupid things such as trying to execute the contents of a data location as if it held an instruction or having the run-time stack overwrite portions of the program itself.

**Basic Instruction Formats and Comments**

A programmer may insert comments which are ignored by the assembler. These comments extend only to the end of the current line and are initiated by the semicolon character ‘;’.

Any blank lines are ignored by the assembler.

Whitespace (space, tab characters) is used as a separator between the three key components of each instruction:

- label (optional)
- opcode (required)
- operand (optional with some instructions, required by others)

If an instruction has a label, it must be the very first thing on the line, i.e., it must not even have whitespace before it.

An opcode must have whitespace before it on the line.

For convenience we will require all alphabetic characters to be upper case. (We could allow lower case, but even then we would not distinguish between upper and lower case when looking at labels or opcodes.)

**Labels**

Labels may contain upper case alpha characters and digits and the underscore character ‘_’ provided the first character is an alpha character. The length of a label is not restricted.

**Opcodes**

An opcode is a symbolic name for an operation code or machine instruction. There are 24 opcodes for the STACK computer. The individual operation’s effects and the corresponding binary encoding can be found in the description of the STACK computer. For convenience we list them according to their operands.

No Operand:

- NOP
- HALT
- ADD
- SUB
NEG
MUL
DIV
TEST
RET
ARG
RESET
IN
OUT

Signed Constant Operand:
  PUSHI

Operand is a Label or a Label+UnsignedNumber Representing an Address :
  PUSH
  POP
  JMP
  JMPZ
  JMPN
  CALL

Operand is Unsigned Constant:
  INC-SP
  DEC-SP

Operand is Signed Constant:
  PUSH-REL-SP
  POP-REL-SP

**Operand**

The operands that are signed or unsigned constants must be in the appropriate range for the STACK computer. Within the instruction format these are 8-bit values.

If an operand is a label + an unsigned number, there must be no whitespace involved. Examples of allowable operands of this type are:

\[
X+5
\]

\[
BOXES+18
\]

This is fairly primitive but does allow accessing cells of an array whose base address is given by the label.

**Assembler Directives**

There are only three assembler directives: WORD, ALLOC, and START. The first two require a label. These directives are located on the line according to the same rules as opcodes.

The formats are:

\[
\begin{align*}
\text{label} & \quad \text{WORD} & \quad \text{numeric\_value} \\
\text{label} & \quad \text{ALLOC} & \quad \text{positive\_numeric\_value} \\
\text{START} & \quad \text{label}
\end{align*}
\]
The WORD directive allocates a single cell of memory and gives it an initial value. The label provides a way to reference that cell.

The ALLOC directive allocates the specified number of contiguous memory calls each of which is initialized to zero. The label references the first cell.

The START directive indicates the label associated with the first instruction that is to be executed when the program is run.

**Style**

Assembly language programs can adopt a number of styles and organizations for programs, but we suggest the following basic style model.

```asm
; File: dumb.asm
; Programmer: U. R. Student
; Description:
; This program adds 23 to a value obtained from the
; user and displays the sum on output.

;;;;; data section ;;;;;;
X WORD -60
BOXES ALLOC 10
TEMP WORD 0

;;;;; instruction section ;;;;;
FIRST PUSHI 23
IN
ADD
OUT
HALT ; end of execution

START FIRST
```

In practice the data section could go after the instruction section, but they should not be intermixed. Remember that blank lines are allowed anywhere. Comments start with a semicolon and extend to the end of the current line.
Examples

We look at four additional examples which show branching, a simple procedure call, a computation using iteration, and the same computation using recursion.

Example 1

; Based on the value of an input the program
; displays 1 for a positive value, 0 for a zero value,
; and -1 for a negative value

ST   IN
TEST
JMPZ ZERO
JMPN NEG
RESET
PUSHI 1
OUT
JMP QUIT
ZERO RESET
PUSHI 0
OUT
JMP QUIT
NEG RESET
PUSHI -1
OUT
QUIT
HALT

Example 2

; We use a procedure with two parameters called SUM.
; It returns the sum of its two arguments. It uses no local variables.

;; function SUM definition
SUM   PUSH-REL-SP-2; get first arg
PUSH-REL-SP-3; get second arg
ADD
RET

;; main method
MAIN   IN
ARG
IN
ARG
PUSHI 2
CALL SUM
OUT
HALT

START MAIN
Example 3

; Iteration example
; We input an integer N and compute 1+2+...+N. If N<=0, the result
; will be set to zero. We output the resulting sum.
; code is similar to:
; input N
; set SUM to 0 and COUNTER to 1
; while COUNTER<=N
; 
; SUM = SUM + COUNTER
; COUNTER = COUNTER + 1
; output SUM
; stop

SUM WORD 0
N WORD 0
COUNTER WORD 1

ST IN ; obtain N from user
POP N
LOOP PUSH N
PUSH COUNTER
SUB ; compute N-COUNTER
JMPN FINISH ; discard N-COUNTER to save stack space
RESET PUSH SUM
PUSH COUNTER
ADD
POP SUM
PUSH COUNTER
PUSHI 1
ADD
POP COUNTER ; COUNTER++
JMP LOOP
FINISH PUSH SUM
OUT
HALT

START ST

Example 4

; Recursive procedure demonstrated

MAIN IN
ARG
PUSHI 1
CALL RECURSUM
OUT
HALT

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

RECURSUM ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; computes the sum of the numbers from 1 to its parameter N
;; using recursion. If N is 0 or negative, the method returns 0
;; int RECURSUM(int N)
;; {
;;     if (N<=0) return 0;
;;     else return N + RECURSUM(N-1);
;; }

RECURSUM    PUSH-REL-SP -2
TEST
JMPZ       ZERO
JMPN       ZERO
; call RECURSUM(N-1). Note N is on top of CPU stack
PUSHI      -1  ; N-1 on top if cpu stack
ARG
PUSHI      1
CALL       RECURSUM
PUSH-REL-SP -2 ; get N onto cpu stack
ADD
RET

ZERO       POP-REL-SP 1 ; get N off cpu stack
PUSHI      0
RET

START       MAIN