

MFM C-- Compiler For the SPARC architecture

April 30.2002

Feng He
Melanie Mason
Michael Weissberger

1.0 Overview of the MFM SPARC compiler. . . .

Compile Overview:

The MFM SPARC compiler runs in several stages, which are all accessible through command-line options. They are, in order, Compile, Assemble, Link. The MFM SPARC compiler uses the filenames a.s and a.o as intermediate files. Any files by this name will be overwritten.

1.1 Usage of the MFM SPARC compiler. . . .

Command Line Format:

```
mfmsparc [ INFILE | -oOUTFILE | -c | -s | -v ]
```

All command line switches and options are optional.

INFILE: Specifies the input file for compilation. If not supplied, takes input from the standard input.

-oOUTFILE: Specifies the output of a linking run. Ignored if *-c* or *-s* options are enabled. If not supplied, a.out is used as the default binary filename.

-v: Verbose mode: Outputs compilation, parsing, symbol table, and intermediate information generated during the compilation process.

-c: Performs Compilation and Assembly, but not linkage. The resulting output file is an object file, 'a.o'.

-s: Performs Compilation, but not Assembly or linkage. The resulting output file is an assembly file, 'a.s'.

All other switches: Displays the command line format string.

Note: Switches such as *-s******, or *-c******, where ******* is any number of characters, has the same effect as the *-s* or *-c* switches alone.

SPARC Stack usage conventions

For the MFM C-- Compiler environment

The stack manipulation scheme discussed in this document is implemented for the C-- Grammar, source is available upon request.

1.0 Sections of the Sparc C--stack frame

Register saving space: The SPARC machine grants 24 registers to each stack frame, and 8 registers global to each process. The machine only holds around 120 registers, so once more than 5 function calls have passed; the stack needs to be utilized to store registers. Each time a save assembly instruction is executed; the frame's registers are saved in the bytes immediately below the stack pointer (%sp). This space is required.

Operand Stack: In order to simplify register usage, an operand stack is used to hold operands inside a statement. As a result, there are many more load and store instructions to the operand stack than would normally exist in a hand optimized program. An Operand Stack such as this is necessitated by the possibility of an extremely long expression, whose rightmost pair is first to be executed, resulting in many operands parsed before any are collapsed up the tree. Storing intermediate operands in registers would limit the number of operands to the number of available registers.

Parameters: Space in the current stack frame is allocated for any function calls which involve parameters.

Local and Temporary Variables: Space on the stack is allocated for any local and temporary variables defined in this stack space.

1.1 Addressing Different Sections of the Stack

Local Variables: $fp - \text{variable address}$

Local variables exist in memory below the frame pointer, not including $fp - 0$. The first addressable local variable begins at $fp - 4$ and ends at $fp - 0$.

Parameters: $fp + 96 + \text{Caller's OpSpace} + (-\text{address})$

Parameters for the current stack frame exist in the stack frame immediately above it on the stack (in higher memory). Normally, we address only negative offsets from the frame pointer. However, by using positive offsets, we can access the stack frame above the current frame, and access parameters. In order to do this, we need two pieces of information: The address of the parameter and the Caller's Operand Stack Space. The address of the parameter is a negative number corresponding to the parameter, left to right.

ie: in foo(int x, int y); x has address 0, and y has address -4.

(In reality, x has address -4 and y has address -8. 4 is added to these values whenever they are used. This quirk exists because giving parameters a negative address is a quick way to show they are parameters.) The Caller pushes the parameters on the operand stack immediately before it calls the function. However, if there are operands on the stack before the function call, the parameters will not be immediately below the register saving space. This is why we need the Caller's OpSpace, which is passed in register %r10.

Operand Stack Space: $sp + 96 + OpSpace$

The next available space on the Operand Stack is reached through this code. The next instruction is always an increment (or decrement) of the Operand Stack Offset, if necessary.

1.2 Notes

Return Values: Placed in the callee's %i0 register.

Return values are placed on the callee's %i0 register. When the return instruction is executed, the callee's %i0 register, through the magic of register windowing, becomes the caller's %o0 register. It is then the caller's responsibility to place the %o0 register onto the top of the OpStack.

Global Variables:

Global variables are placed in the common storage area, generated by the keyword .common.

Variable Sizes:

The two variable types in C-- are char and int. Normally, character is a single byte long and integer is four bytes long. In order to keep alignment of the stack simple, all both characters and integers are four bytes long.

1.3 Example Stack Diagram

