

Mochi/Miloc

User Manual

Copyright © 2010 Lamont Samuels

Toolbar icons Copyright © 2009 Aha-soft

Table of Contents

| | |
|--|---------------|
| Introduction..... | 3 |
| What is Miloc?..... | 3 |
| The Miloc Simulator | 3 |
| The Miloc Instruction Set | 4 |
| Arithmetic..... | 4 |
| Boolean..... | 4 |
| Comparison and Branching | 4 |
| Loads | 4 |
| Stores..... | 5 |
| Invocation..... | 5 |
| Allocation..... | 5 |
| I/O..... | 5 |
| Moves | 5 |
| Notes..... | 5 |
| Mochi | 6 |
| The Mochi Window | 6 |
| Mochi Toolbar..... | 7 |
| Searching Features..... | 7 |
| Miloc Files..... | 8 |
| How to Create/Setup a File..... | 8 |
| Compiling/Running a Miloc File..... | 8 |
| The Debugger..... | 9 |
| The Debugger Window | 9 |
| The Debugger Toolbar | 10 |
| Using the Debugger..... | 11 |
| Add a Breakpoint..... | 11 |
| Remove a Breakpoint..... | 11 |
| Interacting with the Debugger | 11 |

1 Introduction

This section will introduce you to the Miloc language and the Mochi Simulator used to simulate your Miloc files.

What is Miloc?

Miloc represents an intermediate language that contains all the common intermediate language instructions such as:

- Arithmetic
- Boolean
- Comparison and Branching
- Loads
- Stores
- Invocations
- Allocation
- I/O
- Moves

Instructions are made up of a key word that identifies that instruction and combination parameters that could include: registers, immediates, labels, variable names, global names, struct field names and use of the condition code register.

The Mochi Simulator

The Mochi Simulator allows you to simulate your files that contain Miloc instructions.

Mochi also allows you perform the following actions:

- Create Miloc files using the built in text editor
- Save modified files
- Open Miloc files
- Compile files
- Run files
- Debug files

The application also allows you to search for particular line numbers and functions defined within the file.

2 The Miloc Instruction Set

This section lists the instruction set for the Miloc language.

$r_n \Rightarrow$ register

$i_n \Rightarrow$ immediate

$l_n \Rightarrow$ label

$S_d \Rightarrow$ the name of the structure in memory

variable \Rightarrow the name of a local variable defined in the current function

fieldName \Rightarrow the name of the field in a memory struct.

cc \Rightarrow condition code register

Arithmetic

| | | |
|------|------------------------------|---|
| add | r_1, r_2, r_3 | $r_1 + r_2 \Rightarrow r_3$ |
| addi | r_1, i_1, r_2 | $r_1 + i_1 \Rightarrow r_2$ |
| addi | rarp, variable, r_2 | $@[STACK(rarp + @variable)] \Rightarrow r_2$ * stores the address of the variable |
| addi | $r_1, \text{fieldName}, r_2$ | $@[Memory(r_1 + @fieldName)] \Rightarrow r_2$ * stores the address of the field |
| div | r_1, r_2, r_3 | $r_1 / r_2 \Rightarrow r_3$ |
| mult | r_1, r_2, r_3 | $r_1 * r_2 \Rightarrow r_3$ |
| sub | r_1, r_2, r_3 | $r_1 - r_2 \Rightarrow r_3$ |
| subi | r_1, i_1, r_2 | $r_1 - i_1 \Rightarrow r_2$ |

Boolean

| | | |
|------|-----------------|-----------------------------------|
| and | r_1, r_2, r_3 | $r_1 \wedge r_2 \Rightarrow r_3$ |
| or | r_1, r_2, r_3 | $r_1 \vee r_2 \Rightarrow r_3$ |
| xori | r_1, i_1, r_2 | $r_1 \otimes i_1 \Rightarrow r_2$ |

Comparison and Branching

| | | |
|-------|------------|---|
| comp | r_1, r_2 | set cc |
| compi | r_1, i_1 | set cc |
| cbreq | l_1, l_2 | cc == EQ $\Rightarrow l_1 \rightarrow PC$ otherwise $l_2 \rightarrow PC$ |
| cbrge | l_1, l_2 | cc == GE $\Rightarrow l_1 \rightarrow PC$ |
| cbrgt | l_1, l_2 | cc == GT $\Rightarrow l_1 \rightarrow PC$ |
| cbrle | l_1, l_2 | cc == LE $\Rightarrow l_1 \rightarrow PC$ |
| cbrlt | l_1, l_2 | cc == LT $\Rightarrow l_1 \rightarrow PC$ |
| cbrne | l_1, l_2 | cc == GE $\Rightarrow l_1 \rightarrow PC$ |
| jumpi | l_1, l_2 | $l_1 \rightarrow PC$ |

Loads

| | | |
|----------------|------------------------------|--|
| loadi | i_1, r_1 | $i_1 \Rightarrow r_1$ |
| loadai | r_1, i_1, r_2 | $Memory(r_1 + i_1) \Rightarrow r_2$ |
| loadai | rarp, variable, r_2 | $STACK(rarp + @variable) \Rightarrow r_2$ |
| loadai | $r_1, \text{fieldname}, r_3$ | $Memory(r_1 + @fieldName) \Rightarrow r_3$ |
| loadinargument | variable, i_1, r_1 | $Memory(@arg(i_1)) \Rightarrow r_1$ |
| loadglobal | globalName, r_1 | $Memory(@globalName) \Rightarrow r_1$ |
| loadret | $\Rightarrow r_1$ | $Memory(ret) \Rightarrow r_1$ |

Stores

| | | |
|------------------|-------------------------------------|--|
| storeai | r_1, r_2, i_1 | $r_1 \Rightarrow \text{Memory}(r_2 + i_1)$ |
| storeai | $r_1, \text{rarp}, \text{variable}$ | $r_1 \Rightarrow \text{STACK}(\text{rarp} + \text{@variable})$ |
| storeai | $r_1, r_2, \text{fieldName}$ | $r_1 \Rightarrow \text{Memory}(r_2 + \text{@fieldName})$ |
| storeoutargument | r_1, i_1 | $r_1 \Rightarrow \text{Memory}(\text{@outarg}(i_1))$ |
| storeret | r_1 | $r_1 \Rightarrow \text{Memory}(\text{ret})$ |

Invocation

| | | |
|------|-------|---|
| call | i_1 | $r_1 \Rightarrow \text{PC} \rightarrow \text{retaddr}, i_1 \rightarrow \text{PC}$ |
| ret | | $\text{retaddr} \rightarrow \text{PC}$ |

Allocation

| | | |
|-----|---|---------------------------------------|
| new | $S_d, [\text{fieldName} (, \text{fieldName})^*], r_1$ | allocate and store address in r_1 |
| del | r_1 | deallocate memory at address in r_1 |

I/O

| | | |
|---------|-------|--|
| print | r_1 | output integer in r_1 |
| println | r_1 | output integer in r_1 followed by a newline |
| read | r_1 | store integer at address in r_1 ($\Rightarrow \text{Memory}(r_1)$) |

Moves

| | | |
|-------|------------|--|
| mov | r_1, r_2 | $r_1 \Rightarrow r_2$ |
| moveq | i_1, r_1 | if $\text{CC} == \text{EQ}$ then $i_1 \Rightarrow r_1$ |
| movge | i_1, r_1 | if $\text{CC} == \text{GE}$ then $i_1 \Rightarrow r_1$ |
| movgt | i_1, r_1 | if $\text{CC} == \text{GT}$ then $i_1 \Rightarrow r_1$ |
| movle | i_1, r_1 | if $\text{CC} == \text{LE}$ then $i_1 \Rightarrow r_1$ |
| movlt | i_1, r_1 | if $\text{CC} == \text{LT}$ then $i_1 \Rightarrow r_1$ |
| movne | i_1, r_1 | if $\text{CC} == \text{NE}$ then $i_1 \Rightarrow r_1$ |

Notes

- Comments are defined by the “#” symbol.
- Whitespace is ignored. Instructions and special declarations just need to follow their defined grammar.
- Mochi will print “>>” to indicate a read instruction was just executed in the console window. This is where you can enter in your integer value.

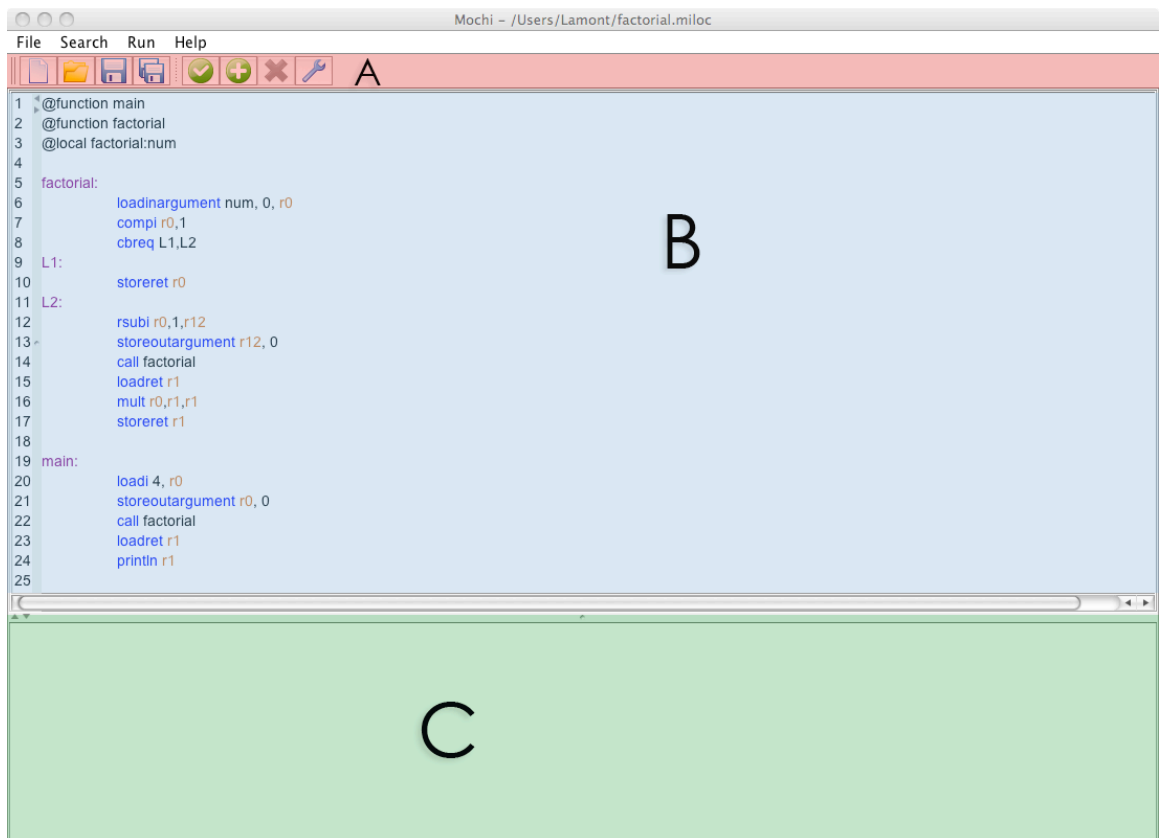
3 Mochi

This section will give an overview of the Mochi Simulator window, its features and functionality.

The Mochi Window

The components that make up the Mochi window include the:

- **(A) Toolbar** – that allows users to create, open, save, save as, compile, run, stop and debug a file
- **(B) Text Editor** – where you can edit and create your miloc files. The line numbers are displayed on the right hand side. The window color-codes function names, labels, instruction terms, and registers.
- **(C) Console Window** – displays the results of running or compiling the file. Users can also type in an integer value when a read instruction executes.



Mochi Toolbar

The following image depicts the Mochi toolbar. The components of the toolbar include:



- A (New Document) – Creates a new document by clearing out the text editor.
- B (Open Document) – Opens a document into the text editor.
- C (Save) – Saves the current opened document.
- D (Save As) – Saves the current document but with a different name and file location defined by you.
- E (Compile) – Compiles the current file by checking to ensure its syntax is correct.
- F (Run) – Compiles the file then runs the file by executing each instruction.
- G (Stop) – Stops the execution of a running file.
- H (Debug) – Opens the debugger window for debugging the file.

Note: All these actions can also be performed within the menu bar of the Mochi window.

Searching Features

Within the Search menu item, you can search based on two different options. First you can search for a particular line number. Once you entered the line number within in the pop-window, the cursor will jump to the line you indicated. Second you can search based off a function name. Once you entered in the name, the cursor will jump to the line where the function name begins.

4 Miloc Files

This section will show you how to create a miloc file and how to compile/run the file within the Mochi Simulator.

How to Create/Setup a Miloc File

A miloc file can be made within the text editor of the Mochi Simulator or any other text editor software, but the file must adhere to the following setup procedures when using the Mochi Simulator:

1. You must list all the function names with the **@function** declaration at the beginning of the file.

```
@function main
```

2. After listing the function names, you must then list all the local variables defined in each function using the **@local FUNCTION_NAME:VARIABLE_NAME** declaration.

```
@local main:myVar
```

3. Once you declared all your functions and variable names, you can begin writing the functions (labels) and instructions associated with those functions.

```
1  @function main
2  @function factorial
3  @local factorial:num
4
5  factorial:
6      loadinargument num, 0, r0
7      compl r0,1
8      cbreq L1,L2
9  L1:
10     storeret r0
11 L2:
12     rsubi r0,1,r12
13     storeoutargument r12, 0
14     call factorial
15     loadret r1
16     mult r0,r1,r1
17     storeret r1
18
19 main:
20     loadi 4, r0
21     storeoutargument r0, 0
22     call factorial
23     loadret r1
24     println r1
25
```

Compiling/Running a Miloc File

After you finish loading or writing your miloc file you can either compile or run the file.

- **Compiling the file** will check for any syntax errors and list them within the console window. The simulator will print "SUCCESS" in the console window if it did not find any syntactical errors in the file.
- **Running the file** will compile the file and if no syntax errors occurred then will begin executing the file. The default starting function is **main**; therefore it will begin executing instructions there first.

Note: You must still declare main in the function declarations.

5 The Debugger

This section will give an overview of the debugger window and the features associated with it.

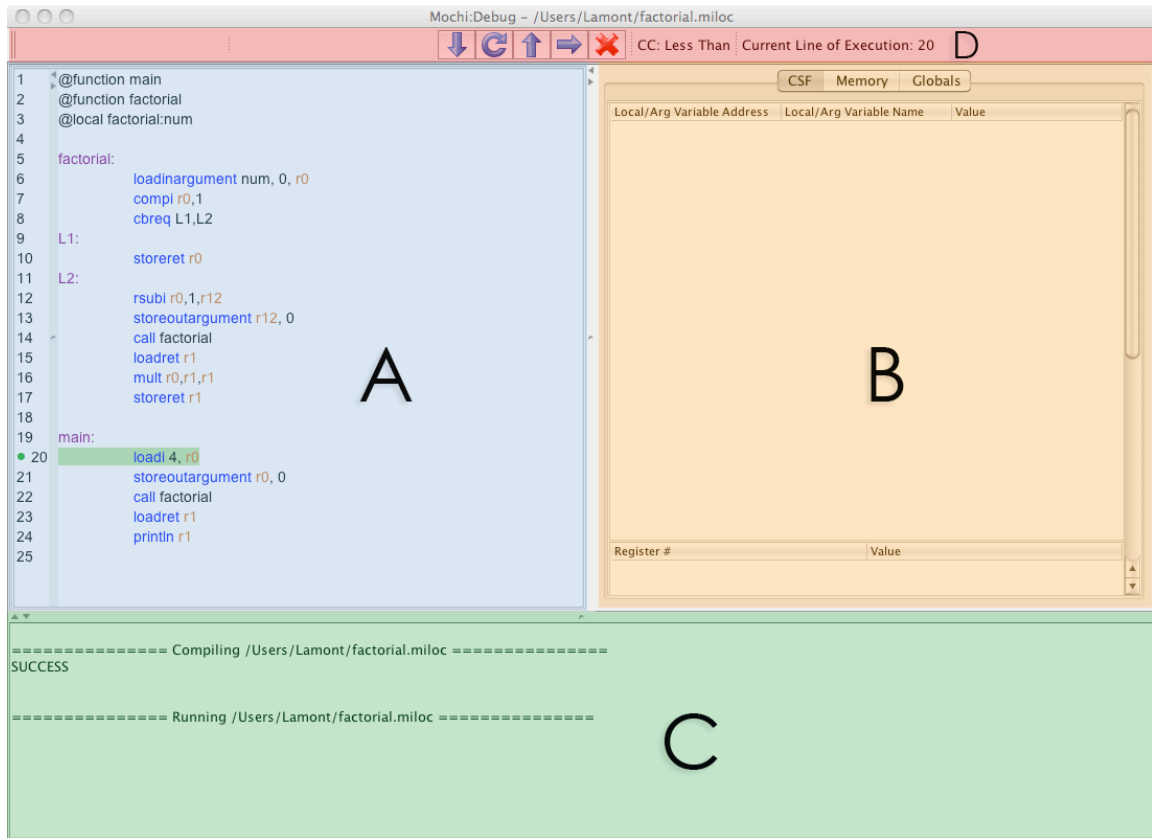
The Debugger Window

The components that make up the Debugger window include the:

- **(A) File of Instructions Panel** – This panel shows all the functions and instructions within the file. Although this window can be edited, the debugger will **not execute** any modifications. If you need to make modifications then do so in the text editor of the main window and rerun the debugger.

Note: The debugger highlights the line that its about to execute in green as indicated in the below figure.

- **(B) The Status Panels** – these panels shows the current state of the CSF (current stack frame), memory, global variables.
 - *Current Stack Frame* displays the address, the name of a local name and/or function argument, and its current value.
 - *Memory* displays all the struct fields that have been allocated and deallocated. The address, field name, and current value are displayed in this panel.
 - *Globals* displays the address, name, and current value of the global variables.
- **(C) Console Window** - that displays the results of debugging the file. Users can also type in an integer value when a read instruction executes.
- **(D) Toolbar** – that allows you to step into, step over, return, continue, and stop the execution of the file.



The Debugger Toolbar

The following image depicts the Debugger toolbar. The components of the toolbar include:



- A (Step Into) – Executes an instruction and calls (steps into) a function when a call instruction was executed. It then begins to execute the functions' instructions.
- B (Step Over) – Executes an instruction but skips (steps over) a function when a call instruction executes. If there are no break points within that function then it executes the function normally but does not allow a line- by-line execution.
- C (Return) – Jumps (Returns) from outside a function.
- D (Continue) – Stops line-by-line execution of the file and continues executing instructions until a breakpoint is hit or it reaches the end of execution.
- E (Stop) – Stop the execution of a debugging file.
- F (CC) – Displays the current value of the CC.
- G (CLE) – Displays the line that is about to be executed.

6 Using the Debugger

This section will show you how to add a breakpoint in a file and run-through a miloc file within the debugger.

Add a Breakpoint

Click on the line number and a green circle will appear next to the line number. This indicates that a breakpoint was added at this line.



Remove a Breakpoint

Double click on the line number to remove the break point.

Interacting with the Debugger

Once you added your breakpoints then click on the debugger icon to begin debugging. Use the debugger toolbar to interact with debugger. If a register, an address in memory, or global was changed after executing a line it will be **highlighted in yellow** within its given status panel.

