Mochi/Miloc User Manual

Table of Contents

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/0
- Moves

Instructions are made up of a key word that indentifies 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 => register
i_n => immediate
l_n => label
S_d => the name of the structure in memory
variable => the name of a local variable defined in the current function
fieldName => the name of the field in a memory struct.
cc => 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)] => r_2 * stores the address of the variable
addi	r_1, fieldName, r_2	@[Memory(r_1 + @fieldName)] => 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	$r_1 - r_2 \Rightarrow r_3$
rsubi	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 => l_1 -> PC otherwise l_2 -> PC
cbrge	l_1, l_2	cc == GE => l_1 -> PC
cbrgt	l_1, l_2	cc == GT => l_1 -> PC
cbrle	l_1, l_2	cc == LE => l_1 -> PC
cbrlt	l_1, l_2	cc == LT => l_1 -> PC
cbrne	l_1, l_2	cc == GE => l_1 -> PC
jumpi	l_1	l_1 -> PC

Loads

loadi	i_1, r_1	$i_1 \Rightarrow r_1$
loadai	r_1, i_1, r_2	Memory ($r_1 + i_1$) => r_2
loadai	rarp, variable, r_2	STACK(rarp + @variable) => r_2
loadai	r_1, fieldname, r_3	Memory(r1 + @fieldName) => r_3
loadinargument	variable, i_1, r_1	Memory(@arg(i_1)) => r_1
loadglobal	globalName, r_1	Memory(@globalName) => r_1
loadret	=> r_1	Memory(ret) => r_1
computeglobaladdress	globalName, r_1	@globalName => r_1

Stores

storeai	r_1, r_2, i_1	r_1 => Memory (r_2 + i_1)
storeai	r_1, rarp, variable	r_1 => STACK(rarp + @variable)
storeai	r_1, r_2, fieldName	r_1 => Memory(r_2 + @fieldName)
storeoutargument	r_1, i_1	r_1 => Memory(@outarg(i_1))
storeglobal	r_1, globalName	r_1 => Memory(@globalName)
storeret	r_1	r_1 => Memory(ret)

Invocation

call	l_1	r_1 => PC -> retaddr, l_1 -> PC
ret		retaddr -> PC

Allocation

new	S_d, [fieldName (,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 (=> Memory(r_1))

Moves

mov	r_1, r_2	r_1 => r_2
moveq	i_1, r_1	if CC == EQ then i_1 => r_1
movge	i_1, r_1	if CC == GE then i_1 => r_1
movgt	i_1, r_1	if CC == GT then i_1 => r_1
movle	i_1, r_1	if CC == LE then i_1 => r_1
movlt	i_1, r_1	if CC == LT then i_1 => r_1
movne	i_1, r_1	if CC == NE then i_1 => r_1

Creating Memory Blocks

The new instruction allows you to create a memory block.

$$\text{new } S_d, [\text{fieldName } (,\text{fieldName})^*], r_1$$

S_d represents the name of the structure or memory block followed by the fields of that memory block and the register to store the beginning location of the memory block.

For example if we had the following structures:

```
struct rectangle {
   int width;
   int height;
};

struct circle {
   int radius;
};
```

To create a new rectangle and circle we would have the following new instructions:

new rectangle [width,height], r1
new circle [radius], r2

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.

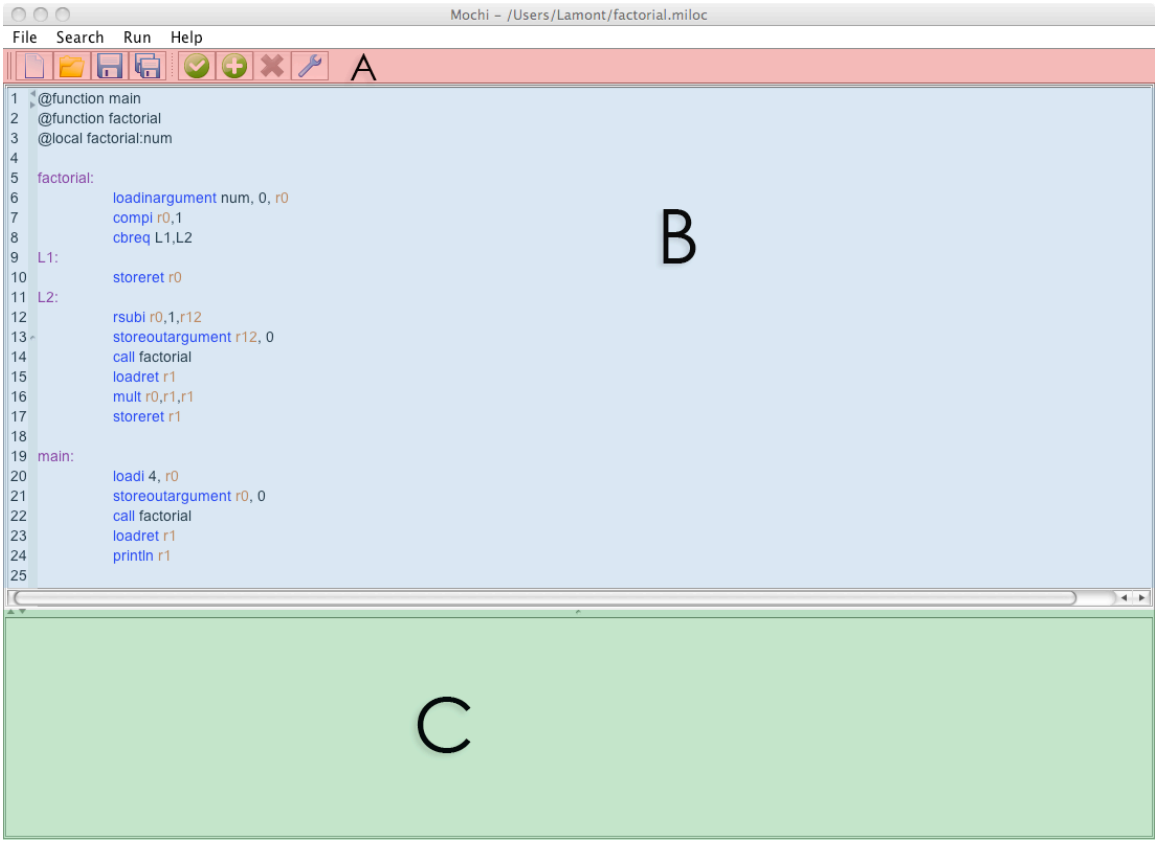
3 Mochi

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.

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:

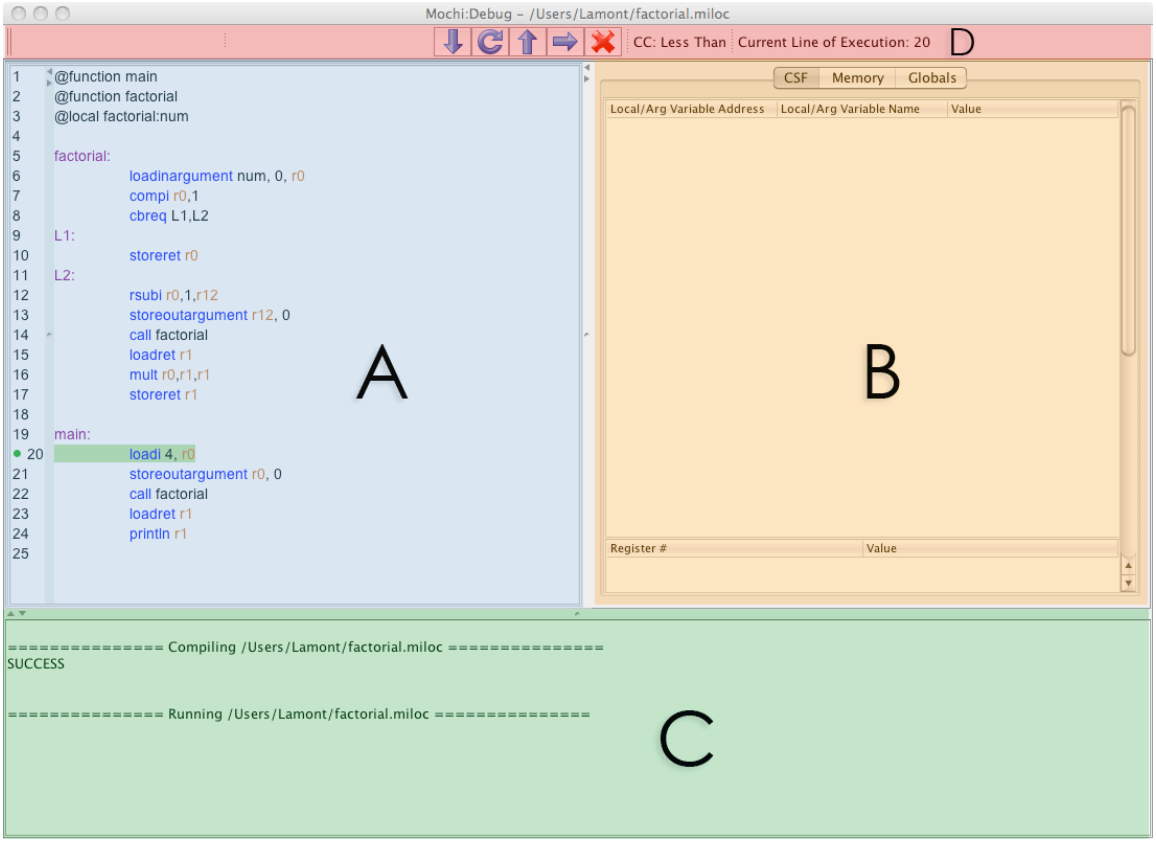
- **(A) File of Instructions Panel** – This panel shows all the functions and instructions within the file. Although this window can be edited, the debugger will **not execute** any modifications. If you need to make modifications then do so in the text editor of the main window and rerun the debugger.

  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:

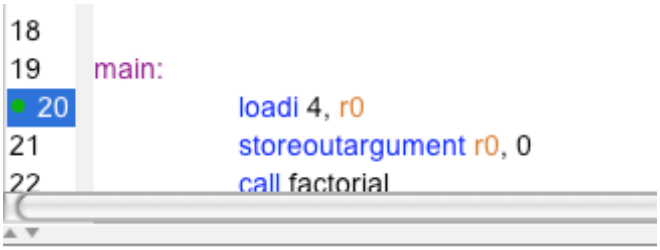
- A (Step Into) – Executes an instruction and calls (steps into) a function when a call instruction was executed. It then begins to execute the functions' instructions.
- B (Step Over) – Executes an instruction but skips (steps over) a function when a call instruction executes. If there are no break 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** with in its given status panel.