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

Gary T. Leavens
Leavens@ucf.edu

November 10, 2024

Abstract

This document defines the assembly language for the Simplified Stack 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 Stack Machine (SSM) 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. This assembly language adopts some conventions from the MIPS processor's assembly language [1].

The assembler assumes that the SSM has 32-bit (4-byte) words and is word-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.ssm`, assuming that the assembler's executable 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.ssm
```

Thus, to put the assembled version of `myProg.ssm` 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.ssm > myProg.bof
```

In addition to the binary object file name, the assembler can be given one command line argument. These options are used for debugging the assembler's code: `-l` to print the lexical tokens read, `-u` to unparse the input's abstract syntax tree (after constructing it), and `-s` to print the symbol table for the file. A binary object file name must always be given after using an option. Only one option can be used for any given invocation of the assembler, and when the `-l` or `-u` options are used, no binary object file is produced.

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.ssm` as follows.

```
./disasm prog.bof > prog.ssm
```

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 Codes

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⟩ character (i.e., `#`) and continue to the end of a line. White space is needed to separate tokens, but is otherwise ignored.

The lexical grammar (in Figure 1) 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 the 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 pound-sign (which starts a comment) ⟨reserved-opcode⟩ or ⟨reserved-data-size⟩, although they are ignored in all other contexts.

2.2 Context-Free Grammar

The syntax of the SSM's assembly language is defined by the (context-free) grammar in Figure 2 and Figure 3. The grammar uses a `typewriter font` for terminal symbols.

⟨section-mark⟩ ::= .text | .data | .end
⟨reserved-opcode⟩ ::= NOP | ADD | SUB | CPW | CPR | AND | BOR
    | NOR | XOR | LWR | SWR | SCA | LWI | NEG | LIT | ARI | SRI
    | MUL |DIV | CFHI | CFLO | SLL | SRL | JMP | CSI | JREL
    | ADDI | ANDI | BORI | NORI | XORI | BEQ | BGEZ | BGTZ | BLEZ
    | BLTZ | BNE | JMPA | CALL | RTN | EXIT | PSTR | PINT | PCH
    | RCH | STRA | NOTR
⟨reserved-data-size⟩ ::= WORD | CHAR | STRING
⟨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
⟨oct-digit⟩ ::= 0| 1 | 2 | 3 | 4 | 5 | 6 | 7

⟨reg⟩ ::= $ ⟨oct-digit⟩ | $gp | $sp | $fp | $r3 | $r4 | $r5 | $r6 | $ra

⟨char-literal⟩ ::= ' ⟨char-elem⟩ '
⟨char-elem⟩ ::= ⟨char⟩ “but not an unescaped ' character”
⟨char⟩ ::= ⟨c-escape-seq⟩ | ⟨printable-ASCII-char⟩
⟨c-escape-seq⟩ ::= ⟨backslash⟩ ⟨escape-code⟩
⟨backslash⟩ ::= “A backslash character (ASCII 92)”
⟨escape-code⟩ ::= n | r | f | t | v | ⟨backslash⟩ | ' | " | 0 | a | b
    | x ⟨hex-digit⟩ ⟨hex-digit⟩ | 0 ⟨oct-digit⟩ ⟨oct-digit⟩ ⟨oct-digit⟩ | s | d | e

⟨string-literal⟩ ::= " {⟨string-elem⟩} "
⟨string-elem⟩ ::= ⟨char⟩ “but not an unescaped " character”

⟨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 SSM assembler.

```
⟨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⟩ ::= ⟨no-arg-instr⟩ | ⟨two-reg-comp-instr⟩ | ⟨two-reg-no-offsets-instr⟩ | ⟨no-target-offset-instr⟩
      | ⟨no-source-offset-instr⟩ | ⟨one-reg-offset-arg-instr⟩ | ⟨one-reg-arg-instr⟩ | ⟨one-reg-offset-instr⟩
      | ⟨shift-instr⟩ | ⟨arg-only-instr⟩ | ⟨immed-arith-instr⟩ | ⟨immed-bool-instr⟩
      | ⟨branch-test-instr⟩ | ⟨jump-instr⟩ | ⟨syscall-instr⟩
⟨no-arg-instr⟩ ::= NOP | JMP | RTN
⟨two-reg-comp-instr⟩ ::= ⟨two-reg-comp-op⟩ ⟨reg⟩ , ⟨offset⟩ , ⟨reg⟩ , ⟨offset⟩
⟨two-reg-comp-op⟩ ::= ADD | SUB | CPW | AND | BOR | NOR | XOR | SCA | LWI | NEG
⟨offset⟩ ::= ⟨number⟩
⟨number⟩ ::= ⟨sign⟩ ⟨unsigned-number⟩
⟨sign⟩ ::= + | - | ⟨empty⟩
⟨two-reg-no-offsets-instr⟩ ::= ⟨two-reg-no-offsets-op⟩ ⟨reg⟩, ⟨reg⟩
⟨two-reg-no-offsets-op⟩ ::= CPR
⟨no-target-offset-instr⟩ ::= ⟨no-target-offset-op⟩ ⟨reg⟩ , ⟨reg⟩ , ⟨offset⟩
⟨no-target-offset-op⟩ ::= LWR
⟨no-source-offset-instr⟩ ::= ⟨no-source-offset-op⟩ ⟨reg⟩ , ⟨offset⟩ , ⟨reg⟩
⟨no-source-offset-op⟩ ::= SWR
⟨one-reg-offset-arg-instr⟩ ::= ⟨one-reg-offset-arg-op⟩ ⟨reg⟩ , ⟨offset⟩ , ⟨arg⟩
⟨one-reg-offset-arg-op⟩ ::= LIT
⟨arg⟩ :  ⟨number⟩
⟨one-reg-arg-instr⟩ ::= ⟨one-reg-arg-op⟩ ⟨reg⟩ , ⟨arg⟩
⟨one-reg-arg-op⟩ ::= ARI | SRI
⟨arg⟩ :  ⟨number⟩
⟨one-reg-offset-instr⟩ ::= ⟨one-reg-offset-op⟩ ⟨reg⟩ , ⟨offset⟩
⟨one-reg-offset-op⟩ ::= MUL | DIV | CFHI | CFLO | JMP | CSI
⟨shift-instr⟩ ::= ⟨shift-op⟩ ⟨reg⟩ , ⟨offset⟩ , ⟨shift⟩
⟨shift-op⟩ ::= SLL | SRL
⟨shift⟩ ::= ⟨unsigned-number⟩
⟨arg-only-instr⟩ ::= ⟨arg-only-op⟩ ⟨arg⟩
⟨arg-only-op⟩ ::= JREL
⟨immed-arith-instr⟩ ::= ⟨immed-arith-op⟩ ⟨reg⟩ , ⟨offset⟩ , ⟨immed⟩
⟨immed-arith-op⟩ ::= ADDI
⟨immed⟩ ::= ⟨number⟩
⟨immed-bool-instr⟩ ::= ⟨immed-bool-op⟩ ⟨reg⟩ , ⟨offset⟩ , ⟨uimmed⟩
⟨immed-bool-op⟩ ::= ANDI | BORI | NORI | XORI
⟨uimmed⟩ ::= ⟨unsigned-number⟩
```

Figure 2: The (context free) grammar of the SSM assembler, part 1 of 2.

⟨branch-test-instr⟩ ::= ⟨branch-test-op⟩ ⟨reg⟩ , ⟨offset⟩ , ⟨immed⟩

⟨branch-test-op⟩ ::= BEQ | BGEZ | BGTZ | BLEZ | BLTZ | BNE

⟨jump-instr⟩ ::= ⟨jump-op⟩ ⟨addr⟩

⟨jump-op⟩ ::= JMPA | CALL

⟨syscall-instr⟩ ::= ⟨offset-only-syscall⟩ | ⟨reg-offset-syscall⟩ | ⟨no-arg-syscall⟩

⟨offset-only-syscall⟩ ::= EXIT ⟨offset⟩

⟨reg-offset-syscall⟩ ::= ⟨reg-offset-syscall-op⟩ ⟨reg⟩ , ⟨offset⟩

⟨reg-offset-syscall-op⟩ ::= PSTR | PINT | PCH | RCH

⟨no-arg-syscall⟩ ::= 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 | CHAR | STRING [⟨unsigned-number⟩]

⟨initializer-opt⟩ ::= = ⟨number⟩ | ⟨char-literal⟩ | ⟨string-literal⟩ | ⟨empty⟩

⟨stack-section⟩ ::= .stack ⟨stack-bottom-addr⟩

⟨stack-bottom-addr⟩ ::= ⟨unsigned-number⟩

Figure 3: The (context free) grammar of the SSM assembler, part 2 of 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 word 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 and CHAR sized data both declaring a word of storage (i.e., 4 bytes) and STRING data taking the number of *words* declared for it.

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 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.

The program's entry point must be strictly less than the static data start address and the static data start address must be strictly less than the stack bottom address.

Furthermore, immediate operands and offsets must fit in the number of bits allowed by the instruction's format.

References

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