Assembler Labels

Bookmark this page

The first thing which makes programming in assembly easier is labels. Labels mark places in the program code. They are marks for positions in the program. They are used instead of offsets or addresses. A label is a text that ends with a colon. Numeric labels are local labels that can be referenced with suffixes 'f' for forward and 'b' for backward. Further, a text label defines a symbol_name. The assembler translates a label into a corresponding address. Let us consider the following example:

```
    addi x1, x0, 1
    beq x1,x0, there
    addi x1, x1, 1
there:
    addi x1, x1, 1
```

The assembler translates the program and inserts the offset associated with the label:

```
0x0:        00100093        addi x1 x0 1
0x4:        00008463        beq x1 x0 8 <there>
0x8:        00108093        addi x1 x1 1

0000000c <there>:
0xc:        00108093        addi x1 x1 1
```

Addressing with Labels

Bookmark this page

If we look at the basic RV32I ISA, then we see that addressing the whole address space might require more than one instruction. The instructions **auipc** and **lui** are used for loading the upper immediates into a register, thus allowing addressing the upper part of an address. The lower part can be addressed by an **addi** instruction with the register. Therefore, we need some tools to split the address of a label into an upper and lower part.

The relocation functions **%hi(symbol)** and **%lo(symbol)** split the address of a label into its higher and lower part. The linker relocates the program and assigns addresses to symbols. E.g.,

```
        lui x1, %hi(there)          #absolute higher 20 bits
        addi x1, x1, %lo(there)     #absolute lower 12 bits
there:
```

is translated into

```
0x10078:    lui x1,0x10
0x1007c:    addi x1,x1,128 # 10080 <there>
```

The address of the label **there** is (after relocation) 0x10080. The functions **%hi** and **%lo** are used to set the correct address in the register x1.

Another example of relocation functions is for pc-relative addressing. The functions **%pcrel_hi(symbol)** and **%pcrel_lo(label)** work together with the **auipc** and **addi** instructions. But as the addressing is relative, they are used differently than global addressing. E.g., the assembler generates from the program

```
1:      auipc x1, %pcrel_hi(there)       #relative higher 20 bits
        addi x1, x1, %pcrel_lo(1b)       #relative lower 12 bits label 1 backwards
there:
```

the following code:

```
1:
0x10078:        auipc x1,0x0
0x1007c:        addi x1,x1,8 # 10080 <there>
```

Again, the address of **there** is 0x10080. The instruction **auipc** needs to add 0x0 to the pc (0x10078) for the upper part of the address of 'there'. The result of the instruction **auipc** yields 0x10078 in register x1. The instruction **addi** adds 8 to the register x1. The value 8 is the lower part of the difference to the label **there** computed from the label **1**.

However, manual work with the relocation functions is cumbersome and unnecessary, as pseudo instructions can be used.

Assembler Directives

Bookmark this page

Assembler directives are for the assembler. They provide the assembler with information on how the text following a directive should be treated, e.g., if the next part contains instructions or data. Directives begin with a dot. The following table shows a list of frequently used directives.

Directive	Arguments	Description
`.text`		change to .text section
`.data`		change to .data section
`.rodata`		change to .rodata section
`.bss`		change to .bss section
`.section`	.text, .data, .rodata, .bss	change to section given by arguments
`.equ`	name, value	define name for value
`.ascii`	"string"	begin string without null terminator
`.asciz`	"string"	begin string with null terminator
`.string`	"string"	same as .asciz
`.byte`	expression [,expression]*	8-bit comma separated words
`.half`	expression [,expression]*	16-bit comma separated words
`.word`	expression [,expression]*	32-bit comma separated words
`.dword`	expression [,expression]*	64-bit comma separated words
`.zero`	integer	zero bytes
`.align`	integer	align to the power of 2
`.globl`	symbol_name	make symbol_name apparent in symbol table

These directives are important when you write a program that is linked for running in an environment, e.g., on an embedded system or an operating system.

The directive **.text** provides the assembler the information that the following source code section is program/instruction code. The directive **.data** provides the information that a data section follows, which is initialized and modifiable. The directive **.rodata** is for an initialized read-only data section (constants). And the directive **.bss** is for an uninitialized modifiable data section. These sections can be defined by directive **.section** following with the kind of section.

A name for a constant value can be defined by **.equ**. Data values for ASCII strings are set by the directives **.ascii**, **.asciz**, and **.string**. The directives .asciz and .string add a null byte after the text. The text string follows a directive.

The directives **.byte**, **.half**, **.word**, and **.dword** are for setting one or more values that follow the directives. The directives result is different in the size of the value(s) following, e.g., **.byte** does only accept 8-bit values.

The directive **.zero** specifies an array of bytes that are all zero. The following integer number after **.zero** specifies the number of zero values following.

The directive **.align** aligns the memory values following the number given by 2 to the power of the parameter. The alignment is achieved by inserting zero bytes.

The directive **.globl symbol_name** makes a **symbol_name**, which is usually defined by a label visible for the linker. The symbol **_start** is required to provide the program's entry point for the linker, though the linker script defines which symbol is the entry point. We do not cover linker scripts and use the default one in the considered Linux Gnu toolchain environment. Nevertheless, linker scripts are important when working in different environments or with embedded systems, as linker scripts specify the memory layout the program uses.

The assembler also knows the dot (.) when processing a program. The dot stands for the current address.

Assembler Directives: Example

Bookmark this page

Now let us look at an example. The hashtag allows comments in the source code:

```
# define exit as 93
.equ exit, 93
# program code
.section .text
# export _start for linker
.globl  _start
_start:
        li      a7, exit
        ecall
# data: init one word (16-bit value) with 1 and read/write
.section .data
counter:
.word 1
# rodata: constant text string
.section .rodata
text_begin:
.asciz  "Text"
text_end:
# current address minus address of text_begin = length of text
.byte .-text_begin
# non initialized block with same size as the text
.section .bss
# start next part by address aligned to multiple of 2^2 = 4
.align 2
copy_begin:
.zero text_end-text_begin
```

We can save this as example.s, build the program in the executable and linkable format (ELF), and analyze the result of the assembly process by the tool objdump:

```
riscv64-linux-gnu-as -o example.o example.s
riscv64-linux-gnu-ld -o example example.o
riscv64-linux-gnu-objdump -f -d -Mno-aliases,numeric example

example:     file format elf64-littleriscv
architecture: riscv:rv64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00000000000100e8

Disassembly of section .text:

00000000000100e8 <_start>:
   100e8:       05d00893                addi    x17,x0,93
   100ec:       00000073                ecall
```

The command **objdump -f -d** shows us the disassembled file (-d) with file header information (-f). The options after the **-Mno-aliases,numeric** specify that objdump prints basic instructions and no pseudo instructions and the registers numbers, not their ABI names (will be covered later). We see only the program code of the section **.text**. With the command **objdump -t**, the symbols of the executable file are retrieved:

```
riscv64-linux-gnu-objdump -t example

example:     file format elf64-littleriscv

SYMBOL TABLE:
00000000000100e8 l    d  .text  0000000000000000 .text
00000000000100f0 l    d  .rodata        0000000000000000 .rodata
00000000000110f6 l    d  .data  0000000000000000 .data
00000000000110fc l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .riscv.attributes      0000000000000000 .riscv.attributes
0000000000000000 l    df *ABS*  0000000000000000 example.o
000000000000005d l       *ABS*  0000000000000000 exit
00000000000100f6 l       .data  0000000000000000 counter
00000000000100f0 l       .rodata        0000000000000000 text_begin
00000000000100f5 l       .rodata        0000000000000000 text_end
00000000000110fc l       .bss   0000000000000000 copy_begin
00000000000118f6 g       *ABS*  0000000000000000 __global_pointer$
00000000000110fa g       .data  0000000000000000 __SDATA_BEGIN__
00000000000100e8 g       .text  0000000000000000 _start
0000000000011108 g       .bss   0000000000000000 __BSS_END__
00000000000110fa g       .bss   0000000000000000 __bss_start
00000000000110f6 g       .data  0000000000000000 __DATA_BEGIN__
00000000000110fa g       .data  0000000000000000 _edata
0000000000011108 g       .bss   0000000000000000 _end
```

The assembling and linking used the symbols and relocated symbols to memory addresses. All numeric values are hexadecimal in the following discussion. The **.section .text** begins at 100e8, **.rodata** at 100f0, **.data** at 110f6, and **.bss** at 110fc. The labels have assigned addresses, e.g., **copy_begin** is in section **.bss** at address 110fc. The linker introduces further symbols.

Finally, we look at the content of the file in hexadecimal format using **objdump -F -s**, whereas **-F** is for additional information of the file:

```
riscv64-linux-gnu-objdump -F -s example

example:     file format elf64-littleriscv

Contents of section .text:  (Starting at file offset: 0xe8)
 100e8 9308d005 73000000                    ....s...
Contents of section .rodata:  (Starting at file offset: 0xf0)
 100f0 54657874 0005                        Text..
Contents of section .data:  (Starting at file offset: 0xf6)
 110f6 01000000                             ....
Contents of section .riscv.attributes:  (Starting at file offset: 0xfa)
 0000 412d0000 00726973 63760001 23000000  A-...riscv..#...
 0010 05727636 34693270 305f6d32 70305f61  .rv64i2p0_m2p0_a
 0020 3270305f 66327030 5f643270 3000      2p0_f2p0_d2p0.
```

The first two double words of the section **.text** is the program code, as shown in the disassembly. Note the order of the bytes has reversed, e.g., 9308d005 instead of 05d00893 in the disassembly. The content is shown as the bytes are stored in memory, respectively the file - and not how a 32-bit value or instruction is composed as RISC-V little endian. In little endian, the 32-bit instruction is built from the lower address as the highest byte to the higher address as the lowest byte. Thus, the four bytes 93 08 d0 05 from 100e8 are taken and put together to the 32-bit value/instruction as 05d00893. Big endian is the mode that combines the values the other way around.

The section **.rodata** begins after the two 32-bit instructions at 100f0 (= 100e8+8) and contains the bytes 54657874 0005. The values 54 65 78 74 are the ASCII codes for T e x t. They are followed by a zero (00) and the length 05 of the text, including the zero text delimiter. The section **.data** follows, which contains the value 1 as 32-bit which has to be encoded as little endian. It follows information about the RISC-V attributes.

There is no content in the section **.bss**; thus, it is not shown. However, as the symbol table shows, section **.bss** starts at 110fc. This is the next address after the section **.data** (ends at 110fa) plus two bytes in order to align it to a multiple of four.

In summary, we see how the assembling and linking put the sections and symbols together in the ELF-Format.

Pseudo Instructions

Bookmark this page

An assembler translates an assembly program into machine code. In this course, we mainly use the GNU assembler. The assembler does not only understand the RISC-V instructions specified by the ISA, but it also understands pseudo instructions and directives.

Pseudo instructions are part of the assembly language and are translated into machine code. In contrast to the ISA instructions, pseudo instructions are easier for the programmer to use. A pseudo instruction can be translated into more than one ISA base instruction. Further, pseudo instructions resolve the issue of using manually relocation functions. The next tables show pseudo instructions and how they are translated into basic instructions.

Pseudo Instructions for Load, Store, and Complement

Bookmark this page

Pseudo instruction	Base instruction(s)	Description
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, symbol[11:0]`	Load address (non position independent code - non-PIC)
`la rd, symbol`	`auipc rd, symbol@GOT[31:12]` `l{w|d} rd, symbol[11:0](rd)`	Load address (position independent code PIC)
`lla ra, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load local address
`lga rd, symbol`	`auipc rd, symbol@GOT[31:12]` `l{w|d} rd, symbol@GOT[11:0](rd)`	Load global address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rs, symbol, rd`	`auipc rd, symbol[31:12]` `s{b|h|w|d} rs, symbol[11:0](rd)`	Store global
`nop`	`addi x0, x0, 0`	No operation
`li rd, imm`	`Different instructions`	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	1's complement
`neg rd, rs`	`sub rd, x0, rs`	2's complement
`negw rd, rs`	`subw rd, x0, rs`	2's complement word

The position-independent code depends on your linker settings and is useful when generating libraries. It goes along with the use of the global offset table (GOT). The GOT is stored in the executable. It allows the operating system to load libraries at program startup to different memory areas. This is an advanced topic and is not used and not covered here. For more information, see the RISC-V ELF Specification. Please note that the pseudo instruction **`lga`** is not available in the GNU toolchain 2.39.

In the table, a symbol is a label though an address is understood, too. Let us consider the following example code and how it is translated by viewing at the disassembly:

```
.text
.globl _start
_start:
    la x1, counter    #load address of counter
    addi x1, x1, 4    #go to next word address += 4
    lw x2, 0(x1)      #load value from address, the 2

    li x1, 2
    lw x3, counter    #load word from address counter, the 0
    add x3, x2, x1    #add: x3 = x2 + x1
    sw x3, counter, x2 #save x3, use x2 for address
.data
counter:
    .word 0, 2
```

```
Disassembly of section .text:

00000000000100e8 <_start>:
   100e8: 00001097           auipc x1,0x1
   100ec: 02808093           addi x1,x1,40 # 11110 <__DATA_BEGIN__>
   100f0: 00408093           addi x1,x1,4
   100f4: 0000a103           lw x2,0(x1)
   100f8: 00200093           addi x1,x0,2
   100fc: 00001197           auipc x3,0x1
   10100: 0141a183           lw x3,20(x3) # 11110 <__DATA_BEGIN__>
   10104: 001101b3           add x3,x2,x1
   10108: 00001117           auipc x2,0x1
   1010c: 00312423           sw x3,8(x2) # 11110 <__DATA_BEGIN__>
```

The memory address of the label counter is 11110. The pseudo instructions are translated into basic instructions:

```
la x1, counter              -> auipc x1,0x1; addi x1, x1, 40
li x1, 2                    -> addi x1, x0, 2
lw x3, counter              -> auipc x3, 0x1; lw x3, 20(x3)
sw x3, counter, x2          -> auipc x2, 0x1; sw x3, 8(x2)
```

Pseudo Instructions for Extending and Conditional Bit Setting

Bookmark this page

The following table shows the pseudo instructions and base instructions for extending and conditional bit setting, e.g., **`seqz x2, x3`** is translated into **`sltiu x2, x3, 1`**.

Pseudo instruction	Base instruction(s)	Description
`sext.{b|h|w} rd, rs`	`different instructions`	sign extend
`zext.{b|h|w} rd, rs`	`different instructions`	zero extend
`seqz rd, rs`	`sltiu rd, rs, 1`	rd = (rs == 0)? 1:0
`snez rd, rs`	`sltu rd, x0, rs`	rd = (rs != 0)? 1:0
`sltz rd, rs`	`slt rd, rs, x0`	rd = (rs < 0)? 1:0
`sgtz rd, rs`	`slt rd, x0, rs`	rd = (rs > 0)? 1:0

Pseudo Instructions for Conditional Branching

Bookmark this page

The next table shows the pseudo instructions for conditional branching. One part of them simplifies branching using comparisons with zero by omitting the register x0. The other part offers comfort functions for more comparisons by rearranging the parameters for base instructions, e.g., branch less or equal **`ble rs, rt, imm`** is the same as **`bge rt, rs, imm`**.

Pseudo instruction	Base instruction(s)	Description
`beqz rs, imm`	`beq rs, x0, imm`	if (rs == 0) PC+=imm
`bnez rs, imm`	`bne rs, x0, imm`	if (rs != 0) PC+=imm
`blez rs, imm`	`bge x0, rs, imm`	if (rs <= 0) PC+=imm
`bgez rs, imm`	`bge rs, x0, imm`	if (rs >= 0) PC+=imm
`bltz rs, imm`	`blt rs, x0, imm`	if (rs < 0) PC+=imm
`bgtz rs, imm`	`blt x0, rs, imm`	if (rs > 0) PC+=imm
`bgt rs, rt, imm`	`blt rt, rs, imm`	if (rs > rt) PC+=imm
`ble rs, rt, imm`	`bge rt, rs, imm`	if (rs <= rt) PC+=imm
`bgtu rs, rt, imm`	`bltu rt, rs, imm`	if (rs > rt) PC+=imm, unsign.
`bleu rs, rt, imm`	`bgeu rt, rs, imm`	if (rs <= rt) PC+=imm, unsign.

Pseudo Instructions for Unconditional Jumping

Bookmark this page

Finally, the pseudo instructions for unconditional jumping simplify programming for the developer. Registers that are not required do not appear with pseudo instructions. And jumping to an address far away is provided by call and tail.

Pseudo instruction	Base instruction(s)	Description
`j imm`	`jal x0, imm`	PC += imm
`jal imm`	`jal x1, imm`	x1 = PC+4; PC += imm
`jr rs`	`jalr x0, rs, 0`	PC = rs
`jalr rs`	`jalr x1, rs, 0`	x1 = PC+4; PC = rs
`ret`	`jalr x0, x1, 0`	PC = x1
`call imm`	`auipc x6, imm[31:12]` `jalr x1, x6, imm[11:0]`	x1 = PC+4; PC = imm
`tail imm`	`auipc x6, imm[31:12]` `jalr x0, x6, imm[11:0]`	PC = imm

Application Binary Interface (ABI) and User-Mode

Bookmark this page

Until now, we have used registers by their numbers. However, RISC-V has an application binary interface (ABI) which is a convention on how registers should be used in a general context. It provides aliases, ABI names, for the registers as well. One convention is the responsibility of storing register values when a jump instruction calls a function. This means the return address of the instruction after the jump instruction in memory is stored in a register and the program counter is the beginning of the function. The function consists of instructions, which are processed, and returns back to the instruction following its call. To return, the function uses the stored return address. It is obvious that there should be a convention which register is used for return addresses when different programmers work together on a larger program. Further, a convention is important, which registers can be assumed as changed or not changed when a function returns. This table summarizes the aliases and storing convention, e.g., the register x1 has the alias ra which stands for return address, and the caller of a function has to store the value of register x1 before the function is called and restore it after the function's return.

Register	ABI alias	Description	Saver
x0	zero	zero constant	-
x1	ra	return address	caller
x2	sp	stack pointer	callee / function
x3	gp	global pointer	- / should not be used from user
x4	tp	thread pointer	- / should not be used from user
x5-x7	t0-t2	temporaries	caller
x8	s0 / fp	saved / Frame pointer	callee / function
x9	s1	saved register	callee / function
x10-x11	a0-a1	function args. / return values	caller
x12-x17	a2-a7	function arguments	caller
x18-x27	s2-s11	saved registers	callee / function
x28-x31	t3-t6	temporaries	caller
pc	-	program counter	-

The stack pointer, global pointer, and thread pointer are used in a system's context. The global pointer and the thread pointer should not be used except from the (operating) system. The stack pointer can be used for storing/restoring values.

We consider here only the unprivileged instructions of the ISA as the focus is on the basics of the RISC-V assembly language. Unprivileged instructions can be used in every privilege level. RISC-V comes with three privilege levels: user (U), supervisor (S), and machine (M). System functions, e.g., of an operating system, run in supervisor-mode or machine-mode. To use system functions from user-mode, the system functions can be called, but require that you stick to the application binary interface.

Stack

Bookmark this page

The stack is a memory area that is used via the stack pointer (register sp). The stack pointer is usually provided by the system; thus, the register sp comes initialized for a user application. The stack grows from a high address to a low address in memory. It is filled from the top to the bottom.

The storing of data on the stack is done by the so-called push action. The value is taken from the stack by the pop action. Let us consider that the register sp has the value 0xff0:

```
li t0, 0xbeabdeaf       # t0 = 0xbeabdeaf
addi sp, sp, -4         # grow stack for four bytes
sw t0, 0(sp)            # push t0 (0xbeabdeaf) to the stack
```

The stack pointer is updated to 0xfec, and the value 0xbeabdeaf is stored at the memory address 0xfec. The value is retrieved, and the stack pointer is restored as follows:

```
lw t0, 0(sp)            # pop 0xbeabdeaf from the stack to t0
addi sp, sp, 4          # shrink stack back
```

Implementation of Control Flow

A program begins with the instruction specified by the program counter. The program counter determines the program flow. Without the branch and jump instructions, instructions would be executed one after the other as they are arranged one after the other in memory. Using the branch and jump instructions allows control of the program's flow. They enable you to implement conditionals and loops, which are basic features of imperative programming. Using unconditional jumps with return addresses (together with the ABI conventions) enables the use of functions (procedures), a feature of procedural programming. As these features build up how we can code our programs, their implementation with assembly instructions is presented.

Conditionals

Bookmark this page

Conditionals check if a condition is true or false. Depending on the result, a branch is taken. This is usually known as an if-then-else-statement.

The implementation of an if-then-else-statement in assembler is done by the instructions for conditional branches. If a condition is (not) true, a branch takes place. Often, the logic of the original if-then-else-statement is reversed using branches; thus, the code for the 'else'-part appears before the code for the 'then'-part, for example:

```
# if condition is true then 'code for then' else 'code for else'
# if (t0 == t1) then branch to label 'then'
   beq t0, t1, then
# else
...  # code for else
# jump over then-part
   j end
then:
...  # code for then
end:
```

A simple if-then-statement can be implemented with a single branch instruction:

```
# if (t0 == t1) then 'code for then' = if (t0 != t1) skip 'code for then'
  bne t0, t1, skip
… # code for then
skip:
…
```

More complex conditionals can be implemented by nested constructs.

Loops

Bookmark this page

Loops or repetitions are implemented by conditionals in assembler. A while-do-statement repeats the do-part, the loop, while a condition is true. For the while-do-statement, the condition is checked at the beginning of the instructions to be repeated. If this condition does not hold, the loop part is skipped. For example:

```
# initialize t0 and t1
# loop shall be taken four times
    li t0, 0
    li t1, 4
while:
# if t0 == t1 goto end, leave loop
    beq t0, t1, end
... # code for the loop
# increment: t0 = t0 + 1
    addi t0, t0, 1
# repeat loop until above condition (t0 == t1) is true
    j while
end:
```

Function

Bookmark this page

A function (or routine) is implemented by saving the return address, jumping to the function code and returning from the function. It has to be taken care to use the correct register for saving the return address. It is therefore best to stick to the ABI which provides a convention about the usage of registers for parameters and return addresses. For example:

```
# parameter in a0
    li a0, 0
# call of func, return address in ra
    jal ra, func
# Continued here after return from func

# ...

# function code
func:
... # code for function, must not change ra
# return results in a0, here the value 1
   li a0, 1
# return to address given by ra
  ret
```

Instructions that push registers on the stack at the beginning of a function are called function prologue, and instructions that pop registers from the stack at the end of a function are called function epilogue. They serve to prepare the use of the registers inside the function's main body. A simple function prologue/epilogue is in the following example of a recursion.

Recursion

Bookmark this page

A recursive function is a function that calls itself; it does a recursion. Recursive functions usually have local variables, which are in registers that need to be saved before the function calls itself and restored after its return. A recursive function can be error-prone, as the (re)storing of register content uses the stack. A stack overflow can happen if the stack does not contain enough space.

Let us consider the computation of the n-th element of the sequence a(n) = a(n-1) + 3, whereby a(0) = 2. Thus, a(0) = 2, a(1) = a(0) + 3 = 5, a(2) = a(1) + 3 = (a(0)+3) + 3 = 8 and so on. The corresponding function calls itself until it reaches the initial value for a(0). The following program uses the recursive function compute to calculate a(n) whereby the parameter n has to be in register a0 for the call. The result is given back in register a0.

```
.globl _start
_start:
        li a0, 5        # compute for n = 5
        call compute

        # exit
        li a7, 93
        ecall
```

```
compute:
        # allocate stack for register ra
        # RV64 registers are 64 bit = 8 bytes
        addi sp, sp, -8
        sd ra, 0(sp)

        # check if recursion ends; yes? then jump
        beq a0, x0, compend

        # otherwise prepare a(n-1)
        addi a0, a0, -1
       # recursion
        call compute
       # result of a(n) = a(n-1) + 3
        addi a0, a0, 3
        # return function
        j compret
compend:
        # if (n ==0) is reached, return a(0) = 2
        li a0, 2
compret:
        # restore ra
        ld ra, 0(sp)
        # free stack
        addi sp, sp, 8
        ret
```