ASSEMBLER–FORMAT(5) Version 1.2.0 | File Formats Manual

February 2024

NAME

assembler–format — format of input assembly files

SYNOPSIS

```
; is a single line comment, you can write everything here
.data
; You can use labels here as well as in memory locations
LABEL:
    .byte   1, 2, 3, LABEL      ; 8 bit numbers
    .half   1, 2, 3, LABEL      ; 16 bit numbers
    .word   1, 2, 3, LABEL      ; 32 bit numbers
    .dword  1, 2, 3, LABEL      ; 64 bit numbers
    .eqv    CONST, 20           ; CONST is set to 20

.text
; You can also use labels here in the .text section
; Write instructions and macros here
START:
    nop
    addi    zero, zero, 0
    addi    t1, zero, 2
    mul     t0, sp, t1
LOD:
    j       LOD                 ; You can use labels in macros
```

DESCRIPTION

The assembler(1) command converts assembly files to output in binary or mif format. The syntax is based on MIPS assembly syntax and is modified to lessen programmer burden and increase compatibility to already established tools and conventions.

SYNTAX

In general, every line contains nothing (blanks), a section, a label or a label and operation. Comments can be added to the end of any line or inserted into a line that otherwise would contain nothing (blanks). The assembler is leniant with the placement of these components to eachother. A operation is an instruction, macro or directive.

A single line:

```
LABEL: <OPERATION> ; COMMENT
```

Multiple lines that fall into a single line:

```
LABEL:
                ; COMMENT
; COMMENT

                          <OPERATION>
     ; COMMENT
```

Similiar to above, but in a different format:

```
LABEL: ; COMMENT
    <OPERATION>                     ; COMMENT
    ; COMMENT
```

All three examples are equal, if the operations are equal.

See OPERATIONS and LABEL DEFINITIONS for details.

TEXT AND DATA SECTIONS

Sections are ordered and used to outline section of particular kind. Sections of any kind can only be defined once. Currently only data and text sections are supported with no plans for supporting other sections.

The use of sections are optional and a assembly file containing no explicit sections, implicitly defines a text section spanning the whole file.

Assembly files must contain a text section and optionally can contain a non–empty data section.

Data sections must come before text sections.

Valid assembly file that includes a data and text section:

```
.data                           ; Outlines a data section spanning to the
    <DIRECTIVE>                 ; next text section
    <DIRECTIVE>

.text                           ; Outlines a text section spanning to the
    <INSTRUCTION>               ; end of the file
    <INSTRUCTION>
```

Valid assembly file that only includes a text section:

```
.text
    <INSTRUCTION>
    <INSTRUCTION>
```

Valid assembly file that implicitly defines a text section:

```
START:
    <INSTRUCTION>
    <INSTRUCTION>
```

Invalid assembly file since the data section is empty (this may be changed in the future):

```
.data
.text
    <INSTRUCTION>
```

Invalid assembly file since it only contains a data section:

```
.data
    <DIRECTIVE>
    <DIRECTIVE>
```

See DIRECTIVES, MACROS and INSTRUCTIONS for details.

LABEL DEFINITIONS

Label definitions are label references that are suffixed with a colon. Labels must be at least one character long and can only contain alphanumeric characters. The first character must be alphabetic.

Correct label definitions include the following:

```
Label0:
A:
LabelVeryNice:
Example215:
```

Incorrect labels definitions include:

```
:
__TEST:
0Lol:
Lol
```

Do note that currently there is no validation of label types and if the label value fits into the instruction. Errors for the former and warnings of the latter are planned.

OPERATIONS

Operations is a definition used to describe instructions, macros and directives. The arguments can be immediates, registers and/or labels. The arguments are separated by commas (,) or commas with a space (,). The operation name and arguments are separated by one or more spaces.

See INSTRUCTIONS, DIRECTIVES and MACROS for details.

REGISTERS

Some instructions and macros require registers to perform actions. There are 31 registers that can be used. Registers can be referenced by either the register number prefixed with an **x**, meaning **x0** to **x31**, or their ABI name.

x0**,** zero
: Immutable register that is always zero.

x1**,** ra
: Return address, callee−saved.

x2**,** sp
: Stack pointer, callee−saved.

x3**,** gp
: General purpose register, caller−saved. Global pointer according to RISC−V spec, but not used as such here.

x4**,** tp
: General purpose register, caller−saved. Thread pointer according to RISC−V spec, but

not used as such here.

x[5|6|7], t[0|1|2]
Temporary register, caller–saved.

x8, s0, fp
Saved register or frame pointer, callee–saved.

x9, s1
Saved register, callee–saved.

x[10|11], a[0|1]
Register for return values and function arguments, caller–saved.

x[12|13|14|15|16|17], a[2|3|4|5|6|7]
Register for function arguments, caller–saved.

x[18|19|20|21|22|23|24|25|26|27], s[2|3|4|5|6|7|8|9|10|11]
Saved register, callee–saved.

x[28|29|30|31], t[3|4|5|6]
Temporary register, caller–saved.

IMMEDIATES

Some instructions, macros and directives require immediates to perform actions. Immediates are either in decimal, binary or hexadecimal format. By default the decimal format is assumed. To interpret immediates as binary or hexadecimal, the prefix **0b** and **0x** must be used respectively. Optionally binary or hexadecimal can be interpreted as signed numbers by suffixing the immediate with **s** or **S**.

The following immediates are valid:

```
0b10          ; 2
0b10s         ; -2
0x14          ; 20
0x14s         ; 20
205
-12
```

The following immediates are invalid:

```
0.1           ; floating point not supported (yet)
b100
x1516
02x3
50-20         ; expressions are not supported
```

DIRECTIVES

Directives are used in data sections and always prefixed with a dot (.). Some common directives are supported and mainly the ones that can be used to store data in the data section.

For some directives the argument is a string, which is delimited by quotation marks. Otherwise general rules apply here as well.

The order of the directives in the assembly file dictate the order of the data in memory. Data starts at address 0 and grows upwards. The first directive is written to address 0.

Currently the following directives are supported:

.byte ***register*|*label*,[*register*|*label*]...**
The registers and labels are stored as 8 bit values in memory.

.half ***register*|*label*,[*register*|*label*]...**
The registers and labels are stored as 16 bit values in memory.

.word ***register*|*label*,[*register*|*label*]...**
The registers and labels are stored as 32 bit values in memory.

.dword ***register*|*label*,[*register*|*label*]...**
The registers and labels are stored as 64 bit values in memory.

.space ***decimal***
Reserve space for data. The *decimal* denotes the space reserved in bytes. It must be a decimal and cannot be negative.

.ascii "***string***"
The *string* is stored as consecutive 8 bit values. The *string* should only contain ASCII characters. All characters are translated to their ASCII code. The *string* is not null terminated.

.asciz "***string***"
Same as **.ascii** but the *string* is null terminated.

.string "***string***"
Alias for **.asciz**.

.eqv ***label, immediate***
The value of the *label* is *immediate*. A *label* emitted this way is a constant that is not written in memory and can be used like a immediate.

These are valid directives:

```
.byte   1, 3,2,LABEL
.half 20, 15, LABEL
.space 10
.eqv                          LABEL,30
```

These are invalid directives:

```
.non                  ; unknown directive
.byte                 ; no arguments
.half 30 15
.space 0b10           ; argument can only be decimal
.asciz "STRING        ; missing closing quotation mark
```

MACROS

Macros are pseudo–instructions that are not and cannot be translated to machine code as is. The syntax is similiar to INSTRUCTIONS. Some of the common macros are supported.

Macros are expanded and mapped to instructions that are machine translatable. Macros cannot be defined by programmers.

The first register for macros that have them is always the register that is written to.

Currently the following macros are supported:

srr ***register, register, immediate***
Shift right rotate. This is implemented as a subroutine thus saving registers is required. Saving registers is not done automatically!

slr ***register, register, immediate***
Shift left rotate. This is implemented as a subroutine thus saving registers is required. Saving registers is not done automatically!

li ***register, immediate|label***
Load immediate. *register* is set to the *immediate* or *label*.

la ***register, immediate|label***
Load address. *register* is set to either the *immediate* or the address of the *label*.

call ***immediate|label***
Jump to a far−away label and handle it as a subroutine. The return address is written to register **ra**. Returning is possible by using the macro **ret** or by the equivalent **jal** instruction.

tail ***immediate|label***
Jump to a far−away label. The return address is voided. Returning is not possible.

push ***register*, [*register*]...**
Save the content of these registers onto the stack. Requires initialization of the stack pointer register **sp**. Multiple registers can be specified to reduce the subtraction overhead. The registers are saved in the given order. The first register is saved at the bottom, the last register at the top of stack.

pop ***register*, [*register*]...**
Load the content of the stack into the registers. Requires initialization of the stack pointer register **sp**. Multiple registers can be specified to reduce the addition overhead. The content is loaded into the registers in the given order. The first register receives the content of the top of stack, the last register of the bottom.

rep ***decimal, instruction|macro***
Repeat the *instruction* or *macro decimal* times. The decimal must be positive and greater than 0. Repeats cannot be nested, meaning a repeat cannot contain a repeat.

mv ***register, register***
Copies the content of the latter register into the former register. This is mapped to either the instruction **addi** or **add**.

nop
No operation. It does not do anything. This is mapped to either the instruction **addi zero, zero, 0** or **add zero, zero, zero**.

ret
Used to return from a subroutine. This is mapped to the instruction **jalr zero, ra, 0**.

j ***immediate|label***
Jump to the *label* or *immediate*. This is mapped to the instruction **jal zero,** *offset*.

jal ***immediate|label***
Jump and link to the *label* or *immediate*. This is mapped to the instruction **jal ra,** *offset*.

jr ***register***
Jump to the address in the *register*. This is mapped to the instruction **jalr zero,** *register*, **0**.

jalr ***register***
: Jump and link to the address in the *register*. This is mapped to the instruction **jalr ra,** *register***, 0**.

See RISC−V Shortened Spec for details.

INSTRUCTIONS

An instruction is machine code in human−readable form. The syntax is similiar to MACROS. All instructions of the RV32I and RV32M extensions are supported.

The first register for instructions that have them is always the register that is written to. A limitation to that rule are branch instructions.

These instructions are used to perform arithmetical, logical and shift operations with registers:

add ***register, register, register***
: Addition of the two latter *register*s.

sub ***register, register, register***
: Subtraction. The minuend is the content of the second *register*, the subtrahend is the content of the last *register*.

xor ***register, register, register***
: Logical bitwise exclusive or of the second and third *register*.

or ***register, register, register***
: Logical bitwise or of the second and third *register*.

and ***register, register, register***
: Logical bitwise or of the second and third *register*.

sll ***register, register, register***
: Logical left shift of the second *register* by the third *register*.

srl ***register, register, register***
: Logical right shift of the second *register* by the third *register*.

sra ***register, register, register***
: Arithmetical right shift of the second *register* by the third *register*.

slt ***register, register, register***
: The first *register* is set to one (1), if the second *register* is less than the last *register*.

sltu ***register, register, register***
: The first *register* is set to one (1), if the second *register* is less than the last *register*. The content of the *register*s compared are interpreted as unsigned numbers.

xnor ***register, register, register***
: Logical bitwise negated exclusive or of the second and third *register*. Note that this is not defined in the RISC−V standard.

equal ***register, register, register***
: Compares the second and third *register*s and sets the first *register* to one (1), if they are equal. Note that this is not defined in the RISC−V standard.

mul ***register, register, register***
: Multiplication of the second and third *register*. The first *register* is set to the lower 32 bits of the result.

mulh ***register, register, register***
: High Multiplication of the second and third *register*. The first *register* is set to the higher 32 bits of the result. The content of both *register*s is interpreted as signed numbers.

mulhu ***register, register, register***
: High Multiplication of the second and third *register*. The first *register* is set to the higher 32 bits of the result. The content of both *register*s is interpreted as unsigned numbers.

mulhsu ***register, register, register***
: High Multiplication of the second and third *register*. The first *register* is set to the higher 32 bits of the result. The content of the second *register* is interpreted as a signed number, the content of the third *register* is interpreted as a unsigned number.

div ***register, register, register***
: Division of the second and third *register*s. The second *register* is the dividend and the third *register* is the divisor. The content of both *register*s are interpreted as signed numbers.

divu ***register, register, register***
: Division of the second and third *register*s. The second *register* is the dividend and the third *register* is the divisor. The content of both *register*s are interpreted as unsigned numbers.

rem ***register, register, register***
: Modulo operation of the second and third *register*s. The second *register* is the dividend and the third *register* is the divisor. The content of both *register*s are interpreted as signed numbers.

remu ***register, register, register***
: Modulo operation of the second and third *register*s. The second *register* is the dividend and the third *register* is the divisor. The content of both *register*s are interpreted as unsigned numbers.

These instructions are used to perform arithmetical, logical and shift operations with immediates:

Shift operations can only take 4 bit immediates.

Note that some instructions cannot use labels. This is WIP.

addi ***register, register, immediate|label***
: Addition of the second *register* and the *immediate* or *label*.

xori ***register, register, immediate***
: Logical bitwise exclusive or of the second *register* and the *immediate*.

ori ***register, register, immediate***
: Logical bitwise or of the second *register* and the *immediate*.

andi ***register, register, immediate***
: Logical bitwise or of the second *register* and the *immediate*.

slli ***register, register, immediate|label***
: Logical left shift of the second *register* by the *immediate* or *label*.

srli ***register, register, immediate|label***
: Logical right shift of the second *register* by the *immediate* or *label*.

srai ***register, register, immediate|label***
Arithmetical right shift of the second *register* by the *immediate* or *label*.

slti ***register, register, immediate***
The first *register* is set to one (1), if the second *register* is less than the *immediate*.

sltiu ***register, register, immediate***
The first *register* is set to one (1), if the second *register* is less than the *immediate*. The content of the *register*s compared are interpreted as unsigned numbers.

These instructions are used to manipulate memory content:

The target byte and half are always the LSBs of the target register.

lb ***register, register, immediate|label***
Loads a byte from memory at the address which is the sum of the second *register* and the *immediate* or *label*.

lh ***register, register, immediate|label***
Loads a half (16 bits) from memory at the address which is the sum of the second *register* and the *immediate* or *label*.

lw ***register, register, immediate|label***
Loads a word (32 bits) from memory at the address which is the sum of the second *register* and the *immediate* or *label*.

lbu ***register, register, immediate|label***
Loads a byte from memory at the address which is the sum of the second *register* and the *immediate* or *label*. The byte is zero extended to 32 bits.

lhu ***register, register, immediate|label***
Loads a half from memory at the address which is the sum of the second *register* and the *immediate* or *label*. The half is zero extended to 32 bits.

sb ***register, register, immediate|label***
Stores a byte into memory at the address which is the sum of the second *register* and the *immediate* or *label*.

sh ***register, register, immediate|label***
Stores a half into memory at the address which is the sum of the second *register* and the *immediate* or *label*.

sw ***register, register, immediate|label***
Stores a word into memory at the address which is the sum of the second *register* and the *immediate* or *label*.

These instructions are used to control the logic flow of the program:

PC−Relative addressing is used for all instructions but **jalr**. For **jalr** absolute addressing is used.

beq ***register, register, immediate|label***
Branch if the *register*s are equal.

bne ***register, register, immediate|label***
Branch if the *register*s are not equal.

blt ***register, register, immediate|label***
Branch if the content of the first *register* is less than the content of the last *register*. The content of the *register*s are interpreted as signed numbers.

bltu ***register, register, immediate|label***
: Branch if the content of the first *register* is less than the content of the last *register*. The content of the *register*s are interpreted as unsigned numbers.

bge ***register, register, immediate|label***
: Branch if the content of the first *register* is greater than or equal to the content of the last *register*. The content of the *register*s are interpreted as signed numbers.

bgeu ***register, register, immediate|label***
: Branch if the content of the first *register* is greater than or equal to the content of the last *register*. The content of the *register*s are interpreted as unsigned numbers.

jal ***register, immediate|label***
: Jump and link to the address which is the sum of the program counter and the *immediate* or *label*. The return address is written to the *register*.

jalr ***register, register, immediate|label***
: Jump and link to the address which is the sum of the second *register* and the *immediate* or *label*. The return address is written to the *register*.

These instructions cannot be categorized:

lui ***register, immediate|label***
: Load upper immediate. The upper 20 bits of the *register* is set to the *immediate* or *label*. The lower 12 bits are zero.

auipc ***register, immediate|label***
: Add upper immediate to program counter. The upper 20 bits of the *immediate* or *label* is added to the program counter and the result is written to the *register*.

See RISC−V Shortened Spec for details.

SEE ALSO

assembler(1), RISC−V Shortened Spec