Simplified RISC Machine Assembler Manual
($Revision: 1.29 $)

Gary T. Leavens
Leavens@ucf.edu

November 13, 2023

Abstract

This document defines the assembly language for the Simplified RISC Machine VM, which is used in the Systems Software class (COP 3402) at UCF. It also defines the interface of the assembler and disassembler.

1 Overview

The assembler for the Simplified RISC Machine (SRM) is simple and has no macro facilities. However, it does let one assemble the instructions that make up a program, resolving symbolic names for jump targets, and it can define the starting address of a program and the program's (static) data section. The opcodes generally follow those for the MIPS processor's ISA [1] albeit in simplified form.

The assembler assumes that the SRM has 32-bit (4-byte) words and is byte-addressable.

1.1 Inputs and Outputs

1.1.1 Assembler

The assembler is passed a single file name as its only command line argument; this file should be the name of a (readable) assembler program; its output (sent to standard output) is a binary object file.

For example, if the program is contained in the file `myProg.srm`, assuming that the assembler is named `asm`, (and both these files are in the current directory), then the assembler can be invoked as follows in the Unix shell to produce a binary object file on standard output.

```
./asm myProg.srm
```

Thus, to put the assembled version of `myProg.srm` into the file `myProg.bof`, one would use a Unix command that redirects the output of the assembler into `myProg.bof`, as follows.

```
./asm myProg.srm > myProg.bof
```

1.1.2 Disassembler

The disassembler is the opposite of the assembler. It is passed a single file name, but that file names a (readable) binary object file, and it produces, on standard output, an assembly language source program.

For example, if the binary object file is found in `prog.bof`, assuming that the disassembler is named `disasm`, (and both these files are in the current directory), then the disassembler can be invoked as follows in the Unix shell to produce (on standard output) an assembly language program that would compile to `prog.bof`.

```
./disasm prog.bof
```

The assembly language program can be redirected into the file `prog.srm` as follows.

```
./disasm prog.bof > prog.srm
```

1.2 Error Outputs

All error messages (e.g., for file permission errors or syntax errors) are sent to standard error output (`stderr`).

1.3 Exit Code

When the either program halts normally, it exits with a zero error code (which indicates success on Unix). However, when `asm` or `disasm` encounters an error, it halts and exits with a non-zero exit code (which indicates failure on Unix).

2 Assembly Language Syntax

2.1 Lexical Grammar

Tokens in the assembler are described by the (regular) grammar of Figure 1. Note that line endings are significant in the context-free grammar of the assembler, as each instruction must be specified on a single line. Lines may be ended either by a newline character (⟨newline⟩ in Figure 1) or by a combination of a carriage-return (⟨cr⟩) followed by a newline. Comments (⟨comment⟩) start with a ⟨comment-start⟩ (i.e., `#`) and continue to the end of a line. White space is needed to separate tokens, but is otherwise ignored.

2.2 Context-Free Grammar

The syntax of the SRM's assembly language is defined by the (context-free) grammar in Figure 2.

3 Initial Values

The initial value of the program counter (*PC*) is set to the address of the program's entry point (i.e., the value of ⟨entry-point⟩), which is declared at the beginning of the ⟨text-section⟩.

The start of the global data in memory is at the address given by the data section's static data start address (i.e., the value of ⟨static-start-addr⟩), declared at the beginning of the ⟨data-section⟩; this value is used as the initial value of the $gp register. The data declared in the data section all have offsets from this address that are computed in declaration order, with `WORD` sized data taking 4 bytes. (`WORD` is the only data size allowed at present.)

The "bottom" of the runtime stack is given in a declaration in the stack section (⟨stack-section⟩); it is the value of ⟨stack-bottom-addr⟩ that follows the `.stack` keyword. This must be divisible by 4 and strictly greater than the static data start address; it is also the initial value put in the $fp and $sp registers at the start of a program's execution.

4 Constraints on Assembly Code

There are some constraints on programs that the assembler checks; the assembler considers violations of these constraints to be an error.

⟨section-mark⟩ ::= `.text` | `.data` | `.end`
⟨reserved-opcode⟩ ::= `ADD` | `SUB` | `AND` | `BOR` | `NOR` | `XOR` | `MUL` | `DIV`
| `SLL` | `SRL` | `MFHI` | `MFLO` | `JR` | `ADDI` | `ANDI` | `BORI` | `XORI` | `BEQ`
| `BGEZ` | `BGTZ` | `BLTZ` | `LBU` | `LW` | `SB` | `SW` | `JMP` | `JAL`
⟨reserved-data-size⟩ ::= `WORD`
⟨ident⟩ ::= ⟨letter⟩ {⟨letter-or-digit⟩} "but not a ⟨reserved-opcode⟩ or ⟨reserved-data-size⟩"
⟨letter⟩ ::= `_` | `a` | `b` | `...` | `y` | `z` | `A` | `B` | `...` | `Y` | `Z`
⟨letter-or-digit⟩ ::= ⟨letter⟩ | ⟨dec-digit⟩

⟨unsigned-number⟩ ::= ⟨dec-digit⟩ {⟨dec-digit⟩}
| `0x` ⟨hex-digit⟩ {⟨hex-digit⟩}
⟨dec-digit⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨hex-digit⟩ ::= ⟨dec-digit⟩ | `a` | `A` | `b` | `B` | `c` | `C` | `d` | `D` | `e` | `E` | `f` | `F`

⟨reg⟩ ::= ⟨dollar-sign⟩ ⟨unsigned-number⟩ | `$at` | `$v0` | `$v1`
| `$a0` | `$a1` | `$a2` | `$a3` | `$t0` | `$t1` | `$t2` | `$t3` | `$t4` | `$t5` | `$t6` | `$t7`
| `$s0` | `$s1` | `$s2` | `$s3` | `$s4` | `$s5` | `$s6` | `$s7` | `$t8` | `$t9`
| `$gp` | `$sp` | `$fp` | `$ra`
⟨dollar-sign⟩ ::= `$`

⟨eol⟩ ::= ⟨newline⟩ | ⟨cr⟩ ⟨newline⟩
⟨newline⟩ ::= "A newline character (ASCII 10)"
⟨cr⟩ ::= "A carriage return character (ASCII 13)"

⟨ignored⟩ ::= ⟨blank⟩ | ⟨tab⟩ | ⟨vt⟩ | ⟨formfeed⟩ | ⟨comment⟩
⟨blank⟩ ::= "A space character (ASCII 32)"
⟨tab⟩ ::= "A horizontal tab character (ASCII 9)"
⟨vt⟩ ::= "A vertical tab character (ASCII 11)"
⟨formfeed⟩ ::= "A formfeed character (ASCII 12)"
⟨comment⟩ ::= ⟨comment-start⟩ {⟨non-nl⟩}
⟨comment-start⟩ ::= `#`
⟨non-nl⟩ ::= "Any character except a newline"

Figure 1: Lexical grammar of the SRM assembler. This grammar uses a `terminal font` for terminal symbols. Note that an underbar (_) and all ASCII letters (a-z and A-Z) are included in the production for ⟨letter⟩. Curly brackets, such as $\{x\}$, mean an arbitrary number of (i.e., 0 or more) repetitions of x. Note that curly braces are not terminal symbols in this grammar. Some character classes are described in English, these are described in a Roman font between double quotation marks (" and "). Note that all characters matched by the nonterminal ⟨ignored⟩ are ignored by the lexer. However, the characters that are part of an ⟨eol⟩ token (i.e., carriage returns and newlines) are not ignored immediately following a semicolon, ⟨reserved-opcode⟩ or ⟨reserved-data-size⟩, although they are ignored in all other contexts.

⟨program⟩ ::= ⟨text-section⟩ ⟨data-section⟩ ⟨stack-section⟩ `.end`
⟨text-section⟩ ::= `.text` ⟨entry-point⟩ ⟨asm-instr⟩ {⟨asm-instr⟩}
⟨entry-point⟩ ::= ⟨addr⟩
⟨addr⟩ ::= ⟨label⟩ | ⟨unsigned-number⟩
⟨label⟩ ::= ⟨ident⟩
⟨asm-instr⟩ ::= ⟨label-opt⟩ ⟨instr⟩ ⟨eol⟩
⟨label-opt⟩ ::= ⟨label⟩ `:` | ⟨empty⟩
⟨empty⟩ ::=
⟨instr⟩ ::= ⟨three-reg-instr⟩ | ⟨two-reg-instr⟩ | ⟨shift-instr⟩ | ⟨one-reg-instr⟩
| ⟨immed-arith-instr⟩ | ⟨immed-bool-instr⟩ | ⟨branch-test-instr⟩ | ⟨load-store-instr⟩
| ⟨jump-instr⟩
⟨three-reg-instr⟩ ::= ⟨three-reg-op⟩ ⟨reg⟩ `,` ⟨reg⟩ `,` ⟨reg⟩
⟨three-reg-op⟩ ::= `ADD` | `SUB` | `AND` | `BOR` | `NOR` | `XOR`
⟨two-reg-instr⟩ ::= ⟨two-reg-op⟩ ⟨reg⟩ `,` ⟨reg⟩
⟨two-reg-op⟩ ::= `MUL` | `DIV`
⟨shift-instr⟩ ::= ⟨shift-op⟩ ⟨reg⟩ `,` ⟨reg⟩ `,` ⟨shift⟩
⟨shift-op⟩ ::= `SLL` | `SRL`
⟨shift⟩ ::= ⟨unsigned-number⟩
⟨one-reg-instr⟩ ::= ⟨one-reg-op⟩ ⟨reg⟩
⟨one-reg-op⟩ ::= `MFHI` | `MFLO` | `JR`
⟨immed-arith-instr⟩ ::= ⟨immed-arith-op⟩ ⟨reg⟩ `,` ⟨reg⟩ `,` ⟨immed⟩
⟨immed-arith-op⟩ ::= `ADDI`
⟨immed⟩ ::= ⟨number⟩
⟨number⟩ ::= ⟨sign⟩ ⟨unsigned-number⟩
⟨sign⟩ ::= `+` | `-` | ⟨empty⟩
⟨immed-bool-instr⟩ ::= ⟨immed-bool-op⟩ ⟨reg⟩ `,` ⟨reg⟩ `,` ⟨uimmed⟩
⟨immed-bool-op⟩ ::= `ANDI` | `BORI` | `XORI`
⟨uimmed⟩ ::= ⟨unsigned-number⟩
⟨branch-test-instr⟩ ::= ⟨branch-test-2-op⟩ ⟨reg⟩ `,` ⟨reg⟩ `,` ⟨offset⟩
| ⟨branch-test-1-op⟩ ⟨reg⟩ `,` ⟨offset⟩
⟨branch-test-2-op⟩ ::= `BEQ` | `BNE`
⟨branch-test-1-op⟩ ::= `BGEZ` | `BGTZ` | `BLEZ` | `BLTZ`
⟨offset⟩ ::= ⟨number⟩
⟨load-store-instr⟩ ::= ⟨load-store-op⟩ ⟨reg⟩ `,` ⟨reg⟩ `,` ⟨offset⟩
⟨load-store-op⟩ ::= `LBU` | `LW` | `SB` | `SW`
⟨jump-instr⟩ ::= ⟨jump-op⟩ ⟨addr⟩
⟨jump-op⟩ ::= `JMP` | `JAL`
⟨syscall-instr⟩ ::= ⟨syscall-op⟩
⟨syscall-op⟩ ::= `EXIT` | `PSTR` | `PCH` | `RCH` | `RSTR` | `STRA` | `NOTR`
⟨data-section⟩ ::= `.data` ⟨static-start-addr⟩ {⟨static-decl⟩}
⟨static-start-addr⟩ ::= ⟨unsigned-number⟩
⟨static-decl⟩ ::= ⟨data-size⟩ ⟨ident⟩ ⟨initializer-opt⟩ ⟨eol⟩
⟨data-size⟩ ::= `WORD`
⟨initializer-opt⟩ ::= `=` ⟨number⟩ | ⟨empty⟩
⟨stack-section⟩ ::= `.stack` ⟨stack-bottom-addr⟩
⟨stack-bottom-addr⟩ ::= ⟨unsigned-number⟩

Figure 2: The (context free) grammar of the SRM assembler, which uses a `typewriter font` for terminal symbols.

The program's entry point (⟨entry-point⟩ value), static data start address (⟨static-start-addr⟩ value), and stack bottom address (⟨stack-bottom-addr⟩ value) must all be divisible by 4. Furthermore, the entry point must be strictly less than the static data start address and the static data strart address must be strictly less than the stack bottom address.

References

[1] Gerry Kane and Joe Heinrich. *MIPS RISC architectures*. Prentice-Hall, Inc., 1992.