

## Learnrisc RISC-V C Compiler - User Manual

### 1. Introduction

This manual describes the usage of a RISC-V C compiler that supports C and assembly procedures. The compiler translates C source code into RISC-V machine code and allows direct input of low-level assembly for performance optimization. It features a teach by running program approach. This document covers compiling, loading, debugging and execution of these programs.

This compiler is not intended as a development tool, but as a learning tool. Its output runs on a simulator and has special control macros for setting its environment. It can setup interrupts, send messages, call utilities and read external commands for example. It is intended for smaller programs and requires no linking capability.

The code generated is 100% RISC-v and its load is comparable with other third party emulators.

### 2. Compiler Workflow

The RISC-V C compiler processes source code through the following stages:

1. **\*\*Preprocessing\*\*** – Handles macros, includes, and conditional compilation.
2. **\*\*Compilation\*\*** – Translates source to assembly code.
3. **\*\*Listing\*\*** – Combines source code with machine instructions for reference
4. **\*\*Load Generation\*\*** – Converts source to an loadable instruction list for simulation or execution on RISC-V hardware.

### 3. Writing C Programs with Assembly

When a c procedure is called the values are passed in register a0,a1,etc. Upon return the return value is in a0. This is also true for assembly procedures. The assembly procedures contain basic assembly code and assembly macros.

When using assembly, the compiler supports the full RV32I instruction set, including ADD, SUB, LW, SW, BEQ, BNE, JAL, and others. Additional macros are provided for convenience, such as loading immediate values or defining constants.

### RISC-5 Assembly Macros

We try to limit macros because the intent is to use the base instructions whenever possible. The following are supported:

#### 1. Load and Move Helpers

Macro (Pseudo-instruction)	Expands to	Purpose
li rd, imm	lui + addi (if big imm)	Load an immediate value into a register
mv rd, rs	addi rd, rs, 0	Copy one register to another
la rd, label	auipc + addi	Load address of a label into a register
nop	addi x0, x0, 0	Do nothing (no operation)

## 2. Branch Shortcuts

Macro	Expands to	Purpose
beqz rs, label	beq rs, x0, label	Branch if register is zero
bnez rs, label	bne rs, x0, label	Branch if register is not zero
bgt rs, rt, label	blt rt, rs, label	Branch if greater than
ble rs, rt, label	bge rt, rs, label	Branch if less or equal

## 3. Call and Return Helpers

Macro	Expands to	Purpose
procedure	auipc ra, ... + jalr	Call a procedure
ret	jalr x0, ra, 0	Return from procedure

## 4. Memory Access Shortcuts

Macro	Expands to	Purpose
lw rd, label	lui + lw (if label far)	Load word from label
sw rs, label	lui + sw	Store word to label

## 5. IO Access

Macro	Expands to	Purpose
cout	addi x0, x0, x31	Output
cin	Addi x0, x0, x30	Input

...

## 4. Load Files

The compiler produces load files for execution. Our load file uses a process called “machine code injection” . The load file consists of actual ASCII instructions and numbers. This is used by our simulator and bypasses the process of converting binary to instructions and instructions to binary.

A memory loader can convert our load file to binary for hardware load. At present this is not offered.

### Load File Example:

```
jal t0,148
sw t0,-4(sp)
addi t1,zero,33
sw t1,0(sp)
addi sp,sp,4

addi t1,zero,33
sw t1,0(sp)

addi sp,sp,4
addi zero,zero,0
addi t1,zero,6
...
```

## 5. IO Files

The IO files defines inputs and outputs for the code. It allow text output and variable values to be displayed. It is generated by embedding a pointer to an output command in a nop instruction. In source this is generated by **cin** and **cout** macros.

## Example IO File Content:

```
00636F7574203C3C20225072
6F6772616D3A204C65644F6E
20656E646C00636F7574203C
3C202022494F3A4C45445322
706C657465223B00
```

## 6. Compiler and Assembler Directives

Directives include:

- `#define` – Defines values to Names
- `#set_address` Defines start of global variables symbols`
- `#end_address` Defines start of global variables symbols`
- `'cout'` Defines request for text output or system access
- `'cin'` Inputs incoming messages to buffer area

## 7. Programming Suggestions

### A. OS Task Switching

1. When you switch from **Task A** to **Task B**, you're not just making a function call — you're replacing the *entire execution context*.

That means **all registers that Task A might use later must be saved**, regardless of whether they're “caller-saved” or “callee-saved” in the ABI.

So in a context switch, you typically save:

- All general-purpose registers: x1–x31 (except x0 which is always zero)
- The program counter (PC)

## B. Interrupt Service Routine

As an option we allow a user define periodic interrupts.

**Save registers** you will use — usually push onto the stack or save in a dedicated interrupt context area

1. Handle the interrupt,
  2. **Restore saved registers**
  3. Return from interrupt
- 

## C. Function calls

Saving and restoring registers is user and application defined.

In RISC-V, **best practice for function calls** revolves around following the **RISC-V calling convention (ABI)**, which clearly defines:

- Which registers a function **must save** (callee-saved)
  - Which registers a function **can freely overwrite** (caller-saved)
  - How arguments and return values are passed
-

### Summary of RISC-V Calling Convention:

Register(s)	Purpose	Caller-saved or Callee-saved?	Notes
x1 (ra)	Return address	Callee-saved	Must be preserved if function calls others
x2 (sp)	Stack pointer	Maintained by caller	Stack pointer management
x3 (gp)	Global pointer	Callee-saved	Usually fixed per program
x4 (tp)	Thread pointer	Callee-saved	Thread-local storage pointer
x5-x7 (t0-t2)	Temporaries	Caller-saved	Not preserved across calls
x8 (s0/fp)	Frame pointer or saved register	Callee-saved	Must be preserved
x9 (s1)	Saved register	Callee-saved	Must be preserved
x10-x17 (a0-a7)	Function arguments and return values	Caller-saved	Up to 8 arguments/return values
x18-x27 (s2-s11)	Saved registers	Callee-saved	Must be preserved
x28-x31 (t3-t6)	Temporaries	Caller-saved	Not preserved across calls