tiny-asm: an assembler for riscv

jacob navia

Contents

List of Tables

List of Figures

1 The RISC-V assembler

1.1 Introduction

The tiny assembler is a "digest" of the GNU `gas` assembler. I have extracted from the 1.3Gb of `binutils-gdb` source code[1] two files: `asm.c` and `asm.h`.

There are two goals here:

1. To produce a small and fast assembler to be used as a compiler back-end. The elimination of features proceeds according to this goal: assemble machine generated output, without consideration for any human user, since all input to the assembler is supposed to be machine generated.

2. To produce a minimal set of sources that is *easy to read and understand* so that people can hack away without a lengthy learning curve. This documentation also, contributes to this objective.

In this version of the tiny-assembler there isn't:

- No input pre-processing. No include files, nor any fancy macro processing.

- No fancy error messages, messages will be emitted only in english. If you want other language error output you are welcome to do it yourself. The rationale behind this is obviously that a high level language user, programming in C++ or C, will be completely unable to understand the assembler messages even if they are translated into his/her native language.

- This assembler is geared to the riscv CPU. All support for any other machine has been dropped, specially support for machines that have ceased to exist for more than 20 years: the Motorola 68000 family, the Sparc, the Z80, etc. I think that even `gas` could drop support for those machines also.

- The code has been cleaned up from all cruft like this:

```
/* The magic number BSD_FILL_SIZE_CROCK_4 is from BSD 4.2 VAX
* flavoured AS.  The following bizarre behaviour is to be
* compatible with above.  I guess they tried to take up to 8
* bytes from a 4-byte expression and they forgot to sign
* extend.  */
#define BSD_FILL_SIZE_CROCK_4 (4)
```

  So, we are still in 2023 keeping bug compatibility with an assembler for a machine that ceased production in 2000?

[1] I have just done a `du -b ./binutils-gdb` Probably is a bit less since I didn't do an extensive search for only .c and .h files.

- All the indirection through macros that are expanded into members of function tables that makes the code impossible to follow are eliminated. Now, if you see code like `foo(bar);` it is highly likely that you are calling function `foo` with argument `bar`...
- All libraries are eliminated. Tiny-asm doesn't use `BFD` nor `libiberty` nor `libopcodes`. The only library used is `zlib`.
- There are only two files: `asm.c` and `asm.h`. No other include files are there, as far as I remember, excepting system includes like `stdio.h` of course.

I have avoided to put much code samples here. There only two source files, and if you want to see the exact code sequences you are free to look them up, it is not very difficult. I see no interest in filling pages with code.

1.1.1 Requirements

I have concentrated in explaining how things work, and that includes talking about specifications and the standards used. You should have:

1. Source code: If you want the official sources of the GNU assembler you should download the binutils-gdb package. It is available in many places, for instance in github binutils.

   You can download the sources of the `tiny-asm` from:
   `https://github.com/jacob-navia/tiny-asm`.

2. Assembler user documentation in "Using as".
   `https://sourceware.org/binutils/docs/as/` This is the official documentation for the Gnu Assembler. Tiny-asm has kept most of it, and the algorithms, names of functions and variables are almost always the same. Knowing what the user specifications are will help you understand what the different assembler directives are doing.

3. The RISC-V Instruction Set Manual Volume I: Unprivileged ISA. There are a lot of versions of this document in the internet. Please try the most recent that you can find, of course. The official sources of the documentation are in riscv-specs, but there are apparently more recent ones. There is a depository in github at github, but they are in a strange format called "adoc" that is difficult to find a translator for, in non-windows systems.

4. DWARF debug information standard, the most recent being DWARF5 (2017) at the DWARF standard. This will enable you to better understand the debug information (cfi) directives of the assembler.

5. The ELF (Executable and Link Format) standard has an official page in the linux foundation at
   `https://refspecs.linuxfoundation.org/elf/elf.pdf`.

   ELF is the object format standard followed by the assembler. This will help you understand the `write_object_file` better.

6. You should obviously have a riscv machine. If you don't use a simulator (slow) buy a cheap board that can run linux. The chinese propose several machines, like pine64. This is the machine I am using, for around 110 US$. You can buy similar ones directly from the chinese, for instance
   waveshare, or buy it from amazon.com, there are several boards available there. The Sifive company sells riscv boards also, but they are not interested in retail sales. Demands for price and availability go into the bit bucket unless you are a huge company with orders of several hundred boards probably. But you can always try at

https://www.sifive.com.
Recently, the Chinese company sipeed has released a clone of the raspberry pi using riscv. It is called `Lichee pi 4A` and comes in two versions: 8GB and 16GB RAM. It sells for around 150 euros at the "Ali-baba" Chinese retailer: ali-express. This machine is much faster than many other boards and features a processor with the vector instructions extension.

1.2 Building tiny-asm

The build process runs as follows:

1. Download the software from github
2. Build it:

   ```
   $ gcc -o asm -g asm.c -lz
   ```

That is it. There is no Makefile but you can write one. I wrote this one:

```
star64:~/tiny-asm$ cat Makefile
asm: asm.o
gcc -o asm asm.o  -g -lz

asm.o: asm.c asm.h
gcc  -W -Wall -Wstrict-prototypes -Wmissing-prototypes\
-Wshadow -Wwrite-strings -g -c asm.c
clean:
rm -f asm.o asm
```

The Makefile for `gas` is 2268 lines... an impressing piece of software. However I think that 9 lines is much easier to understand. The user wants to use an assembler, maybe modify it, so there is no point in making him/her try to modify a 2 thousand line Makefile.

1.3 What is an assembler?

In these times of hype and exaggeration we speak about "artificial intelligence", of "programming without coding", and many other things that are designed to make us believe that we must buy the next gizmo.

Let me tell you one thing: All that a machine can do will be described here, in the assembler. And you will see no intelligence, no high level concepts, just extremely simple operations like adding a number to another, or reading and storing data in memory in small sized chunks.

Machines can't do anything else, and the smallest living being is many orders of magnitude more complex than these machines.

Why assembly language?

Because it is the only way to understand what is *really* happening inside a computer, the only way to pierce the hype from the abstractions of software and get into the last interface: the interface between the hardware interpreter and the software.

All computers are that: an interpreter. They receive encoded instructions and executes those, producing results that are written to memory.

This interface is called the "ISA" or "Instruction Set Architecture". It is the whole set of *instructions* i.e. small pieces of operations that are encoded in the wires of the machine and give software the set of operations it can perform.

The machine that tiny-asm encodes instructions for is called a RISC-V (risc five) ISA. It is a **R**educed **I**nstruction **S**et **C**omputer, i.e it is a descendant of an idea of an IBM engineer. The first prototype computer to use reduced instruction set computer (RISC) architecture was designed by IBM researcher John Cocke and his team in the late 1970s. This idea is not new, but it has come to fruition lately. Almost all the smart phones in the world use an "**A**corn **R**isc **M**achine" (ARM) architecture.

Assembly is not really a complicated language. Your learn it in a few words. It consists of instructions for the machine, and instructions for the assembler, also called directives. They are written one per line, in a simple format:

```
<instruction name> <arguments...>
```

where `<arguments>` is just a comma separated list.

All instructions in the riscv ISA have the first argument as the *destination* where the result of the operation is written to. The rest of the arguments are the inputs that the machine uses to perform the operation. For instance an addition needs two numbers as input and is written like this:

```
add t1,t2,t3
```

where those t_n are special memory locations called *registers*[2]. The machine will take `t2` and add to it `t3` and put the result into the `t1` register.

Now, a machine can't read. It can't read the "add" above. So, here comes "tiny-asm", the super-hero of our story. It will take the text above "add t1,t2,t3" and figure out that "add" is a legal operation that is present in its opcode table. It will take the *code* of the operation (a series of small numbers), it will put the codes of the register t1, t2, and t3 in the places the instruction format has setup for them, and will ship that to the growing program.

The integrated circuit will decode the instruction encoded by the assembler, separate the different parts, the destination register, the data sources, etc, and dispatch the whole to the specialized part within the CPU that handles the instruction.

Low level programming is difficult and sometimes tedious. You are just telling the machine to move this piece of data here, do that micro-operation and put the result somewhere. The *reason* why you are doing these operations is written nowhere. There are no classes nor structures, just nothing but micro-instructions and data movements. You get numbed down by the sheer size of things you have to write just to do something that you write in a single instruction in a high level language.

To avoid this, it is better to let the machine work for you. Just use a compiler that can output assembly language, and work using the program that the compiler gives you.

Of course do not use a high level of optimizations when you do that, because the code that the compiler will hand you will be completely unreadable.

And remember: The compiler has to output a program that is correct for all types of programs. You, when programming using tiny-asm, you have just to better *this* program, so you can do things that the compiler will never do in the general case.

Since you have the source code of the assembler, after reading this documentation you will be able to understand how it works and modify it as you like.

Since the assembler is at the base of everything, many subjects are discussed here: cryptographic operations, bit operations, instruction formats, object file formats, debug information format, etc. Each of those subjects would need a book of its own. Do not expect an in-depth treatment of any of those subjects. They will be discussed with the point of view of the assembler, nothing more.

1.4 Overview

Like all assemblers, this assembler has a **parser**, where the text of the input file is converted into logical units that represent either instructions for the machine, or for the assembler

[2] See table §1.1 page 28

itself, called *pseudo instructions*, and an **encoder**, where the instruction and its arguments are encoded into a 32 or 16 bit instruction and added to the current fragment. And then there is the object file generation, where the instructions and associated information are packed into the ELF format.

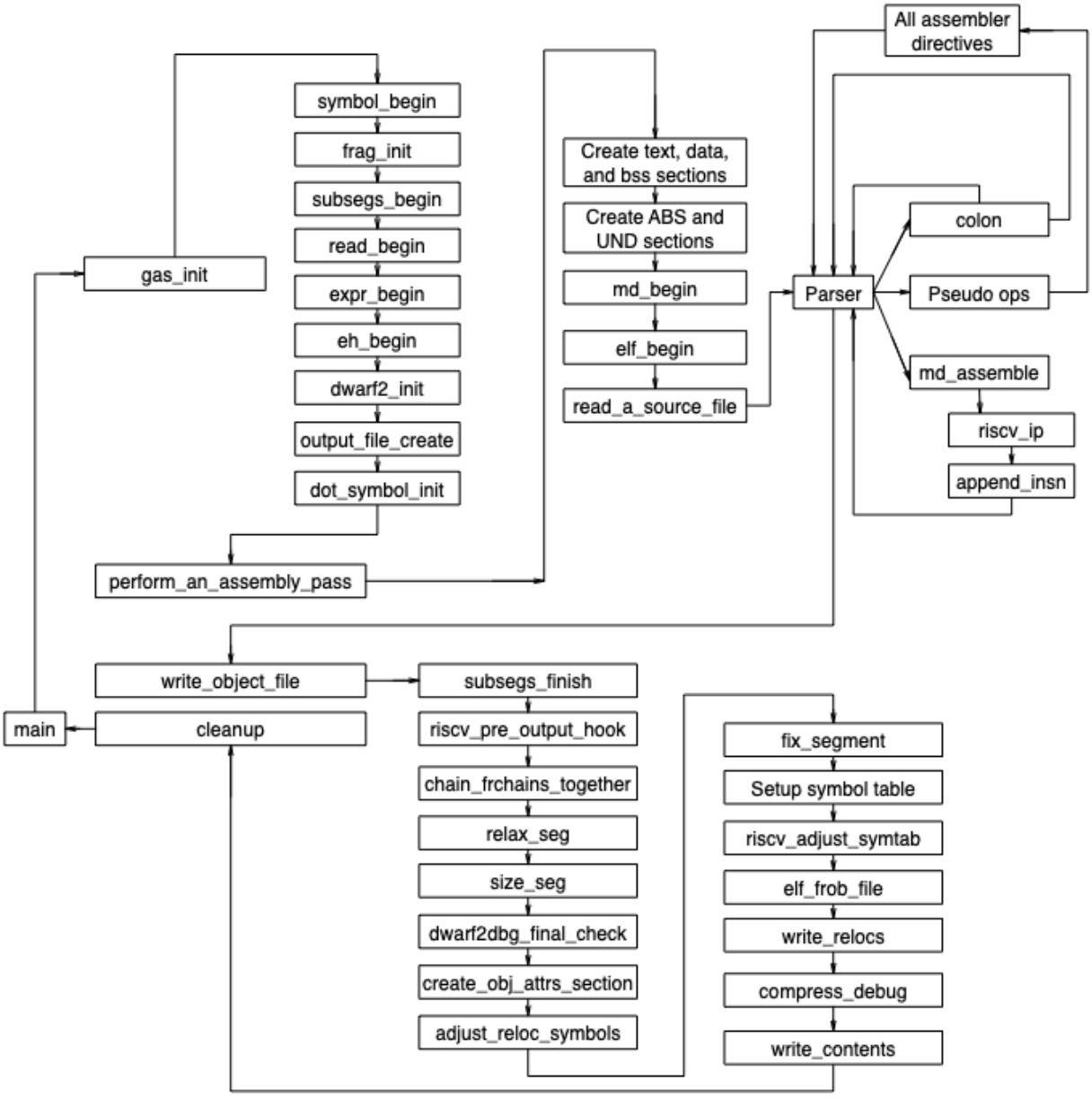


Figure 1.1: Overview of the assembler control flow

In figure 1.1 (page 13) we have these three main parts. Please keep in mind that this is a high level abstraction of the control flow. Obviously, if we would put each statement in the diagram we would have to cram 40 000 lines into a diagram... too much.

We start with `main` that organizes all three parts [3]. It calls the initialization, `gas_init`, that initializes the symbols (`symbol_begin`,),the fragments initialization, the sub-segments, etc.

"Fragments" are understood in the assembler as pieces of code already assembled but that can grow, getting new instructions or other data. They are of variable length, and they will be strung together in a process called "relaxation" at the end of the assembly.

The initialization of the "sub-segments" means the text, data, and bss sections are created. Are "sub-segments" just plain object file sections? Not quite. There are "sections" like the "ABS" (absolute) section or the "UND" (undefined) sections that will never be written out in the object file.

There are other initializations that give us the opportunity of explaining some concepts that will be important later on. The `eh_begin` function, for instance, initializes the "exception handling" stuff. This is a complicated system that allows languages like C++ to walk the stack at run time, searching for a handler that will accept handling the exception that has just occurred.

This process involves an impressive machinery that contains a set of tables that associate addresses in the code to descriptions of the stack contents that allow a debugger or a runtime interpreter to see what functions have in terms of local variables and the space that each stack frame uses in the stack. And even if you are programming in C and you do not have any need for exceptions you will get them anyway since your C code could be called from a C++ program.

Other initializations concerns the start of the `dwarf2` debug information generation. Yes, the assembler can emit debug information for the program it is assembling. This way, the assembly programmer can follow the program line by line. `tiny-asm` has kept this even if it is highly unlikely that the compiler, that emits its own and much richer debug information, will need this.

The initialization of the "dot symbol" needs also some explaining. The current location when assembling a program is called "dot", i.e. a point. This symbol is always associated with the current address following a long assembler tradition that goes back to the start of the micro-computer age.

Eventually we come to the `perform_an_assembly_pass` function. This one continues the initialization process by creating the standard sections of the object file:

- The text section. This is a misnomer since there isn't anything textual inside. It contains the binary codes that will be interpreted by the integrated circuit. This is the most important output of the whole assembly process.
- The data section. This contains the tables, constants, structures and everything that the programmer has defined as static data that will be loaded at the start of the program by the program loader.
- The BSS section that contains nothing. It is just a reserved memory space that will be allocated by the program loader when it loads the program and contains always zeroes at the start.
- There are many other sections in an ELF format file. Let's stop here.

Then, we finish the setup process by calling `md_begin` and `elf_begin` functions.

[3] Please be aware that in the diagram there is a direct link between, for instance, the function `dot_symbol_init` and `perform_an_assembly_pass`. This does NOT mean that the first calls the second directly. It means that the flow of the program returns to `gas_init` and then returns to the main function , and it is `main` function that calls `perform_an_assembly_pass`.
That would be quite complicated to draw, however. So, the diagram simplifies this.

The `md_begin` function reads all the static tables and builds hash tables from the for fast access. The opcodes are stored in hash tables, together with other data like the register names, the Control and Status Registers (CSRs) and what have you.

The `elf_begin` function builds symbols for each section in the object file. This allows to emit relocations or symbol addresses as an offset from the start of the section.

The setup phase behind us, we start the real work of the assembler: the well named `read_a_source_file`. This function does the parsing and the encoding of the instructions and directives.

In the diagram below, the functions aren't shown with their actual names but with their functional description. The GAS developers took (as you can see) a lot of effort to choose clear names that describe quite well what each function is doing. Still, I thought that here we will use functional boxes instead of function names, since some of the functions described here do not exist as a separated subroutine but they are just pieces of `read_a_source_file`.

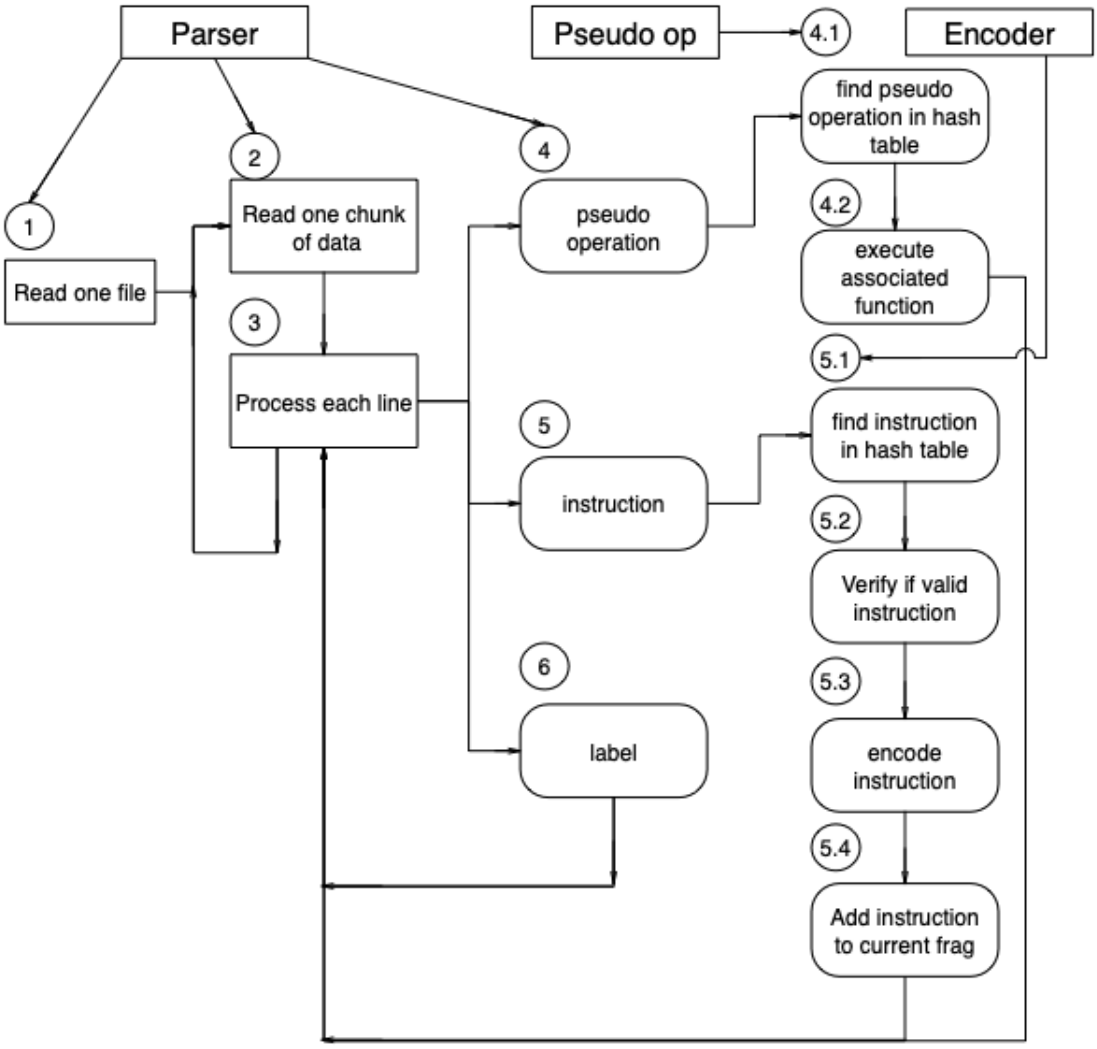


Figure 1.2: A more detailed view of the parser

We assume that the assembler input is a single file containing instructions, data, and

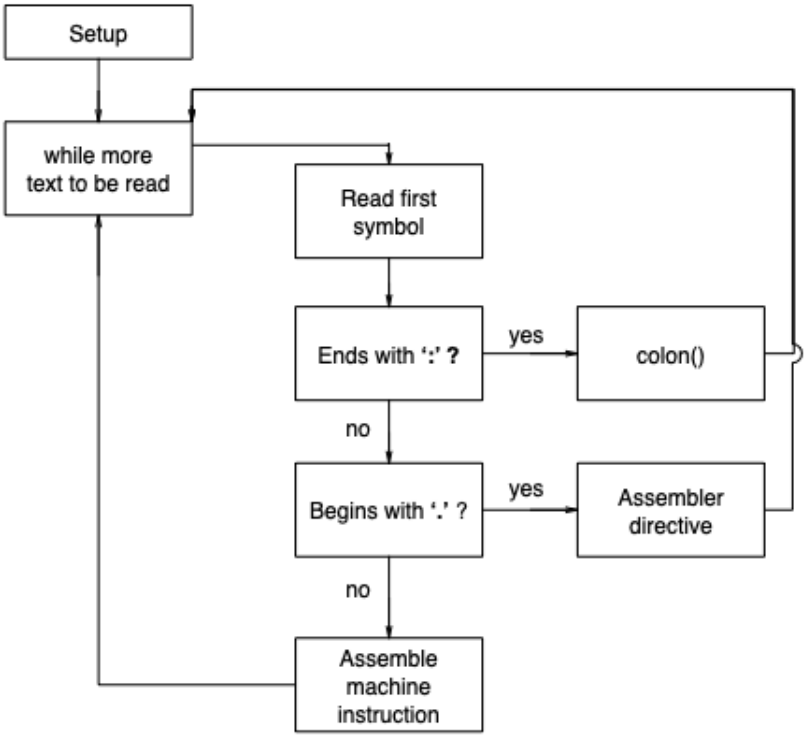


Figure 1.3: `read_a_source_file function`

assembler directives. In this version of the assembler, parsing is reduced to a bare minimum since we assume that we are assembling compiler output, and all the sophistication that is needed for an assembler adapted to human use is not needed for an assembler that is used to parse machine output.

We start with the function `read_a_source_file` that organizes the parsing and the instruction generation[4].

1. Setup. Here, we setup the input file name, in variable `physical_input_file` and we care about writing a file name record if we are emitting debug information.

2. We read a chunk of the input file. Currently, `BUFFER_SIZE` is set to 256 * 1024, and can be changed just by editing the corresponding line in asm.h

3. We start parsing lines. The first thing we read should be a symbol. If it ends with a colon, it is a label definition. We call the corresponding function `colon()` and continue parsing. If it is not finished by a colon, we see if the first letter is a point. If it is, it is an assembler directive. We call the corresponding function stored in the pseudo-ops structure (called `pseudo_typeS`) and we go fishing for the next line. If it is not a pseudo-operation, it *must* be a machine instruction. We call the `md_assemble` function.

The `md_assemble` function does basically following things:

1. Test if the instruction is valid using the current set of RISCV specifications. There are instructions that can be issued only with 64 or even 128 bits, or floating point instructions that depend on floating point being implemented in hardware, etc. RISCV

[4] Actually, the initialization phase is executed before, but we will abstract that away for the time being

machines can have a number of extensions implemented, since the basic ISA (Instruction Set Architecture) doesn't even have multiplication or division!

Each "extension" has a letter that characterizes it. For instance, in the machine I am using we have in /proc/cpuinfo a line with:

```
isa : rv64imafdc
```

This means that the machine is a risc V 64 bits machine (rv64), with the integer (i), multiplication (m), a (Atomic), f (single precision floating point), d (double precision floating point) and c (Compressed instructions in 16 bits) extensions. The assembler should test if any instruction is legal in the current subset, and reject those that do not comply.

Since we are an assembler for reading compiler output, we just assume the compiler doesn't emit wrong instructions and skip this test.

2. We call the `riscv_ip` function to encode the instruction. Basically it uses the `args` character string to know what arguments are expected. It verifies that those are correct, and inserts all necessary bits at the required positions. We will see later how these formats are defined.

3. If assembly succeeded the new instruction is added to the current fragment.

The `riscv_ip` function is basically a huge switch statement. The function will go through each one of the characters present in the `args` string of the opcode and add the necessary bits to the instruction.

In the tables below you will find the description of the different formats defined for each of the instructions in a riscv machine. This tables will help you understand how `riscv_ip` works.

1.5 General concepts and data structures

1.5.1 Binary File Descriptor (bfd)

This structure is at the heart of the BFD library. In the context of an assembler, there is only one of these beasts around, called `stdoutput`.

The main things stored here are:

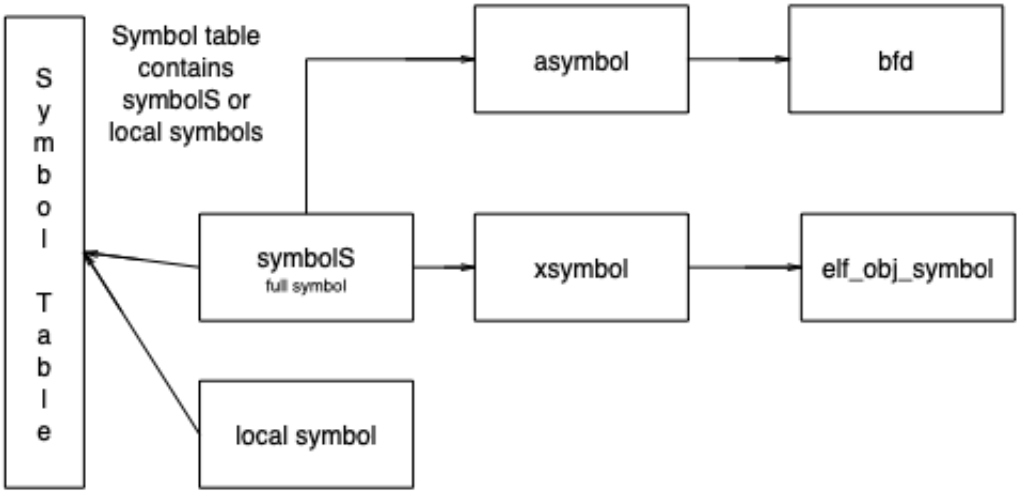
- The file name.
- A table of function pointers that dispatches to the back end for doing most of the work.
- The input output stream
- A pointer to the back-end private data.

Of course there are a lot of other fields that you can study by reading the definition of this structure in asm.h.

It is important to underscore here that the table of function pointers has been completely eliminated in `tiny-asm`. There are no more indirection through the `xvec` field, since tiny-asm will only assemble riscv instructions. The front end and the back end have been merged into a monolithic whole. Still, this design is essential for understanding `gas`.

1.5.2 Symbols

There are several types of different structures that together represent a symbol. They will be described below, but in general they reflect the need by the bfd library to make a back-end independent structure that holds some general information, a high level abstraction of a symbol. Back-ends can differ in the object format, and in the cpu used, so the information that is common to all those very different contexts is rather minimal.



asymbol

This is the bfd-internal format, holding the following things:

- `the_bfd`. This points to the bfd that this symbol refers to. Since under tiny-asm there is only one bfd, called stdoutput, this is redundant. In other contexts, for instance in the linker where there are a lot of binary files opened for reading and one for writing, this makes much more sense.
- Name The name of the symbol.
- Value. Here there is either a pointer to some other symbol, or a numeric value.
- Flags. A long list of different flags. Some of them aren't used in tiny-asm, but their definition is still there since they are used in the linker.
- A pointer to the section.
- A pointer to special data used by the back end. It is a union of a generic pointer and an address.

To access the fields of an `asymbol` inline functions with rather lengthy names are provided. These functions look like this:

```
1 static inline asection * bfd_asymbol_section (const asymbol *sy)
2 {
3     return sy→section;
4 }
```

This function accesses the `section` field[5]

[5] Is this really necessary? What are the real advantages of using `bfd_asymbol_section(foo)` instead of `foo→section`?
Yes, I know. It is called "information hiding". But the problem is that information hiding **hides** information, precisely, and if you are trying to understand what the code is doing, you do not know beforehand if that is a call to a lengthy function or just a field access.

`asymbols` are global, and used for data that concerns a whole section. The constructor for these objects is `elf_make_empty_symbol`

symbolS

This structure is used for all kinds of symbols (labels, functions) that the assembler extracts from the source code. In the symbol table, we find pointers to this one and to the `local_symbol` structures. The size of this should be the same or less than `struct local_symbol`, and fields that do not fit in that size go into an overflow structure called `xsymbol` for e**x**tension of symbol.

Listing 1.1: symbolS, xsymbol, elf_obj_sy

```
typedef struct symbol {
    struct symbol_flags flags; /* Symbol flags. */
    hashval_t hash;            /* Hash value calculated from name. */
    const char *name;          /* The symbol name. */
    fragS *frag;    /* Pointer to the frag of this symbol,if any. Otherwise NULL.*/
    asymbol       *bsym;       /* BFD symbol */
    struct xsymbol *x;         /* Extra symbol fields that won't fit. */
} symbolS;

/* Extra fields to make up a full symbol. */
struct xsymbol {
    expressionS value;  /* Symbol value. Note that this is NOT a pointer */
    /* Forwards and backwards chain pointers. */
    struct symbol *next;
    struct symbol *previous;
    struct elf_obj_sy obj; /* Yet another symbol structure (YASS!) */
};

/* Additional information we keep for each symbol. */
struct elf_obj_sy {
    unsigned int local : 1;  /* Whether the symbol has been marked as local. */
    unsigned int rename : 1;  /* Whether the symbol has been marked for rename with
        @@@. */
    unsigned int bad_version:1; /* Whether the symbol has a bad version name. */
    /* Whether visibility of the symbol should be changed. */
    ENUM_BITFIELD (elf_visibility) visibility : 2;
    /* Keep track of .size expressions that involve yet unresolved differences */
    expressionS *size;
    /* The list of names specified by the .symver directive. */
    struct elf_versioned_name_list *versioned_name;
};
```

The constructors for `symbolS` are:

- `symbol_make`. This constructor is simple, code below:

```
    static symbolS *symbol_make(const char *name)
    {
        /* Let the machine description default it,e.g. for register names. */
        symbolS *symbolP = md_undefined_symbol((char *)name);
        if (!symbolP) symbolP = symbol_new(name,undefined_section,&
            zero_address_frag,0);
        return (symbolP);
    }
```

  So, if you read this you would think that first, as the commentary says, is calling to some function... Actually, for the riscv backend, we have a `#define` in asm.h:

`#define␣md_undefined_symbol(name)␣(0)`
`symbol_make` is just an alias for `symbol_new`.

- `symbol_create`. This function allocates space for the new symbol, sets some default fields, and then calls `symbol\_init` that will finish the construction of the new symbol.
- `symbol_new`. This is a small function that calls `symbol_create` and then links the new symbol into the global list of symbols using the function `symbol_append`.
- `symbol_find_or_make(const char *name)`. This function searches for a symbol and if not found creates an undefined symbol, returning a pointer to it. When creating a symbol, it checks if it is a local symbol. Then either calls the constructor for a local or a true symbol.

In the symbol table, full fledged symbols or local symbols appear. The distinction between them is that for many symbols like labels, or similar, all the huge amount of information described above make no sense. A shorter and smaller structure is used, what makes considerable gains in memory space.

The `symbol_flags` structure

Flags that annotate an object are normally defined as `#defines` for instance

```
#define FLAG_FOO 1
#define FLAGG _BAR 2
... etc
```

But there are better ways, for instance by defining a structure that can be accessed in a much clearer way.

Listing 1.2: The symbol_flags structure

```
struct symbol_flags {
    unsigned int  local_symbol:1; /* Whether the symbol is a local_symbol or not. */
    unsigned int  written:1;     /* Whether symbol has been written. */
    /* Whether symbol value has been completely resolved (used during final pass
     * over symbol table). */
    unsigned int  resolved:1;
    /* Whether the symbol value is currently being resolved (used to detect loops
     * in symbol dependencies). */
    unsigned int  resolving:1;
    /* Whether the symbol value is used in a reloc. This is used to ensure that
     * symbols used in relocs are written out,even if they are local and would
     * otherwise not be. */
    unsigned int  used_in_reloc:1;

    /* Whether the symbol is used as an operand or in an expression. */
    unsigned int  used:1;

    unsigned int  volatil:1;  /* Whether the symbol can be re-defined. */

    /* * Whether the symbol is a forward reference,and whether such has been
     * determined. */
    unsigned int  forward_ref:1;
    unsigned int  forward_resolved:1;

    /* This is set if the symbol is set with a .weakref directive. */
    unsigned int  weakrefr:1;

    /* This is set when the symbol is referenced as part of a .weakref
     * directive, but only if the symbol was not in the symbol table
```

```
     * before. It is cleared as soon as any direct reference to the symbol
     * is present. */
    unsigned int   weakrefd:1;

    /* Whether the symbol has been marked to be removed by a .symver
     * directive. */
    unsigned int   removed:1;

    /* Set when a warning about the symbol containing multibyte characters
     * is generated. */
    unsigned int   multibyte_warned:1;
};
```

Many of these fields are never used directly. The field `used_in_reloc`, for instance, is never accessed directly. The reason is clear when you look at the code for its access functions:

```
/* Mark a symbol as having been used in a reloc. */
static void symbol_mark_used_in_reloc(symbolS * s)
{
    if (s→flags.local_symbol)
        s = local_symbol_convert(s);
    s→flags.used_in_reloc = 1;
}
/* Return if a symbol has been used in a relocation */
static int symbol_used_in_reloc_p(symbolS * s)/* Return whether a symbol has been
     used in a reloc. */
{
    if (s→flags.local_symbol)
        return 0; /* Local symbols can't be used in relocations */
    return s→flags.used_in_reloc;
}
```

local_symbol

This structure can represent only symbols with offsets within the same `frag`.

Listing 1.3: local symbol

```
struct local_symbol {
    struct symbol_flags flags; /* Flags: Only local_symbol and resolved relevant.*/
    hashval_t  hash;          /* Hash value calculated from name. */
    const char    *name;      /* The symbol name. */
    fragS         *frag;      /* The symbol frag. */
    asection      *section;   /* The symbol section. */
    valueT     value;         /* The value of the symbol. */
};
```

Constructor for the `local_symbol` structure is the function `local_symbol_make`.

Symbol table

Listing 1.4: union symbol_entry_t

```
/* This structure makes up the entries of the symbol table */
typedef union symbol_entry {
    struct local_symbol lsy;
    struct symbol  sy;
} symbol_entry_t;
```

The symbol table is a hash table called `sy_hash`, created at initialization in the function `symbol_begin` called from `gas_init`.

Adding symbols into the symbol table is done with `symbol_table_insert`, the function `symbol_find` searches for a given symbol.

1.5.3 Fragments

As the assembler reads the source file, it stores the information it parses into `frags`. These structures contain two parts: a fixed part, i.e. the bytes that are known to be emitted, and a variable part, whose length can change. Memory is allocated with a worst case philosophy: the maximum size of the variable part is known, and will be used when reserving memory for each fragment.

When the source file has been fully parsed the relaxation process will go through all these fragments and build the linear structures needed in the object file.

Listing 1.5: struct frag

```
1  struct frag {
2      /* Object file address. This is zero until relaxation writes it. */
3      addressT fr_address;
4      /* When relaxing multiple times, remember the address the frag had in the last
5         relax pass. */
6      addressT last_fr_address;
7
8      /* (Fixed) number of octets we know we have. May be 0. */
9      valueT fr_fix;
10     /* May be used for (Variable) number of octets after above.
11     The generic frag handling code no longer makes any use of fr_var. */
12     offsetT fr_var;
13     /* For variable-length tail. */
14     offsetT fr_offset;
15     /* For variable-length tail. Points to the symbol to use according to fr_type */
16     symbolS *fr_symbol;
17     /* Points to opcode low addr byte, for relaxation. */
18     char *fr_opcode;
19
20     /* Chain forward; ascending address order. Rooted in frch_root. */
21     struct frag *fr_next;
22
23     /* Where the frag was created, or where it became a variant frag. */
24     const char *fr_file;
25     unsigned int fr_line;
26
27     /* A serial number for a sequence of frags having at most one alignment
28        or org frag, and that at the tail of the sequence. */
29     unsigned int region:16;
30
31     /* Flipped each relax pass so we can easily determine whether fr_address
32        has been adjusted. */
33     unsigned int relax_marker:1;
34
35     /* Used to ensure that all insns are emitted on proper address boundaries. */
36     unsigned int has_code:1;
37     unsigned int insn_addr:6;
38
39     /* This field indicates the interpretation of fr_offset, fr_symbol and the
40      * variable-length tail of the frag, as well as the treatment it gets in various
```

```
41      * phases of processing. It does not affect the initial fr_fix characters;
42      * they are always supposed to be output verbatim (fixups aside). */
43    relax_stateT fr_type;
44    /* This is used as a size for this frag and is updated during relaxation */
45    relax_substateT fr_subtype;
46
47    /* A simple structure (see below) that is used when creating a mapping symbol and
          when
48       checking if the mapping symbol is still useful. Mapiing symbols are create
             when
49       transitioning from a given mapping state to another */
50    struct riscv_frag_type tc_frag_data;
51
52    /* Data begins here. */
53    char fr_literal[1];
54 };
55
56 struct riscv_frag_type { symbolS *first_map_symbol, *last_map_symbol; };
```

1.5.4 Fixups

In many situations, the assembler can't finish a calculation because all data needed for it isn't available. For instance a symbol is yet unresolved, or the exact location for some instruction component is absent.

In those situations the assembler emits a `fixup`. This is nothing else than an instruction on how to patch the output later, when all the data is known.

Fixups are described in a structure called `fixS` that holds mainly following kinds of information:

- `next` The `fixS` structures are linked in a list.
- `fx_frag` The fragment where the fix should be applied.
- `fx_where` The position within that fragment where the fix should be applied.
- The quantity to be added or subtracted. If it is a symbol, a pointer to that symbol will be stored in the fields `fx_addsy` or `fx_subsy`. Otherwise, if it is just a number it will stored in the field `fx_offset`.
- `fx_size`. The size (in bytes) of the fixup, i.e. how many bytes should be written at the given location.
- There are many other fields that you can look up in the definition in asm.h. They are described in the comments surrounding their definition.

Constructors

Two functions build a fixup: `fix_new` and `fix_new_exp`. The second one is for a fixup referring to an expression, the first is for a symbol with an optional offset. They differ only in that `fix_new_exp` determines the symbol to add or subtract from the given expression. Both call `fix_new_internal` to do the actual fix.

Applying a fixup

A fixup is resolved by the function `md_apply_fix`. It uses the type of fixup to determine the sequence of actions to be performed: to fix the high 20 bits of a 32 bit address, or the lower 12, or add to a 64 bit address an addend, etc. The code consists (yes, you guessed it!) of a big `switch` statement with all the handled types of relocation existing for riscv machines, and it is not very difficult to follow.

1.5.5 Relocations

Sometimes a fixup can't be resolved. For instance this C code:

```
#include <stdio.h>
int main(void) {
    printf("hello\n");
}
```

gcc translates this to:

```
    .section   .rodata
.LC0:
    .string "hello"
    .text
main:
    /* irrelevant stuff ellided */
    lla a0,.LC0
    call   puts@plt
    /* further stuff ellided */
```

The address of the puts procedure can't be established by the assembler, nor the linker, only by the program loader that will know at load time the address of the shared library `libc6.so`. The assembler makes the same thing as when establishing a fixup. It makes a new fixup, this time for the linker, that will tell it where the address of the puts function needs to be stored.

This kind of fixup is called a *relocation*.

The linker can't resolve the address either, so it will make a relocation for the program loader, that will patch the code accordingly when the program starts[6].

Relocations, contrary to simple fixups have a standard format prescribed in the object file format, in our case ELF.

Listing 1.6: Elf relocation structure

```
typedef struct {
    unsigned char r_offset[8]; /* Location at which to apply the action */
    unsigned char r_info[8]; /* index and type of relocation */
    unsigned char r_addend[8]; /* Constant addend used to compute value */
} Elf64_External_Rela;
```

This format doesn't exactly correspond to the internal one used by the assembler. The function `bfd_elf64_swap_reloca_out` converts from the bfd format to the ELF one.

Data structures for relocations

The central structure for relocations is the `howto_table`, that describes in detail the types of relocation and how they are handled. It is a table of structures defined by the following type:

```
struct reloc_howto_struct {
    /* Contains the relocation type according to the riscv standard */
    unsigned int type;

    /* The size of the item to be relocated in bytes. */
    unsigned int size:4;

    /* The number of bits in the field to be relocated. This is used
```

[6] The process is obviously much more complicated. Here we leave all the details out, to take a high level view.

```
9     when doing overflow checking. */
10    unsigned int bitsize:7;
11
12    /* The value the final relocation is shifted right by. This drops
13    unwanted data from the relocation. */
14    unsigned int rightshift:6;
15
16    /* The bit position of the reloc value in the destination.
17    The relocated value is left shifted by this amount. */
18    unsigned int bitpos:6;
19
20    /* What type of overflow error should be checked for when
21    relocating. */
22    ENUM_BITFIELD (complain_overflow) complain_on_overflow:2;
23
24    /* The relocation value should be negated before applying. */
25    unsigned int negate:1;
26
27    /* The relocation is relative to the item being relocated. */
28    unsigned int pc_relative:1;
29
30    /* This field is true only in 3 relocation types that refer to thread storage.
31       They are: R_RISCV_TLS_DTPREL32, R_RISCV_TLS_DTPREL64, and
32       R_RISCV_TPREL_HI20  */
33    unsigned int partial_inplace:1;
```

The original comment above this field read like this

> Some formats record a relocation addend in the section contents rather than with the relocation.

This means that the place for the relocation can be filled either with zero bits or with a number to which the relocation is added.

> For ELF formats this is the distinction between USE_REL and USE_RELA (though the code checks for USE_REL == 1/0). The value of this field is TRUE if the addend is recorded with the section contents; when performing a partial link (ld -r) the section contents (the data) will be modified. The value of this field is FALSE if addends are recorded with the relocation (in arelent.addend); when performing a partial link the relocation will be modified. All relocations for all ELF USE_RELA targets should set this field to FALSE (values of TRUE should be looked on with suspicion). However, the converse is not true: not all relocations of all ELF USE_REL targets set this field to TRUE. Why this is so is peculiar to each particular target. For relocs that aren't used in partial links (e.g. GOT stuff) it doesn't matter what this is set to.

I have read many times that, trying to make sense of it. In tiny-asm, all relocations are `reloca`. Still, in the code this field is used, and some relocations, specifically the thread local storage relocations, do use this field.

```
34
35    /* When some formats create PC relative instructions, they leave
36    the value of the pc of the place being relocated in the offset
37    slot of the instruction, so that a PC relative relocation can
38    be made just by adding in an ordinary offset (e.g., sun3 a.out).
39    Some formats leave the displacement part of an instruction
40    empty (e.g., ELF); this flag signals the fact. */
41    unsigned int pcrel_offset:1;
```

```
42
43      /* This field is not used in tiny-asm */
44      unsigned int install_addend:1;
45
46      /* src_mask selects the part of the instruction (or data) to be used
47      in the relocation sum. If the target relocations don't have an
48      addend in the reloc, eg. ELF USE_REL, src_mask will normally equal
49      dst_mask to extract the addend from the section contents. If
50      relocations do have an addend in the reloc, eg. ELF USE_RELA, this
51      field should normally be zero. Non-zero values for ELF USE_RELA
52      targets should be viewed with suspicion as normally the value in
53      the dst_mask part of the section contents should be ignored. */
54      bfd_vma src_mask;
55
56      /* dst_mask selects which parts of the instruction (or data) are
57      replaced with a relocated value. */
58      bfd_vma dst_mask;
59
60      /* If this is non null, then, the supplied function is called rather than the
61      normal one. Under tiny-asm, this field is never NULL. Three functions are used.
62      See below for further expplanations.*/
63      bfd_reloc_status_type (*special_function)
64      (bfd *, arelent *, struct bfd_symbol *, void *, asection *,
65      bfd *, char **);
66
67      /* The textual name of the relocation type. */
68      const char *name;
69  };
```

The `special_function` field has 3 possible values:

1. `bfd_elf_generic_reloc`. This function adds the input section position if the symbol meets certain conditions:

```
1       if (output_bfd ≠ NULL && (symbol→flags & BSF_SECTION_SYM) == 0
2           && (!reloc_entry→howto→partial_inplace
3                   || reloc_entry→addend == 0)) {
4           // Section relative
5           reloc_entry→address += input_section→output_offset;
6           // Stop any processing, this relocation has been handled
7           return bfd_reloc_ok;
8       }
9
10      return bfd_reloc_continue; // Relocation not handled, continue processing
```

   The `output_bfd` variable is never NULL since it is always `stdoutput`.

2. `riscv_elf_add_sub_reloc`. This function does exactly the same tests that the preceding one, and returns exactly the same values. They differ in that this one has a lot of code for the case when the `output_bfd` is NULL, what could be the case in the linker, but not in the assembler.

3. `riscv_elf_ignore_reloc`. Beware of these explicit names! This function doesn't ignore the relocation at all, but does a

   `reloc_entry->address += input_section->output_offset;` *without* any tests at all. [7]

[7] After an enquiry in the group binutils, Alain Modra answered:

1.5.6 Sections and subsections

Assembled data falls into four sections: opcodes, initialized data, uninitialized data and debug information. You may have separate groups of data in those sections that you want to end up near to each other in the object file, even though they are not contiguous in the assembler source.

Tiny-asm allows you to use subsections for this purpose. Within each section, there can be numbered subsections with values from 0 to 8191 [8].

Objects assembled into the same subsection go into the object file together with other objects in the same subsection. For example, a compiler might want to store constants in the text section, but might not want to have them interspersed with the program being assembled. In this case, the compiler could issue a `.text 0` before each section of code being output, and a `.text 1` before each group of constants being output.

Subsections are optional. If you do not use subsections, everything goes in subsection number zero.

Each subsection is zero-padded up to a multiple of four bytes.

Subsections appear in your object file in numeric order, lowest numbered to highest. The object file contains no representation of subsections; ld, objdump and other programs that manipulate object files see no trace of them. They just see all your text subsections as a single text section, and all your data subsections as a data section.

To specify which subsection you want subsequent statements assembled into, use a numeric argument to specify it, in a `.text <number>` or a `.data <number>` statement. If you just say `.text` then an implicit zero is assumed. Likewise `.data` means `.data 0`.

In the source code, sometimes subsections are called "subsegments".

1.6 Instruction formats and encoding

Yes, there are several parts in an assembler, but there is a fundamental part that makes the purpose of the whole program: **encoding instructions**. The essential part is here: transforming ASCII text representing instructions into a series of 16 or 32 bit sequences that encode each operation that the machine can do, including operations that are seldom, if ever, used.

To understand how the assembler works, it is important to keep in mind how the machine works, the names of its parts, and the intricacies of instruction encoding. Yes, yes, that looks awfully dry and uninteresting. But (to me) it is interesting, and if you do not like to *understand* how things work, please go to tik-tok and play some games...

There are several types of instruction encoding, named **R**, **I**, **S**, **B**, **U**, **J**.

- All are 32 bits, like the ARM.
- The first 7 bits are reserved for the opcode (bits 0 to 6).
- The same operand, for instance the source register 1 (sr1) is at the same position, bits 15 to 19.
- All instructions have at least one register operand.
- Since we have 32 registers, all register encoding take 5 bits.

There is nothing wrong with the name. No relocation of section contents is done. The only change made here is for ld -r to keep the relocation associated with the same location in the output as it was in the input.

That said, I think it would be better for the `R_RISCV_SET_ULEB128`, and `R_RISCV_SUB_ULEB128`, howto to use a special function something like `ppc64_elf_unhandled_reloc`.

[8] This limit is mentioned in the GAS documentation. In the software, actually, there isn't a single test to enforce this limit, so you can write any number between 1 and MAX_INT.

The risc v introduces a more functional naming schema, where registers are assigned usage names, instead of the register numbers. Here is a correspondence table between them:

Table 1.1: RISCV symbolic register names

Register name	ABI name	Description	Register name	ABI name	Description
Integer registers					
x0	zero	Hard-wired zero	x16	a6	Seventh argument
x1	ra	Return Address	x17	a7	Eighth argument
x2	sp	Stack pointer	x18	s2	Saved 2
x3	gp	Global pointer	x19	s3	Saved 3
x4	tp	Thread Pointer	x20	s4	Saved 4
x5	t0	Temporary/Alternate link register	x21	s5	Saved 5
x6	t1	Temporary	x22	s6	Saved 6
x7	t2	Temporary	x23	s7	Saved 7
x8	fp/s0	Frame pointer	x24	s8	Saved 8
x9	s1	Saved 1	x25	s9	Saved 9
x10	a0	First argument / Return value	x26	s10	Saved 10
x11	a1	Second Argument / Return value	x27	s11	Saved 11
x12-x15	a2-a5	Argument 3-5	x28-x31	t3-t6	Temporary registers
Floating point registers					
f0-f7	ft0-ft7	Fp temps	f2-f7	fa2-fa7	function arguments
f8-f9	fs0-fs1	Fp saved registers	f18-f27	fs2-fs11	saved registers
f10-f11	fa0-fa1	Fp arguments/return value	f28-f31	ft8-ft11	Temporary registers

The difference between the ABI names and the actual register numbers is due to the fact that the ranges of registers are not contiguous. For instance the range of saved registers has two of them as x8 and x9, then the rest is x18 to x27.

1.7 The instruction formats

Each format is designed to be used by similar type of instructions.

- **R** Register to register ALU instructions.
- **I** Immediate and load.
- **S** Store and comparisons.
- **B** Branch.
- **U J** Jump and jump with link (call) instructions.

The RISC-V manual comments these formats like this

> The RISC-V ISA keeps the source (rs1 and rs2) and destination (rd) registers at the same position in all formats to simplify decoding. Except for the 5-bit immediates used in CSR instructions, immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit

for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry. [9]

All instructions are 32 bits. This requirement, that riscv shares with ARM and other machines, is necessary to make possible parallel decoding of instructions. In the x86, for instance, each instruction is of variable length, what makes parallel decoding of instructions an incredibly difficult undertaking.

The `C` extension compresses 32 bit instructions into 16 bits, what makes for more compact code. They expand into 32 bits instructions.

Table 1.2: The different instruction formats

"R" format:
struct rFormat {
unsigned extra7:7;
unsigned src2:5;
unsigned src1:5;
unsigned extra3:3;
unsigned dst:5;
unsigned opcode:7;
};
	"I" format
	struct iFormat {
	unsigned imm12:12;
	unsigned src1:5;
	unsigned extra3:3;
	unsigned dst:3;
	unsigned opcode:7;
	};
		"U" format
		struct uFormat {
		unsigned imm20:20;
		unsigned dst:5;
		unsigned opcode:7;
		};

"S" format
struct sFormat {
unsigned imm12_2:7;
unsigned src2:5;
unsigned src1:5,
unsigned extra3:3;
unsigned imm12_1:5;
unsigned opcode:7;
};
	"B" format
	struct bFormat {
	unsigned imm12_sign:1;
	unsigned imm12_10_5:6;
	unsigned src2:5;
	unsigned src1:5;
	unsigned extra3:3;
	unsigned imm12_1_4:4;
	unsigned imm12_11:1;
	unsigned opcode:7;
	};
		"J" format
		struct jFormat {
		unsigned imm12_sign:1;
		unsigned imm12_1_10:10;
		unsigned imm12_11:1;
		unsigned imm12_12_19:7;
		unsigned dst:5;
		unsigned opcode:7;
		};

1.7.1 The "R" format

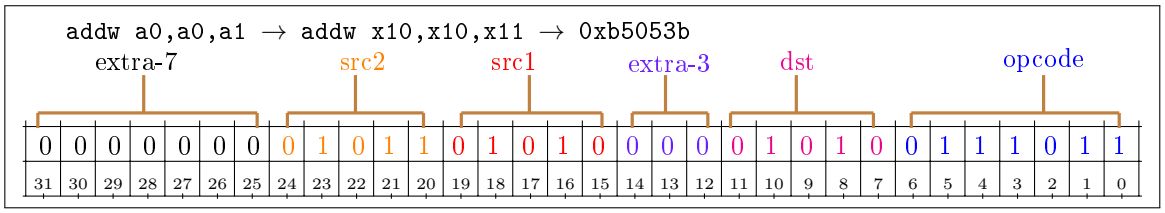
This format features 3 registers (destination, source 1 and source 2) and has two fields of 3 and seven bits available for use to customize the opcodes. We use a 32 bit addition as an example of this format: `addw a0,a0,a1`. The addition using ABI names is `addw a0,a0,a1` but using actual register numbers we have `addw x10,x10,x11`. For this instruction the 10 bits of `extra-3` and `extra-7` are empty.

We have then:

- Opcode: 0 1 1 1 0 1 1 → 0x3b (59 decimal).
- Destination register: 0 1 0 1 0 → 0xA (10 decimal). Register 10 is `a0`, that contains the first argument and is loaded with the result.

[9] RISC-V User level ISA V 2.2 §2.2. They add further down: Decoding register specifiers is usually on the critical paths in implementations, and so the instruction format was chosen to keep all register specifiers at the same position in all formats at the expense of having to move immediate bits across formats

Figure 1.4: **R** Instruction layout



- Source 1: 0 1 0 1 0 → 0xA (10 decimal). Register 10 (a0) is the first source.
- Source 2: 0 1 0 1 1 → 0xB (11 decimal). Register 11 (a1) is the second source.

Software handling

We have an instruction with the `args` format of `"Cs,Cw,Ct"` that expects source and destination to be identical (`s` and `w`) followed by a target register in the expected range for compressed registers. All of that is true, and we succeed with a compressed 16 bit instruction.

Obviously this is not what we wanted. We wanted a 32 bit 'R' instruction. To be able to do that, we add the following instruction at the top of our assembler file

```
.option arch -c
```

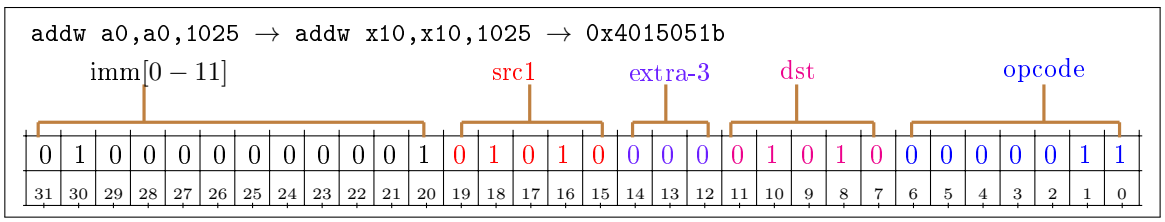
I.e. we disable all compressed instructions.

We see here that the *order* in the layout of the opcode table is very important. The instructions that are **more** constrained should come first, and the general formats should come last. For instance the compressed instruction should come first, and non-compressed last, since the software stops at the first match.

1.7.2 The "I" format

This format changes the `"R"` format by merging `src2` with `extra-7` to give a 12 bit field where an immediate integer value caan be stored (up to $2^{12}-1 \rightarrow 4095$ values can be stored).

Figure 1.5: **I** Instruction layout



Software handling

The first instruction that the software tries has its `args` string: `"Cs,Cw,Ct"`, we expect a source register in compressed format, i.e. register 8-15, followed by the *same* register. The second condition succeeds, and the software passes to the third argument: we expect a register, and we find the constant 1025. Nope, this instruction is not the one.

The next `addw` instruction to be tested has the string `"Cs,Ct,Cw"`, a permutation of the above that fails also, for the same reasons.

More instructions are tried, with strings `d,Cu,Co` that fails, `"d,s,t"` that fails also since we have an immediate constant and not a register in the third position (`'t'` field). At last we arrive at an instruction with `args` field of `d,s,j"`, i.e. a sign extended immediate (`'j'`) in the third position. This time the software succeeds and we are done. Accessing the different fields is done with macros. Here is one example of a series of macros that extracts the immediate field of the immediate value in the instruction above

```
#define RV_X(x, s, n)  (((x) >> (s)) & ((1 << (n)) - 1))
```

This macro extracts <n> bits from <x>, beginning in bit position <s>. It has two parts:

1. The left side of the "and" operation that shifts the given number <s> bits to the right to bring it to position zero, and

2. An expression that builds a mask of <n> 1 bits by shifting a 1 <n> positions to the right and subtracting one, what gives a power of two minus 1. A power of two minus 1 is a field full of ON bits in two's complement notation. For instance `1 << 3` $\rightarrow$ 8 (1000). You subtract 1 from that and you obtain 0111 (7), i.e. 3 bits "on", a mask to extract the lower 3 bits from a number.

```
#define RV_IMM_SIGN(x) (-(((x) >> 31) & 1))
```

This macro returns either -1 or 0, depending if the sign of the 32 bit number is negative or positive. Since -1 is 32 bits of "1" bits, it can be used to sign extend a number.

The two macros above are used in these new ones:

```
#define EXTRACT_ITYPE_IMM(x) (RV_X(x,20,12)|(RV_IMM_SIGN(x) << 12))
#define ENCODE_ITYPE_IMM(x) (RV_X(x, 0, 12) << 20)
```

The first macro extracts 12 bits from the given number (<x>) and sign-extends its sign. The second extracts the lower 12 bits of the value, and puts them at position 20-31 [10]

1.7.3 The "U" format

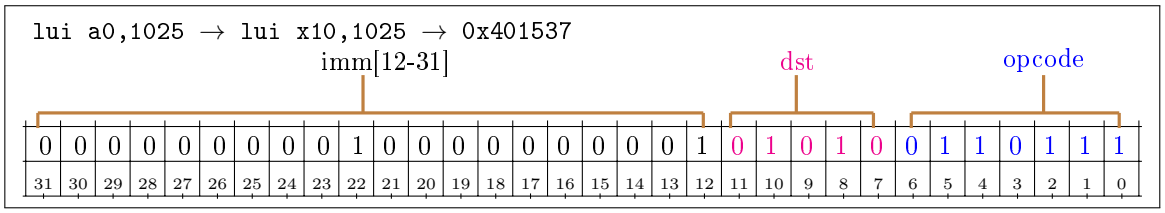
A variant of the **I** format featuring more space for immediate constants is the **U** format, that can hold immediate constants with 20 bits.

The `lui`[11]. instruction loads an unsigned 20 bits immediate stored in the bits 12 to 31 of the instruction into the upper 20 bits of the destination and sets the lower 12 bits to zero. In C language notation we have: `dst = (imm20 << 12);`. The authors justify these choices with:

[10] It is a pity that machines implementing the boolean extension aren't widely available yet. I miss the ARM boolean instructions that will reduce many of those macros to a couple of instructions.

[11] `lui` stands for **l**oad **u**pper **i**mmediate

Figure 1.6: **U** Instruction layout



> In practice, most immediates are either small or require all XLEN bits. We chose an asymmetric immediate split (12 bits in regular instructions plus a special load upper immediate instruction with 20 bits) to increase the opcode space available for regular instructions.Immediates are sign-extended because we did not observe a benefit to using zero-extension for some immediates as in the MIPS ISA and wanted to keep the ISA as simple as possible.[12]

Software handling

Looking up the `args` description for this instruction, we find the character string `"d,Cu"`. This means we should expect a register name, followed by a comma, and an immediate value to be able to use a `C` (compressed) instruction. But that doesn't work, our constant is beyond bounds of the compressed immediate.

The software continues its search for the correct instruction and we come to the next instruction in the list that has the `args` string `"d,u"`, without any compression requirements. This time a match is found, and necessary bits are inserted as shown in figure 1.6 page 32.

Obviously, loading an immediate constant that will be shifted by12 bits is seldom used. This is thought for loading the upper 20 bits of an *address*, then adding the lower 12 bits with another instruction. This constant was choosen in this example so that it has a 1 bit at the end of 10 bits, and 1 at the start to be visible in the drawing.

To extract the J type immediate we use the following macro:

```
1    #define EXTRACT_UTYPE_IMM(x) ((RV_X(x, 12, 20) << 12) | (RV_IMM_SIGN(x) << 32))
```

1.7.4 The "S" format

In this format, the `dst` field disappears and its bits are used to hold the lower 4 bits of an immediate value. An instruction that uses this format is the `sd` (store double word) instruction.

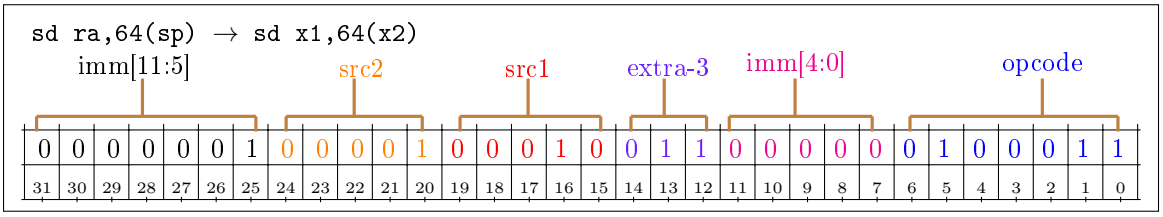
We use the instruction `sd ra,64(sp)` as example. This instruction means: Store the contents of the return address register `(ra)` at the memory address obtained by adding 64 to the contents of the sp register. We have here an address that is obtained by adding the contents of a register and a *displacement* that must fit into 12 bit. As you can see here, this is a much easier format than the ARM jungle of different types of offsets where you never really know which one to use. The Risc-V manual specifies that all offsets are signed.[13]

[12] Riscv ISA Architecture §2.2

[13] They say:

> Except for the 5-bit immediates used in CSR instructions, immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry.

Figure 1.7: **S** Instruction layout



We have then for this instruction:

- `src1` is 0 0 0 1 0, or register 2.
- `src2` is 0 0 0 0 1, or register 1.
- The immediate is the concatenation of `imm[4:0]` and `imm[11:5]` i.e; 0 0 0 0 0 0 1 0 0 0 0 0 0 or 64.

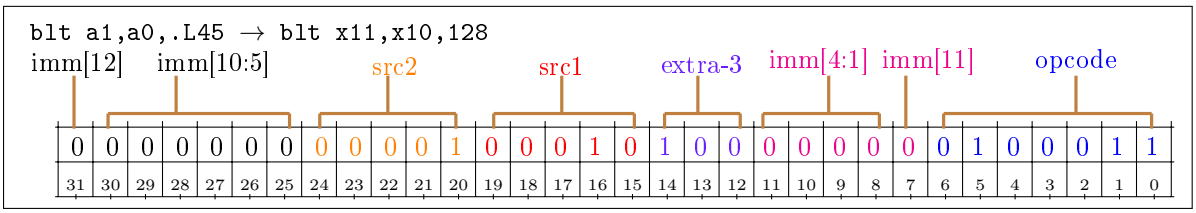
For extracting the immediate from the instruction we use the macro

```
1  #define EXTRACT_STYPE_IMM(x) \
2  (RV_X(x, 7, 5) | (RV_X(x, 25, 7) << 5) | (RV_IMM_SIGN(x) << 12))
```

This macro extracts five bits beginning at position seven, then 7 bits from position 25 upwards, shifted by 5 left, so that they come right after the first five. The whole is sign extended in the same way as explained in section 1.7.2 page 31.

1.7.5 The "B" format

Figure 1.8: **B** Instruction layout



In this format, we have a 13 bit immediate for branches. The immediate represents the amount that will be added to the program counter to reach the specified location, in multiples of 2. Since the lowest bit of the immediate will be always zero, it has been replaced by bit 11 (the twelfth bit) adding one bit to the quantity being written. The range of the branch is ± 4K.

The different conditional branches are specified in the `extra-3` group, with

Table 1.3: Encoding of conditional branches

extra-3	Instruction	Description
0 0 0	beq	branch if equal

Table 1.3: Encoding of conditional branches

0 0 1	bne	branch if different
1 0 0	blt	branch if less than
1 0 1	bge	branch if greater/equal
1 1 0	bltu	branch if less than unsigned
1 1 1	bgeu	branch if greater equal unsigned

All these instructions share the same opcode: 99. The `extra-3` field is used to extend the opcode for different instructions.

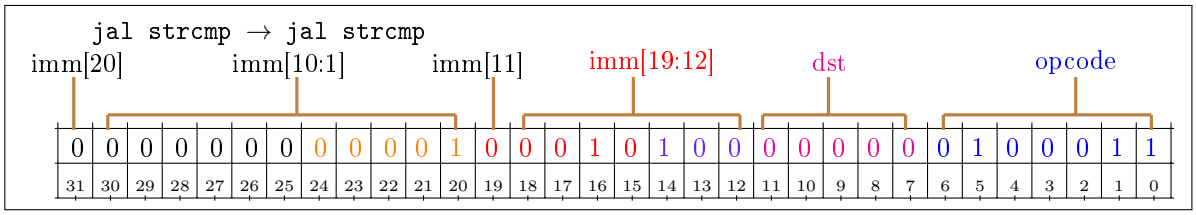
The macro to access the immediate value is way more complicated due to the bit scrambling...

```
#define EXTRACT_BTYPE_IMM(x) ((RV_X(x, 8, 4) << 1) | \
(RV_X(x, 25, 6) << 5) | (RV_X(x, 7, 1) << 11) | (RV_IMM_SIGN(x) << 12))
```

1.7.6 The "J" format

The only difference between the U and J formats is that the 20-bit immediate is shifted left by 12 bits to form U immediates and by 1 bit to form J immediates. In the "J" format, the immediate represents an offset in pairs of 16 bit instructions from the current PC.

Figure 1.9: **J** Instruction layout



Why this scrambled layout? Citing the Risc-v manual:

> Although more complex implementations might have separate adders for branch and jump calculations and so would not benefit from keeping the location of immediate bits constant across types of instruction, we wanted to reduce the hardware cost of the simplest implementations. By rotating bits in the instruction encoding of B and J immediates instead of using dynamic hardware muxes to multiply the immediate by 2, we reduce instruction signal fanout and immediate mux costs by around a factor of 2.
>
> The scrambled immediate encoding will add negligible time to static or ahead-of-time compilation. For dynamic generation of instructions, there is some small additional overhead, but the most common short forward branches have straightforward immediate encodings.

The macro to extract this monster from its hiding place looks like this

```
#define EXTRACT_JTYPE_IMM(x) ((RV_X(x, 21, 10) << 1)|(RV_X(x, 20, 1) << 11) | \
(RV_X(x, 12, 8) << 12) | (RV_IMM_SIGN(x) << 20))
#define ENCODE_JTYPE_IMM(x) ((RV_X(x, 1, 10) << 21)|(RV_X(x, 11, 1) << 20) | \
(RV_X(x, 12, 8) << 12) | (RV_X(x, 20, 1) << 31))
```

1.8 The compressed instructions

The Risc-v instructions are normally 32 bits in length. The "C" extension (C for **C**ompressed) encodes certain instructions in 16 bits, what leads to big savings in code size. These instructions aren't enabled by default in the assembler. You can enable them (if your machine actually supports them) with the instruction: `.option arch, +c`. Enabling them or not is not that important, since the linker will replace longer with shorter instruction whenever possible. For instance the jumps can't be really calculated until all the instructions are compressed, what only the linker can know.

The compressed instructions are enabled when one of these conditions is true:

- The compressed 16 bit instructions have the lowest 2 bits of the opcode set to either 00, 01, or 10.
- 32 bits instructions have their lowest two bits set to 11. The following 3 bits should have any value different from 111.
- The 48 bit instructions have their lowest 6 bits set to 011111. (5 bits set)
- 64 bit instructions have the 7 lower bits set to 0111111. (6 bits set)

The criteria for making a compressed instruction are as follows:

- The immediate or the address offset is small.
- One of the registers used is the `zero` register (x0), the return address register or link register `ra` (x1), or the stack pointer `sp` (x2).
- The destination and first source register are the same.
- The registers used belong to the 8 most popular ones, described with 3 bits in the table below[14].

Table 1.4: Compressed register numbers

number	000	001	010	011	100	101	110	111
int reg. number	x8	x9	x10	x11	x12	x13	x14	x15
ABI name	s0	s1	a0	a1	a2	a3	a4	a5
FP reg number	f8	f9	f10	f11	f12	f13	f14	f15
FP ABI name	fs0	fs1	fa0	fa1	fa2	fa3	fa4	fa5

There are nine different compressed instruction layouts.
In the table below the registers that use the 3 bit number are marked with a '.

Table 1.5: Compressed formats

Meaning	Code	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register	CR	Extra-4				dst/src1					src2					op	
Immediate	CI	Extra-3			S	rd/rs1					immediate					op	
Store local	CSS	Extra-3			imm						rs2					op	
Wide imm	CIW	Extra-3			imm								rd'			op	
Load	CL	Extra-3			imm			rs1'			imm		rd'			op	
Store	CS	Extra-3			imm			rs1'			imm		rs2'			op	
Arithmetic	CA	Extra-6						rd'/rs1'			Extra-2		rs2'			op	

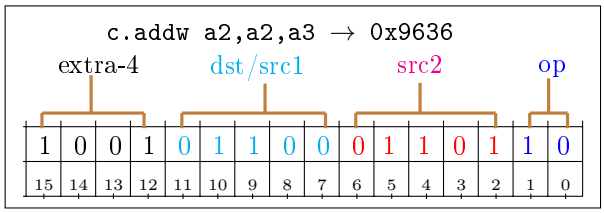
[14] Actually those numbers are just the normal register number modulo 8.

Table 1.5: Compressed formats

Branch	CB	Extra-3	offset	rs1'	offset	op
Jump	CJ	Extra-3	jump target			op

1.8.1 The compressed register (CR) format

Figure 1.10: Compressed **CR** Instruction layout



This format acccepts instructions where the destination and the first source register are the same. It has four fields, here from right to left, i.e. from bit 0 to 15:

1. OP: Bits 0-1. Value: 2.
2. Src2: Bits 2-6. The second source register. Note that it is specified in 5 bits, like dst/src1, so any register of the set of 32 is possible, except the zero register. In this case it is 13, i.e. register a3 (x13).
3. dst / src1: Bits 7-11. The source 1 and the destination register are the same. Also specified in 5 bits, in this case it is 12: the a2 (x12) register.
4. Extra-4: Bits 12-15. Value: 9. Complements the opcode. This field can have two values that correspond to `mv` (move) or, in the example, `add`.

The software side

The argument description for `addw,a2,a2,a3` is the character string `Cs,Cw,Ct`. The first argument is a compressed format source register (Cs), followed by a compressed format register that should be equal to the preceding one (Cw), followed by a compressed format second source register, (Ct).

The code for the 's' case in `riscv_ip` is as follows:

```
1    case 's':  /* RS1 x8-x15. */
2        if (!reg_lookup(&asarg,RCLASS_GPR,&regno)
3            || !(regno ≥ 8 && regno ≤ 15))
4        break;
5        INSERT_OPERAND(CRS1S,*ip,regno % 8);
6        continue;
```

It is a typical sample of the code in the encoder (`riscv_ip`). We search for a register name with `reg_lookup` and we ensure that is between 8 and 15. If that is not the case, the matching process for this instruction candidate fails, and we look for the next one (`break`).

If it is, we insert the operand in the right position and continue with this candidate.

Note that the identifier `CRS1S` doesn't appear in ANY macro, variable or enumeration in the whole program.

It is a literal name argument! When we look at the definition of `INSERT_OPERAND` we find:

```
#define INSERT_OPERAND(FIELD,INSN,VALUE) \
  INSERT_BITS ((INSN).insn_opcode,VALUE,OP_MASK_##FIELD,OP_SH_##FIELD)
```

The `##` operand before the `FIELD` macro argument makes the preprocessor convert it to `OP_MASK_CRS1S` what is defined with `#define OP_MASK_CRS1S 0x7` in asm.h.

The first level expansion converts this to:

```
#define INSERT_OPERAND(FIELD,INSN,VALUE) \
   INSERT_BITS ((INSN).insn_opcode,VALUE,OP_MASK_CRS1S,OP_SH_CRS1S)
```

The `INSERT_BITS` macro is defined as follows:

```
#define INSERT_BITS(STRUCT, VALUE, MASK, SHIFT) \
      (STRUCT) = (((STRUCT) & ~((insn_t)(MASK) << (SHIFT))) \
         | ((insn_t)((VALUE) & (MASK)) << (SHIFT)))
```

This macro has two parts, separated by an | (or) sign:

```
((STRUCT) & ~((insn_t)(MASK) << (SHIFT))) and
((insn_t)((VALUE) & (MASK)) << (SHIFT)
```

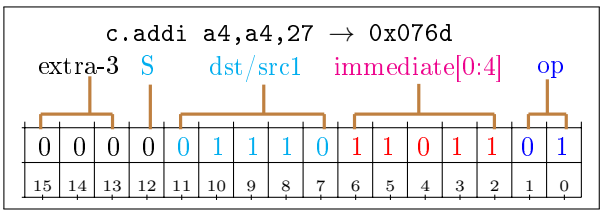
In the first one we set to zero all bits in the field that will be written. The second one introduces the bits into the right position. The *or* operation joins those parts into a single value.

The encoder works like an interpreter for a "language" of single letters that represent pieces of instruction fields. They indicate what to expect at the given position. Its actions can be only be "break" (discard the current candidate) or insert the correct bits and "continue" with it.

1.8.2 The compressed immediate (CI) format

These instructions perform operations between a register and a small immediate encoded in only 6 bits. The register can't be the zero register, and the immediate can't be zero. There

Figure 1.11: Compressed immediate **CI** Instruction layout



are four instructions that use the compressed immediate format. They differ in the `extra-3` field. From least significant bit to the most significant one we have:

1. OP: Bits 0-1, always with value 1 for the CI format.
2. The immediate field, in bits 2 to 6 that encodes immediate bits 0 to 4. In the example above this is 27, 1 1 0 1 1 in binary.
3. The destination and the source register number over 5 bits. In the example we have 14 since the register a4 has the number 14.
4. The sign of the immediate value in a single bit (index 12th).

5. The `Extra-3` field, that allows for 3 instructions to be distinguished: `addi, addiw,` and `addi16sp`. The last one adds a number of 16 bits quantities to the stack and is used to adjust the stack at the prologue or at the epilogue of a function. Since the stack must be aligned to a multiple of 16, there is no need to keep the lower 4 bits. This makes for adjustments of -512 to 496 bytes.

To access the immediate value we use

```
1    #define EXTRACT_CITYPE_IMM(x) (RV_X(x, 2, 5) | (-RV_X(x, 12, 1) << 5))
2    #define ENCODE_CITYPE_IMM(x) ((RV_X(x, 0, 5) << 2) | (RV_X(x, 5, 1) << 12))
```

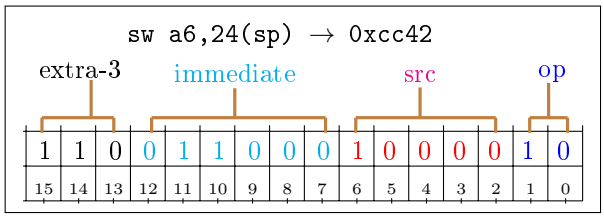
The first macro uses the same technique for sign extending that our `RV_IMM_SIGN` uses (see 1.7.2 page 31). We just need another expression since the other was fixed for 32 bits.

1.8.3 The stack relative store (CSS) format

Table 1.6: Compressed store instructions with the CSS format

Syntax	Operation
`c.swsp rs2 (uimm6)(sp)`	Store word to an offset from sp. $mem[sp + (uimm6 << 2)] \leftarrow rs2[0..31]$
`c.sdsp rs2 (uimm6)(sp)`	Store double word to an offset from sp. $mem[sp + (uimm6 << 3)] \leftarrow rs2[0..63]$
`c.fswsp fs2 (uimm6)(sp)`	Store single precision to an offset from sp. $mem[sp + ()uimm6 << 2)] \leftarrow fs2[0..31]$
`c.fsdsp fs2 (uimm6)(sp)`	Store double precision to an offset from sp. $mem[sp + (uimm6 << 3)] \leftarrow fs2[0..63]$

Figure 1.12: Store to stack offset (**CSS**) instructions layout



In our example instruction we have an `op` field of 2, an `src` field of 16 (10000) and the cryptic "011000" sequence that is translated into 00110 (6 decimal) since the bits are scrambled: they are stored as bits 5 4 3 2 7 6 The macros to access the immediate displacement here are:

```
1    #define EXTRACT_CSSTYPE_IMM(x) (RV_X(x, 7, 6) << 0)
2    #define ENCODE_CSSTYPE_IMM(x) (RV_X(x, 0, 6) << 7)
```

The encoding of instruction `c.swsp` needs only one source register: the source of the 32 bit data to store in memory. Any register will do since we have a register number in 5 bits. The value of the immediate displacement will be added to the stack pointer scaled by 4 to form the effective address. In the example above the 6 binary is scaled to 24. [15]

[15] By an unfortunate coincidence the scrambled bits of the constant are 011000, what is 24 in binary. Beware, nothing in this business is simple, and a 24 can be scrambled to 6, then scaled to 24 back again.

The argument description string is `"CV,CM(Cc)"`: We need a register name (`CV`), followed by a small constant (`CM`) that is a displacement (the parentheses) of the stack pointer (`Cc`). The constant value will be zero extended, since obviously negative offsets for the stack aren't very useful!

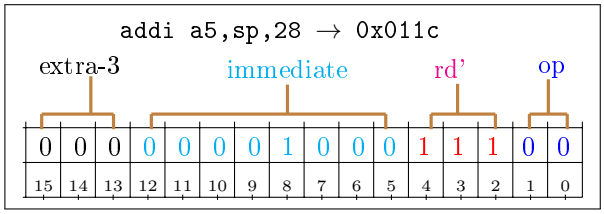
The reach of this instruction is $2^7 - 1$ values since we have 7 bits. Scaled by 4, i.e. 127 * 4 → 508.

And... "one more thing" as Steve Jobs liked to say, there is a problem with zero offsets from the stack pointer. Normally a zero offset is omitted, i.e. you do NOT write `sw a6,0(sp)`, you just write `sw a6,(sp)`. The handling of the `CM` directive tests for this with the function `riscv_handle_implicit_zero_offset`.

1.8.4 The wide immediate (CIW) format

This format is used to encode a constant in bits 5 to 12. It is used in the `addi4spn` instruction. The constant encoded in those 8 bits is scaled by 4, i.e. the two lower bits are implicit zeroes. The scaled value will be added to the stack pointer and written to the register whose index is stored in the 3 bits `rd'`. This instruction builds then pointers to values stored in the local stack frame.

Figure 1.13: Store to stack offset (**CIW**) instructions layout



1. The OP field is zero.

2. The destination (`rd'`) is 7, the register number in 3 bits of the a5 register

3. Now, this is more complicated to explain. The poor immediate bits are *scrambled*, i.e. they are **not** in the natural order but in the order: 5, 4, 9, 8, 7, 6, 2, 3. The bits 1 and 0 are implicitly zero. The quantity (128) has a single bit on at the position 7, what in our scrambled layout corresponds to bit 8. [16]. The Risc-V ISA manual justifies this saying:

   > The immediate fields are scrambled in the instruction formats instead of in sequential order so that as many bits as possible are in the same position in every instruction, thereby simplifying implementations.[17]

   The "simpliying" above refers to hardware simplification.

4. The Extra-3 field is zero.

The macros used to access the immediate are:

[16] The number 128 is 1000 0000 in binary. Bit 7 is one. In the scrambled order we have bit 7 in the fourth position of the immediate field, counting from left to right, as shown in the figure 1.13

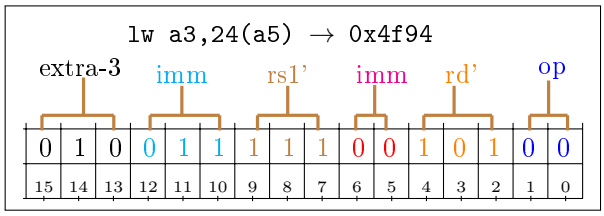
[17] Risc-V Unprivileged ISA V20191213 §16.2

```
#define EXTRACT_CIWTYPE_ADDI4SPN_IMM(x) ((RV_X(x, 6, 1) << 2) |\
(RV_X(x, 5, 1) << 3) | (RV_X(x, 11, 2) << 4) | (RV_X(x, 7, 4) << 6))
#define ENCODE_CIWTYPE_ADDI4SPN_IMM(x) ((RV_X(x, 2, 1) << 6) |\
(RV_X(x, 3, 1) << 5) | (RV_X(x, 4, 2) << 11) | (RV_X(x, 6, 4) << 7))
```

The argument description string for this instruction is `"Ct,Cc,CK"`

1.8.5 The compressed load (CL) format

Figure 1.14: Compressed load **CL** Instruction layout



1. The OP field is zero.
2. The destination register is 5 (a3).[18]
3. This field corresponds to an offset from a register. The constant should be aligned by a multiple of 4, since we are loading 4 bytes. The two lower bits then should be zero and they are implicit, i.e. they are absent from the encoding. The value is split between two bits at positions 5 and 6, and the rest in positions 10, 11, and 12. The two bits in positions 5 and 6 are scrambled, and bit 6 corresponds to bit 2 of the immediate and bit 5 is bit 6 of the immediate value, they are not consecutive.
4. The `rs1'` field contains 1 1 1, what corresponds to x15 (a5).
5. We have in bits 10, 11, and 12 the bits 3, 4, and 5 of the immediate value.
6. The `extra-3` field contains constant 2.

1.9 The opcode table

The full table of opcodes (called `riscv_opcodes`) consists of entries with the following structure:

```
struct riscv_opcode {
const char *name;
```

The name of the instruction in lower case. This is also the used as the key to the hash table. Several instructions can share the same name, and they are recognized by their different arguments.

```
unsigned xlen_requirement;
```

The word bit length (32, 64, or 128) that is required to use this instruction. A zero here means no requirement.

[18] These values are in table 1.4

```
enum riscv_insn_class insn_class;
```

The instruction class to which it belongs. For instance the instructions belonging to the basic integer operations are `INSN_CLASS_I` one of the member of the `enum riscv_insn_class`. This was used to decide whether or not this instruction is legal in the current machine architecture context, but this test has been dropped since we assume that the compiler will not generate instructions that are illegal for the target machine.

```
const char *args;
```

A string describing the arguments for this instruction. This string will be interpreted by the `riscv_ip` function in a rather big set of nested `switch` statements.

```
insn_t match;
insn_t mask;
```

The basic opcode for the instruction. When assembling, this opcode is modified by the arguments to produce the actual instruction that is used. If `pinfo` is `INSN_MACRO`, then this is 0. Otherwise the `mask` field is a bit mask used to isolate the relevant portions of the opcode when disassembling. If `pinfo` is `INSN_MACRO` then this field contains the macro identifier, encoded as a member of an anonymous enumeration and casted to an integer.

```
int (*match_func) (const struct riscv_opcode *op, insn_t word);
```

A function to determine if a word corresponds to this instruction. Usually, this computes `((word & mask) == match)`.

```
unsigned long pinfo;
```

Additional information about the instruction. They are:

Table 1.7: Opcode flags

Symbol	Description
`INSN_ALIAS`	Just an alias, for example "`mv`" for "`addi dest,src,zero`
`INSN_BRANCH`	Unconditional branch
`INSN_CONDBRANCH`	Conditional branch
`INSN_JSR`	Jump to a subroutine
`INSN_DREF`	Data reference
`INSN_V_EEW64`	Instruction allowed only when the machine is a 64 bit machine or more
`INSN_XX_BYTE`	5 different data size specifiers, for XX=1, 2, 4, 8, or 16 bytes

```
};
```

The field `args` above needs more explanation. It is a one (or more) letters that represent the type of argument that can be expected in an instruction. This can be a register, a constant within a certain range, or other things. During assembly, the assembler reads and interprets this character string to weed out wrong choices or emit warnings, and to verify that all constrains are met.

The table below should document all the letters used by the `riscv_ip` function. They are listed in the order they appear there; only for the first level. If a letter has a continuation (for instance for the compressed instructions), the secondary switch statement is explained in another table[19].

[19] Nested tables are as difficult to read as nested `switch` statements.

Table 1.8: Opcode arguments letters

Character	Argument expected
\0	End of the argument string. Here are done the final checks, for instance that this instruction corresponds to the bit length of the machine (64 bit instructions can't be done in a 32 bit machine). It checks also if the end of the argument string coincides with the end of the actual arguments present. If everything goes well it sets the errors to zero and branchs to the end of the `riscv_ip` function.
,	Synchronization. Arguments are separated by commas. The software tests this and ignores the separators.
() []	Displacement or index. Same behavior as for commas.
0	Expects a zero displacement. For instance: `lr.w a5,0(sp)`.
1	Used for thread local storage.
<	Shift amount for shifts less than 32.
>	Shift amount for 0 to word length - 1. Normally 63.
A	Requests a symbol
a	20 bit relative offset.
B	Requests a symbol or a constant.
C	Compressed format instructions. This leads to a nested switch statement, since all the compressed argument descriptions begin with a `C` letter. This switch is described in table 1.10 page 44.
c	Call using the global object table
D	Floating point destination register
d	Destination register.
E	Control register number. This is used only in privileged instructions.
F	Expects a bit field, that is defined by the following character. Used in the `.inst` directive only.
I	`M_LI` macro. Immediate value.
j	Sign extended immediate.
m	Rounding mode. This argument expects a character string representing the rounding mode. It can be one of `"rne","rtz", "rdn","rup", "rmm", "dyn"`. See table 1.9 page 43.
O	Opcode field
o	Expects a load/store displacement.
P	Fence predecessor
p	PC relative offset
Q	Fence successor
q	Expects a register store displacement.
R	Floating point RS3 for `.insn` directive.
r	RS3. Integer register for the `.insn` directive.
S	Floating point source 1.
s	First source register. Also called src1 in the documentation.
T	Floating point source 2
t	Second source register. The `'t'` is for target. It is also called src2 in the documentation.
U	Floating point source.
u	Expects a 20 bit immediate
V	Vector instructions. This leads to a nested switch statement.

Table 1.8: Opcode arguments letters

Character	Argument expected
y	Expects a `bs` immediate used in the cryptography sm4 instructions `sm4es` and `sm4ks`. It is a 2 bit constant that tells the sm4 algoritrhm which byte to choose: it represents the number of bytes to shift right `rs2`, selecting thus a single byte. See §1.36 page 106.
Y	Expects an `rnum` immediate. This is a constant that appears in the instruction `aes64ks1i`. It is used only in that instruction, and should be $0 \leq rnum \leq 10$.
W	Expects an offset for the prefetch instruction. The offset should have the 5 lower bits at zero. Followed by letters "if".
X	Integer immediate
Xu	eXtract unsigned n bits starting at position m. These arguments look like this: `Xu2@25`, meaning e**X**tract 2 bits starting at bit 25.
Z	Expects a CSR number, a CSRRxI Immediate. **C**ontrol and **S**tatus **R**egisters are specified in a different instruction format. For this to work, you have to have access to a CPU with the 'z' extension.
z	Expects a zero

Below is the set of rounding modes for the `m` parameter. It has been taken from the Sifive site[20]. Edited in May 27th 2020.

Table 1.9: Accepted rounding modes for the 'm' parameter

Binary Value	Mnemonic	Meaning
000	rne	Round to Nearest, ties to Even
001	rtz	Round towards Zero
010	rdn	Round Down (towards $-\infty$)
011	rup	Round Up (towards $+\infty$)
100	rmm	Round to Nearest, ties to Max Magnitude
101		Invalid. Reserved for future use.
110		Invalid. Reserved for future use.
111	dyn	In instruction's rm field, selects dynamic rounding mode; In Rounding Mode register, Invalid.

These rounding modes are recognized in the assembler using the `riscv_rm` table, a simple table of 8 character strings.

The C (compressed) instructions are differentiated by the following letters:

Table 1.10: Compressed instruction types

Char	Description
5	Five bit field
6	Six bit numeric field
8	Eight bit field
a	Jump. Expects 20 bit PC relative offset
c	Source 1 constrained to be sp

[20] sifive-blog
The URL seems truncated but it is not...

Table 1.10: Compressed instruction types

D	Floating point source 2
F	Field of 6, 4, 3, or 2 bits
U	Source 1 and destination the same.
j	Non-zero immediate of 6 bits
k	Immediate (possibly zero)
K	scaled by 4 stack addend
l	Load immediate (64 bits)
L	Stack offset scaled by 16
m	Load immediate
n	Immediate offset from SP
M	Scaled by 4 stack displacement(32 bits store)
N	Data reference with offset from stack scaled by 4(64 bits store)
o	`C.addiw, c.li,` and `c.andi` allow zero immediate. C.addi allows zero immediate as hint. Otherwise this is same as 'j'.
s	Source register 1
S	Floating point source 1
t	Integer register source 2
T	Floating point source 2
u	Immediate for jumps
V	Second source integer register rs2
v	Compressed I type immediate
x	Source 2 and destination are the same.
w	Source 1 constrained to be equal to the destination.
z	Source 2 should be the zero register
>	Shift amount between 0 and word length - 1

This is an example for an instruction entry in the opcodes table:

```
{"addi",0,INSN_CLASS_C,"Ct,Cc,CK",MATCH_C_ADDI4SPN,MASK_C_ADDI4SPN,\
match_c_addi4spn,INSN_ALIAS},
```

After parsing the name of the instruction, the `riscv_ip` function examines entries in the opcode table starting with the first one that has this name. It copies this entry into temporary storage because it will modify it later (using the `create_insn` function).

Then, it uses the letter in the `args` character string to check if there is a match. If there is, it stores immediately the bits into the instruction copy. But, as mentioned above, if there isn't any match, all the work is discarded and `riscv_ip` starts over using a saved pointer to the start of the arguments.

This way it ensures that eventually, the good instruction will be discovered, if at all. It is a slow process, since in many cases 4 other 5 instructions will be parsed and discarded until the correct one is found. Since the order of the opcodes is crucial the most used instructions can be the last ones to be found, what compounds the problem.

Several solutions can be imagined to speed up things, but the question arises if the speed of the assembler encoding is really the limiting factor for the compilation process. In a very cheap riscv machine assembling a 3.6Mb file takes 1.7 seconds, including the time for i/o from disk.

1.10 A more detailed view of instruction encoding

If the first symbol in a line of assembler text is not a label or a directive, the parser calls `md_assemble`.[21]

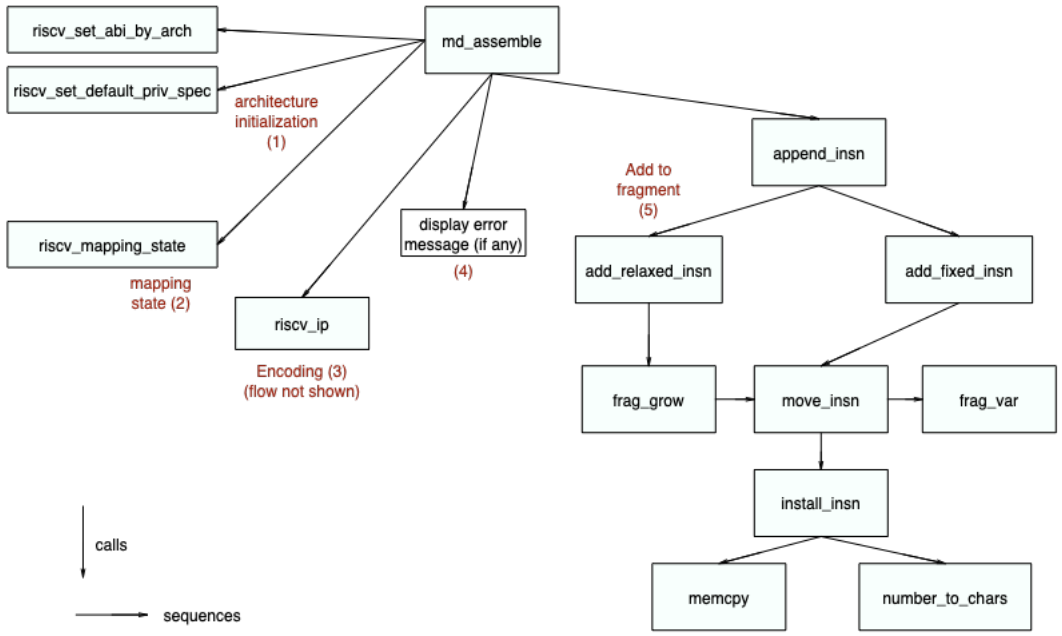


Figure 1.15: md_assemble control flow

```
1  static void md_assemble(char *str)
2  {
3      struct riscv_cl_insn insn;
4      expressionS imm_expr;
5      bfd_reloc_code_real_type imm_reloc = BFD_RELOC_UNUSED;
6
7      /* The architecture and privileged elf attributes should be set
8       * before assembling. */
9      if (!start_assemble) {
10         start_assemble = true;
11         riscv_set_abi_by_arch();                                    (1)
12         if (!riscv_set_default_priv_spec(NULL)) return;
13     }
14     riscv_mapping_state(MAP_INSN,0,false /* fr_align_code */ );    (2)
15     const struct riscv_ip_error error = riscv_ip(str,&insn,
16                                  &imm_expr,&imm_reloc,op_hash); (3)
17     if (error.msg) {                                                (4)
18         if (error.missing_ext)
19             as_bad("%s `%s',extension `%s' required",error.msg,
20                     error.statement,error.missing_ext);
21         else as_bad("%s `%s'",error.msg,error.statement);
22         return;
23     }
24     if (insn.insn_mo->pinfo == INSN_MACRO)
```

[21] The `md` prefix is probably a short for machine dependent.

```
25        macro(&insn,&imm_expr,&imm_reloc);
26     else append_insn(&insn,&imm_expr,imm_reloc);                    (5)
27 }
```

1. If it is the first time that we emit an instruction, initialize things.[22] The functions called are:

   - `riscv_set_abi_by_arch`. This function sets the ABI, and makes a lot of consistency checks.
   - `riscv_set_default_priv_spec` sets the privileged instruction set that will be used.

   This settings allow the assembler to check if an instruction is valid within the subset that is established. The problem with this approach is that if the assembler makes a bad guess about the machine, it will not assemble perfectly legal instructions, as is the case with the U74 CPU that I am using. The GAS assembler doesn't want to assemble the bit manipulation instructions and several other extensions that the U74 supports. Since there wasn't any hope of convincing the "binutils" group that it would be nice if the assembler supported all instructions available in the machine it is running on, I decided to drop those tests[23]

2. The `riscv_mapping_state` function adds "mapping symbols" if there is a transition from another section to the `.text` section. The signification of those is not completely clear, to me at least. In any case they are produced in important quantities, and the function `riscv_check_mapping_symbols` is tasked with removing them if there are too many of them.

   The mapping state can be either `MAP_INSN` or `MAP_DATA`. It is stored in a field of the `riscv_segment_info_type` structure that is a member of the `segment_info` block associated with each section. Within tiny-asm, this structure is set but never really used outside the mapping state function, for coding a state transition from another section to the `.text` section. Only those transitions are monitored.

3. We come now to the central task of an assembler: encoding instructions. This is done in the `riscv_ip` function.

   As you have seen in the encoding of the instructions, each instruction has several pieces of information written in pieces of its 32 bits: the source register number, the opcode, etc. These pieces are the parameters that fill the instruction, besides the fixed bits of the opcode.

   In the opcode table those parameters are grouped into a sequence of letters that represent each one of the different pieces that an instruction can receive. `riscv_ip` interprets those letters and acts accordingly, putting into the specified parts of the instruction the register numbers, the immediate constants or all other parameters that build an instruction. Each letter can have other modifiers, for instance the C letter (Compressed) has several modifiers that specify the parts of the 16 bit compressed instruction. For a full description of each letter see the table in §1.9 page 41.

4. `riscvv_ip` returns either without an error, or with an error description stored in a structure that receives as a parameter.

[22] The assembler uses a global variable (`start_assemble`) that is used only here or in `riscv_write_out_attrs` to test if we have an empty file with no instructions. In the later case we do not write the elf attributes to the executable.

[23] Actually, the situation is more complicated. The GNU people argue that it is a security that the assemble checks if the instructions being assembled are valid. They say that if you invoke gcc with the **undocumented** option `march=rv64gc_zbb` those instructions are assembled. Well, you can judge by yourself.

5. If the instruction found by `riscv_ip` is a macro, the macro-expansion procedure is called, otherwise we append the new instruction to the growing fragment using `append_insn`. `append_insn`'s task is to decide how the new instruction will be added to the current fragment, and if it is necessary to create a new fragment. If the new instruction has a relocation, either it is stored with `add_relaxed_insn` when possible, or a new fixup is created, and the instruction stored with `append_fixed_insn`. If there are no relocations, the instruction is immediately stored with `append_fixed_insn`.

   After storing the instruction, the assembler creates a new fragment for all relocations that could be changed by the linker, to avoid calculating wrong offset between symbols, since those could change by the actions of the linker.

1.11 Writing the object file

After we have encoded all instructions and setup all the static data, processed all the assembler directives, we arrive at the end of the file, and we start preparing for writing the result of our efforts: the object file.

This file is written according to the ELF (**E**xecutable and **L**ink **F**ormat.)[24] standard. This file format is extensively described in a lot of documentation floating in the internet, so it is not necessary to repeat all that here.

Before we start writing out things we must finish the assembling process.

- We have a long list of "fragments", each holding a piece of the final section... we have to stitch all that together.
- We have some symbols that still haven't got a specific location. We should resolve them.
- We have to prepare to write the file header and the section headers.
- We have symbols in an internal format. We have to prepare to write them out in the ELF symbol format.
- References to symbols (fixups) must be resolved as far as it is possible. Of course some symbols are just externals, and can't be resolved anyway.

1.11.1 Write the object file

The `write_object_file` function is a very long one (more than 250 lines). Here is a detailed account of it:

- `subsegs_finish` This function does mainly two things:
  1. Correctly align the section.
  2. Finish the last fragment, so that there isn't any half done fragment.
- `riscv_pre_output_hook` This function finishes optimizations of the `eh_frame` output. Basically, if a subtraction from two symbols is performed, it is feasible to substitute the subtraction by a constant when the two symbols are in the same fragment. Sometimes, however, it is impossible to know if that is the case. In that case the optimization is postponed to the end of the assembly. This is done here.
- The assembler creates some sections to store its own data. They need to be discarded now, since they aren't needed any more. Once we do that, the sections need to be renumbered since we have thrown away some.

[24] Unix is fond of mythological names: We have magic numbers, Elfs, dwarfs, daemons...

- `chain_frchains_together` This function manipulates the next and previous pointer of the fragment chains to make a single list. Now, since we have chained everything in a single list, any new relocations must be done not relative to a fragment, but relative to the start of the big list. We record that we have done the fragment reorganization in the variable `frags_chained`. This global variable is used in the function `fix_new_internal` when making a new fixup:

```
fixS **seg_fix_rootP = (frags_chained ? &seg_info(now_seg)→fix_root
                                      : &frchain_now→fix_root);
```

- `merge_data_into_text`. If the user specified (with the `-R` flag) that data sections should go into the text segment to make the data read-only, we should merge the data and the text sections. This is done now.

- We keep calling `relax_segment` until we record that there isn't any more changes.

```
1       rsi.pass = 0;
2       while (1) {
3           rsi.changed = 0;
4           map_over_sections(relax_seg,&rsi);
5           rsi.pass++;
6           if (!rsi.changed)
7               break;
8       }
```

  `rsi` is a variable of type `struct relax_seg_info`[25]. The function `map_over_sections` just calls the function given in argument for each section in the output file.

- `size_seg`. Now that the address and size of all fragments is known, we can calculate the total size of each segment. This is done in the following stages:

  1. Set the current segment to the one we are measuring.
  2. For each fragment in this section convert them to fragments without any variable part, to be able to size them. This is done in the function `cvt_frag_to_fill`.
  3. Go through the list to the last element. Then:
     `size = fragp->fr_address + fragp->fr_fix;`
  4. Then, the section is padded to alignment if necessary.

- `dwarf2dbg_final_check`. This is interesting stuff. There is a proposal from Alexandre Oliva[26] that introduces the concept of "view numbers" where the same program counter can belong to several views. The underlying need for this are inlined functions, where the inlined code can belong to the current function, or it can be understood as part of the inlined function, allowing the debugger to trace through the inlined function as if it were a normal function call.[27]

[25] A very simple structure:
`struct relax_seg_info {int pass; int changed;}`
The `pass` member is incremented but never used. It is there to allow debugging infinite loops that could arise.

[26] https://www.fsfla.org/~lxoliva/

[27] The whole proposal text is here:
This proposal introduces a new implicit column to the line number table, namely "view numbers", so that multiple program states can be identified at the same program counter, and extends loclists with means to add view numbers to address ranges, enabling locations to start or end at specific views.
This may improve debug information, enabling generators to indicate inlined entry points and preferred breakpoints for statements even if instructions associated with the corresponding source locations were not emitted at the given PC, and to emit variable locations that indicate the initial values of inlined arguments,

- `create_obj_attrs_section` creates a section to hold all program attributes. The attributes should refer to the CPU type where the program can run.
- All relocations refer to symbols. So we have to resolve symbols before doing the relocations. this is done

```
1       if (symbol_rootP) {
2           symbolS       *symp;
3
4           for (symp = symbol_rootP; symp; symp = symbol_next(symp))
5           resolve_symbol_value(symp);
6       }
7       resolve_local_symbol_values();
8       resolve_reloc_expr_symbols();
```

  The `resolve_symbol_value` function tries to determine the value of a possibly very complex expression and assigning it to the symbol.

  The `resolve_local_symbol_value` organizes a traversal of the hash symbol table to resolve all local symbols.

- `elf_frob_file_before_adjust` will go through all symbols and will eliminate unneeded versions of versioned symbols.
- `adjust_reloc_syms` will go through all symbols and try to replace the references to symbols by references to the section symbol + offset.
- `fix_segment`. This function will go through all fixups of a segment and resolve those that can be resolved at this stage. For instance if a fragment's address has been resolved any fixup mentioning this address can be resolved too. Or when a symbol has been resolved, the fixup can be eliminated.
- Now it's time to write the symbol table. The code goes through all symbols checking that:

  1. Local labels are defined.
  2. Splice out symbols that should be ignored, like symbols that were equated to bss or to undefined symbols.
  3. `elf_frob_symbol` Will take care of symbol versioning and associated complexities...
  4. Take care of "warning" symbols, i.e. symbols that are there just to generate a warning. They are just skipped.
  5. Take care of the infinite possibilities of bugs... For instance there could be symbols that were emitted before an alignment that ended as a zero byte alignment. They are unnecessary. Get rid of them.

- `set_symtab`. This function counts the symbols, and allocates a table that will be used to store the symbols to be written out.
- `elf_frob_file`. This function does two things:

  1. In the case we are emitting `stabs` debug information, fill the header with the number of stabs, and other information.

---

and side effects of operations as they would be expected to take effect from the source code, even when multiple statements have their side effects all encoded at the same PC: with view numbers, debug information consumers may be able to logically advance the perceived program state, so as to reflect user-expected changes specified in the source code, even if the operations were reordered or optimized out in the executable code.

2. Do the checks necessary for putting in the elf file flags, the necessary description of the target machine.

- `write_relocs`. Write out all relocations.
- `elf_frob_file_after_relocs`. If we have a group of sections, and we have established the number of relocations, it could be that a section has no longer any relocations or that the number of relocations has changed. In that case the size of the group must be adjusted.
- Once the relocations have been prepared for writing, we can compress the debug section, if necessary. This must be done before anything is written out since it makes the size of the file change.
- `write_contents`. this function organizes the actual writing out of the data. It writes the fixups, the section contents and the fill data to align sections. This is done using the `set_section_contents` function. This function makes some checks and then calls `elf_set_section_contents`.

  This one makes some further checks, copies the contents into the image of the section in RAM and calls `generic_set_section_contents` that makes some checks and positions the file pointer at the correct position, then finally calls `bfd_bwrite` that will send the data to the disk with `fwrite`.

  Described like that, this whole bunch of stacked procedures seems bloated but it is not. Each one takes a piece of the work. The GAS code is written by defensive programmers and defensive programming is not a bad idea. It pays when you have clear error messages and not bad results. Bugs provoked by missing sanity tests are very difficult to find, bugs with clear error messages spare you the time consuming search for "where is the bug?". They pop up with an error message and you instantly know where the problem is.

1.12 Assembler directives

Directives are defined in a table of structures of type `pseudo_typeS`:

Listing 1.7: struct pseudo_typeS

```
typedef struct _pseudo_type {
    /* Assembler mnemonic in lower case, without the implicit dot '.' */
    const char *poc_name;
    /* Function that will be called to handle this directive */
    void (*poc_handler) (int);
    /* Value to pass to handler. */
    int poc_val;
} pseudo_typeS;
```

The assembler defines several tables of this structures. We have the main one, `potable` and several others: `cfi_pseudo_table` for the debug information, `elf_pseudo_table` for the directives concerning the object code format, and a `riscv_pseudo_table` for several riscv specific directives.

All of them will be called from `read_a_source_file` function. Here is the relevant code snippet:

```
if (*s == '.') {
    /* PSEUDO - OP. WARNING: Next_char may be end-of-line. We lookup the pseudo-op
     * table with s+1 because we already know that the pseudo-op begins with a '.' */
    pop = str_hash_find(po_hash,s + 1);
```

```
5     if (pop && !pop→poc_handler)
6         pop = NULL;
7 // ... code elided
8     /* Input_line is restored. Input_line_pointer→1st non-blank char after
9      * pseudo-operation. */
10    (*pop→poc_handler) (pop→poc_val);
11 }
```

The po_hash table is built when the assembler starts, containing the different tables mentioned above. The function that does this is very simple:

Listing 1.8: pop_insert

```
1 static void pop_insert(const pseudo_typeS * table)
2 {
3     const pseudo_typeS *pop;
4     for (pop = table; pop→poc_name; pop++) {
5         if (str_hash_insert(po_hash,pop→poc_name,pop,0) ≠ NULL) {
6             if (!pop_override_ok)
7                 as_fatal("error constructing %s pseudo-op table",
8                     pop_table_name);
9         }
10        //else printf("%s\n",pop→poc_name);
11    }
12 }
```

Just a loop inserting each member of the given table. The variable pop_override_ok is a global that will be zero if we don't accept any insertions with the same name.

That function will be called from pobegin, that looks like this:

Listing 1.9: pobegin

```
1 static void pobegin(void)
2 {
3     po_hash = str_htab_create();
4     pop_table_name = "md"; /* Do the target-specific pseudo ops. */
5     pop_override_ok = 0;  /* Do not accept any shadowing */
6     pop_insert(riscv_pseudo_table);
7     pop_table_name = "obj"; /* Object specific. Skip any already present */
8     pop_override_ok = 1;
9     pop_insert(elf_pseudo_table);
10    pop_table_name = "standard";/* Now portable ones. Skip any already present */
11    pop_insert(potable);
12    pop_table_name = "cfi";   /* Now CFI ones. */
13    pop_insert(cfi_pseudo_table);
14 }
```

This code ensures that machine specific directives shadow any object or standard directives since they are inserted first. The global variable pop_table_name is used for error messages only, as we have seen in the code of pop_insert[28].

1.12.1 .align, .p2align, p2alignw, p2alignl

Entries in the table:

```
1     {"align",s_align_ptwo,0},
2     {"p2align",s_align_ptwo,0},
```

[28] Looking at this code I do not quite understand why there isn't an additional parameter to pop_insert instead of a global variable. Probably it is difficult to modify the syntax for all back-ends of GAS.

```
3     {"p2alignw",s_align_ptwo,-2},
4     {"p2alignl",s_align_ptwo,-4},
```

These four entries lead to calls to the same function, albeit with different arguments.

```
1     void s_align_ptwo(int arg) { s_align(arg,0); }
```

`s_align` receives two arguments. The first one, if positive, defines a default alignment. If negative, it defines a length of a fill pattern. The second argument, if positive, should be interpreted as a byte boundary, not as a power of two. Now, if the first argument was negative, the second argument should contain the fill pattern.

All arguments are optional. If none is given, the alignment defaults to the argument that will be given to `s_align_ptwo`.

The `s_align` function calls eventually `do_align`. The comment at the start of this function says it all:

```
1     /* Guts of .align directive: N is the power of two to which to align. A value
2     * of zero is accepted but ignored: the default alignment of the section will
3     * be at least this. FILL may be NULL,or it may point to the bytes of the fill
4     * pattern. LEN is the length of whatever FILL points to,if anything. If LEN
5     * is zero but FILL is not NULL then LEN is treated as if it were one. MAX is
6     * the maximum number of characters to skip when doing the alignment,or 0 if
7     * there is no maximum. */
```

But we aren't done yet. `do_align` calls `md_do_align` that is actually a macro:

```
1     #define md_do_align(N, FILL, LEN, MAX, LABEL)   \
2     if ((N) ≠ 0 && !(FILL) && subseg_text_p (now_seg)) \
3     {                                               \
4         if (riscv_frag_align_code (N))              \
5         goto LABEL;                                 \
6     }
```

The actual call sequence looks like this:

```
1     md_do_align(n,fill,len,max,just_record_alignment);
```

Yes, there is *still* another level. And in this level we discover that we just can't align anything. The riscv linker changes the size of some instructions, allowing compressed instructions where possible, what will change the adresses of all subsequent instructions. So, the only thing that `riscv_frag_align_code` can do is just emit an alignment relocation that will tell the linker that this fragment needs to be aligned.

Obviously, all this lengthy process could be simplified a lot, but I have tried to keep the original structure, it may be useful to understand GAS in the context of other machines.

1.12.2 .ascii, .asciiz, .string, .string8, .string16, .string32, .string64

All these directives lead to the `stringer` function. The entries are as follows:

```
1     {"ascii",stringer,8 + 0},
2     {"asciz",stringer,8 + 1},
3     {"string8",stringer,8 + 1},
4     {"string16",stringer,16 + 1},
5     {"string32",stringer,32 + 1},
6     {"string64",stringer,64 + 1},
```

The `stringer` receives an odd argument when it should append a zero to its output. The numbers represent how many bytes should it use for each character. The input is done by following `input_line_pointer` that is a global pointer to the assembler text. `stringer`'s code is easy to follow, so it is not further described here.

1.12.3 .bss

Changes (if necessary) the current section to he bss. This section contains uninitialized data and will set to zero at the program's start by the loader. This directive will calls `obj_elf_bss`, a small function that realizes this change.

```
/* Change to the .bss section. */
static void obj_elf_bss(int i ATTRIBUTE_UNUSED)
{
    int    temp;
    obj_elf_section_change_hook();
    temp = get_absolute_expression(); // Optional subsection. Normally blank
    subseg_set(bss_section,(subsegT) temp);
    demand_empty_rest_of_line();
}
```

Function `obj_elf_section_change_hook` remembers the section before the change so that a `.section previous` directive can find it. See §1.12.22, page 66 for `subseg_set`.

1.12.4 .byte, .dc, .dc.a, .dc.b, .dc.d, .dc.l, .dc.s, .dc.w, etc

```
    {"byte",cons,1},
    {"dc",cons,2},
    {"dc.a",cons,0},
    {"dc.b",cons,1},
    {"dc.d",float_cons,'d'},
    {"dc.l",cons,4},
    {"dc.s",float_cons,'f'},
    {"dc.w",cons,2},
    {"hword",cons,2},
    {"int",cons,4},
    {"octa",cons,16},
    {"quad",cons,8},
    {"short",cons,2},
    {"long",cons,4},
    {"quad",cons,8},
    {"word",cons,2},
    {"2byte",cons,2},
    {"4byte",cons,4},
    {"8byte",cons,8},
    {"half",cons,2},
```

GAS likes to be compatible. The consequence of that is the above list. All those directives lead to the same function. You can write a two byte constant with `.short`, `.dc`, `.dc.w`, `.hword`, `.2byte` and `.half`. [29]

So, what does this `cons` function do?

It is a fairly simple function, consisting in a loop reading expressions separated by commas. In the original code, the crucial lines look like this:

```
    do {
        TC_PARSE_CONS_RETURN_TYPE ret = TC_PARSE_CONS_RETURN_NONE;
        ret = TC_PARSE_CONS_EXPRESSION(&exp,(unsigned int)nbytes);

        if (rva) {
            if (exp.X_op == O_symbol)
                exp.X_op = O_symbol_rva;
```

[29] The directives `.2byte`, `.4byte`, etc are used by gcc mainly within the debug information.

```
8             else
9                 as_fatal(("rva without symbol"));
10        }
11        emit_expr_with_reloc(&exp,(unsigned int)nbytes,ret);
12        ++c;
13    } while (*input_line_pointer++ == ',');
```

The problem with macros such as those here (lines 2 and 3), is that they make impossible to know what is going on actually in the program. Translated into C, these two lines expand into:

```
1   do {
2       bfd_reloc_code_real_type ret = BFD_RELOC_NONE;
3       ret = (expr(0,&exp,expr_normal),BFD_RELOC_NONE);
4       ... // The rest is the same
5   }
```

Line 2 shows that `ret` is a member of the enumeration `bfd_reloc_code_real_type` that is assigned zero.

Line 3 is a comma expression, that in its first statement evaluates a call to `expr`, that reads an expression from `input_line_pointer` and in the second (and last) one evaluates to a constant that is assigned to the `ret` variable.

Besides this small problem, `cons` doesn't present any big difficulties.

1.12.5 data

Tells the assembler to change (if necessary) to the data section. This directive is handled by the `s_data` function:

Listing 1.10: s_data

```
1  static void s_data(int ignore ATTRIBUTE_UNUSED)
2  {
3      segT       section;
4      int    temp;
5
6      temp = get_absolute_expression();
7      if (flag_readonly_data_in_text) {
8          section = text_section;
9          temp += 1000;
10     } else section = data_section;
11     subseg_set(section,(subsegT) temp);
12     demand_empty_rest_of_line();
13 }
```

If the data section is readonly, a special subsegment in the text section is used.[30]

See §1.12.22, page 66 for `subseg_set`.

1.12.6 Other data directives

These directives allow you to control the size of integer data being emitted at the current position.

Directive	Description
.half	Emit an integer of 16 bits

[30] It could be possible to set the flags of the data section to read-only, but GAS prefers this methods for portability reasons... Not all systems probably support that.

.word	Emit an integer of 32 bits
.dword	Emit integer of 64 bits
.dtprelword .dtpreldword	Emit a word or double word thread relative symbol for DWARF debug information in thread local variables.
.uleb128 .sleb128	Emit an unsigned or signed leb128 integer at the current position. This must be a number. No symbols allowed. These two will be fully explained in §1.12.23 page 68.

1.12.7 debug, extern, format, lflags, name, noformat, spc, xref

All those directives have only *one* thing in common: they are completely **ignored** by the GNU assembler. It just advances the line pointer to the end of the line.

Why this?

As you guessed, it is just a compatibility feature.

```
{"debug",s_ignore,0},
{"extern",s_ignore,0},/* We treat all undef as ext. */
{"format",s_ignore,0},
{"lflags",s_ignore,0},/* Listing flags. */
{"name",s_ignore,0},
{"noformat",s_ignore,0},
{"spc",s_ignore,0},
{"xref",s_ignore,0},
```

As the comment shows, declaring a symbol *extern* doesn't do anything. The assembler declares all undefined symbols `extern`. This implies that if a misspelled name appears in your assembler program you will see it at link time, not at assembly time. No big deal anyway.

More problematic is ignoring directives like `xref` or `debug`. These directives are expected to *do* something, and silently accepting and ignoring them will provoke in people that expect some result from their directives to search in vain **why** the assembler is not doing what they have written.

This is worst than a clear error message: "unknown directive". Much worst. That is why those directives aren't accepted any more in `tiny-asm`, except the **extern** one, because that one *does* what the user is expecting.

1.12.8 equ, equiv, eqv, set

```
.equ symbol, expression
```

This directive sets the value of symbol to expression. It is synonymous with '.set';

This is something similar to

`#define␣name␣another_name`

in C. There are some subtleties though. The `equiv` directive will complain if the first symbol is already defined. The `eqv` directive announces to the assembler that the right hand side is a forward reference.

```
{"equ",s_set,0},
{"equiv",s_set,1},
{"eqv",s_set,-1},
{"set",s_set,0},
```

The `s_set` function is simple to follow.

1.12.9 globl

```
.global symbol[, symbol, symbol, ...]
```

`.global` makes the symbol visible to ld. If you define symbol in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, symbol takes its attributes from a symbol of the same name from another file linked into the same program.

In the `potable` we have:

```
1 Table: (potable)
2     {"global",s_globl,0},
3     {"globl",s_globl,0},
```

Unix has a big problem with vowels. They are shunned everywhere. Why write `globl`? Is the absence of a poor vowel *really* that shorter? Or is the necessary effort of *remembering its absence* when writing the program (taking precious memory space in the brain) even costlier?

Well, at least the assembler lets you decide, you can use both.

Coming back to our source code, the `s_globl` function is a very simple and short one. It just scans names and adds the `EXTERNAL` bit to each of the symbols scanned in a loop (not shown).

```
1     if ((name = read_symbol_name()) == NULL)
2         return;
3     symbolP = symbol_find_or_make(name);
4     S_SET_EXTERNAL(symbolP);
```

1.12.10 attach_to_group

```
Syntax:
     .attach_to_group <name>
Table: (elf_pseudo_table)
     {"attach_to_group",obj_elf_attach_to_group,0},
```

This will attach the current section to the named group. If the group doesn't exist it will be created. The `obj_attach_to_group` function just changes a pointer and the flags of the current section. The relevant lines (without error checking etc) of this function are:

```
1     elf_group_name(now_seg) = gname;
2     elf_section_flags(now_seg) |= SHF_GROUP;
```

1.12.11 .comm, .common, .lcomm

Only the directive `.comm` and `.lcomm` are documented in the official documentation.

```
Syntax:
.comm symbol , length
Table: (elf_pseudo_table)
{"comm",obj_elf_common,0},
{"common",obj_elf_common,1},
{"lcomm",obj_elf_lcomm,0},
```

`.comm` declares a common symbol named symbol. When linking, a common symbol in one object file may be merged with a defined or common symbol of the same name in another object file. If ld does not see a definition for the symbol–just one or more

common symbols–then it will allocate length bytes of uninitialized memory. length must be an absolute expression. If ld sees multiple common symbols with the same name, and they do not all have the same size, it will allocate space using the largest size.

`.lcomm` (local common) has the same syntax as `comm` but the symbol is just declared in the bss section and not make visible.

`.common` is a synonym for `comm` even if it receives a different argument because actually... the argument is ignored!

```
1 static void obj_elf_common(int is_common ATTRIBUTE_UNUSED)
2 {
3     s_comm_internal(0,elf_common_parse);
4 }
```

The function `s_comm_internal` is mostly parsing and error checking. The essential lines are at the end:

```
1     S_SET_VALUE(symbolP,(valueT) size);
2     S_SET_EXTERNAL(symbolP); // This is absent in lcomm
3     S_SET_SEGMENT(symbolP,bfd_com_section_ptr);
```

1.12.12 hidden

```
Syntax:
    .hidden symbol-name [, symbol-name, ...]
```

Sets the visibility of a symbol, i.e. if it is visible for modules outside the one being assembled. This directive implies *protected* as well.

It is handled by the `obj_elf_visibility` function.

1.12.13 ident

```
Syntax:
    .ident "A string"
Table: elf_pseudo_table
{"ident",obj_elf_ident,0},
```

This directive writes any string into the comments section of the file. For instance:

```
.ident "I love you Barbie"
```

Assembling your file, you can display it to your girlfriend with:

```
star64:~/tiny-asm$ asm sample.s
star64:~/tiny-asm$ objdump -s -j .comment a.out

a.out:     file format elf64-littleriscv

Contents of section .comment:
0000 0049206c 6f766520 796f7520 42617262  .I love you Barb
0010 696500                               ie.
```

She will be surely greatly impressed... The `obj_elf_ident` function creates the `.comments` section if it is not already present. Then, it calls the stringer for parsing. You can write any number of these comments.

1.12.14 insn

```
Syntax:
.insn type, operand [,...,operand_n]
.insn insn_length, value
.insn value
Table: riscv_pseudo_table
{"insn",s_riscv_insn,0},
```

This directive assembles an unknown instruction into the instruction stream. For instance, using the first type of syntax, let's say you want to want to issue the instruction `add a0,a1,a2`. First, you have to look up what type of instruction it is. It is an "R" type of instruction. You write as first argument "r".

After the type, you should give the fields of the R format that are fixed: the opcode, the extra-3 and the extra-7 fields. In this case both are zero. And then, you should give the arguments of the instruction, i.e. the register names.

You should write then:

```
1    .insn r 0x33, 0, 0, a0,a1,a2
```

Note that there isn't any comma between the "r" and the 0x33! The "r" is understood as a part of the opcode.

Now where does this 0x33 come from?

If you go to the opcode table, and search for the `"add"` entries, you will see several of them. You should choose this one:

```
1    {"add",0,INSN_CLASS_I,"d,s,t",MATCH_ADD,MASK_ADD,match_opcode,0},
```

since the other ones further up are compressed (`INSN_CLASS_C`) and we do not want compression. The opcode is in the `MATCH_ADD` field, that is defined in asm.h to be... 0x33. After the two zeroes of the bit fields associated with class "R" we write the 3 required register names.

How can we know that this is OK?

Easy: just write following assembler program:

```
1    add a0,a1,a2
2    .insn r 0x33, 0, 0, a0,a1,a2
```

Then assemble it, and then display the contents with

```
1    star64:~/tiny-asm$ objdump -d sample.o
2
3    sample.o:    file format elf64-littleriscv
4
5    Disassembly of section .text:
6
7    0000000000000004 <main>:
8    4:  00c58533          add a0,a1,a2
9    8:  00c58533          add a0,a1,a2
```

We find the 0x33 in the lower 7 bits of the opcode field.

The other syntax variants of the directive are trivial.

Another example: the instruction `addw a0,a1,a2`. The entry in the opcode table is:

```
␣{"addw",64,INSN_CLASS_I,"d,s,t",MATCH_ADDW,MASK_ADDW,match_opcode,0},
```

We look the constant `MATCH_ADDW` in asm.h, what gives 0x3b. So, as shown in 1.4 page 30, the two fields "extra-3" and "extra-7" are zero. We write then:

```
addw a0,a1,a2
.insn r 0x3b, 0, 0, a0,a1,a2
```

and when disassembling we get:

```
4:  00c58533          add  a0,a1,a2
8:  00c58533          add  a0,a1,a2
c:  00c5853b          addw a0,a1,a2
10: 00c5853b          addw a0,a1,a2
```

The `s_riscv_insn` function essentially just calls `riscv_ip`. The lookup of the "r" letter yields an entry into the `riscv_insn_types` table, that looks like this:
`␣{"r",0,INSN_CLASS_I,"O4,F3,F7,d,s,t",0,0,match_opcode,0},`
where we see the length of the instruction (4 bytes) and the names of the 3 and 7 bits extra fields. Then, we find the usual denominations ("d,s,t") that we discussed when analyzing the string arguments to each opcode, see table 1.8 page 43.

Conclusion This is quite difficult stuff, because precisely the point of an assembler is to avoid you to encode manually the instructions. It is a *very* error prone process. And in the end if you write:

```
.word 0xc58533
```

it will work in the same way. The justification advanced by the GNU folks is that in future versions of the assembler you will *not* see this as just data, but as a real instruction.

Maybe. But I think a more real justification is that the riscv architecture itself allows for instruction extensions, and has a whole part of the instruction space available for standard or non-standard extensions to the accepted opcodes. The existence of an `insn` extension here, would allow the assembler to assemble code that uses those extensions.

1.12.15 internal

```
Syntax:
.internal symbol-name [, symbol-name, ...]
```

Sets the visibility of a symbol, i.e. if it is visible for modules outside the one being assembled. This directive implies *protected* as well.

It is handled by the `obj_elf_visibility` function. function.

1.12.16 loc

```
Syntax:
    .loc fileno lineno [column] [options]
Table:  elf_pseudo_table
     {"loc",dwarf2_directive_loc,0},
```

Ahhh the old days, when everything was simple and clear! Remember when the debug information for the line number was just a triplet of address, file, line?

Say goodbye to that now, and welcome to DWARF[31]. The line number is a series of instructions to an interpreted language executed by a state machine.

Yes, you read correctly.

Conceptually we have a table of addresses, each one with as many properties as desired:

---

[31] Critiques to DWARF abound. See for instance: https://tobast.fr/doc/publications/oopsla19-dwarf.pdf

address	source file	source line	source column	state-ment?	basic block	... other columns
0x40260	1	23	12	0	0	
0x40264	1	23	12	1	1	

.
.
.

"... we design a byte-coded language for a state machine and store a stream of bytes in the object file instead of the matrix. This language can be much more compact than the matrix. When a consumer of the line number information executes, it must "run" the state machine to generate the matrix for each compilation unit it is interested in."[32]

The arguments for the `.loc` directive then, are as follows:

- `fileno`. The file index in the assembler's file table.
- `lineno`. Line number.
- `column`. This field is optional.
- `options`. They are the following:
  - `basic_block` This instruction represents the start of a basic block.[33]
  - `prologue_end`. End of the setup of the stack frame. This changes the state of the interpreter. In C it corresponds to the opening brace of a function.
  - `is_stmt␣value` Start of a statement sequence.
  - `isa␣value` Sets the instruction set architecture register to *value*
  - An unsigned integer identifying the block to which the current instruction belongs. Discriminator values are assigned arbitrarily by the DWARF producer and serve to distinguish among multiple blocks that may all be associated with the same source file, line, and column. Where only one block exists for a given source position, the discriminator value should be zero. This is necessary because the compiler can move instructions around to keep the pipeline busy. Then, instructions belonging a one or several blocks could be mixed.
  - `view`. This is not in the 4th edition of the DWARF standard nor in the 5th. It has been added later probably. The documentation says:

    > This option causes a row to be added to .debug_line in reference to the current address (which might not be the same as that of the following assembly instruction), and to associate value with the view register in the .debug_line state machine. If value is a label, both the view register and the label are set to the number of prior .loc directives at the same program location. If value is the literal 0, the view register is set to zero, and the assembler asserts that there aren't any prior .loc directives at the same program location. If value is the literal -0, the assembler arrange for the view register to be reset in this row, even if there are prior .loc directives at the same program location.

    Crystal clear isn't? [34]

[32] DWARF Debugging Information Format Version 4, page 108

[33] A basic block is a sequence of instructions where only the first instruction may be a branch target and only the last instruction may transfer control. A procedure invocation is defined to be an exit from a basic block.

[34] There is no other documentation anywhere that would state what this thing does in a more understandable way... Sorry.

The function `dwarf2_directive_loc` is interesting as an example of the functions used to parse data within the assembler. To make things a bit clearer I have added comments to everything.

Listing 1.11: Parsing .loc directive

```
1  static void  dwarf2_directive_loc(int dummy ATTRIBUTE_UNUSED)
2  {
3      /* If we see two .loc directives in a row,force the first one to be output now.*/
4      if (dwarf2_loc_directive_seen) dwarf2_emit_insn(0);
5      offsetT filenum = get_absolute_expression();
6      SKIP_WHITESPACE();
7      offsetT line = get_absolute_expression();
8      /* error checking: */
9      if (filenum < 1) {
10         /* DWARF5 specifies that a file number of zero indicates that
11            the file is unknown */
12         if (filenum == 0 && dwarf_level < 5) dwarf_level = 5;
13         /* All other values are just nonsense */
14         if (filenum < 0 || DWARF2_LINE_VERSION < 5) {
15             as_bad("file number less than one");
16             return;
17         }
18     }
19     if ((valueT) filenum ≥ files_in_use || files[filenum].filename == NULL) {
20         as_bad("unassigned file number %ld",(long)filenum);
21         return;
22     }
23     gas_assert(debug_type == DEBUG_NONE);
24     current.filenum = filenum;
25     current.line = line;
26     current.discriminator = 0;
27     SKIP_WHITESPACE();
28     /* test for an optional column number */
29     if (ISDIGIT(*input_line_pointer)) {
30         /* We have the optional column number */
31         current.column = get_absolute_expression(); SKIP_WHITESPACE();
32     }
33     /* Now we start parsing the "options" field */
34     while (ISALPHA(*input_line_pointer)) {
35         char          *p,c = get_symbol_name(&p);
36         offsetT       value;
37         if (strcmp(p,"basic_block") == 0) {
38             current.flags |= DWARF2_FLAG_BASIC_BLOCK;
39             *input_line_pointer = c; // Restore character
40         } else if (strcmp(p,"prologue_end") == 0) {
41             if (dwarf_level < 3) dwarf_level = 3;
42             current.flags |= DWARF2_FLAG_PROLOGUE_END;
43             *input_line_pointer = c;
44         } else if (strcmp(p,"epilogue_begin") == 0) {
45             if (dwarf_level < 3) dwarf_level = 3;
46             current.flags |= DWARF2_FLAG_EPILOGUE_BEGIN;
47             *input_line_pointer = c;
48         } else if (strcmp(p,"is_stmt") == 0) { // is_stmt <boolean value>
```

get_absolute_expression

debug_type

current

get_symbol_name

```
49          (void)restore_line_pointer(c);
50          value = get_absolute_expression();
51          if (value == 0) current.flags &= ~DWARF2_FLAG_IS_STMT;
52          else if (value == 1) current.flags |= DWARF2_FLAG_IS_STMT;
53          else { as_bad("is_stmt value not 0 or 1"); return; }
54      } else if (strcmp(p,"isa") == 0) { // "isa" numbers are defined by the ABI
55          if (dwarf_level < 3) dwarf_level = 3;
56          (void)restore_line_pointer(c);
57          value = get_absolute_expression();
58          if (value ≥ 0) current.isa = value;
59          else {
60              as_bad("isa number less than zero");
61              return;
62          }
63      } else if (strcmp(p,"discriminator") == 0) {
64          (void)restore_line_pointer(c);
65          value = get_absolute_expression();
66          if (value ≥ 0) current.discriminator = value;
67          else {
68              as_bad(("discriminator less than zero"));
69              return;
70          }
71      } else if (strcmp(p,"view") == 0) {
72      /* Now we parse the mysterious "view" statement. */
73          symbolS       *sym;
74          (void)restore_line_pointer(c);
75          SKIP_WHITESPACE();
76          if (ISDIGIT(*input_line_pointer) || *input_line_pointer == '-') {
77              /*
78               * Now, we expect either "0" or "-0"
79               */
80              bool       force_reset = *input_line_pointer == '-';
81              value = get_absolute_expression();
82              if (value ≠ 0) {
83                  as_bad("numeric view can only be asserted to zero"); return;
84              }
85              if (force_reset && force_reset_view) sym = force_reset_view;
86              else {
87                  sym = symbol_temp_new(absolute_section,&zero_address_frag,value);
88                  if (force_reset) force_reset_view = sym;
89              }
90          } else { // We have a symbol that will be put into the "view" register.
91              char          *name = read_symbol_name();
92              // We silently accept .loc view followed by nothing, without
93              // any warning or error.
94              if (!name) return;
95              sym = symbol_find_or_make(name);
96              free(name); // read_symbol_name allocates memory for its result
97              if (S_IS_DEFINED(sym) || symbol_equated_p(sym)) {
98                  if (S_IS_VOLATILE(sym)) sym = symbol_clone(sym,1);
99                  else if (!S_CAN_BE_REDEFINED(sym)) {
100                     as_bad("symbol `%s' is already defined",S_GET_NAME(sym));
                            return; }
101                 }
102                 S_SET_SEGMENT(sym,undefined_section); S_SET_VALUE(sym,0);
103                 symbol_set_frag(sym,&zero_address_frag);
104             }
105             current.u.view = sym;
```

restore_line_pointer

```
106         } else {
107             as_bad("unknown .loc sub-directive `%s'",p);
108             (void)restore_line_pointer(c); return;
109         }
110         /* This macro differs from SKIP_WHITESPACE in that it ignores a double quotes
111          * after the name */
112         SKIP_WHITESPACE_AFTER_NAME();
113     }
114     demand_empty_rest_of_line();
115     dwarf2_any_loc_directive_seen = dwarf2_loc_directive_seen = true;
116     /* If we were given a view id,emit row now */
117     if (current.u.view) dwarf2_emit_insn(0);
118 }
```

6 demand_empty_rest_of_line

1. The function `get_absolute_expression` reads a constant from the global line pointer and sets it to just after the last character of the constant. If any error occurs, it emits an error message and returns zero. If the expression is absent, it returns zero without any error message. This makes many things default to a convenient zero.

2. The value in the global variable `debug_type` will be turned off by the function `dwarf2_directive_filename`, and if we don't have a dwarf style `.file` directive in between, then `files_in_use` will be zero and the error in line 15 will trigger.
   Note: The global `debug_type` will be left to zero, effectively disabling the emission of any debug information by the assembler.

3. `current` is a structure of type `dwarf2_line_info` that holds the current context. We update it AFTER all error checking is done, to preserve a correct context in case of an error

4. The `get_symbol_name` function parses a symbol using `input_line_pointer`. It writes a zero immediately after the expected symbol and returns the value of the character at the position where zero was written. Its result is left in its pointer argument, that will point to the start of the symbol.

5. `restore_line_pointer` writes the previous character into the line pointer, advances to the next character and if it is a double quote, it ignores it by advancing again.

6. `demand_empty_rest_of_line` advances the line pointer to the next newline character. If there is anything in that part of the line it will complain with an error.

1.12.17 local

```
Syntax:
    .local symbol,symbol,...
    Table: elf_pseudo_table
    {"local",obj_elf_local,0},
```

This directive makes the given symbol a local symbol, not visible to other modules. Since all symbols are local unless declared extern or undefined, the utility of this is not clear.

The important lines of `obj_elf_local` are:

```
1     symbolP = get_sym_from_input_line_and_check();
2     S_CLEAR_EXTERNAL(symbolP);
3     symbol_get_obj(symbolP)→local = 1; // See below
```

The function `symbol_get_obj` returns a pointer to a small structure that keeps several disjoint pieces of information about a symbol, among them, whether it is a local symbol. We can't access directly the field because of local symbols precisely.

```
/* Get a pointer to the object format information for a symbol. */
static struct elf_obj_sy *symbol_get_obj(symbolS * s)
{
    if (s→flags.local_symbol)
        s = local_symbol_convert(s);
    return &s→x→obj;
}
```

If the symbol is local, we have to convert it first. See §1.5.2, page 18 to see where this piece fits in the general schema of things.

1.12.18 .option

```
Syntax:
.option <option-name>
Table: riscv_pseudo_table
{"option",s_riscv_option,0},
```

This handles the update of several riscv related options. The example given in the GAS documentation runs as follows:

```
    .option push
    .option norelax
    la gp, __global_pointer$
    .option pop
```

In the "relaxation" process, the assembler tries to find shorter, compressed, sequences for instructions. It tries to substitute loading a global directly, for a shorter sequence that loads the address from an offset from the `__global_pointers` table. The problem arises when you want to load the address of the `__global_pointers` table itself. In that case you do NOT want the assembler to pick an offset since the `__global_pointers` table is not loaded. Then, you disable for a single instruction, this feature and all goes well.

Of course this happens only to people that are writing the startup code, or other assembler wizards. This kind of fiddling is *for them only*. Please do not mess around with any of this things yourself.

The code for `s_riscv_options` is trivial: a long series of:

```
    if (strcmp(name,"push") == 0) { /* code for push option */}
    else if (strcmp(name,"pop") == 0 {/* code for pop option */})
    etc...
```

Other interesting values for `.option` are:

- `pic` or `nopic`. Enable or disable the position independent code generation. This corresponds to the `-fPic` flag in gcc.
- `rvs` or `norvc`. Enable or disable the compressed instructions generation.
- `relax` or `norelax`. Enable or disable relaxation.
- `csr-check` or `nocsr-check`. Enables or disables checking when using the CSR registers.
- Etc. There are many other obscure things to peruse here: binutils-docs

1.12.19 org

```
Syntax:
.org new-location-counter , fill byte
```

Advance the location counter of the current section to *new-location-counter*. It should be either an absolute expression or an expression with the same section as the current subsection. That is, you can't use .org to cross sections: if it has the wrong section, the .org directive is ignored. To be compatible with former assemblers, if the section of new-lc is absolute, as issues a warning, then pretends the section of new-lc is the same as the current subsection.

1.12.20 protected

```
Syntax:
.protected symbol-name [, symbol-name, ...]
```

Sets the visibility of a symbol, i.e. if it is defined for modules outside the one being assembled, the definition in this module will be used.

It is handled by the `obj_elf_visibility` function. function.

1.12.21 reloc

The documentation of GAS says about this directive:

```
Syntax:
.reloc offset, reloc_name[, expression]
```

> Generate a relocation at `offset` of type `reloc_name` with value `expression`. If offset is a number, the relocation is generated in the current section. If offset is an expression that resolves to a symbol plus offset, the relocation is generated in the given symbol's section. `expression`, if present, must resolve to a symbol plus addend or to an absolute value, but note that not all targets support an addend. e.g. ELF REL targets such as i386 store an addend in the section contents rather than in the relocation. This low level interface does not support addends stored in the section.

The last part of the description needs maybe a clarification. In the x86 systems, the addend to the relocation is stored in the data itself, so the program loader should only add the load address. This makes constructing relocations with an addend impossible.

Why is this directive necessary? Mystery, the official documentation gives no examples, and (with my limited imagination) I just can't figure out its use.[35]

Well, the only way of figuring out this, is to use it and see what it does. I write this in C:

```
1  long double mm = 3.1415926534564321;
2  int main(void) {}
```

I compile it with: `gcc -c -S tld.c` and obtain a `tld.s` assembler file:

```
1  .file  "tld.c"
2  .size  mm, 16
3  mm:
4  .word  0
5  .word  -1610612736
6  .word  -1253836416
7  .word  1073779231
8  .text
9  .globl main
```

[35] In the documentation of the ARM assembler I found a similar RELOC directive that (seems) to force the assembler to put either a symbol or the preceding instruction at a specific address, like the `.org` directive, but I am not sure

```
10      .type   main, @function
11      main:
12      jr  ra
```

We have then, a long double in the data section. I start gdb:

```
(gdb) print &mm
$1 = (<data variable, no debug info> *) 0x2aaaaac010 <mm>
```

OK, now I add the line: `.reloc 8,BFD_RELOC_32,mm` after the last `.word` in the definition of mm. I start gdb with the new program and...

```
(gdb) print &mm
$1 = (<data variable, no debug info> *) 0x2aaaaac010 <mm>
```

The address is the same, the contents of the long double constant are the same, nothing changed. Weird.

Next thing: Change the text segment? I add the same reloc directive just before the `jr ra` at the end of `main`. Now I obtain:

```
(gdb) b main
Breakpoint 1 at 0x66c
(gdb) run
Starting program: /home/jacob/tiny-asm/tld-reloc
/home/jacob/tiny-asm/tld-reloc: error while loading shared libraries:\
unexpected reloc type 0x01
[Inferior 1 (process 1474) exited with code 0177]
```

Great! Now something seems to have changed. I can't run the program. The relocation is probably disturbing something in the program loader.

Conclusion

1) Do not mess around with this unless you know exactly what you are doing...

2) If you know what you are doing... please let me know.

1.12.22 text

Tells the assembler to change (if necessary) to the text section. Data and instructions will go at the end of that section.[36]

The change is handled by the `s_text` function:

Listing 1.12: s_text function

```
1 static void s_text(int ignore ATTRIBUTE_UNUSED)
2 {
3     int temp = get_absolute_expression();
4     subseg_set(text_section,(subsegT) temp);
5     demand_empty_rest_of_line();
6 }
```

The function `subseg_set` is used in several other functions to change the current section/segment.

[36] Remember that "text" in this context has *nothing* to do with a text format, in the usual sense of the word. There is no text sequences here, unless you put a text sequence yourself.

Listing 1.13: Code of subseg_set

```
1  static void subseg_set(segT secptr,subsegT subseg)
2  {
3      if (!(secptr == now_seg && subseg == now_subseg))
4          subseg_set_rest(secptr,subseg);
5  }
6  static void subseg_set_rest(segT seg,subsegT subseg)
7  {
8      frchainS      *frcP;  /* crawl frchain chain */
9      frchainS     **lastPP; /* address of last pointer */
10     frchainS      *newP;  /* address of new frchain */
11     segment_info_type *seginfo;
12
13     if (frag_now && frchain_now)
14         frchain_now→frch_frag_now = frag_now;                          (1)
15     subseg_change(seg,(int)subseg);                                    (2)
16     seginfo = seg_info(seg);                                           (3)
17     /* Should the section symbol be kept? Yes. */
18     seg→symbol→flags |= BSF_SECTION_SYM_USED;                          (4)
19     /* Attempt to find or make a frchain for that subsection. We keep the
20     * list sorted by subsection number. */
21     for (frcP = *(lastPP = &seginfo→frchainP); frcP ≠ NULL;
22             frcP = *(lastPP = &frcP→frch_next))
23             if (frcP→frch_subseg ≥ subseg)
24                 break;
25     if (frcP == NULL || frcP→frch_subseg ≠ subseg) {
26     /* Not found. Make a new. This should be the only code that creates a frchainS.*/
27         newP = (frchainS *) obstack_alloc(&frchains,sizeof(frchainS));
28         newP→frch_subseg = subseg;
29         newP→fix_root = NULL;
30         newP→fix_tail = NULL;
31         obstack_begin(&newP→frch_obstack,CHUNKSIZE);
32         obstack_alignment_mask(&newP→frch_obstack) = __alignof__(fragS) - 1; (5)
33         newP→frch_frag_now = frag_alloc(&newP→frch_obstack);
34         newP→frch_frag_now→fr_type = rs_fill;
35         newP→frch_cfi_data = NULL;
36         newP→frch_root = newP→frch_last = newP→frch_frag_now;
37         *lastPP = newP; // Insert in chain
38         newP→frch_next = frcP;
39         frcP = newP;
40     }
41     frchain_now = frcP;
42     frag_now = frcP→frch_frag_now;
43 }
```

1. Make sure that `frchain_now` has a correct pointer in `frch_frag_now`.
2. `subseg_change` is a small function that sets the global variables `now_seg` and `now_subseg` to the values given, and, if necessary, allocates the `seg_info` structure.
3. `seg_info` is just a macro that accesses the structure in the `userdata` of the bfd.
4. The original code used a function call and was just too complicated for setting a flag. It was: `if (bfd_keep_unused_section_symbols(stdoutput))` that returned always true...
5. The gnu C construct `__alignof__` has an equivalent in the C standard of 2011: `_Alignof`. In the code above it should be `_Alignof(fragS)`.

1.12.23 uleb128, sleb128

```
Syntax:
.uleb128 value
.sleb128 value
Table: riscv_pseudo_table
{"uleb128",s_riscv_leb128,0},
{"sleb128",s_riscv_leb128,1},
```

These instructions encode a number using a special format. There is also a general directive for all machines that has the same syntax.

To encode an unsigned number:

1. Split the number in 7 bit chunks
2. Read the 7 bits of the lowest significant bits into a byte.
3. Set the most significant bit of the byte to 1 if more bytes follow, to zero otherwise.
4. Output 1 byte and shift the value right by 7 bits.

Listing 1.14: output_uleb128

```
1  static unsigned int output_uleb128(char *p,valueT value)
2  {
3      char          *orig = p;
4      unsigned byte;
5
6      do {
7          byte = (value & 0x7f);
8          value >≥  7;
9          if (value ≠ 0)
10         /* More bytes to follow. */
11             byte |= 0x80;
12         *p++ = byte; // If value was zero, byte is zero
13     } while (value ≠ 0);
14     return p - orig;
15 }
```

A signed number has a different encoding. Example: Encode `-98765432`

1. Ignore the minus sign. Binary representation is
   `0101 1110 0011 0000 1010 0111 1000`, a 27 bit number padded to 28 with zero.
2. Negate all bits, what gives:
   `1010 0001 1100 1111 0101 1000 0111`
3. Add 1, what gives:
   `1010 0001 1100 1111 0101 1000 1000`
4. Split into 7 bit groups:
   `1010000 1110011 1101011 0001000`
5. Add high 1 bit in all but the most significant one
   `01010000 11110011 11101011 10001000` → `0x50F3EB88`

The code for this is written in a quite complicated way, maybe because the code doesn't do step 1 above or because some machine under some OS is behaving badly...

Listing 1.15: output_sleb128

```
1  static inline unsigned int output_sleb128(char *p,offsetT value)
2  {
3      char          *orig = p;
4      int    more;
5
6      do { unsigned byte = (value & 0x7f);
7      /* Sadly,we cannot rely on typical arithmetic right shift behaviour. Fortunately,
8       * we can structure things so that the extra work reduces to a noop on systems
9       * that do things "properly". */
10        value = (value >> 7)|~(-(offsetT) 1 >> 7);
11        more = !((((value == 0) && ((byte & 0x40) == 0))
12        || ((value == -1) && ((byte & 0x40) ≠ 0))));
13        if (more) byte |= 0x80;
14        *p++ = byte;
15     } while (more);
16     return p - orig;
17 }
```

1.12.24 Other directives

In general, the code for handling directives is simple and easy to follow. There is no need to detail all of that here.

1.13 The cfi directives

CFI stands for **C**all **F**rame **I**nformation. The objective of these directives is to furnish to a debugger enough information so that at any address within the program, the layout of the stack is clear.

The C++ language uses also this kind of information for another purpose: to rewind the stack, looking for a procedure that will *catch* an exception that has been thrown somewhere in the program. To be able to reconstruct the stack at any moment, big tables are generated, that give the stack unwinding machinery all the information needed to rewind the stack.

Before we get into the details, we need to explain some concepts. We begin with the concept of the *stack frame*, i.e. the portion of the stack used by the currently running function. When a function call is executed, both the riscv CPU and the ARM cpu copy the address of the next instruction into a special register. At the end of the called function, the last instruction that is executed is a jump to the address stored into that register.

Other machines like the x86 family, do not have a link register and the machine pushes the return address into the stack, decreasing the stack by the address size and writing into the new space the return address. Under the riscv/ARM RISC machines we have a link register that allows to avoid (sometimes) to store the return address in memory.

The stack address at the moment of the call is called **C**anonical **F**rame **A**ddress or CFA.

The first thing the called procedure does is to save the permanent registers that it will use. All machines have in their ABI a list of registers that are preserved across calls (the permanent registers) and other scratch registers that are used freely, without any obligation to preserve their contents. A procedure then, needs to store the current values of those registers in the stack to be able to restore them at the end to their previous values.

To be able to reconstruct the data that is active at procedures higher in the stack, the debugger or the stack unwinding machinery must restore the values of the saved registers, so the addresses and register numbers must be stored in the tables for each procedure. Starting with the current instruction pointer, the debugger restores the values of the previous CFA,

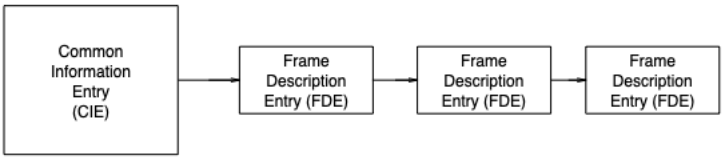
virtually returning from a procedure, what allows it to show the values of all the variables of that procedure, and so on.

The debug information is independent of the type of machine being used, what complicates further things.

- Compilers can duplicate the epilogue to avoid executing a jump instruction to a common one.
- Sometimes a procedure uses a frame pointer register, sometimes they use directly the stack pointer.
- Within the prologue or epilogue, the stack can change. Some compilers will use a push instruction for each register saved, some others will subtract from the stack a fixed amount, and save the registers at fixed offsets from the stack or frame pointer.
- Sometimes a preserved register will be saved in a scratch register, and restored later without using a stack frame...
- Some machines use a bit-mask for saving the registers in a single instruction.
- Etc. There are many other special conditions, weird designs that needed not to be mentioned here.

1.13.1 Concepts

The `.eh_frame` section contains two things: a CIE (Common Information Entry) and several FDEs or Frame Description Entry) records.



The CIE

Table 1.13: Common Information Entry fields

Field	Description
Length	A 4 byte unsigned value indicating the length in bytes of the CIE structure, not including the Length field itself. If Length contains the value 0xffffffff, then the length is contained in the Extended Length field. If Length contains the value 0, then this CIE shall be considered a terminator and processing shall end.
Extended Length	Optional, see above. In practice, this is never used.
CIE-ID	A 4 byte value that is used to distinguish between CIEs and FDEs. In CIEs it will be always zero.
Version	This is a single byte and should be 1.
Augmentation	This is a series of byte codes that are interpreted (sounds familiar?) See below.
Code alignment	An unsigned leb128 encoded value that represents the units used in the "advance location" instructions in this CIE and its associated FDEs.

Table 1.13: Common Information Entry fields

return address register	This field is only mentioned in the MaskRay blog. All other official documents do not mention it[37]
Data alignment factor	Similar to the code alignment factor above
Augmentation length	Unsigned leb128 encoded value. This field is only present if the augmentation string contains the 'z' character.
Augmentation data	A block of data, that is interpreted according to the augmentation string.
The augmentation string characters	
'z'	Indicates there is some data there. Must be the first character.
'L'	The FDEs contain pointers to language specific data. This is a single byte that indicates how those pointers are encoded.
'P'	This indicates the presence of two items: 1) A single byte that specifies how the second item, a pointer, is encoded. 2) The second item is encoded according to the type of encoding described by the first, and it represents a pointer to a **personality** routine, i.e. some routine that will be used to unwind the stack according to the language preferences.
'S'	An associated FDE describes a signal frame, i.e. an interrupt procedure [38].

The FDE

FDE stands for Frame Description Entry.

Table 1.14: FDE fields

Field	Description
Length	In 4 bytes
Extended Length	Same specs as in the CIEs above
CIE pointer	A 4 byte unsigned value that when subtracted from the offset of the the CIE Pointer in the current FDE yields the offset of the start of the associated CIE.
Program Counter begin	This is a pointer encoded according to the method specified by the 'R' character in the CIE[39]
PC range	An absolute value that tells how long the code section is.
Augmentation length	Unsigned leb128 encoded value that contains the length of the following data

[37] The riscv specification mentions explicitly that other registers could contain the return address.

> There is no dedicated stack pointer or subroutine return address link register in the Base IntegerISA; the instruction encoding allows any x register to be used for these purposes. However, the standard software calling convention uses register x1 to hold the return address for a call, with register x5 available as an alternate link register. The standard calling convention uses register x2 as the stack pointer.

RISC-V Unprivileged ISA V20191214-draft, page 14 For RISC-V machines then, this field *could* be useful. In any case, the software representation has a field "return column".

[38] This letter is not mentioned in the Linux Standard Base specifications release 5, but it is mentioned in the MaskRay blog.

[39] ... as far as I have understood this mess.

Table 1.14: FDE fields

Augmentation data	Contains pointers encoded according to the prescriptions of the CIE
Call frame instructions	A set of call frame instructions.

Software representation

A CIE will be described by the following structure in asm.h:

```
struct cie_entry {
    struct cie_entry *next;
    symbolS       *start_address;
    unsigned   return_column;
    unsigned   signal_frame;
    unsigned char fde_encoding;
    unsigned char per_encoding;
    unsigned char lsda_encoding;
    expressionS personality;
    struct cfi_insn_data *first,*last;
};
```

The `cie_entry` structure will be built in the function `cfi_finish` .
An FDE is described by the following structure:

```
struct fde_entry {
    struct fde_entry *next;         Linked list
    symbolS       *start_address;   start
    symbolS       *end_address;     end
    struct cfi_insn_data *data;
    struct cfi_insn_data **last;
    unsigned char per_encoding;     Always DW_EH_PE_omit
    unsigned char lsda_encoding;    Always DW_EH_PE_omit
    int    personality_id;          Not supported in riscv
    expressionS personality;
    expressionS lsda;
    unsigned   return_column;
    unsigned   signal_frame;
    int    eh_header_type;
    /* Compact unwinding opcodes,not including the PR byte or LSDA. */
    int    eh_data_size;
    uint8_t       *eh_data;
    symbolS       *eh_loc;           Not used in riscv
    int    sections;
};
```

The constructor is `cfi_new_fde`. It receives a label symbol as argument, and the fde will start at that label. Calls `alloc_fde_entry` to allocate and fill the new structure with default values. The default "return column" is 1, as the ABI specifies[40].

1.13.2 An example

Let's see how the debug information is organized with a simple example. Given the following C program:

[40] This is a misnomer. It is not a "column" but a register number actually. Columns in the virtual table correspond to register numbers.

```
#include <stdio.h>
int main(void)
{
    printf("hello\n");
}
```

... sorry for this lack of any imagination. Now, if we compile this with:

```
star64:~/tiny-asm $ gcc -c -S -g hello.c
```

We obtain then:

Listing 1.16: hello.s

```
star64:~/tiny-asm cat hello.s
    .file   "hello.c"       set the file name
    .option pic             see §1.12.18 page 64
    .text                   assemble in the text section
.Ltext0:
    .cfi_sections .debug_frame See §1.13.3 page 74.
    .file 0 "/home/jacob/tiny-asm" "hello.c"
    .section   .rodata      assemble in the read only section
    .align 3                align to multiple of 8 (2^3)
.LC0:
    .string "hello"         see §1.12.2 page 52
    .text
    .align 1
    .globl  main
    .type   main, @function
main:
.LFB0:                      "main" will be known as LFB0 in some debug statements
    .file 1 "hello.c"
    .loc 1 3 1              See 1.12.16, page 59.
    .cfi_startproc          first executable instruction of "main"
    addi    sp,sp,-16       reserve space for stack frame
    .cfi_def_cfa_offset 16 record that with CFI
    sd  ra,8(sp)            store return address at sp+8
    sd  s0,0(sp)            store previous frame pointer at (sp).
    .cfi_offset 1, -8       return address is at s0-8. See §1.13.6 page 77
    .cfi_offset 8, -16      previous frame pointer is at s0-16
    addi    s0,sp,16        set s0 (frame pointer)
    .cfi_def_cfa 8, 0       See §1.13.8 page 77
    .loc 1 4 2              Start line 4 col 2 in the C text above
    lla a0,.LC0             load the address of .LC0 into a0
    call    puts@plt        call puts (and not printf)
    li  a5,0                put zero into scratch register a5
    .loc 1 5 1              we start line 5 col 1 of the program text
    mv  a0,a5               put the zero into the result register
    ld  ra,8(sp)            restore the return address
    .cfi_restore 1          tell that to CFI
    ld  s0,0(sp)            restore the frame pointer
    .cfi_restore 8          tell that to CFI
    .cfi_def_cfa 2, 16      See §1.13.8 page 77
    addi    sp,sp,16        restore the stack
    .cfi_def_cfa_offset 0  tell that to CFI
    jr  ra                  jump to the return address
    .cfi_endproc            tell CFI that we returned
.LFE0:                      alias for the end of "main"
    .size   main, .-main    subtract from the current position the address of "main"
                            label. That will be the size of this procedure.
```

```
47 # Further lines snipped
```

We see here that there are only 7 `.cfi_*` directives used. In bigger files, for instance in `asm.c` we find that the only directives used are exactly the same ones. And that file makes around 35 000 lines. We will document here those ones that are used by gcc. The other are documented in the GAS documentation.

Let's go to each of those `cfi` directives in detail.

1.13.3 cfi_sections

```
Syntax:
.cfi_sections <section_list>
Table: cfi_pseudo_table
 {"cfi_sections",dot_cfi_sections,0},
```

The directive `.cfi_sections` is used to specify the type of format that should be used: whether CFI directives should emit `.eh_frame` section, `.debug_frame` section and/or `.sframe` section. To emit multiple sections, specify them together in a list. For example, to emit both `.eh_frame` and `.debug_frame`, use `.eh_frame, .debug_frame`. The default if this directive is not used is `.cfi_sections .eh_frame`.

The `.eh_frame` is required for exceptions to work. It must contain sufficient info to unwind from all the places where exception may be raised, but doesn't have to include anything beyond that. For example, it does not need to contain info needed to unwind through function prologue or epilogue, since no exception can be raised there.

The `.debug_frame` (and other `.debug_*` sections) is only needed for debugging (and also for "self-aware" programs which unwind their own stack on e.g. crashes). It should contain sufficient info for debugger to unwind the stack from arbitrary place in the program, though in practice it may not.

The differences between the two formats are:[41]

- `.eh_frame` is based on `.debug_frame` introduced in DWARF v2.
- `.eh_frame` has the flag of `SHF_ALLOC` (indicating that a section should be part of the process image) but `.debug_frame` does not, so the latter has very few usage scenarios.
- `.debug_frame` supports DWARF64 format (supports 64-bit offsets but the volume will be slightly larger) but `.eh_frame` does not support (in fact, it can be expanded, but lacks demand)
- In the CIE (Common Information Entry) of `.debug_frame`, augmentation instead of `augmentation_data_length` and `augmentation_data` is used.
- The version field in CIEs is different.
- The meaning of `CIE_pointer` in FDEs is different. `.debug_frame` indicates a section offset (absolute) and `.eh_frame` indicates a relative offset. This change made by `.eh_frame` is great. If the length of `.eh_frame` exceeds 32-bit, `.debug_frame` has to be converted to DWARF64 to represent `CIE_pointer`. Relative offsets do not need to worry about this issue (if the distance between FDE and CIE exceeds 32-bit, add a CIE OK)
- In `.eh_frame`, augmentation typically includes R and the FDE encoding is `DW_EH_PE_pcrel | DW_EH_PE_sdata4` for small code models of AArch64, PowerPC64, x86-64.
- `initial_location` has 4 bytes in GCC (even if `-mcmodel=large`). In `.debug_frame`, 64-bit architectures need 8-byte `initial_location`. Therefore, `.eh_frame` is usually smaller than an equivalent `.debug_frame`

[41] see maskray-blog

1.13.4 cfi_startproc

`.cfi_startproc` is used at the beginning of each function that should have an entry in `.eh_frame`.

```
Syntax:
.cfi_startproc [simple]
Table: cfi_pseudo_table
{"cfi_startproc",dot_cfi_startproc,0}
```

The `.cfi_startproc` directive is handled by `dot_cfi_startproc`, that performs following actions:

- Verifies that an `cfi_endproc` has been issued or that we are at the start of the program.
- Allocates and initializes a new FDE.
- If present parses the `simple` argument, and sets an internal flag accordingly.
- If `simple` wasn't present, it generates the initial instructions for the virtual machine, in this case it sets the stack pointer to x [42].

1.13.5 cfi_def_cfa_offset

```
Syntax:
.cfi_def_cfa_offset offset
Table: cfi_pseudo_table
{"cfi_def_cfa_offset",dot_cfi,DW_CFA_def_cfa_offset}
```

`.cfi_def_cfa_offset` modifies a rule for computing CFA. Register remains the same, but offset is new. Note that it is the absolute offset that will be added to a defined register to compute CFA address. In the example of hello.s line 22 we see that the new offset is emitted right after we subtract 16 from the stack. Right after that instruction, the CFA is 16 bytes from the value of `sp`, obviously.

This instruction (and several others) are handled by the `dot_cfi` function that receives as its argument the instruction for the virtual machine.

This function does the following:

- Check that a previous `cfi_startproc` has been issued.
- If the last address wasn't the current address, emit an instruction to advance to the current address.
- And now... a big switch statement that will perform the actions needed for each instruction.

In this case (`DW_CFA_def_cfa_offset`) the code is:

```
1     case DW_CFA_def_cfa_offset:
2         offset = cfi_parse_const();
3         cfi_add_CFA_def_cfa_offset(offset);
4         break;
```

The function `cfi_add_CFA_def_cfa_offset` is as follows:

```
1 /* Add a DW_CFA_def_cfa_offset record to the CFI data. */
2 static void cfi_add_CFA_def_cfa_offset(offsetT offset)
3 {
4     cfi_add_CFA_insn_offset(DW_CFA_def_cfa_offset,offset);
```

[42] This instruction is repeated for each procedure in the program. It would be much easier to set this information in the CIE, since there isn't any program that will switch the stack register on a procedure basis...

```
5     frchain_now→frch_cfi_data→cur_cfa_offset = offset;
6 }
7
8 static void cfi_add_CFA_insn_offset(int insn,offsetT offset)
9 {
10    struct cfi_insn_data *insn_ptr = alloc_cfi_insn_data();
11
12    insn_ptr→insn = insn;
13    insn_ptr→u.i = offset;
14 }
```

Each action is split in several functions, a side-effect of object oriented design. The function `alloc_cfi_insn_data` allocates space for a new data packet.

These data packets are defined like this:

Listing 1.17: cfi_insn_data

```
1 struct cfi_insn_data {
2     struct cfi_insn_data *next;        Linked list
3     int    insn;                       The instruction in question
4     union {                            Depending on the instruction, only one
5         struct {                       of these fields is active.
6             unsigned   reg;
7             offsetT    offset;
8         }      ri;
9         struct {
10            unsigned   reg1;
11            unsigned   reg2;
12        }      rr;
13        unsigned   r;
14        offsetT    i;
15        struct {
16            symbolS       *lab1;
17            symbolS       *lab2;
18        }      ll;
19        struct cfi_escape_data *esc;
20        struct {
21            unsigned   reg   ,encoding;
22            expressionS exp;
23        }      ea;
24        const char    *sym_name;
25    } u;
26 };
```

The function `alloc_cfi_insn_data` let us see immediately how everything is organized:

```
1 static struct cfi_insn_data *alloc_cfi_insn_data(void)
2 {
3     struct cfi_insn_data *insn = XCNEW(struct cfi_insn_data);
4     struct fde_entry *cur_fde_data = frchain_now→frch_cfi_data→cur_fde_data;
5
6     *cur_fde_data→last = insn;       Link the new item in the linked list
7     cur_fde_data→last = &insn→next;
8     SET_CUR_SEG(insn,is_now_linkonce_segment());
9     return insn;
10 }
```

The macro `XCNEW` is just a call to `xcalloc` with a corresponding `sizeof` its argument, that should be a type. It is saved as the current FDE data pointer, added to the linked list. And that is all.

No? You want me to explain to you the impressing code
`SET_CUR_SEG(insn,is_now_linkonce_segment());`
Well, I don't know what it should do, since in asm.h we have the definition:
`#define␣SET_CUR_SEG(structp,seg)␣(void)␣(0␣&&␣seg)` [43]

So, all that complex statement is actually nothing!

1.13.6 cfi_offset

```
Syntax:
.cfi_offset register, offset
Table: cfi_pseudo_table
{"cfi_offset",dot_cfi,DW_CFA_offset}
```

The previous value of register is saved at offset *offset* from the CFA. Processing goes to `dot_cfi` (see above in `cfi_def_cfa_offset`). The relevant lines in `dot_cfi` are:

```
1    case DW_CFA_offset:
2        reg1 = cfi_parse_reg();
3        cfi_parse_separator();
4        offset = cfi_parse_const();
5        cfi_add_CFA_offset(reg1,offset);
6    break;
```

1.13.7 cfi_restore

```
Syntax:
.cfi_restore register [, register]
Table: cfi_pseudo_table
{"cfi_restore",dot_cfi,DW_CFA_restore}
```

The argument is a list of one or more registers. Again, we use the workhorse `dot_cfi`. The relevant lines are below:

```
1    case DW_CFA_restore:
2    for (;;) {
3        reg1 = cfi_parse_reg();
4        cfi_add_CFA_restore(reg1);
5        SKIP_WHITESPACE();
6        if (*input_line_pointer ≠ ',')
7            break;
8        ++input_line_pointer;
9    }
10   break;
```

1.13.8 cfi_def_cfa

```
Syntax:
.cfi_def_cfa register, offset
Table: cfi_pseudo_table
{"cfi_def_cfa",dot_cfi,DW_CFA_def_cfa},
```

`.cfi_def_cfa` defines a rule for computing CFA as: take address from register and add offset to it. The relevant lines in `dot_cfi` are:

[43] Yes, I should eliminate all those fake statements from the code of tiny-asm... but I haven't since it is quite a lot of work, to find them, and to get rid of them. In other CPUs that statement does something, surely.

```
    case DW_CFA_def_cfa:
        reg1 = cfi_parse_reg();
        cfi_parse_separator();
        offset = cfi_parse_const();
        cfi_add_CFA_def_cfa(reg1,offset);
    break;
```

1.13.9 .cfi_endproc

Syntax:
.cfi_endproc
Table: cfi_pseudo_table
{"cfi_endproc",dot_cfi_endproc,0},

.cfi_endproc is used at the end of a function where it closes its unwind entry previously opened by .cfi_startproc and emits it to .eh_frame.

The dot_cfi_endproc procedure is as follows:

```
static void dot_cfi_endproc(int ignored ATTRIBUTE_UNUSED)
{
    if (!cfi_test_startproc()) return;
    last_fde = frchain_now→frch_cfi_data→cur_fde_data;

    cfi_end_fde(symbol_temp_new_now());
    demand_empty_rest_of_line();

    cfi_sections_set = true;
    if ((cfi_sections & CFI_EMIT_target) ≠ 0)
        tc_cfi_endproc(last_fde);
}
```

- Requires a previous open startproc
- sets globals like last_fde, a variable that is set, kept current, but *never used*. It is there just for fun.
  Or not?
  Actually, it is used when SUPPORT_COMPACT_EH is defined. Since this is not supported under the riscv version of GAS, what you see are just leftovers of its former self... [44]
- cfi_end_fde sets several globals to mark the end of a function.
- tc_cfi_endproc is #defined as nothing, so the last two lines are empty.

1.13.10 .cfi_remember_state and .cfi_restore_state

This complementary directives save the current state of the virtual table of register values and restore it later. The usage facilitates cases where a lot of .cfi* directives are issued that need to be ignored for the rest of the code. One example is when we have repeated exit procedures instead of a single one. At each repeated function epilogue there are a lot of .cfi instructions issued:

[44] It can be asked why these variables are still there if they do not fill any purpose. There are several reasons why. The first is that they do not cost a lot of space or execution time. And the second is that, of course, maybe tiny-asm will one day support compact eh_frames, and if the skeleton of places where the variable is set and updated is erased, that would be impossible. And the third one is that I haven't found the time to enclose all usages of that variable in a conditional compilation like SUPPORT_COMPACT_EH what would be actually the correct solution.

```
    beq label
    cfi_remember_state       We save the state here
    ld  ra,24(sp)
    .cfi_restore 1           Several cfi directives
    ld  s0,16(sp)            that describe the function epilogue
    .cfi_restore 8
    .cfi_def_cfa 2, 32
    addi   sp,sp,32
    .cfi_def_cfa_offset 0
    jr  ra
label:
    .cfi_restore_state       We restore it here, voiding all previous cfi directives
    ...
```

Within the assembler those instructions are added using the general utilities for adding `DW_CFA` instructions. Here is `cfi_add_CFA_remember_state`

```
static void cfi_add_CFA_remember_state(void)
{
    struct cfa_save_data *p;

    cfi_add_CFA_insn(DW_CFA_remember_state);

    p = XCNEW(struct cfa_save_data); // Allocate space
    // remember the offset
    p→cfa_offset = frchain_now→frch_cfi_data→cur_cfa_offset;
    // Push it into the top of the list
    p→next = frchain_now→frch_cfi_data→cfa_save_stack;
    frchain_now→frch_cfi_data→cfa_save_stack = p;
}
```

1.14 Risc-v instructions

OK, we know now how to build tiny-asm, how to write directives, how the operations are encoded, let's start now to *do* something with that knowledge. Let's see how the common operations are done.

We will start by showing programs generated by the C compiler. It is the best way to get a feeling for this machine, its instructions and its possibilities.

1.14.1 Loads, stores and addition

Here we cover the basics: loading data from memory, performing an operation, and storing the result in memory again. The riscv is a RISC machine, i.e. like the ARM, it can't work directly on data in memory like the x86 family. Data must be first loaded into memory, before it can be used for calculations.

Consider the following C program:

```
int main(void)
{
    short sa=5,sb=6,sc=sa+sb; // 16 bit addition
    int ia=5,ib=6,ic=ia+ib; // 32 bit
    long long lla=5,llb=6,llc=lla+llb; // 64 bit
    float fa=5,fb=6,fc=fa+fb; // single precision
    double da=5,db=6,dc=da+db;// double precision
    return sc+ic+llc+fc+dc; // Should be 55 isn't it?
}
```

We translate this with:
`gcc -c -S add.c` obtaining the following assembler file:

Listing 1.18: add.s, no optimizations

```
1     .file  "add.c"        // Standard instructions at the beginning of any file.
2     .option pic           // PC relative code
3     .text                 // Ensure code section
4     .align 4              // Align to multiple of 16 bits (2^4 bytes)
5     .globl main           // Visible outside this module
6     .type  main,@function // Debug statement
7 main:
8     addi   sp,sp,-112     // add immediate -112 to the value in the stack
9     sd  s0,104(sp)        // Store doubleword: old frame pointer (s0)
10    addi   s0,sp,112      // Setup the new frame pointer
```

At this point the prologue of this function is finished. The old value of the frame pointer has been saved and a new one established. We start compiling the first C statement.

```
11                          // short sa=5,sb=6,sc=sa+sb; // 16 bit addition
12    li  a5,5              // Put constant 5 in a5
13    sh  a5,-18(s0)        // Store halfword (16 bits)
14    li  a5,6              // Put 6 into a5
15    sh  a5,-20(s0)        // Store it
16    lhu a4,-18(s0)        // Load half word unsigned
```

Note that the compiler uses the "`lhu`" instruction for loading an *unsigned* instead of the correct one `lh` that does a sign extension and is used for loading signed data, as it should be since we have declared the data as `short` and not `unsigned short`!

```
17    lhu a5,-20(s0)        // Same as above
18    addw   a5,a4,a5       // At last! 32 bit aaddition
19    slli   a5,a5,48       // shift left a5 48 bits
20    srli   a5,a5,48       // shift right a5 48 bits
21    sh  a5,-22(s0)        // Store 16 bits: store halfword, sh
```

The compiler emits code to load the data as unsigned, do the addition, and select the lower 16 bits. We can see better what is going on if we follow this sequence in the debugger but using `-5` instead of a positive constant.

```
=> 0x2aaaaaa63e <main+22>: lhu a5,-20(s0)
(gdb) print/x $a4
$2 = 0xfffb
=> 0x2aaaaaa642 <main+26>: addw a5,a5,a4
(gdb) print/x $a5
$3 = 0x6
=> 0x2aaaaaa644 <main+28>: slli a5,a5,0x30
(gdb) print/x $a5
$4 = 0x10001
=> 0x2aaaaaa646 <main+30>: srli a5,a5,0x30
(gdb) print/x $a5
$5 = 0x1000000000000
=> 0x2aaaaaa648 <main+32>: sh a5,-22(s0)
(gdb) print/x $a5
$6 = 0x1
```

We see now that the addition was done in an unsigned form, producing 0x10001, that after the shifts was converted to 1. So, -5 + 6 → 1. We are saved for this time... [45]

```
22                          // int ia=5,ib=6,ic=ia+ib; // 32 bit
23      li  a5,5            // Same as before: 5 into a5
24      sw  a5,-28(s0)      // Initialize "ia" to 5
25      li  a5,6            // Put 6 into a5
26      sw  a5,-32(s0)      // Store it into "ib"
27      lw  a5,-28(s0)      // load ia
28      mv  a4,a5           // copy it to a4
29      lw  a5,-32(s0)      // load "ib"
30      addw    a5,a4,a5    // Do the addition
31      sw  a5,-36(s0)      // store the result
32                          // long long lla=5,llb=6,llc=lla+llb; // 64 bit
33      li  a5,5            // load 5
34      sd  a5,-48(s0)      // Store doubleword this time
35      li  a5,6            //
36      sd  a5,-56(s0)      // Same
37      ld  a4,-64(s0)      // Load "lla" into a4 (directly this time)
38      ld  a5,-48(s0)      // Load "llb" into a5
39      add a5,a4,a5        // 64 bit addition
40      sd  a5,-64(s0)      // Store 64 bits
41                          // float fa=5,fb=6,fc=fa+fb; // single precision
42      lla a5,.LC0         // Load the address of.LC0 into a5
43      flw fa5,0(a5)       // Load single precision from the address in a5
44      fsw fa5,-68(s0)     // Store it at "fa"
45      lla a5,.LC1         // Same process for "fb".
46      flw fa5,0(a5)
47      fsw fa5,-72(s0)     // "fb" at -72
48      flw fa4,-68(s0)     // Load fa4 with "fa"
49      flw fa5,-72(s0)     // Load fa5 with "fb"
50      fadd.s fa5,fa4,fa5  // Add single precision
51      fsw fa5,-76(s0)     // Store result at -76
52                          // double da=5,db=6,dc=da+db; double precision
53      lla a5,.LC2         // Load the address of .LC2 into a5
54      fld fa5,0(a5)       // Load double precision from that address
55      fsd fa5,-88(s0)     // Initialize "da"
56      lla a5,.LC3         // Same for "db"
57      fld fa5,0(a5)
58      fsd fa5,-96(s0)
59      fld fa4,-88(s0)     // Load "da" into fa4
60      fld fa5,-96(s0)     // Load "db" at fa5
61      fadd.d fa5,fa4,fa5  // Add double precision
62      fsd fa5,-104(s0)    // Store result
63                          // return sc+ic+llc+fc+dc; Should be 55
64      lh  a5,-22(s0)      // "sc" into a5
65      sext.w a5,a5        // Sign extend it
66      lw  a4,-36(s0)      // "ic" goes into a4
67      addw    a5,a4,a5    // Add both a,nd accumulate into a5
68      sext.w a5,a5        // Sign extend result to 64 bits
69      mv  a4,a5           // Copy it to a4
70      ld  a5,-64(s0)      // Load "llc" to a5
71      add a5,a4,a5        // Add accumulating into a5
```

[45] Why does the compiler do this instead of loading everything as signed and doing a signed addition? Nobody knows, at least not me. Note that we are using the compiler without any optimizations, we will see later what happens when some of those are turned on.

In any case, the sequence of loading sign extended 16 bit data and making a 32 bit addition gives exactly the same results.

```
72    fcvt.s.l   fa4,a5    // Convert integer in a5 into float in fa4
73    flw fa5,-76(s0)      // Load "fc" into fa5
74    fadd.s fa5,fa4,fa5   // Add single precision fa4 and fa5
75    fcvt.d.s   fa4,fa5   // Convert that result into double precision
76    fld fa5,-104(s0)     // Load "dc" into fa5
77    fadd.d fa5,fa4,fa5   // Add double precision fa4+fa5 --> fa5
78    fcvt.w.d a5,fa5,rtz  // Truncate result into a45
79    sext.w a5,a5         // Sign extend
80    mv  a0,a5            // Put result into the result register
81                         // Start of epilogue --------------------
82    ld  s0,104(sp)       // Restore previous frame pointer
83    addi   sp,sp,112     // Restore stacl
84    jr  ra               // Jump to return address
85                         // End of code of "main"
86    .size  main, .-main  // Compute size of main at assembly time
87    .section   .rodata   // New section: read only data
88    .align 2             // Align to 4 byte boundary
89 .LC0:
90    .word  1084227584    // 5.0 in single precision
91    .align 2
92 .LC1:
93    .word  1086324736    // 6.0 in single precision
94    .align 3             // Align to 8 byte boundary
95 .LC2:
96    .word  0
97    .word  1075052544    // 5.0 in double precision
98    .align 3
99 .LC3:                   // 6.0 in double precision
100   .word  0
101   .word  1075314688
102                        // End of module add.o GNU specific stuff follows
103   .ident "GCC: (GNU) 11.3.0"
104   .section   .note.GNU-stack,"",@progbits
```

This simple program allows us to see the instructions in action. How data is loaded from, and written to memory, how to convert from integer to floating point and vice versa, and how to add in several formats.

- Load from memory all integer data into the a5 register
- Once in memory, copy the data to its eventual destination.
- Target the result of the operations into a5, to save it into memory.

What happens with higher optimization levels? Trying with `gcc -c -S -O1 add.c` we obtain:

```
1 main:
2    li  a0,55
3    ret
```

WOW... there is nothing left even at the lowest optimization level. To avoid this we change the program like this:

```
1 cat add1.c
2 int main(int argc,char *argv[])
3 {
4    short sa=argc,sb=6,sc=sa+sb;
5    int ia=argc,ib=6,ic=ia+ib;
6    long long lla=argc,llb=6,llc=lla+llb;
7    float fa=argc,fb=6,fc=fa+fb;
```

```
8     double da=argc,db=6,dc=da+db;
9     return sc+ic+llc+fc+dc;
10 }
```

The compiler can't possibly know what "argc" will contain and will be forced to do the hard work.

This yields the following program:

Listing 1.19: add1.s

```
1  main:                  // argc is in a0 (first argument)
2     addiw   a5,a0,6     // add argc + 6. Result in a5
3     slliw   a5,a5,16    // 16 bit left shift of result
4     sraiw   a5,a5,16    // 16 bit right shift of result. "sa" is in a5
5     addiw   a4,a0,6     // 32 bit add of argc and 6
6     addw    a5,a5,a4    // Accumulate addition into a5
7     addi    a4,a0,6     // Add 64 bits argc + 6 into a4
8     add a5,a5,a4        // Accumulate into a5
9     fcvt.s.l   fa5,a5   // Convert sum sc+ic+llc to double
10    fcvt.s.w   fa4,a0   // Convert argc into float in fa4
11    flw fa3,.LC0,a5     // auipc instruction
```

This instruction, that the assembly code of gcc represents as "lw" is actually the "auipc" instruction that was introduced to the specifications in 2014, version 2.0. "auipc" adds a 20 bit upper immediate to the program counter to form an address where the data will be loaded. This constant will be filled by the linker, that can establish the definitive distance between the program counter and the variable in question[46].

If you look at the entry of "auipc" in the opcodes table you will find:

```
{"auipc",0,INSN_CLASS_I,"d,u",MATCH_AUIPC,MASK_AUIPC,match_opcode,0},
```

Now, looking at table §1.8 page 43 you will see that the 'u' letter means a 20 bit immediate will be supplied. Our label ".LC1" is precisely that.

But, I hear your question, how come that I see "lw" in the assembler source text and an "auipc" instruction gets written out ???

Well, that the magic of tiny-asm. It will be explained below, after we finish with this small program.

```
12    fadd.s fa4,fa4,fa3   // Add single precision: fa4 = fa4 + fa3
13                         // fa4 contains argc in single precision
14                         // fa3 contains 6
15    fadd.s fa5,fa5,fa4   // Accumulate in fa5 that contains the sum of sc+ic
16    fcvt.d.s   fa5,fa5   // Convert from single precision to double precision.
17    fcvt.d.w   fa4,a0    // Convert argc to double precision
18    fld fa3,.LC1,a5      // The same auipc instruction to acces 6.0 in double prec.
19    fadd.d fa4,fa4,fa3   // Double precision add: fa4 = argc+6.0
20    fadd.d fa5,fa5,fa4   // Add toaccumulator fa5
21    fcvt.w.d a0,fa5,rtz  // Convert to integer
22    sext.w a0,a0         // sign extend
23    ret                  // Done.
24    .size  main, .-main
25    .section   .rodata.cst4,"aM",@progbits,4
26    .align 2
27 .LC0:
28    .word  1086324736
29    .section   .rodata.cst8,"aM",@progbits,8
30    .align 3
```

[46] **A**dd **U**pper **I**mmediate to **P**rogram **C**ounter → **AUIPC**.

```
31 .LC1:
32     .word  0
33     .word  1075314688
```

We see here what it means to optimize:

- The compiler keeps all data in registers, there isn't even a stack frame.
- More operations do actual calculations than loading or storing data from/to memory. In the unoptimized version of add.c we have only 15 out of 76 instructions that do arithmetic. In the optimized version we have 15 out of 33, mainly because there are so few loads and no stores
- Use of more advanced instructions

Load and store instructions in short

Table 1.15: Standard load and store operations

Instruction	Description
`lb rd, imm12(rs1)`	Load 8 bits. Sign extension.
`lb rd,lab`	Macro: Load 8 bits from address `lab`.
`sb rs2, imm12(rs1)`	Store 8 bits at rs1+imm12
`sb rs1, lab,rs2`	Macro: store 8 bits at lab's address. Contents of rs2 destroyed
`lbu rd, imm12(rs1)`	Load 8 bits. Zero extension
`lh rd, imm12(rs1)`	Load 16 bits Sign extension
`sh rs2, imm12(rs1)`	Store 16 bits
`lhu rd, imm12(rs1)`	Load 16 bits. Zero extension
`lw rd, imm12(rs1)`	Load 32 bits Sign extension
`sw rs2, imm12(rs1)`	Store 32 bits
`lwu rd, imm12(rs1)`	Load 32 bits Zero extension
`ld rd, imm12(rs1)`	64 bit load
`sd rs2, imm12(rs1)`	Store 64 bits at rs1+imm12
`lui rd,imm20`	Load a 20 bit address constant into rd.
`auipc rd,imm20`	Adds the 20 bit immediate to the program counter and stores the result in rd.

The `imm12` is always sign extended.

Addressing modes

- Absolute addressing.

```
lui  a0, %hi(message)
addi a0, %lo(message)
```

  The `%hi` and the `%lo` constructs mean the higher 20 and the lower 12 bits of the address.

- Relative addressing

```
auipc  a0, %pcrel_hi(msg + 1)
addi   a0, a0, %pcrel_lo(message)
```

- GOT (Global Object Table) relative addressing

```
.L1:
   auipc  a0, %got_pcrel_hi(message)
   ld     a0, %pcrel_lo(.L1)(a0)
```

  Note that the last two are the same: either PC relative or GOT relative, the instructions

Recognizing addressing modes

The assembler recognizes these keywords using tables of the following structure:

```
struct percent_op_match {
    const char *str; // Name without the percentage sign
    bfd_reloc_code_real_type reloc; // Relocation type invoked
};
const struct percent_op_match percent_op_utype[];
const struct percent_op_match percent_op_itype[];
const struct percent_op_match percent_op_stype[];
const struct percent_op_match percent_op_rtype[];
const struct percent_op_match percent_op_null[];
```

These tables will be used in the function `parse_relocation` to recognize (or not) a relocation directive.

```
/* Return true if *STR points to a relocation operator. When returning true, move
* *STR over the operator and store its relocation code in *RELOC. Leave both *STR
* and *RELOC alone when returning false. */
bool parse_relocation(char **str,bfd_reloc_code_real_type * reloc,
                      const struct percent_op_match *percent_op)
```

This function will set up a pointer to the first table, and will scan each name in all the tables, assuming they are in consecutive order. The last "table" is a terminator with only zeroes. It is crucial then, that, unaware of this, you insert something in between those tables. That would totally screw up things...

`parse_relocation` will be called when parsing an expression that should yield a small immediate constant ot offset. Its single use will be in `my_getSmallExpression`.

```
/* Parse string STR as a 16-bit relocatable operand. Store the expression in
* *EP and the relocation,if any,in RELOC. Return the number of relocation
* operators used (0 or 1).
*
* On exit,EXPR_PARSE_END points to the first character after the expression. */
size_t my_getSmallExpression(expressionS * ep, bfd_reloc_code_real_type * reloc,
                             char *str,const struct percent_op_match *percent_op)
```

Now, this function, `my_getSmallExpression` will be called from two places:

1. `my_getOpcodeExpression`, a function used in `riscv_ip`.
2. `riscv_ip` directly, and in an extensive fashion.

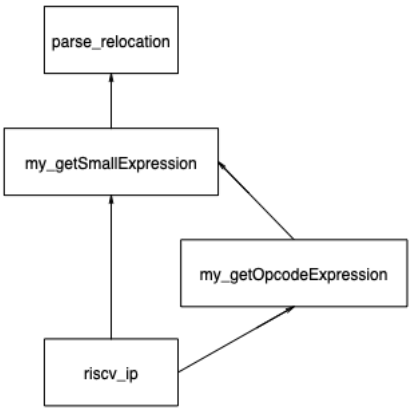


Figure 1.16: Who calls the `parse_relocation`hn,vc=:m:= function

1.14.2 Digression: assembler macros

We have seen above that the expression `flw fa3,.LC0,a5` gets translated into two instructions:

```
1    auipc a5,0x0
2    flw fa3,52(a5)
```

Looking at the opcode table, we find that there several entries for the "flw" instruction.

```
1 {"lw",0,INSN_CLASS_C,"d,Cm(Cc)",MATCH_C_LWSP,MASK_C_LWSP, \
2            match_rd_nonzero,INSN_ALIAS|INSN_DREF|INSN_4_BYTE},
3 {"lw",0,INSN_CLASS_C,"Ct,Ck(Cs)",MATCH_C_LW,MASK_C_LW,\
4            match_opcode,INSN_ALIAS|INSN_DREF|INSN_4_BYTE},
5 {"lw",0,INSN_CLASS_I,"d,o(s)",MATCH_LW,MASK_LW, \
6            match_opcode,INSN_DREF|INSN_4_BYTE},
7 {"lw",0,INSN_CLASS_I,"d,A",0,(int)M_LW,match_never,INSN_MACRO},
```

At line 7, we find an instruction whose flag field has the `INSN_MACRO` set. In the `md_assemble` function, we find the sequence:

```
if (insn.insn_mo→pinfo == INSN_MACRO)
    macro(&insn,&imm_expr,&imm_reloc);
else
    append_insn(&insn,&imm_expr,imm_reloc);
```

If this instruction is actually a macro, expand it, if not, append the new instruction.

What does the `macro` procedure do?

- It decomposes its arguments into 4 parts: the destination register (rd), the two source registers (rs1 and rs2), and a mask. According to the mask, different actions are performed. In our case we have `M_LW`, as wecan see in line 7 of the opcodes listing above. In the same line we find that the function for matching the opcode is `match_never` a function that will always fail, excluding that the macro will be understood as another opcode. [47]
- Using the mask value, it dispatches in a long `switch` statement for each mask. In our case:

```
case M_LW:
   pcrel_load(rd,rd,imm_expr,"lw",
```

[47] `M_LW` is a member of an anonymous enumeration defined in asm.h

```
                BFD_RELOC_RISCV_PCREL_HI20, BFD_RELOC_RISCV_PCREL_LO12_I);
        break;
```

`pcrel_load` and its companion `pcrel_store` call `pcrel_access` with slightly different arguments:

Listing 1.20: pcrel load and store

```
1  void pcrel_load(int destreg,int tempreg,expressionS * ep,const char *lo_insn,
2          bfd_reloc_code_real_type hi_reloc, bfd_reloc_code_real_type lo_reloc)
3  {
4      pcrel_access(destreg,tempreg,ep,lo_insn,"d,s,j",hi_reloc,lo_reloc);
5  }
6
7  void pcrel_store(int srcreg,int tempreg,expressionS * ep,const char *lo_insn,
8           bfd_reloc_code_real_type hi_reloc,bfd_reloc_code_real_type lo_reloc)
9  {
10     pcrel_access(srcreg,tempreg,ep,lo_insn,"t,s,q",hi_reloc,lo_reloc);
11 }
```

And, to make a disgression within a disgression, long and explicit type names can be nice, but sometimes they can lead to *really* verbose code... What if we substitute in the code above the long names with something like `Reloc` ?

Listing 1.21: pcrel load and store improved

```
1      void pcrel_load(int destreg,int tempreg,expressionS * ep,const char *lo_insn,
2                  Reloc hi_reloc, Reloc lo_reloc)
```

Is this code less lisible?

Anyway, both functions call `pcrel_acces` [48]

```
1  static void pcrel_access(int destreg,int tempreg,expressionS * ep,
2                       const char *lo_insn,const char *lo_pattern,
3                 bfd_reloc_code_real_type hi_reloc,bfd_reloc_code_real_type lo_reloc)
4  {
5      expressionS ep2;
6      ep2.X_op = O_symbol; // expression is a symbolic expression
7      ep2.X_add_symbol = make_internal_label(); // Symbol to attach the relocation
8      ep2.X_add_number = 0;
9      macro_build(ep,"auipc","d,u",tempreg,hi_reloc); // First insn
10     macro_build(&ep2,lo_insn,lo_pattern,destreg,tempreg,lo_reloc); // Second
11 }
```

`pcrel_access` builds a symbolic expression and calls `macro_build` twice. The first one to build the `auipc` instruction, and the second for the actual load using the temporary register. The function `macro_build` receives as arguments:

1. An expression.
2. A name for the instruction to generate.
3. A format string that will be used, in a similar manner to printf, as a template for the extraction of the corresponding arguments from the rest.

[48] The problem with `bfd_reloc_code_real_type` (besides the fact that is a pain to type!) is that many of the words used do not convey any new information... Real type? Are other types "unreal"? What did they want to say?

In our case we give it first the `.LC0` label, the name of the first instruction that we want to generate ("auipc"), and a format string of 'd' and 'u'.

The meaning of those letters is as follows:

Table 1.16: Macro letter arguments

Letter	action
'V'	Vector macro. It needs a further letter for fully specifying which action is needed.
'd'	`INSERT_OPERAND(RD,insn,va_arg(args,int)); continue;`
's'	`INSERT_OPERAND(RS1,insn,va_arg(args,int)); continue`
't'	`INSERT_OPERAND(RS2,insn,va_arg(args,int)); continue`
'q','u' and 'j'	`r=va_args(args,int); continue;` "r" is the relocation type".

Then, just before exiting, `macro_build` will call: `append_insn(&insn,ep,r);`

Let's see the output of `objdump` when we ask for disassembly and relocations:

```
 $ # First we generate the object file
 $ ./asm -o tauipc.o tauipc.s
 $ # Let's look at it with objdump
 $ objdump -d -r tauipc.o
0000000000000000 <main>:
0:  00000697          auipc  a3,0x0
0: R_RISCV_PCREL_HI20 .LC0
0: R_RISCV_RELAX  *ABS*
4:  0006a787          flw fa5,0(a3) # 0 <main>
4: R_RISCV_PCREL_LO12_I .L0
4: R_RISCV_RELAX  *ABS*
```

As expected, we have a relocation of 20 bits and another one for the next instruction for the lower 12 bits. There are also 'relax"' relocations, that we will meet later, when we study relocations.

1.14.3 Subtraction

Replacing all additions with subtractions in our C source doesn't change much to the overall shape of the program. The subtraction instructions are:

- `sub`. 64 bit subtraction.
  Syntax: `sub rd,rs1,rs2`
  Operation: `rd` ← `rs1 - rs2`.
- The `subw` instruction does a 32 bit subtraction. Same syntax and operation as above.
- The `fsub.s` does a single precision subtraction
  Syntax: `fsub.s fd,fs1,fs2`
  Operation: `fd` ← `fs1 - fs2`
- The `fsub.d` instruction does a double precision subtraction. Same as above.

1.14.4 Comparisons

- `slti`. Set less than immediate. (Signed)
  Syntax: `slti rd,rs1,immediate`
  Operation: `rd` ← `(rs1 < immediate) ?  1 :  0`

- `sltiu` Set less than immediate unsigned.
  Syntax:
  `sltiu rd, rs1, imm`
  `sltu rd,rs1,rs2`
  Operation: `rd` ← `(rs1 < rs2/imm ) ? 1 : 0`

- The pseudo instruction `SEQZ rd,rs` sets rd to 1 if rs is equal to zero. This is actually an alias for `sltiu rd,rs,1`.

- `flt.s` and `flt` perform floating point comparisons for single and double precision floating point respectively.
  Syntax: `flt rd,fsrc1,fsrc2`
  Operation: `rd` ← `fsrc1 < fsrc2) ? 1 : 0`
  `rd` is an integer register, `fsrc1` and `fsrc2` are floating point.

- `feq` and `feq.s` do an equality comparison.
  Syntax: `feq rd,fsrc1,fsrc2`
  Operation: `rd` ← `(fsrc1 == fsrc2) ? 1 : 0`
  `rd` must be an integer register, `fsrc1` and `fsrc2` are floating point.

An instruction alias that uses subtraction is `neg` that is actually just `sub rd,x0,rs1` i.e. subtract rs1 from zero.

1.14.5 Multiplication and Division

Multiplication

These instructions are present if the processor implements the 'M' extension.

- `mul` performs a 64 by 64 bits multiplication, returning the lower 64 bits.
- `mulh` performs a signed 64 bit by a signed 64 bit multiplication and returns the higher 64 bits of the result.
- `mulhu` multiplies unsigned by unsigned 64 bit quantities and returns the upper 64 bits.
- `mulhsu` multiplies a signed rs1 by an unsigned rs2 and returns the higher 64 bits. [49]
- `mulw` is a 32 bit multiplication. The lower 32 bits are returned, with sign extension.

XuanTie-OpenC910

This processor features several new instructions for multiplication.

Table 1.17: Thead Multiplication extensions

Instruction	Operation	Description
`th.mula rd,rs1,rs2`	$rd \leftarrow rd + (rs1 \times rs2)$	Accumulate in `rd`
`th.mulah rd,rs1,rs2`	$t[0:31] \leftarrow rd + (rs1[0:15] \times rs2[0:15])$ $rd \leftarrow sign\_extend(t)$	Accumulate with result of 16 bit multiplication.

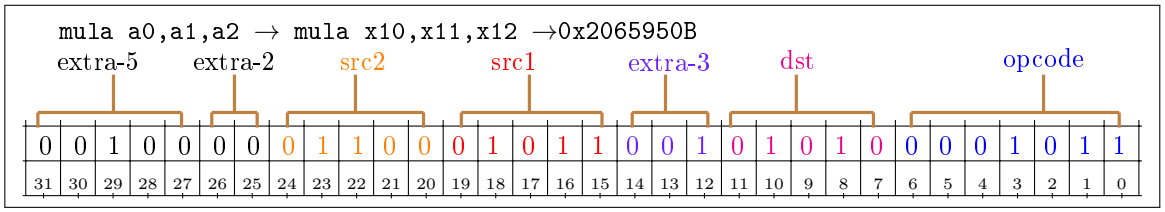
[49] In a multiple precision context, this instruction can be used to multiply the higher 64 bits that contain the sign, with the lower 64 bits of the other multiplicand, that has no sign.

Table 1.17: Thead Multiplication extensions

`th.mulaw rd,rs1,rs2`	$t[0:31] \leftarrow rd + (rs1[0:31] \times rs2[0:31])$ $rd \leftarrow sign\_extend(t)$	Accumulate with result of 32 bit multiplication.
`th.muls rd,rs1,rs2`	$rd \leftarrow rd - (rs1 \times rs2)$	Subtract from `rd` the result of the multiplication
`th.mulsh rd,rs1,rs2`	$t[0:31] \leftarrow rd - (rs1[0:15] \times rs2[0:15])$ $rd \leftarrow sign\_extend(t)$	Subtract from rd result of 16 bit multiplication.
`th.mulsw rd,rs1,rs2`	$t[0:31] \leftarrow rd - (rs1[0:31] \times rs2[0:31])$ $rd \leftarrow sign\_extend(t)$	Subtract from rd result of 32 bit multiplication.

These operations are encoded using a modified form of the "R" format. Here is the encoding for the `mula` instruction for instance: `mula a0,a1,a2`.

Figure 1.17: Modified C910 **R** Instruction layout



As you can see, the `extra-7` field of the "R" format has been split into a 5+2 bit field. The meaning of those 2 bits is described in §1.14.14 page 96

Division

A change of the standard allows now to implement processors that have multiplication but not division. For those that do feature division, we have:

- `div` features a signed 64 bit division with rounding towards zero.
  Syntax:
  `div rd,rs1,rs2`
  Operation:
  `rd ← rs1 / rs2`

- `divu` Unsigned division. Syntax and mode of operation the same as DIV.

- `rem` Signed remainder
  Syntax:
  `rem rd,rs1,rs2`
  Operation:
  `rd ← rs1 % rs2`

- `remu` Unsigned remainder. Same as `REM` but for unsigned data.

- `remw` and `remuw` 32 bit versions.

Division by zero returns a result with all bits set, without any trap. [50]

The riscv ISA doesn't provide an instruction for calculating the remainder and the division with only one division operation. The sequence: `DIV[U] rdq, rs1, rs2; REM[U] rdr, rs1, rs2` ()where rdq can't be the same as rs1 or rs2) is proposed for optimization.

1.14.6 Shifts

Table 1.18: Standard shift operations

Syntax	Operation
`slli rd,rsrc1,imm`	Shift left logical immediate.
`srli rd,rsrc1,imm`	Shift right logical immediate (Shifts in zeros)
`srai rd,rsrc1,imm`	Shift right arithmetic (propagating the sign bit).
`sll rd,rsrc1,rsrc2`	Shift left logical (shifts in zeroes).
`sllw rd,rsrc1,rsrc2`	As `sll` but works on lower 32 bits.
`srl rd,rsrc1,rsrc2`	Shift right logical (shifts in zeroes).
`srlw rd,rsrc1,rsrc2`	As `srl` but works on lower 32 bits.
`sra rd,rsrc1,rsrc2`	Shift right arithmetic (propagating the sign bit).
`sraw rd, rsrc1, rsrc2`	As `sra` but works on lower 32 bits.

In all this instructions `rsrc1` is the quantity to be shifted, and `rsrc2` or `imm` contain the number of bits to shift.

1.14.7 Control flow

Inconditional Jumps

Table 1.19: Standard inconditional jumps

Pseudo instruction	Base instruction	Operation
`j label`	`jal x0 label`	Jump inconditional `pc← pc+sign_extend(imm20 * 2)`
`jal fn`	`jal x1,fn`	Call subroutine
`jr register`	`jalr x0,register`	Call function pointer in register

The `jal` instructions uses the 'j' instruction format (See §1.7.6 page 34). The offset immediate (in multiples of 2 bytes) is added to the current program counter value to form the target address. It has a reach of 1MB forward or backwards.

[50] The riscv standard justifies this with:

> We considered raising exceptions on integer divide by zero, with these exceptions causing a trap in most execution environments. However, this would be the only arithmetic trap in the standard ISA (floating-point exceptions set flags and write default values, but do not cause traps) and would require language implementors to interact with the execution environment's trap handlers for this case. Further, where language standards mandate that a divide-by-zero exception must cause an immediate control flow change, only a single branch instruction needs to be added to each divide operation, and this branch instruction can be inserted after the divide and should normally be very predictably not taken, adding little runtime overhead.The value of all bits set is returned for both unsigned and signed divide by zero to simplify the divider circuitry.
>
> The value of all 1s is both the natural value to return for unsigned divide, representing the largest unsigned number, and also the natural result for simple unsigned divider implementations. Signed division is often implemented using an unsigned division circuit and specifying the same overflow result simplifies the hardware.

The indirect jumps through a register are **not** in multiples of two bytes, beware. The address must be the real address of the target.

1.14.8 Conditional expressions

Table 1.20: Standard conditional expressions

Inst	Operation
`beq rs1, rs2, label`	`if (rs1 =` $=$ `rs2) pc←pc+sign_extend(imm12 << 1)`
`bge rs1, rs2, label`	`if (rs1` $\geq$ `rs2) pc←pc+sign_extend(imm12 << 1)`
`bgeu rs1, rs2, label`	`if (rs1` $\geq$ `rs2) pc←pc+sign_extend(imm12 << 1)`
`blt rs1, rs2, label`	`if (rs1` $\leq$ `rs2) pc←pc+sign_extend(imm12 << 1)`
`bltu rs1, rs2, label`	`if (rs1` $\leq$ `rs2) pc←pc+sign_extend(imm12 << 1)`
`bne rs1, rs2, label`	`if (rs1` $\neq$ `rs2) pc←pc+sign_extend(imm12 << 1)`

All these instructions have a range of $\pm$4K.

Exercise 1: *The instruction `bgt` is an alias. How would you build it from the other instructions?*

Exercise 2: *Write a small program that uses a conditional branch.*

Exercise 3: *Disassemble the program. What you see instead of bgt?*

Exercise 4: *How is the change achieved? Look at the source asm.c.*

1.14.9 And, Or, Xor

Table 1.21: Standard boolean instructions

Inst	Operation
`and rd,rsrc1,rsrc2`	$rd \leftarrow rsrc1 \wedge rsrc2$
`andi rd,rsrc1,imm12`	$rd \leftarrow rsrc1 \wedge imm12$
`or rd,rsrc1,rsrc2`	$rd \leftarrow rsrc1 \vee rsrc2$
`ori rd,rsrc1,imm12`	$rd \leftarrow rsrc1 \vee imm12$
`xor rd,rsrc1,rsrc2`	$rd \leftarrow rsrc1 \oplus rsrc2$
`xori rd,rsrc1,imm12`	$rd \leftarrow rsrc1 \oplus imm12$

Exercise 5: *Use the XOR instruction to invert all bits in an integer register*

1.14.10 Reading timers

The "Zinctr" extension prescribes at least 3 counters/timers that should be present in all implementations.

- Cycles.The `rdcycle` pseudo instruction reads the low XLEN bits of the cycle special register which holds the number of clock cycles executed by the processor core on which the hardware thread is running from an arbitrary start time somewhere in the past, probably, when the machine was powered on.
- Time. The `rdtime` instructions returns the wall clock time since start, sometime in the past.

- Instructions retired. The `rdinstret` instruction returns the number of instructions retired, i.e. executed (roughly) since some time in the past.

Reading standard counters

Table 1.22: Counter reading

Instruction	Description
`rdtime rd`	Reads a 64 bit timer counter
`rdcycle rd`	Reads a 64 bit cycle counter
`rdinstret rd`	Reads a 64 bit counter for the number of instructions retired, i.e. executed

The `rd` placeholder represents a 64 bit register.

Exercise 6: *Write a program in assembler to print these 3 counters.*

Exercise 7: *Try to verify that time corresponds to a time measure*

1.14.11 CSR instructions

"CSR" stands for **C**ontrol and **S**tatus **R**egister. These registers are used primarily in the privileged part of the instruction set, but there are some uses in the unprivileged instructions (the subject of this book).

1.14.12 Boolean instructions

These instructions correspond to the `"Zbb"`. In the opcode table they have `INISN_CLASS_ZBB` in the class field. extension.

The instructions that work only in a 32 bit environment have been excluded.

The Sifive U74-MC supports the standard Zbb extension. The XuanTie-OpenC910 has two somehow similar instructions, `"ff1"` and `"ff0"`.

Note: GCC doesn't recognize these instructions in machines using the U74 CPU. You have to force it by adding the (undocumented) option `-march=rv64gc_zbb` to the compilation command line. These problems do not affect tiny-asm: it will generate the correct instructions without problems.

Table 1.23: Zbb boolean extension instructions

Inst.	Description
`bclr rd,rs1,rs2`	Returns rs1 with a single bit cleared at the index specified in rs2. The index is read from the lower log2(XLEN) bits of rs2.
`bclri rd,rs1,nr`	Returns rs1 with a single bit cleared at the index specified in nr. The index is read from the lower log2(XLEN) bits of nr.
`bext rd, rs1, rs2`	Returns a single bit extracted from rs1 at the index specified in rs2. The index is read from the lower log2(XLEN) bits of rs2.
`bexti rd, rs1, nr`	Returns a single bit extracted from rs1 at the index specified in nr. The index is read from the lower log2(XLEN) bits of nr.
`binv rd, rs1, rs2`	Returns rs1 with a single bit inverted at the index specified in rs2
`binvi rd, rs1,nr`	Returns rs1 with a single bit inverted at the index specified in nr.

Table 1.23: Zbb boolean extension instructions

`bset rd, rs1,rs2`	Returns rs1 with a single bit set at the index specified in rs2.
`bseti rd, rs1,nr`	Returns rs1 with a single bit set at the index specified in nr.
`clmul rd,rs1,rs2`	clmul produces the lower half of the 2×XLEN carry-less product.
`clmulh rd, rs1, rs2`	Produces the upper half of the 2× XLEN carry-less product.
`clmulr`	Produces bits 2× XLEN-2 to XLEN-1 of the 2×XLEN carry-less product. [51]
`clz rd,rs1`	Counts the number of 0 bits before the first 1 bit, starting at the most significant bit and progressing to bit 0. If the input is 0, the output is 64. If the most-significant bit of the input is 1, the output is 0.
`clzw rd,rs1`	Counts the number of 0 bits before the first 1 bit, starting at bit 31 and progressing to bit 0. If the least-significant word is 0, the output is 32. If the most-significant bit of the word is 1, the output is 0.
`ctz rd,rs1`	Counts the number of 0 bits before the first 1 bit, starting at the least-significant bit and progressing to the most-significant bit. If the input is 0, the output is 64. If the least-significant bit of the input is 1, the output is 0.)
`ctzw rd,rs1`	Counts the number of 0 bits before the first 1 bit, starting at the least-significant bit and progressing to the most-significant word. If the least significant word is 0, the output is 32. If the least significant bit of the input is 1, the output is 0.
`cpop rd,rs1`	Counts the number of 1 bits in the source register. This operations is also known as "population count" or "Hamming weight".
`cpopw rd,rs1`	Counts the number of 1 bits in the least-significant word of the source register
`max rd,rs1,rs2`	Returns the larger of two signed integers.
`maxu rd,rs1,rs2`	Returns the larger of two unsigned integers.
`min rd,rs1,rs2`	Returns the smaller of two signed integers.
`minu rd,rs1,rs2`	Returns the smaller of two unsigned integers.
`orc.b rd,rs`	Combines the bits within each byte using bitwise logical OR. This sets the bits of each byte in the result rd to all zeros if no bit within the respective byte of rs is set, or to all ones if any bit within the respective byte of rs is set.
`orn rd,rs1,rs2`	performs the bitwise logical OR operation between rs1 and the bitwise inversion of rs2. $rd \leftarrow rs1 \mid \sim rs2$
`rev8 rd,rs1`	Reverses the order of the bytes in rs1.
`rol rd,rs1,rs2`	Rotate left. Performs a rotate left of rs1 by the amount in least-significant 6 bits of rs2.
`rolw rd,rs1,rs2`	Rotate left. Performs a rotate left of rs1 by the amount in least-significant 5 bits of rs2. The resulting 32 bit vvalue is sign extended to 64.

[51] The clmulr instruction is used to accelerate CRC calculations. The r in the instruction's mnemonic stands for reversed, as the instruction is equivalent to bit-reversing the inputs, performing a clmul, then bit-reversing the output.

Table 1.23: Zbb boolean extension instructions

`ror rd,rs1,rs2`	Rotate right. Uses the least significant 6 bits of rs2 for the amount to rotate.
`rori rd,rs1,imm`	Rotate right immediate. Uses the least significant 6 bits of *imm* for the amount to rotate.
`sext.b rd,rs`	Sign-extends the least-significant byte in the source to 64 by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits.
`sext.h rd,rs`	Sign-extends the least-significant 16 bits in the source to 64 by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits.
`sh1add rd,rs1,rs2`	Shifts rs1 left by 1 and adds it to rs2.
`sh2add rd,rs1,rs2`	Shifts rs1 left by 2 and adds it to rs2.
`sh1add_uw rd,rs1,rs2`	This instruction performs an 64 bit wide addition of two addends. The first addend is rs2. The second addend is the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 1 place.
`sh2add_uw rd,rs1,rs2`	Same as above but the shift is by 2 places.
`sh3add_uw rd,rs1,rs2`	Same as above but the shift is 3 places.
`slli.uw rd, rs1, imm6`	Takes the least-significant word of rs1, zero-extends it, and shifts it left by the immediate.
`xnor rd, rs1, rs2`	Performs the bit-wise exclusive-NOR operation on rs1 and rs2. `rd = ~(rs1 ^ rs2);` [52]

Exercise 8: *Use the "max" instruction to calculate the absolute value of a signed integer.*

The Zbkb extension

Table 1.24: Zbkb instructions

Inst.	Description
`pack rd,rs1,rs2`	Packs the XLEN/2-bit lower halves of rs1 and rs2 into rd, with rs1 in the lower half and rs2 in the upper half.
`packh rd,rs1,rs2`	Packs the least-significant bytes of rs1 and rs2 into the 16 least-significant bits of rd, zero extending the rest of rd.
`packw rd,rs1,rs2`	packs the low 16 bits of rs1 and rs2 into the 32 least-significant bits of rd, sign extending the 32-bit result to the rest of rd.
`rvb rd,rs1`	Reverses the order of the bits in every byte of a register.
`xperm.b rd, rs1, rs2,`	The xperm.b instruction operates on bytes. The rs1 register contains a vector of 8 8-bit elements. The rs2 register contains a vector of 8 8-bit indexes. The result is each element in rs2 replaced by the indexed element in rs1, or zero if the index into rs2 is out of bounds. This instruction is in the extension `Zxbkx`.

[52] The XNOR operation of two inputs returns 1 if the two inputs are equal, zero otherwise.

Table 1.24: Zbkb instructions

`xperm.n rd, rs1, rs2`	The xperm.n instruction operates on nibbles. The rs1 register contains a vector of XLEN/4 4-bit elements. The rs2 register contains a vector of XLEN/4 4-bit indexes. The result is each element in rs2 replaced by the indexed element in rs1, or zero if the index into rs2 is out of bounds. This instruction is in the extension `Zxbkx`.
`zext.h rd, rs`	This instruction zero-extends the least-significant halfword of the source to XLEN by inserting 0's into all of the bits more significant than 15.

1.14.13 Pause instruction

```
Syntax:
pause
```

The `pause` instruction is a HINT that indicates the current hart's rate of instruction retirement should be temporarily reduced or paused. The duration of its effect must be bounded and may be zero. No state is changed. The standard says about this:

> Software can use the PAUSE instruction to reduce energy consumption while executing spin-wait code sequences. Multithreaded cores might temporarily relinquish execution resources to other harts when PAUSE is executed. It is recommended that a PAUSE instruction generally be included in the code sequence for a spin-wait loop.

Exercise 9: *Calculate how long takes a pause instruction in your machine*

1.14.14 Floating point

Floating point operations are controlled with the status register, `fcsr`. It is a 32-bit read-/write register that selects the dynamic rounding mode for floating-point arithmetic operations and holds the accrued exception flags.

- Bit 0: NX Inexact
- Bit 1: UF Underflow
- Bit 2: OF Overflow
- Bit 3: DZ Divide by zero
- Bit 4: NV Invalid operation
- Bits 5-7: Rounding mode. This will be used when the instruction uses the dynamic rounding mode. See §1.26 page 98.
- Bits 8-31 Reserved.

The `fcsr` register can be read and written with the FRCSR and FSCSR instructions, which are assembler pseudo instructions, built on the underlying CSR access instructions.

The fields of the `csr` can also be accessed individually. The instruction `frrm` reads the rounding mode field. The instruction `fsrm` writes to it. In a similar fashion `frflags` and `fsflags` read and write to the flags field.

```
Syntax:
    frrm rd          rd← rounding mode
    fsrm rd,rs1      rounding mode ← rs1, rd ← old rounding mode
    frflags rd       rd ← flags
    fsflags rd,rs1   flags ← rs1, rd ← old flags
```

Exercise 10: *Write an assembler program to show the CSR flags in the console*

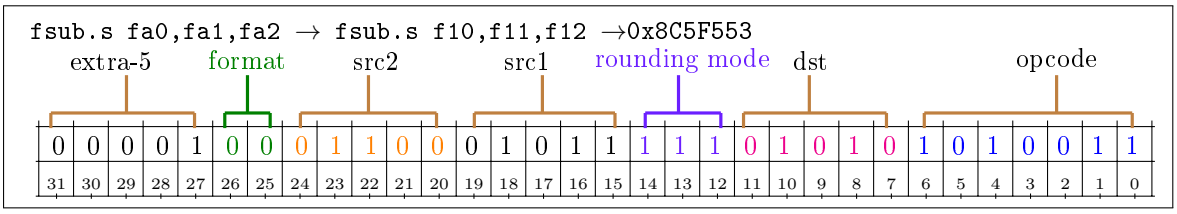
Exercise 11: *Write a subroutine that returns the flags of the CSR as a 32 bit integer*

Floating point can have 4 possible precision settings: half (16 bits), single (32), double (64) and quad(128). Most machines implement single and double precision, some implement half precision, and (till now) none has implemented 128 bit precision.

Encodings

Floating point instructions use a slightly modified "R" format. Bits 25 and 26 define a "format" field that is used to differenciate between half, single and double precision.

Figure 1.18: Modified **R** Instruction layout



In the figure above we have:

- Bits 0-6, the opcode, 83 (0x53).
- Bits 7-11, the destination, 10 (0xA)
- Bits 12-14, the rounding mode, 7 (Dynamic rounding mode)
- Bits 15-19, the first source register, 11 (a1)
- Bits 20-24, the second source register 12 (a2)
- Bits 25-26, The format, in this case 0, single precision
- Bits 27-31, The code for the operation, in this case 1, the code for FADD or FSUB.

Table 1.25: Format bits (Bits 25-26)

Bits value	Instruction mnemonic	Description
0 0	S	Single precision
0 1	D	Double precision
1 0	H	Half precision
1 1	Q	128 bit Quad precision

Table 1.26: Rounding mode bits (Bits 12-14)

Bits value	Mode mnemonic	Description
000	RNE	Round to nearest
001	RTZ	Round to zero.
010	RDN	Round down towards $-\infty$
011	RUP	Round up towards $+\infty$

Table 1.26: Rounding mode bits (Bits 12-14)

100	RMM	Round to nearest. Ties towards max magnitude
101	Reserved	
110	Reserved	
111	DYN	If in the instruction, selects dynamic rounding mode. If in the rounding mode register, it is *reserved*

The recognition of the rounding modes for an instruction is done in the case for the letter 'm', in the `riscv_ip` function. It uses the `riscv_rm` table of rounding modes.

```
/* Table of legal rounding modes. */
const char *const riscv_rm[8] = {"rne","rtz","rdn","rup","rmm",0,0,"dyn"};
/* Code snippet for case 'm': rounding modes in riscv_ip */
    case 'm':  /* Rounding mode. */
        if (arg_lookup(&asarg,riscv_rm,
            ARRAY_SIZE(riscv_rm),&regno)) {
            INSERT_OPERAND(RM,*ip,regno);
            continue;
        }
    break;
```

Floating point instructions

Single precision floating point is called extension "F", double "D" and half "H". There is provision for a "Q" extension for 128 bit numbers but none of the machines I have used implements that yet.

In the instructions below, an adddress is formed by adding the contents of the source register with a sign extended imm12.

Table 1.27: Floating point load/store instructions

Instruction	Description
`flw frd,imm12(fs1)`	Load single precision data from address at into frd.
`fsw fs2,imm12(rs1)`	Store single precision data from fs2 at address
`fld frd,imm12(fs1)`	Load double precision data from address at into frd
`fsd fs2,imm12(rs1)`	Store double precision data from fs2 at address
`flh frd,imm12(fs1)`	Load half precision data from address at into frd
`fsh fs2,imm12(rs1)`	Store half precision data from fs2 at address

In the instructions above the data will be moved without any changes.

Table 1.28: Floating point arithmetic instructions

Instruction	Description
`fadd.{h|s|d} frd,frs1,frs2`	Add. $frd \leftarrow frs1 + frs2$
`fsub.{h|s|d} frd,frs1,frs2`	Subtraction. $frd \leftarrow frs1 - frs2$
`fmul.{h|s|d} frd,frs1,frs2`	Multiplication. $frd \leftarrow frs1 \times frs2$
`fdiv.{h|s|d} frd,frs1,frs2`	Division. $frd \leftarrow frs1 \div frs2$

Table 1.28: Floating point arithmetic instructions

`fmadd.{h|s|d}` `fd,fs1,fs2,fs3`	Fused multiply add. $fd \leftarrow (fs1 \times fs2) + fs3$
`fmsub.{h|s|d}` `fd,fs1,fs2,fs3`	Fused multiply subtract. $fd \leftarrow (fs1 \times fs2) - fs3$
`fnmadd.{h|s|d}` `fd,fs1,fs2,fs3`	Fused negative multiply add simple precision. $fd \leftarrow (-fs1 \times fs2) - fs3$ [53] !
`fnmsub.{h|s|d} fs1,fs2,fs3`	Fused negative subtraction simple precision. $fs1 \leftarrow (-fs1 \times fs2) + fs3$

Table 1.29: Floating point square root, min, max instructions

Instruction	Description
`fsqrt.{h|s|d} rd,rs1`	Square root. $rd \leftarrow \sqrt{rs1}$
Minimum/Maximum	
`fmin.{h|s|d} rd,rs1,rs2`	Minimum of two inputs.`rd`← `rs1<rs2 ? rs1 : rs2`
`fmax.{h|s|d} rd,rs1,rs2`	Maximum of two inputs.`rd`← `rs1<rs2 ? rs2 : rs1`
`fsgnj.{s|d|h} fd,fs1,fs2`	Sign injection of fs2 into fs1. $fd[xlen-1] \leftarrow rs2[xlen-1]$ $fd[0..xlen-2] \leftarrow rs1[0..xlen-2]$
`fsgnjn.{s|d|h}} fd,fs1,fs2`	Sign injection of neg(fs2) into fs1. $fd[xlen-1] \leftarrow !\, rs2[xlen-1]$ $fd[0..xlen-2] \leftarrow rs1[0..xlen-2]$
`fclass.{h|s|d} rd,fs1`	Classify fs1, returning a classification in rd, that must be an integer register. The bits in the result are explained in table §1.30, page 100. Only one bit will be set.

Table 1.30: `fclass` results

Bit number	Meaning
0	$-\infty$
1	$rs1 < 0$
2	subnormal $rs1 < 0$
3	$rs1 \equiv -0$
4	$rs1 \equiv 0$
5	subnormal $rs1 > 0$
6	$rs1 > 0$

[53] The official riscv manual acknowledges that fnmadd is a **misnomer**. They try to justify this error with:

> The FNMSUB and FNMADD instructions are counter intuitively named, owing to the naming of the corresponding instructions in MIPS-IV. The MIPS instructions were defined to negate the sum, rather than negating the product as the RISC-V instructions do, so the naming scheme was more rational at the time. The two definitions differ with respect to signed-zero results. The RISC-V definition matches the behavior of the x86 and ARM fused multiply-add instructions, but unfortunately the RISC-V FNMSUB and FNMADD instruction names are swapped compared to x86 and ARM.
>
> RISC-V Unprivileged ISA V20191214-draft page 77

In my opinion, this "explanation" doesn't explain why this misnomer is maintained...

Table 1.30: `fclass` results

7	$+\infty$
8	signaling NAN
9	quiet NAN

Conversions

Table 1.31: Floating point conversion instructions

Instruction	Description
`fcvt.w.s rd,rs1`	Converts a single-precision floating-point number to a signed 32-bit integer. Sign-extends the32-bit result to the destination register width.
`fcvt.s.w rd,rs1`	Converts a signed 32-bit integer to a single-precision floating-point number
`fcvt.wu.s rd,rs1`	Converts a single-precision floating-point number to an unsigned 32-bit integer. Sign-extends the 32-bit result to the destination register width.
`fcvt.s.wu rd,rs1`	Converts a unsigned 32-bit integer to a single-precision floating-point number
`fcvt.l.s rd,rs1`	Converts a single-precision floating-point number to a signed 64-bit integer.
`fcvt.s.l rd,rs1`	Converts a signed 64-bit integer to a single-precision floating-point number
`fcvt.lu.s rd,rs1`	Converts a single-precision floating-point number to an unsigned 64-bit integer.
`fcvt.s.lu rd,rs1`	Converts a unsigned 64-bit integer to a single-precision floating-point number
Other precisions	
`fcvt.{l|w|lu|wu}.{s|d|h}`	Convert floating point to integer in the different sizes and precisions
`fcvt.{s|d|h}.{l|lu|w|wu}`	Convert integer to floating point in the different sizes and precisions
`fcvt.{h|s|d}.{h|s|d}`	Convert between different floating point formats.

Table 1.32: Move to/from integer registers

Instruction	Description
`fmv.x.w rd,rs1`	Moves the single-precision value in floating-point register rs1represented in IEEE 754-2008 encoding to the lower 32 bits of integer register rd. The higher 32 bits of the destination register are filled with copies of the floating-point number's sign bit.
`fmv.w.x`	Moves the single-precision value encoded in IEEE 754-2008 standard encoding from the lower 32 bits of integer register rs1 to the floating-point register rd.

Absent from the table of instructions above are the ones introduced in 2023: the "Zfa" extension, that will make possible to load some immediates into fp registers, minimum/max-

imum operations with NANs and others. [54]

Comparisons

Floating-point compare instructions (`feq, flt, fle`) perform the specified comparison between floating-point registers (rs1 = rs2, rs1 < rs2, rs1 ≤ rs2) writing 1 to the integer register rd if the condition holds, and 0 otherwise.

Table 1.33: Floating point comparison instructions

Instruction	Description
`feq.{h|s|d} rd,fs1,fs2`	Equality. $rd \leftarrow (fs1 = fs2)$
`flt.{h|s|d} rd,fs1,fs2`	Less than comparison. $rd \leftarrow (fs1 < fs2)$
`fle.{h|s|d} rd,fs1,fs2`	Less equal comparison. $rd \leftarrow (fs1 \leq fs2)$

`flt.{h|s|d}` and `fle.{h|s|d}` perform signaling comparisons: they set the invalid operation exception flag if either input is NaN. `feq` performs a quiet comparison: it only sets the invalid operation exception flag if either input is a *signaling* NaN. For all three instructions, the result is 0 if either operand is NaN.

1.14.15 Atomic instructions

These instructions read, modify, and write memory atomically to provide synchronisation across several RISC-V harts in the same memory space. The theoretical model whereupon riscv is based is described in the paper **Memory consistency and event ordering in scalable shared-memory multiprocessors**.In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

Complex atomic memory operations on a single memory word or doubleword are performed with the load-reserved (`lr`) and store-conditional (`sc`) instructions.

Table 1.34: load reserved/store conditional instructions

Instruction	Description
`lr.{d|w}.aqrl rd,rs1`	Loads a double or a single word from the address in rs1, places the sign-extended value in rd, and registers a reservation set, a set of bytes that subsumes the bytes in the addressed double word. For `lr.w` the value is extended to 64 bits.
`sc.{d|w}.aqrl rd,rs1,rs2`	Conditionally writes a word in rs2 to the address in rs1: the `sc.{w|d}` succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If it succeeds, the instruction writes the word in rs2 to memory, and it writes zero to rd. If it fails, the instruction does not write to memory, and it writes a nonzero value to rd. Regardless of success or failure, executing an `sc` instruction invalidates any reservation held by this hart.

[54] They are not supported in tiny-asm, nor do they have any implementation in actual hardware yet.

The aq and rl bits

The riscv specifications explain:

> To follow more closely the specifications in the paper cited above for release consistency, each atomic instruction has two bits, `aq` and `rl`, used to specify additional memory ordering constraints as viewed by other riscv harts.The bits order accesses to one of the two address domains, memory or I/O, depending on which address domain the atomic instruction is accessing.
>
> If both bits are clear, no additional ordering constraints are imposed on the atomic memory operation.If only the aq bit is set, the atomic memory operation is treated as an acquire access,i.e., no following memory operations on this RISC-V hart can be observed to take place before the acquire memory operation.
>
> If only the rl bit is set, the atomic memory operation is treated as a release access, i.e., the release memory operation cannot be observed to take place before any earlier memory operations on this RISC-V hart. If both the aq and rl bits are set, the atomic memory operation is sequentially consistent and cannot be observed to happen before any earlier memory operations or after any later memory operations in the same riscv hart and to the same address domain.[55]

These bits can be specified by writing their state after the instruction, for instance:

```
amoxor.d rd,rs1,rs2,aq
amoxor.d rd,rs1,rs2,aqrl
amoxor.d rd,rs1,rs2,rl
```

Encodings

New bits are: bit 26, for `aq` and bit 25, for `rl`. The width of the instruction is written in bits 12 to 14. A value of 3, for instance, means $2^3 \rightarrow 8$.

Figure 1.19: Atomic instructions layout

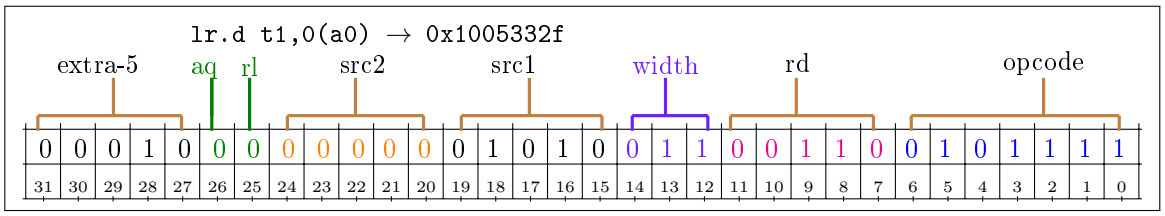


Table 1.35: Atomic Memory Operation instructions

Instruction	Description
`amoswap.{w|d}[.aq|aqrl|rl] rd,rs2,(rs1)`	Loads a data value from the address in rs1 , place the value into register rd , swaps the loaded value and the original value in rs2 , then stores the result back to the address in rs1

[55] §10.1, page 55

Table 1.35: Atomic Memory Operation instructions

`amoadd.{w|d}[.aq|aqrl|rl] rd,rs2,(rs1)`	Loads a data value from the address in rs1 , place the value into register rd , adds the loaded value and the original value in rs2 , then stores the result back to the address in rs1
`amoand.{w|d}[.aq|aqrl|rl] rd,rs2,(rs1)`	Loads a data value from the address in rs1 , place the value into register rd , does and AND operation between the loaded value and the original value in rs2 , then stores the result back to the address in rs1
`amoor.{w|d}[.aq|aqrl|rl] rd,rs2,(rs1)`	Loads a data value from the address in rs1 , place the value into register rd , does an OR operation between the loaded value and the original value in rs2 , then stores the result back to the address in rs1
`amoxor.{w|d}[.aq|aqrl|rl] rd,rs2,(rs1)`	Loads a data value from the address in rs1 , place the value into register rd , does an XOR operation between the loaded value and the original value in rs2 , then stores the result back to the address in rs1
`amomax[u].{w|d}[.aq|aqrl|rl] rd,rs2,(rs1)`	Loads a data value from the address in rs1 , place the value into register rd , finds the signed (or the unsigned) maximum value between the loaded value and the original value in rs2 , then stores the result back to the address in rs1
`amomin[u].{w|d}[.aq|aqrl|rl] rd,rs2,(rs1)`	Loads a data value from the address in rs1 , place the value into register rd , finds the signed (or unsigned) minimum value between the loaded value and the original value in rs2 , then stors the result back to the address in rs1

An example

To show one of those instructions in action, we write a small program. The results can only be traced in the debugger since there is no call to printf.

```
1     .globl main
2 main:
3     addi sp,sp,-32       // Boilerplate code for building the stack frame
4     sd ra,8(sp)
5     sd s0,0(sp)
6
7     li t1,123            // Put 123 into t1
8     sd t1,(sp)           // Store it at the bottom of the stack
9     li t2,47             // Put 47 in t2
10    amoadd.d t3,t2,0(sp) // Atomic Memory Operation
11    ld t4,(sp)           // Load the result into t4
12
13    ld ra,8(sp)          // Boiler plate code to destroy the stack frame
14    ld s0,0(sp)
15    add sp,sp,32
16    ret
```

We follow the execution in the debugger

```
star64:~/tiny-asm$ gdb a.out
GNU gdb (GDB) 11.2
... the usual stuff from gdb not shown
Reading symbols from a.out...
(gdb) b main                          // Set a breakpoint at main
Breakpoint 1 at 0x640
(gdb) display/i $pc                   // Disassemble each instruction and show it
1: x/i $pc
<error: No registers.>                // ... Normal, we haven't started yet
(gdb) run                             // Let's go
Starting program: /home/jacob/tiny-asm/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/libthread_db.so.1".

Breakpoint 1, 0x0000002aaaaaa640 in main ()
1: x/i $pc
=> 0x2aaaaaa640 <main+24>: amoadd.d t3,t2,(sp) // GDB stops BEFORE the instruction
    is executed
(gdb) print $t3
$1 = 274743607392                     // t3 contains nonsense
(gdb) print $t2
$2 = 47                               // As it should
(gdb) nexti                           // Execute next instruction
0x0000002aaaaaa644 in main ()
1: x/i $pc
=> 0x2aaaaaa644 <main+28>: ld t4,0(sp)
(gdb) print $t3                       // Contains the old value at the stack bottom
$3 = 123
(gdb) nexti
0x0000002aaaaaa648 in main ()
1: x/i $pc
=> 0x2aaaaaa648 <main+32>: ld ra,8(sp)
(gdb) print $t4
$4 = 170                              // The stack bottom contains 123 + 47
(gdb)
```

Exercise 12: *Change the amoadd by amoswap. What are the values of t2, t3 and t4 after the operation?*

1.15 Cryptographic Extensions

There are two main branches for these specifications: scalar and vector opcodes. Only scalar opcodes are supported in tiny-asm for the time being. Even then, there isn't any easily available board that implements them as of this writing (2023).

The scalar opcodes have two sides too, of course: encryption (extension`Zkne`) and decription (`Zknd`).

Table 1.36: Atomic Memory Operation instructions

Instruction	Description
AES	
`aes64ks1i rd, rs1, rcon`	Key schedule 1 decryption. Implements the rotation, SubBytes and Round Constant addition steps.

Table 1.36: Atomic Memory Operation instructions

`aes64ks2 rd, rs1, rs2`	Key schedule 2 decryption. Implements the remaining xor operations.
`aes64im rd, rs1`	Applies the inverse MixColumns transformation to two columns of the state array, packed into a single 64-bit register. It is used in decryption to create the inverse cipher KeySchedule,according to the equivalent inverse cipher construction.
`aes64esm rd, rs1, rs2`	Performs the (Inverse) SubBytes, ShiftRows and Mix-Columns Transformations for encryption.
`aes64es rd, rs1, rs2`	Perform the (Inverse) SubBytes and ShiftRows Transformations.It is used for the last round of encryption only.
`aes64dsm rd, rs1, rs2`	SubBytes, ShiftRows and Mix-Columns Transformation for decryption.
`aes64ds rd, rs1, rs2`	SubBytes and ShiftRows Transformations. It is used for the last round of decryption only
Hash functions	
`sha256sig0 rd,rs1`	Implements the Sigma0 transformation function as used in the SHA2-256 hash function [56]
`sha256sig1 rd,rs1`	Implements the Sigma1 transformation function as used in the SHA2-256 hash function.
`sha256sum0 rd,rs1`	Implements the Sum0 transformation function as used in the SHA2-256 hash function (Section 4.1.2).
`sha256sum1 rd,rs1`	Implements the Sum1 transformation function as used in the SHA2-256 hash function (Section 4.1.2).
`sha512sig0 rd,rs1,rs2`	Implements the Sigma0 transformation, as used in the SHA2-512 hash function (Section 4.1.3).
`sha512sigl rd,rs1,rs2`	Implements the Sigma1 transformation, as used in the SHA2-512 hash function (Section 4.1.3).
`sha512sum0 rd,rs1`	Implements the Sum0 transformation function as used in the SHA2-512 hash function (Section 4.1.3). $rd \leftarrow ror64(rs1, 28) \oplus ror64(rs1, 34)$
`sha512sum1 rd,rs1`	Implements the Sum1 transformation function as used in the SHA2-512 hash function (Section 4.1.3). rd←ror64(rs1,14)⊕ror64(rs1,18)⊕ror64(rs1,41)
`sm3p0 rd,rs1`	P0 transformation function as used in the SM3 hash function. [57]
`sm3p1 rd,rs1`	P1 transformation function as used in the SM3 hash function.

[56] NIST, "Secure Hash Standard (SHS)." Federal Information Processing Standards Publication FIPS 180-4,Aug. 2015, [Online]. Available: doi.org.

[57] "GB/T 32905-2016: SM3 Cryptographic Hash Algorithm." Also GM/T 0004-2012. Standardization Administration of China, Aug. 2016 Available at gmbz.org

Table 1.36: Atomic Memory Operation instructions

`sm4ks rd,rs1,rs2,bs`	Implements a T-tables in hardware style approach to accelerating the SM4 Key Schedule. A byte is extracted from rs2 based on bs, to which the SBox and linear layer transforms are applied, before the result is XOR'd with rs1 and written back to rd. `bs` is a 2 bit constant that is multiplied by 8, and represents the number of bits to shift right to select the first byte of `rs2`. $$sbin \leftarrow (rs2[0..31] >> bs * 8)[0..8]$$
`sm4ed rd, rs1, rs2, bs`	Accelerates the block encrypt/decrypt operation of the SM4 block cipher Implements a T-tables in hardware style approach to accelerating the SM4 round function. A byte is extracted from rs2 based on bs, to which the SBox and linear layer transforms are applied, before the result is XOR'd with rs1 and written back to rd. The `bs` parameter is encoded in bits 30 and 31. The two bits are multiplied by 8, and represent the shift amount for the algorithm.
`xperm4 rd,rs1,rs2`	The xperm4 instruction operates on nibbles. The rs1 register contains a vector of 16 4-bit elements, two per byte. The rs2 register contains a vector of 16 4-bit indexes. The result is each element in rs2 replaced by the indexed element in rs1, or zero if the index into rs2 is out of bounds.
`xperm8 rd,rs1,rs2`	The rs1 register contains a vector of 8 8-bit elements. The rs2 register contains a vector of 8 8-bit indexes. The result is each element in rs2 replaced by the indexed element in rs1, or zero if the index is out of bounds.

1.16 Instructions specific to the Thead processor

All these instructions are prefixed with the letters `"th."`.

Table 1.37: Thead instructions

Instruction	Description
`th.addsl rd,rs1,rs2,imm2`	$rd \leftarrow rs1 + (rs2 << imm2)$ Add with shifted register.
`th.ext rd,rs1,imm1,imm2`	$rd \leftarrow rs1[imm1 : imm2]$. Extract bits `imm1` to `imm2` with sign extension
`th.extu rd,rs1,imm1,imm2`	$rd \leftarrow rs1[imm1 : imm2]$. Extract bits `imm1` to `imm2` with zero extension

Table 1.37: Thead instructions

`th.ff0 rd,rs`	Finds the first bit with the value of 0 from the highest bit of rs1 and writes the result back into the rd register. If the highest bit of rs1 is 0, the result 0 is returned. If all the bits in rs1 are 1, the result 64 is returned.
`th.ff1 rd,rs`	Finds the first bit with the value of 1 from the highest bit of rs1 and writes the index of this bit back into rd. If the highest bit of rs1 is 1, the result 0 is returned. If all the bits in rs1 are 1, the result 64 is returned.
`th.lbia rd,(rs1),imm5,imm2` Post increment	Load byte with post increment of `rs1`. The destination register is filled with the sign extended contents of `mem[rs1]`. Then, rs1 is incremented (or decremented) by `imm5 « imm2`
`th.lbuia rd,rs1,imm5,imm2` Post increment unsigned	Load byte unsigned with post increment of `rs1`. The destination register is filled with the zero extended contents of `mem[rs1]`. Then, rs1 is incremented (or decremented) by `imm5 « imm2`
`th.lbib rd,rs1,imm5,imm2`	Pre-increment load byte. Increments rs1 by `imm5 « imm2`. Then, loads a sign extended byte into `rd` from `mem[rs1]`
`th.lhia rd, (rs1), imm5, imm2. Post-increment.`	Loads sign extended half-word into `rd`.After the load, `rs1` is incremented by `imm5 « imm2`.
`th.lhib rd, (rs1), imm5, imm2. Pre-increment.`	Loads sign extended half-word into `rd`.Before the load, `rs1` is incremented by `imm5 « imm2`.
`th.lhuia rd, (rs1), imm5, imm2. Post-increment.`	Loads zero extended half-word into `rd`.After the load, `rs1` is incremented by `imm5 « imm2`.
`th.lhuib rd, (rs1), imm5, imm2. Pre-increment.`	Loads zero extended half-word into `rd`.Before the load, `rs1` is incremented by `imm5 « imm2`.
`th.lwia rd, (rs1), imm5, imm2. Post-increment.`	Loads sign extended word into `rd`.After the load, `rs1` is incremented by `imm5 « imm2`.
`th.lwib rd, (rs1), imm5, imm2. Pre-increment.`	Loads sign extended word into `rd`.Before the load, `rs1` is incremented by `imm5 « imm2`.
`th.lwuia rd, (rs1), imm5, imm2. Post-increment.`	Loads zero extended word into `rd`.After the load, `rs1` is incremented by `imm5 « imm2`.
`th.lwuib rd, (rs1), imm5, imm2. Pre-increment.`	Loads zero extended word into `rd`.Before the load, `rs1` is incremented by `imm5 « imm2`.
`th.lwd rd,imm7(rs1)`	Load two sign extended words into two consecutive registers. The `rd` register receives the first 32 bits, sign extended, and the register `rd+1` receives the sign extended bits 32 to 63.
`th.ldia rd, (rs1), imm5, imm2` Post-increment	Loads 64 bits from `*(rs1)` into `rd`. Then increments `rs1` by a sign extended `(imm5 « imm2)`
`th.ldib rd, (rs1), imm5, imm2` Pre-increment	Increments `rs1` by a sign extended `(imm5 « imm2)`. Then loads 64 bits from `*(rs1)` into `rd`.

`th.rev rd,rs1`	Reverses the bytes in rs1.
	`rd[7]← rs[0] rd[6]← rs[1] rd[5]← rs[2]`
	`rd[4]← rs[3] rd[3]← rs[4] rd[2]← rs[5]`
	`rd[1]← rs[6] rd[0]← rs[7]`

Table 1.37: Thead instructions

`th.revw rd,rs1`	Reverses the bytes in lower word of rs1.
	`rd[3]`← `rs[0]` `rd[2]`← `rs[1]` `rd[1]`← `rs[2]`
	`rd[0]`← `rs[3]`

`th.tst rd,rs1,imm6`	rd contains the bit at position imm6 of rs1.
`th.tstnbz rd, rs1`	Tests for a zero byte in rs1. Each byte of rd will be either 0xff (the corresponding byte is zero) or 0 (the corresponding byte is different than zero)
`th.lbia rd,(rs1),imm5,imm2`	Post-increment. $signExt(rd \leftarrow mem[rs1])$; $rs1 \leftarrow rs1 + imm5 << imm2$ rd and rs1 must be different registers.
`th.lbib rd,(rs1),imm5,imm2`	Pre-increment. $rs1 \leftarrow rs1 + imm5 << imm2$; $signExt(rd \leftarrow mem[rs1])$ rd and rs1 must be different registers.
`th.lbuia rd,(rs1),imm5,imm2`	Post-increment. $zeroExt(rd \leftarrow mem[rs1])$; $rs1 \leftarrow rs1 + imm5 << imm2$ rd and rs1 must be different registers.
`th.lbuib rd,(rs1),imm5,imm2`	Pre-increment. $rs1 \leftarrow rs1 + imm5 << imm2$; $zeroExt(rd \leftarrow mem[rs1])$ rd and rs1 must be different registers.
`th.ldd rd1,rd2, (rs1),imm2`	Load pair of registers. $address \leftarrow rs1 + zero\_extend(imm2 << 4)$ $rd1 \leftarrow mem[address + 7 : address]$ $rd2 \leftarrow mem[address + 15 : address + 8]$
`th.ldia rd,(rs1),imm5,imm2`	Load byte with post increment $rd \leftarrow signExt(mem[rs1 + 7 : rs1])$ $rs1 \leftarrow rs1 + signExt(imm5 << imm2)$
`th.ldib rd,(rs1),imm5,imm2`	Load byte with pre increment $rs1 \leftarrow rs1 + signExt(imm5 << imm2)$ $rd \leftarrow signExt(mem[rs1 + 7 : rs1])$
`th.lhia rd,(rs1),imm5,imm2`	Load half word with post increment $rd \leftarrow signExt(mem[rs1 + 1 : rs1])$ $rs1 \leftarrow rs1 + signExt(imm5 << imm2)$
`th.lhib rd,(rs1),imm5,imm2`	Load half word with pre increment $rd \leftarrow signExt(mem[rs1 + 1 : rs1])$ $rs1 \leftarrow rs1 + signExt(imm5 << imm2)$
`th.lhuia rd,(rs1),imm5,imm2`	Load half word with post increment and zero extend. $rd \leftarrow zeroExt(mem[rs1 + 1 : rs1])$ $rs1 \leftarrow rs1 + signExt(imm5 << imm2)$
`th.lhib rd,(rs1),imm5,imm2`	Load half word with pre increment and zero extend. $rd \leftarrow zeroExt(mem[rs1 + 1 : rs1])$ $rs1 \leftarrow rs1 + signExt(imm5 << imm2)$
`th.lrb rd,rs1,imm2`	Load sign extended byte with shifted register. $rd \leftarrow signExt(mem[rs1 + (rs2 << imm2)])$

Table 1.37: Thead instructions

`th.lrbu rd,rs1,imm2`	Load zero extended byte with shifted register. $rd \leftarrow zeroExt(mem[rs1 + (rs2 << imm2)])$
`th.lrd rd,rs1,imm2`	Load double word with shifted register. $rd \leftarrow mem[rs1 + (rs2 << imm2)]$
`th.lrh rd,rs1,imm2`	Load half word with sign extend and shifted register. $rd \leftarrow mem[rs1 + (rs2 << imm2)]$
`th.lrhu rd,rs1,imm2`	Load half word with zero extend and shifted register. $rd \leftarrow mem[rs1 + (rs2 << imm2)]$
`th.lrw rd,rs1,imm2`	Load word with sign extend and shifted register. $rd \leftarrow mem[rs1 + (rs2 << imm2)]$
`th.lrwu rd,rs1,imm2`	Load word with zero extend and shifted register. $rd \leftarrow mem[rs1 + (rs2 << imm2)]$
`th.lurb rd,rs1,rs2,imm2`	Load byte, shift it, then sign extend result. $rd \leftarrow signExt(mem[rs1 + zeroExt(rs2[0:31])] << imm2)$
`th.lurbu rd,rs1,rs2,imm2`	Load byte, shift it, then zero extend result. $rd \leftarrow zeroExt(mem[rs1 + zeroExt(rs2[0:31])] << imm2)$
`th.lurd rd,rs1,rs2,imm2`	Load double word, shift the result. $rd \leftarrow mem[rs1 + zeroExt(rs2[0:31])] << imm2$
`th.lurh rd,rs1,rs2,imm2`	Load half word, shift the result. $rd \leftarrow signExt(mem[rs1 + zeroExt(rs2[0:31])] << imm2)$
`th.lurhu rd,rs1,rs2,imm2`	Load half word, shift the result. $rd \leftarrow zeroExt(mem[rs1 + zeroExt(rs2[0:31])] << imm2)$
`th.lurw rd,rs1,rs2,imm2`	Load word, shift the result. $rd \leftarrow signExt(mem[rs1 + zeroExt(rs2[0:31])] << imm2)$
`th.lurwu rd1,rd2,(rs1),imm2`	Load word register pair unsigned. $address \leftarrow rs1 + zeroExt(imm2 << 3)$ $rd1 \leftarrow signExt(mem[address + 3 : address])$ $rd2 \leftarrow signExt(mem[address + 7 : address + 4])$
`th.lwd rd,rs1,(rs2),imm2`	
`th.mula rd,r1,r2`	Multiplies $r1 \times r2$ and adds the sign extended result to rd. The result is then stored in rd.
`th.mulah rd,r1,r2`	Multiplies the lower 16 bits of each $r1 \times r2$ and adds the sign extended result to the lower 16 bits of rd. The result is then stored in rd.
`th.mulaw rd,r1,r2`	Multiplies the lower 32 bits of each $r1 \times r2$ and adds the sign extended result to the lower 32 bits of rd. The result is then stored in rd.
`th.mulsw rd,r1,r2`	Multiplies the lower 32 bits of each $r1 \times r2$ and subtracts the sign extended result to the lower 32 bits of rd. The result is then stored in rd.
`th.mveqz rd,r1,r2`	Assigns r1 to rd if r2 is equal to zero. Otherwise rd remains unchanged
`th.mvneqz rd,r1,r2`	Assigns r1 to rd if r2 is not equal to zero. Otherwise rd remains unchanged

Table 1.37: Thead instructions

`th.srri rd, rs1, imm6`	Shifts the original value of rs1 to the right by the amount specified , then rotates the result one bit to the right.
`th.srriw rd, rs1, imm5`	Shifts the lower 32 bits of rs1 to the right, then rotates the result one bit to the right in 32 bits.

Exercise 13: *Calculate ceil(log2(x))*
This function in C calculates $\lceil log_2(x) \rceil$:

```
1  static unsigned int bfd_log2(unsigned long x)
2  {
3      unsigned int  result = 0;
4
5      if (x ≤ 1) return result;
6      --x;
7      do
8          ++result;
9      while ((x >≥  1) ≠ 0);
10     return result;
11 }
```

Do the same in 3 assembler instructions.

1.17 Pseudo instructions

The accepted policy under risc-v is the opposite to the ARM64 assembler. Under ARM64, the assembler will issue an error if an instruction alias expands to more than one instruction. Here, it is quite the opposite, the assembler (and above all, the linker) is responsible for expanding high level macros.

Table 1.38: Pseudo instructions

Pseudo	Base instruction	Meaning
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`call offset`	`auipc x1, offset[31:12]; jalr x1, x1, offset[11:0]`	Call far-away subroutine
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value. Just an alias.
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fence`	`fence iorw, iorw`	Fence on all memory and I/O

Table 1.38: Pseudo instructions

`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]; fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fs{w|d} rd,symbol, rt`	`auipc rt, symbol[31:12]; fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jalr rs jalr`	`x1, rs, 0`	Jump and link register
`jr rs`	`jalr x0, rs, 0`	Jump register
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]; l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`la rd, symbol`	`auipc rd, symbol@GOT[31:12]; lw|d rd, symbol@GOT[11:0](rd)`	Load address With .option pic
`la rd, symbol`	`auipc rd, symbol[31:12]; addi rd, rd, symbol[11:0]`	Load address With .option nopic (Default)
`li rd, immediate`	`Myriad sequences`	Load immediate
`lla rd, symbol`	`auipc rd, symbol[31:12]; addi rd, rd, symbol[11:0]`	Load local address
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`nop`	`addi x0, x0, 0`	No operation
`not rd, rs`	`xori rd, rs, -1`	Ones' complement
`pause`	`fence w, 0`	PAUSE hint
`ret`	`jalr x0, x1, 0`	Return from subroutine
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]; s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`sext.b rd, rs`	`slli rd, rs, XLEN - 8; srai rd, rd, XLEN - 8`	Sign extend byte It will expand to another instruction sequence when B extension is available
`sext.h rd, rs`	`slli rd, rs, XLEN - 16; srai rd, rd, XLEN - 16`	Sign extend half word It will expand to another instruction sequence when B extension is available
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero

Table 1.38: Pseudo instructions

`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`tail offset`	`auipc x6, offset[31:12];` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine.
`zext.b rd, rs`	`andi rd, rs, 255`	Zero extend byte
`zext.h rd, rs`	`slli rd, rs, XLEN - 16;` `srli rd, rd, XLEN - 16`	Zero extend half word It will expand to another instruction sequence when B extension is available
`zext.w rd, rs`	`slli rd, rs, XLEN - 32;` `srli rd, rd, XLEN - 32`	Zero extend word It will expand to another instruction sequence when B extension is available

Exercise 14: *Mismatch between source and disassembly. Explain*

Consider this program:

```
1    .globl main
2 main:
3    la a0,.L2
4    jr ra
5 .L2:
```

When we assemble this with `gcc -o addr.o addr.s` we obtain an object file. When we disassemble it, we obtain:

```
1 Disassembly of section .text:
2
3 0000000000000000 <main>:
4 0:  00000517          auipc  a0,0x0
5 4:  00050513          mv  a0,a0
6 8:  00008067          ret
```

Explain why this mismatch.

1.18 Interfacing with high level languages

- The first 8 arguments of a function, if they fit in 64 bits, are passed in the registers a0-a7.
- When more than 8 arguments are present, they are passed in the stack.
- Scalars that are 2×64 bits wide are passed in a pair of argument registers, with the low-order 64 bits in the lower-numbered register and the high-order 64 bits in the higher-numbered register. Scalars wider than 128 bits are passed by reference. In C, the address of a copy is passed.
- Floating point arguments (16, 32 and 64 bits) are passed in the registers f10 to f17.
- Structures with two floating point fields (complex numbers, for instance) are passed using two floating points registers. If there isn't two floating point regs available they are passed using the integer protocol.
- Functions with variable number of arguments use the integer protocol.
- Values up to 128 bits are returned using the a0-a1 register pair, or the f0,f1 registers for floating point.

1.19 The full opcode table

Symbols used in the opcode table:

Sign	Description
$\oplus$	Xor
$\not\wedge$	Not and
$\pm$	Sign extension
$+$	Zero extension
$\triangledown$	Macro.
$\triangledown$	Alias
$\curvearrowleft$	Rotate right
op_u	Unsigned variant of `op`, for example $<_u$
$reg[idx]$	Bit `idx` of `reg`
$reg[n..m]$	Bits n through m inclusive of `reg`
`rm`	Rounding mode
`uim`n	unsigned immediate with n bits
`imm`n	immediate with n bits
$\mathcal{H}$	Hypervisor instruction
$\approx$	Rounding mode

Notes:

- The symbol $\leftarrow$ is used for assignment. $=$ means equality.
- If the rounding mode is not explicitly mentioned, the rounding mode stored in the floating point control register is used.

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
`add sp,sp,imm6`	C	Cc,Cc,CL	$sp \leftarrow sp + imm6 << 4$
`add rd',sp,imm8`	C	Ct,Cc,CK	
`add rd,rd,rs2'`	C	d,CU,CV	
`add rd,rd,imm6`	C	d,CU,Co	
`add rd,rs1',rd`	C	d,CV,CU	
`add rd,x0,rs2'`	C	d,Cz,CV	
`add rd,rs1,rs2`	I	d,s,j	
`add rd,rs1,rs2`	I	d,s,t	
`add rd,rs1,rs2,%tprel_add`	I	d,s,t,1	
`add.uw`	Zba	d,s,t	
`addi`	C	Cc,Cc,CL	
`addi`	C	Ct,Cc,CK	
`addi rd,rd,imm6`	C	d,CU,Cj	$rd \leftarrow rd + imm6$
`addi`	C	d,CU,z	
`addi`	C	d,CV,z	
`addi`	C	d,Cz,Co	
`addi rd,rs1,imm12`	I	d,s,j	$rd \leftarrow rs1 + imm12$
`addiw rd,rs1,imm12`	C	d,CU,Co	$rd \leftarrow \pm rs1_{0..31} + imm12$
`addiw rd,rs,imm12`	I	d,s,j	$rd \leftarrow \pm rs_{0..31} + imm12$
`addw`	C	Cs,Ct,Cw	
`addw`	C	Cs,Cw,Ct	
`addw`	C	d,CU,Co	
`addw rd,rs1,imm12`	I	d,s,j	$rd \leftarrow rs1 + imm12$

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
add rd,rs1,rs2w	I	d,s,t	$rd \leftarrow rs1 + rs2$
aes64ds rd, rs1, rs2	Zknd	d,s,t	Round: InvShiftRows, InvSubBytes
aes64dsm rd, rs1, rs2	Zknd	d,s,t	Round: InvShiftRows, InvSubBytes, InvMixColumns
aes64es rd, rs1, rs2	Zkne	d,s,t	Round: ShiftRows, SubBytes
aes64esm rd, rs1, rs2	Zkne	d,s,t	Round: ShiftRows, SubBytes, MixColumns
aes64im rd, rs1	Zknd	d,s	KeySchedule: InvMixColumns for Decrypt
aes64ks1i rd, rs1, rcon	Zknd | Zkne	d,s,Y	KeySchedule: SubBytes, Rotate, Round Const
aes64ks2 rd, rs1, rs2	Zknd | Zkne	d,s,t	KeySchedule: XOR summation
amoadd.d rd,rs2,(rs1),	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 + rd
amoadd.d.aq rd,rs2,(rs1)	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 + rd. Acquire.
amoadd.d.aqrl rd,rs2,(rs1)	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 + rd. Acquire/release
amoadd.d.rl rd,rs2,(rs1)	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 + rd. Release
amoadd.w rd,rs2,(rs1)	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 + rd. 32 bit.
amoadd.w.aq	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 + rd. 32 bit acquire
amoadd.w.aqrl rd,rs2,(rs1)	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 + rd.Acquire/Release
amoadd.w.rl rd,rs2,(rs1)	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 + rd. Release.
amoand.d rd,rs2,(rs1)	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 ∧ rd
amoand.d.aq rd,rs2,(rs1)	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 ∧ rd. Acquire
amoand.d.aqrl rd,rs2,(rs1)	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 ∧ rd. Acquire/Release
amoand.d.rl rd,rs2,(rs1)	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 ∧ rd. Release
amoand.w rd,rs2,(rs1)	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 ∧ rd. 32 bit
amoand.w.aq rd,rs2,(rs1)	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 ∧ rd
amoand.w.aqrl rd,rs2,(rs1)	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 ∧ rd.Acquire/Release
amoand.w.rl rd,rs2,(rs1)	A	d,t,0(s)	rd←*(rs1) *(rs1)←rs2 ∧ rd. Release. 32 bits
amomax.d	A	d,t,0(s)	rd←*(rs1) *(rs1)← max(rs1,rs2)
amomax.d.aq	A	d,t,0(s)	
amomax.d.aqrl	A	d,t,0(s)	
amomax.d.rl	A	d,t,0(s)	
amomax.w	A	d,t,0(s)	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
amomax.w.aq	A	d,t,0(s)	
amomax.w.aqrl	A	d,t,0(s)	
amomax.w.rl	A	d,t,0(s)	
amomaxu.d	A	d,t,0(s)	
amomaxu.d.aq	A	d,t,0(s)	
amomaxu.d.aqrl	A	d,t,0(s)	
amomaxu.d.rl	A	d,t,0(s)	
amomaxu.w	A	d,t,0(s)	
amomaxu.w.aq	A	d,t,0(s)	
amomaxu.w.aqrl	A	d,t,0(s)	
amomaxu.w.rl	A	d,t,0(s)	
amomin.d	A	d,t,0(s)	
amomin.d.aq	A	d,t,0(s)	
amomin.d.aqrl	A	d,t,0(s)	
amomin.d.rl	A	d,t,0(s)	
amomin.w	A	d,t,0(s)	
amomin.w.aq	A	d,t,0(s)	
amomin.w.aqrl	A	d,t,0(s)	
amomin.w.rl	A	d,t,0(s)	
amominu.d	A	d,t,0(s)	
amominu.d.aq	A	d,t,0(s)	
amominu.d.aqrl	A	d,t,0(s)	
amominu.d.rl	A	d,t,0(s)	
amominu.w	A	d,t,0(s)	
amominu.w.aq	A	d,t,0(s)	
amominu.w.aqrl	A	d,t,0(s)	
amominu.w.rl	A	d,t,0(s)	
amoor.d	A	d,t,0(s)	
amoor.d.aq	A	d,t,0(s)	
amoor.d.aqrl	A	d,t,0(s)	
amoor.d.rl	A	d,t,0(s)	
amoor.w	A	d,t,0(s)	
amoor.w.aq	A	d,t,0(s)	
amoor.w.aqrl	A	d,t,0(s)	
amoor.w.rl	A	d,t,0(s)	
amoswap.d	A	d,t,0(s)	
amoswap.d.aq	A	d,t,0(s)	
amoswap.d.aqrl	A	d,t,0(s)	
amoswap.d.rl	A	d,t,0(s)	
amoswap.w	A	d,t,0(s)	
amoswap.w.aq	A	d,t,0(s)	
amoswap.w.aqrl	A	d,t,0(s)	
amoswap.w.rl	A	d,t,0(s)	
amoxor.d	A	d,t,0(s)	$rd \leftarrow mem[rs1]$ $mem[rs1] \leftarrow rd \oplus rs2$
amoxor.d.aq	A	d,t,0(s)	$rd \leftarrow mem[rs1]$ $mem[rs1] \leftarrow rd \oplus rs2$ acquire *(rs1)
amoxor.d.aqrl rd,rs2,(rs1)	A	d,t,0(s)	$rd \leftarrow mem[rs1]$ $mem[rs1] \leftarrow rd \oplus rs2$ acquire/release *(rs1)

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
amoxor.d.rl	A	d,t,0(s)	$rd \leftarrow mem[rs1]$ $mem[rs1] \leftarrow rd \oplus rs2$ release *(rs1)
amoxor.w	A	d,t,0(s)	$rd \leftarrow mem[rs1]_{0..32}$ $mem[rs1]_{0..32} \leftarrow rd \oplus rs2$
amoxor.w.aq	A	d,t,0(s)	$rd \leftarrow mem[rs1]_{0..32}$ $mem[rs1]_{0..32} \leftarrow rd \oplus rs2$ acquire *(rs1)
amoxor.w.aqrl	A	d,t,0(s)	$rd \leftarrow mem[rs1_{0..32}]$ $mem[rs1]_{0..32} \leftarrow rd \oplus rs2$ acquire/release *(rs1)
amoxor.w.rl	A	d,t,0(s)	$rd \leftarrow mem[rs1]_{0..32}$ $mem[rs1]_{0..32} \leftarrow rd \oplus rs2$ release *(rs1)
and	C	Cs,Ct,Cw	
and	C	Cs,Cw,Co	
and	C	Cs,Cw,Ct	
and rd,rs1,imm12	I	d,s,j	$rd \leftarrow rs1 \wedge imm12$
and rd,rs1,rs2	I	d,s,t	$rd \leftarrow rs1 \wedge rs2$
andi	C	Cs,Cw,Co	
andi	I	d,s,j	
andn rd,rs1,imm12	Zbb | Zbkb	d,s,j	$rd \leftarrow rs1 \nwedge imm12$
andn rd,rs1,rs2	Zbb | Zbkb	d,s,t	$rd \leftarrow rs1 \nwedge rs2$
auipc rd,imm20	I	d,u	$rd \leftarrow pc + imm20$
bclr rd,rs1,im6	Zbs	d,s,>	$rd \leftarrow (rs1 \wedge \sim (1 << im6))$
bclr	Zbs	d,s,t	
bclri	Zbs	d,s,>	
beq	C	Cs,Cz,Cp	
beq	I	s,t,p	
beqz rs,lab	C	Cs,Cp	$pc \leftarrow pc + lab \times (rs1 = rs2)$
beqz rs,lab	I	s,p	$pc \leftarrow pc + lab \times (rs = rs2)$
bext	Zbs	d,s,>	
bext	Zbs	d,s,t	
bexti	Zbs	d,s,>	
bge	I	s,t,p	
bgeu	I	s,t,p	
bgez	I	s,p	
bgt rs1,rs2,lab	I	t,s,p	$pc \leftarrow pc + lab \times (rs2 < rs1)$
bgtu r1,r2,lab	I	t,s,p	$pc \leftarrow pc + lab \times (r2 <_u r1)$
bgtz r1,lab	I	t,p	$pc \leftarrow pc + lab \times (0 < r1)$
binv rd,rs1,imm6	Zbs	d,s,>	alias for binvi
binv rd,rs1,rs2	Zbs	d,s,t	$rd \leftarrow (\sim rs1[rs2]) \vee \sim (1 << rs2)$
binvi rd,rs1,imm6	Zbs	d,s,>	$rd \leftarrow (\sim rs1[imm6]) \vee \sim (1 << imm6)$ [58]
ble	I	t,s,p	
bleu	I	t,s,p	
blez	I	t,p	
blt rs1,rs2,lab	I	s,t,p	$pc \leftarrow pc + lab \times (rs1 < rs2)$
bltu r1,r2,lab	I	s,t,p	$pc \leftarrow pc + lab \times (r1 <_u r2)$

[58] rd is assigned the value of rs1 with the bit at position imm6 inverted

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
bltz rs1,lab	I	s,p	$pc \leftarrow pc + lab \times (rs1 < 0)$
bne rs1,rs2,label	C	Cs,Cz,Cp	$pc \leftarrow pc + label \times (rs1 \neq rs2)$
bne rs1,rs2,label	I	s,t,p	$pc \leftarrow pc + label \times (rs1 \neq rs2)$
bnez rs,label	C	Cs,Cp	∇ $pc \leftarrow pc + label \times (rs \neq 0)$
bnez rs, label	I	s,p	∇ $pc \leftarrow pc + label \times (rs \neq 0)$
brev8 rd, rs	Zbkb	d,s	Reverse bits in byte. for (i=0;i<4;i++) $rd[i]_{0..7} \leftarrow rs[i]_{7..0}$
bset rd,rs,imm6	Zbs	d,s,>	Alias of bseti
bset rd,rs1,rs2	Zbs	d,s,t	Set bit: $rd \leftarrow rs$ $rd[rs2] \leftarrow 1$
bseti rd,rs1,imm6	Zbs	d,s,>	Set bit: $rd \leftarrow rs$ $rd[imm6] \leftarrow 1$
c.add	C	d,CV	
c.addi	C	d,Co	
c.addi16sp	C	Cc,CL	
c.addi4spn	C	Ct,Cc,CK	
c.addiw	C	d,Co	
c.addw	C	Cs,Ct	
c.and	C	Cs,Ct	
c.andi	C	Cs,Co	
c.beqz	C	Cs,Cp	
c.bnez	C	Cs,Cp	
c.ebreak	C	--	
c.fld	D&C	CD,Cl(Cs)	
c.fldsp	D&C	D,Cn(Cc)	
c.flw	F&C	CD,Ck(Cs)	
c.flwsp	F&C	D,Cm(Cc)	
c.fsd	D&C	CD,Cl(Cs)	
c.fsdsp	D&C	CT,CN(Cc)	
c.fsw	F&C	CD,Ck(Cs)	
c.fswsp	F&C	CT,CM(Cc)	
c.j	C	Ca	
c.jalr	C	d	
c.jr	C	d	
c.ld	C	Ct,Cl(Cs)	
c.ldsp	C	d,Cn(Cc)	
c.li	C	d,Co	
c.lui	C	d,Cu	
c.lw	C	Ct,Ck(Cs)	
c.lwsp	C	d,Cm(Cc)	
c.mv	C	d,CV	
c.nop	C	--	
c.nop	C	Cj	
c.or	C	Cs,Ct	
c.sd	C	Ct,Cl(Cs)	
c.sdsp rs,uim6(sp)	C	CV,CN(Cc)	$mem[sp + uim6 << 3] \leftarrow rs$
c.slli	C	d,C>	
c.slli64	C	d	
c.srai	C	Cs,C>	
c.srai64	C	Cs	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
c.srli	C	Cs,C>	
c.srli64	C	Cs	
c.sub	C	Cs,Ct	
c.subw	C	Cs,Ct	
c.sw	C	Ct,Ck(Cs)	
c.swsp	C	CV,CM(Cc)	
c.unimp	C	--	
c.xor	C	Cs,Ct	
call	I	c	
call	I	d,c	
cbo.clean	Zicbom	0(s)	
cbo.flush	Zicbom	0(s)	
cbo.inval	Zicbom	0(s)	
cbo.zero	Zicboz	0(s)	
clmul	Zbc | Zbkc	d,s,t	
clmulh	Zbc | Zbkc	d,s,t	
clmulr	Zbc	d,s,t	
clz	Zbb	d,s	
clzw	Zbb	d,s	
cpop	Zbb	d,s	
cpopw	Zbb	d,s	
csrc csreg,x0	Zicsr	E,Z	Clears bits in control reg
csrc creg,rs1	Zicsr	E,s	
csrci creg,imm12	Zicsr	E,Z	Clears bits of the control reg
csrr	Zicsr	d,E	
csrrc	Zicsr	d,E,Z	
csrrc	Zicsr	d,E,s	
csrrci	Zicsr	d,E,Z	
csrrs	Zicsr	d,E,Z	
csrrs	Zicsr	d,E,s	
csrrsi	Zicsr	d,E,Z	
csrrw	Zicsr	d,E,Z	
csrrw	Zicsr	d,E,s	
csrrwi	Zicsr	d,E,Z	
csrs	Zicsr	E,Z	
csrs	Zicsr	E,s	
csrsi	Zicsr	E,Z	
csrw	Zicsr	E,Z	
csrw	Zicsr	E,s	
csrwi	Zicsr	E,Z	
ctz	Zbb	d,s	
ctzw	Zbb	d,s	
div	M	d,s,t	
divu	M	d,s,t	
divuw	M	d,s,t	
divw	M	d,s,t	
dret	I	--	
ebreak	C	--	
ebreak	I	--	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
ecall	I	--	
fabs.d rd,rs	D	D,U	$rd \leftarrow (rs < 0)? - rs : rd$
fabs.h rd,rs	Zfh	D,U	$rd \leftarrow (rs < 0)? - rs : rd$
fabs.q rd,rd	Q	D,U	$rd \leftarrow (rs < 0)? - rs : rd$
fabs.s rd,rs	F	D,U	$rd \leftarrow (rs < 0)? - rs : rd$
fadd.d rd,rs	D	D,S,T	
fadd.d rd,rs1,rs2	D	D,S,T,m	$rd \leftarrow rs1 + rs2$
fadd.h	Zfh	D,S,T	$rd \leftarrow rs1 + rs2$
fadd.h rd,rs1,rs1,rm	Zfh	D,S,T,m	$rd \leftarrow rs1 + rs2, \approx rm$
fadd.q	Q	D,S,T	
fadd.q	Q	D,S,T,m	
fadd.s	F	D,S,T	
fadd.s	F	D,S,T,m	
fclass.d	D	d,S	
fclass.h	Zfh	d,S	
fclass.q	Q	d,S	
fclass.s	F	d,S	
fcvt.d.h frd,frh	Zfhmin&D	D,S	
fcvt.d.l frd,rs1	D	D,s	$frd \leftarrow rs1$
fcvt.d.l frd,rs1,rm	D	D,s,m	$frd \leftarrow rs1 \approx rm$
fcvt.d.lu	D	D,s	
fcvt.d.lu	D	D,s,m	
fcvt.d.q	Q	D,S	
fcvt.d.q	Q	D,S,m	
fcvt.d.s	D	D,S	
fcvt.d.w	D	D,s	
fcvt.d.wu	D	D,s	
fcvt.h.d	Zfhmin&D	D,S	
fcvt.h.d	Zfhmin&D	D,S,m	
fcvt.h.l	Zfh	D,s	
fcvt.h.l	Zfh	D,s,m	
fcvt.h.lu	Zfh	D,s	
fcvt.h.lu	Zfh	D,s,m	
fcvt.h.q	Zfhmin&Q	D,S	
fcvt.h.q	Zfhmin	D,S,m	
fcvt.h.s	Zfhmin	D,S	
fcvt.h.s	Zfhmin	D,S,m	
fcvt.h.w	Zfh	D,s	
fcvt.h.w	Zfh	D,s,m	
fcvt.h.wu	Zfh	D,s	
fcvt.h.wu	Zfh	D,s,m	
fcvt.l.d	D	d,S	
fcvt.l.d	D	d,S,m	
fcvt.l.h	Zfh	d,S	
fcvt.l.h	Zfh	d,S,m	
fcvt.l.q	Q	d,S	
fcvt.l.q	Q	d,S,m	
fcvt.l.s	F	d,S	
fcvt.l.s	F	d,S,m	
fcvt.lu.d	D	d,S	
fcvt.lu.d	D	d,S,m	
fcvt.lu.h	Zfh	d,S	
fcvt.lu.h	Zfh	d,S,m	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
fcvt.lu.q	Q	d,S	
fcvt.lu.q	Q	d,S,m	
fcvt.lu.s	F	d,S	
fcvt.lu.s	F	d,S,m	
fcvt.q.d	Q	D,S	
fcvt.q.h	Zfhmin&Q	D,S	
fcvt.q.l	Q	D,s	
fcvt.q.l	Q	D,s,m	
fcvt.q.lu	Q	D,s	
fcvt.q.lu	Q	D,s,m	
fcvt.q.s	Q	D,S	
fcvt.q.w	Q	D,s	
fcvt.q.wu	Q	D,s	
fcvt.s.d	D	D,S	
fcvt.s.d	D	D,S,m	
fcvt.s.h fd,fs	Zfhmin	D,S	$fd_{[0..32]} \leftarrow fs_{[0..16]}$
fcvt.s.l	F	D,s	
fcvt.s.l	F	D,s,m	
fcvt.s.lu	F	D,s	
fcvt.s.lu	F	D,s,m	
fcvt.s.q	Q	D,S	
fcvt.s.q	Q	D,S,m	
fcvt.s.w	F	D,s	
fcvt.s.w	F	D,s,m	
fcvt.s.wu	F	D,s	
fcvt.s.wu	F	D,s,m	
fcvt.w.d	D	d,S	
fcvt.w.d	D	d,S,m	
fcvt.w.h rd,fs	Zfh	d,S	$rd[0..32] \leftarrow fs[0..16]$
fcvt.w.h rd,fs,rm	Zfh	d,S,m	$rd[0..32] \leftarrow fs[0..16] \approx rm$
fcvt.w.q	Q	d,S	
fcvt.w.q	Q	d,S,m	
fcvt.w.s	F	d,S	
fcvt.w.s	F	d,S,m	
fcvt.wu.d	D	d,S	
fcvt.wu.d	D	d,S,m	
fcvt.wu.h	Zfh	d,S	
fcvt.wu.h	Zfh	d,S,m	
fcvt.wu.q	Q	d,S	
fcvt.wu.q	Q	d,S,m	
fcvt.wu.s	F	d,S	
fcvt.wu.s	F	d,S,m	
fdiv.d	D	D,S,T	
fdiv.d	D	D,S,T,m	
fdiv.h	Zfh	D,S,T	
fdiv.h	Zfh	D,S,T,m	
fdiv.q	Q	D,S,T	
fdiv.q	Q	D,S,T,m	
fdiv.s	F	D,S,T	
fdiv.s	F	D,S,T,m	
fence	I	--	▽ fence iorw,iorw
fence	I	P,Q	
fence.i	Zifencei	--	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
fence.tso	I	--	
feq.d	D	d,S,T	
feq.h	Zfh	d,S,T	
feq.q	Q	d,S,T	
feq.s	F	d,S,T	
fge.d	D	d,T,S	
fge.h	Zfh	d,T,S	
fge.q	Q	d,T,S	
fge.s	F	d,T,S	
fgt.d	D	d,T,S	
fgt.h	Zfh	d,T,S	
fgt.q	Q	d,T,S	
fgt.s	F	d,T,S	
fld	D	D,A,s	
fld	D	D,o(s)	
fld	D&C	CD,Cl(Cs)	
fld	D&C	D,Cn(Cc)	
fle.d	D	d,S,T	
fle.h	Zfh	d,S,T	
fle.q	Q	d,S,T	
fle.s	F	d,S,T	
flh	Zfhmin	D,A,s	
flh	Zfhmin	D,o(s)	
flq	Q	D,A,s	
flq	Q	D,o(s)	
flt.d	D	d,S,T	
flt.h	Zfh	d,S,T	
flt.q	Q	d,S,T	
flt.s	F	d,S,T	
flw	F	D,A,s	
flw	F	D,o(s)	
fmadd.d fd,fs1,fs2, fs3	D	D,S,T,R	
fmadd.d fd,fs1,fs2, fs3,rm	D	D,S,T,R,m	
fmadd.h fd,fs1,fs2, fs3	Zfh	D,S,T,R	
fmadd.h fd,fs1,fs2, fs3,rm	Zfh	D,S,T,R,m	
fmadd.q fd,fs1,fs2, fs3	Q	D,S,T,R	
fmadd.q fd,fs1,fs2, fs3,rm	Q	D,S,T,R,m	
fmadd.s fd,fs1,fs2, fs3	F	D,S,T,R	
fmadd.s fd,fs1,fs2, fs3	F	D,S,T,R,m	
fmax.d fd,fs1,fs2	D	D,S,T	$fd \leftarrow fs1 \lceil rs2$
fmax.h fd,fs1,fs2	Zfh	D,S,T	
fmax.q fd,fs1,fs2	Q	D,S,T	
fmax.s fd,fs1,fs2	F	D,S,T	
fmin.d fd,fs1,fs2	D	D,S,T	
fmin.h fd,fs1,fs2	Zfh	D,S,T	
fmin.q fd,fs1,fs2	Q	D,S,T	
fmin.s fd,fs1,fs2	F	D,S,T	
fmsub.d fd,fs1,fs2, fs3	D	D,S,T,R	
fmsub.d fd,fs1,fs2, fs3,rm	D	D,S,T,R,m	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
fmsub.h fd,fs1,fs2, fs3	Zfh	D,S,T,R	
fmsub.h fd,fs1,fs2, fs3,rm	Zfh	D,S,T,R,m	
fmsub.q fd,fs1,fs2, fs3	Q	D,S,T,R	
fmsub.q fd,fs1,fs2, fs3,rm	Q	D,S,T,R,m	
fmsub.s fd,fs1,fs2, fs3	F	D,S,T,R	
fmsub.s fd,fs1,fs2, fs3,rm	F	D,S,T,R,m	
fmul.d fd,fs1,fs2	D	D,S,T	
fmul.d fd,fs1,fs2	D	D,S,T,m	
fmul.h fd,fs1,fs2	Zfh	D,S,T	
fmul.h fd,fs1,fs2	Zfh	D,S,T,m	
fmul.q fd,fs1,fs2	Q	D,S,T	
fmul.q fd,fs1,fs2	Q	D,S,T,m	
fmul.s fd,fs1,fs2	F	D,S,T	
fmul.s fd,fs1,fs2	F	D,S,T,m	
fmv.d fd,fs1	D	D,U	
fmv.d.x fd,fs1	D	D,s	
fmv.h fd,fs1	Zfh	D,U	
fmv.h.x fd,fs1	Zfhmin	D,s	
fmv.q	Q	D,U	
fmv.s	F	D,U	
fmv.s.x	F	D,s	
fmv.w.x	F	D,s	
fmv.x.d	D	d,S	
fmv.x.h	Zfhmin	d,S	
fmv.x.s	F	d,S	
fmv.x.w	F	d,S	
fneg.d	D	D,U	
fneg.h	Zfh	D,U	
fneg.q	Q	D,U	
fneg.s	F	D,U	
fnmadd.d fd,fs1,fs2, fs3	D	D,S,T,R	
fnmadd.d fd,fs1,fs2, fs3,rm	D	D,S,T,R,m	
fnmadd.h fd,fs1,fs2, fs3	Zfh	D,S,T,R	
fnmadd.h fd,fs1,fs2, fs3,rm	Zfh	D,S,T,R,m	
fnmadd.q fd,fs1,fs2, fs3	Q	D,S,T,R	
fnmadd.q fd,fs1,fs2, fs3,rm	Q	D,S,T,R,m	
fnmadd.s fd,fs1,fs2, fs3	F	D,S,T,R	
fnmadd.s fd,fs1,fs2, fs3,rm	F	D,S,T,R,m	
fnmsub.d fd,fs1,fs2, fs3	D	D,S,T,R	
fnmsub.d fd,fs1,fs2, fs3,rm	D	D,S,T,R,m	
fnmsub.h fd,fs1,fs2, fs3	Zfh	D,S,T,R	
fnmsub.h fd,fs1,fs2, fs3,rm	Zfh	D,S,T,R,m	
fnmsub.q fd,fs1,fs2, fs3	Q	D,S,T,R	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
fnmsub.q fd,fs1,fs2, fs3,rm	Q	D,S,T,R,m	
fnmsub.s fd,fs1,fs2, fs3	F	D,S,T,R	
fnmsub.s fd,fs1,fs2, fs3,rm	F	D,S,T,R,m	
frcsr rd	F	d	
frflags rd	F	d	
frrm rd	F	d	
frsr rd	F	d	
fscsr rd,rs	F	d,s	
fscsr rd	F	s	
fsd fs2,imm12(rs1)	D	T,A,s	
fsd	D	T,q(s)	
fsd	D&C	CD,Cl(Cs)	
fsd	D&C	CT,CN(Cc)	
fsflags	F	d,s	
fsflags	F	s	
fsflagsi	F	Z	
fsflagsi	F	d,Z	
fsgnj.d	D	D,S,T	
fsgnj.h	Zfh	D,S,T	
fsgnj.q	Q	D,S,T	
fsgnj.s	F	D,S,T	
fsgnjn.d	D	D,S,T	
fsgnjn.h	Zfh	D,S,T	
fsgnjn.q	Q	D,S,T	
fsgnjn.s	F	D,S,T	
fsgnjx.d	D	D,S,T	
fsgnjx.h	Zfh	D,S,T	
fsgnjx.q	Q	D,S,T	
fsgnjx.s	F	D,S,T	
fsh	Zfhmin	T,A,s	
fsh	Zfhmin	T,q(s)	
fsq	Q	T,A,s	
fsq	Q	T,q(s)	
fsqrt.d	D	D,S	
fsqrt.d	D	D,S,m	
fsqrt.h	Zfh	D,S	
fsqrt.h	Zfh	D,S,m	
fsqrt.q	Q	D,S	
fsqrt.q	Q	D,S,m	
fsqrt.s	F	D,S	
fsqrt.s	F	D,S,m	
fsrm	F	d,s	
fsrm	F	s	
fsrmi	F	Z	
fsrmi	F	d,Z	
fssr	F	d,s	
fssr	F	s	
fsub.d frd,frs1,frs2	D	D,S,T	
fsub.d frd,frs1,frs2	D	D,S,T,m	
fsub.hi frd,frs1,frs2	Zfh	D,S,T	
fsub.h frd,frs1,frs2	Zfh	D,S,T,m	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
fsub.q frd,frs1,frs2	Q	D,S,T	
fsub.q frd,frs1,frs2	Q	D,S,T,m	
fsub.s frd,frs1,frs2	F	D,S,T	
fsub.s frd,frs1,frs2	F	D,S,T,m	
fsw	F	T,A,s	
fsw	F	T,q(s)	
hfence.gvma	H	--	
hfence.gvma	H	s	
hfence.gvma	H	s,t	
hfence.vvma	H	--	
hfence.vvma	H	s	
hfence.vvma	H	s,t	
hinval.gvma	svinval	s,t	
hinval.vvma	svinval	s,t	
hlv.b rd,0(rs1)	H	d,0(s)	$\mathcal{H}$ $rd \leftarrow \pm(rs1)[0..7]$
hlv.bu rd,0(rs1)	H	d,0(s)	$\mathcal{H}$ $rd \leftarrow_u (rs1)[0..7]$
hlv.d	H	d,0(s)	$\mathcal{H}$
hlv.h rd,0(rs1)	H	d,0(s)	$\mathcal{H}$ $rd \leftarrow (rs1)[0..15]$
hlv.hu rd,0(rs1)	H	d,0(s)	$\mathcal{H}$ $rd \leftarrow_u (rs1)[0..15]$
hlv.w rd,0(rs1)	H	d,0(s)	$\mathcal{H}$ $rd \leftarrow (rs1)[0..31]$
hlv.wu rd,0(rs1)	H	d,0(s)	$\mathcal{H}$ $rd \leftarrow_u (rs1)[0..31]$
hlvx.hu	H	d,0(s)	
hlvx.wu	H	d,0(s)	
hret	I	--	
hsv.b rs1,0(rs2)	H	t,0(s)	$\mathcal{H}$ $mem[rs2] \leftarrow rs1[0..7]$
hsv.d rs1,0(rs2)	H	t,0(s)	$\mathcal{H}$
hsv.h rs1,0(rs2)	H	t,0(s)	$\mathcal{H}$ $mem[rs2] \leftarrow rs1[0..15]$
hsv.w rs1,0(rs2)	H	t,0(s)	$\mathcal{H}$ $mem[rs2] \leftarrow rs1[0..31]$
j	C	Ca	
j	I	a	
jal	I	a	
jal	I	d,a	
jalr	C	d	
jalr	I	d,o(s)	
jalr	I	d,s	
jalr	I	d,s,j	
jalr	I	o(s)	
jalr	I	s	
jalr	I	s,j	
jr	C	d	
jr	I	o(s)	
jr	I	s	
jr	I	s,j	
jump	I	c,s	
la	I	d,B	
la.tls.gd	I	d,A	
la.tls.ie rd,symbol	I	d,A	∇ $rd \leftarrow \&symbol$
lb rd,lab	I	d,A	∇ $rd \leftarrow mem[lab][0..7]$
lb rd,imm12(rs1)	I	d,o(s)	$rd \leftarrow mem[rs1+imm12][0..7]$
lbu	I	d,A	
lbu	I	d,o(s)	
ld	C	Ct,Cl(Cs)	
ld	C	d,Cn(Cc)	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
ld	I	d,A	
ld	I	d,o(s)	
lh rd,lab	I	d,A	∇ $rd \leftarrow mem[lab]_{0..15}$
lh rd,im12(rs1)	I	d,o(s)	$rd \leftarrow \pm mem[rs1 + im12]_{0..15}$
lhu rd,im12(rs1)	I	d,A	$rd \leftarrow +mem[rs1 + im12]_{0..15}$
lhu	I	d,o(s)	
li	C	d,Co	
li	C	d,Cv	
li	I	d,I	
li	I	d,j	
lla	I	d,B	
lr.d	A	d,0(s)	
lr.d.aq	A	d,0(s)	
lr.d.aqrl	A	d,0(s)	
lr.d.rl	A	d,0(s)	
lr.w	A	d,0(s)	
lr.w.aq	A	d,0(s)	
lr.w.aqrl	A	d,0(s)	
lr.w.rl	A	d,0(s)	
lui	C	d,Cu	
lui	I	d,u	
lw	C	Ct,Ck(Cs)	
lw	C	d,Cm(Cc)	
lw rd,lab	I	d,A	∇ $rd \leftarrow \pm mem[lab]_{0..32}$
lw rd,im12(rs1)	I	d,o(s)	$rd \leftarrow \pm mem[rs1 + im12]_{0..32}$
lwu rd,lab	I	d,A	∇ $rd \leftarrow mem[lab]_{0..32}$
lwu rd,im12(rs1)	I	d,o(s)	$rd \leftarrow mem[rs1 + im12]_{0..32}$
max rd,rs1,rs2	Zbb	d,s,t	$rd \leftarrow (rs1 < rs2)\ ?\ rs2 : rs1$
maxu rd,r1,r2	Zbb	d,s,t	$rd \leftarrow (r1 <_u r2)\ ?\ r2 : r1$
min rd,r1,r2	Zbb	d,s,t	$rd \leftarrow (r1 <_u r2)\ ?\ r1 : r2$
minu rd,r1,r2	Zbb	d,s,t	$rd \leftarrow (r1 <_u r2)\ ?\ r1 : r2$
move rd,rs1	C	d,CV	∇ $rd \leftarrow rs1$
move rd,rs1	I	d,s	∇ $rd \leftarrow rs1$
mret	I	--	Return from exception
mul rd,rs1,rs2	Zmmul	d,s,t	$rd \leftarrow rs1 \times rs2$
mulh rd,rs1,rs2	Zmmul	d,s,t	$rd \leftarrow (rs1 \times rs2)_{63..127}$
mulhsu rd,rs1,rs2u	Zmmul	d,s,t	Signed rs1 $\times$ unsigned rs2u $rd \leftarrow (rs1 \times rs2u)_{63..127}$
mulhu	Zmmul	d,s,t	
mulw	Zmmul	d,s,t	
mv	C	d,CV	
mv	I	d,s	
neg	I	d,t	
negw	I	d,t	
nop	C	--	
nop	I	--	
not	I	d,s	
or	C	Cs,Ct,Cw	
or	C	Cs,Cw,Ct	
or	I	d,s,j	
or	I	d,s,t	
orc.b	Zbb	d,s	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
ori	I	d,s,j	
orn	Zbb | Zbkb	d,s,t	
pack rd,rs1,rs2	Zbkb	d,s,t	
packh rd,rs1,rs2	Zbkb	d,s,t	
packw rd,rs1,rs2	Zbkb	d,s,t	
pause	Zihintpause	--	
prefetch.i imm12(rs1)	Zicbop	Wif(s)	
prefetch.r imm12(rs1)	Zicbop	Wif(s)	
prefetch.w imm12(rs1)	Zicbop	Wif(s)	
rdcycle	I	d	
rdcycleh	I	d	
rdinstret	I	d	
rdinstreth	I	d	
rdtime	I	d	
rdtimeh	I	d	
rem	M	d,s,t	
remu	M	d,s,t	
remuw	M	d,s,t	
remw	M	d,s,t	
ret	C	--	
ret	I	--	
rev8	Zbb | Zbkb	d,s	
rev8	Zbb | Zbkb	d,s	
rol	Zbb | Zbkb	d,s,t	
rolw	Zbb | Zbkb	d,s,t	
ror	Zbb | Zbkb	d,s,>	
ror	Zbb | Zbkb	d,s,t	
rori	Zbb | Zbkb	d,s,>	
roriw	Zbb | Zbkb	d,s,<	
rorw	Zbb | Zbkb	d,s,<	
rorw	Zbb | Zbkb	d,s,t	
sb rs1,var,rs2	I	t,A,s	▽ auipc + sb $mem[var]_{0..7} \leftarrow rs1_{0..7}$
sb rd, imm12(rs1)	I	t,q(s)	$rd \leftarrow mem[rs1 + imm12]_{0..7}$
sbreak	C	--	
sbreak	I	--	
sc.d	A	d,t,0(s)	
sc.d.aq	A	d,t,0(s)	
sc.d.aqrl	A	d,t,0(s)	
sc.d.rl	A	d,t,0(s)	
sc.w	A	d,t,0(s)	
sc.w.aq	A	d,t,0(s)	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
sc.w.aqrl	A	d,t,0(s)	
sc.w.rl	A	d,t,0(s)	
scall	I	--	
sd	C	CV,CN(Cc)	
sd	C	Ct,Cl(Cs)	
sd	I	t,A,s	
sd	I	t,q(s)	
seqz	I	d,s	
sext.b	I	d,s	
sext.b	Zbb	d,s	
sext.h	I	d,s	
sext.h	Zbb	d,s	
sext.w	C	d,CU	
sext.w	I	d,s	
sfence.inval.ir	svinval	--	
sfence.vm	I	--	
sfence.vm	I	s	
sfence.vma	I	--	
sfence.vma	I	s	
sfence.vma	I	s,t	
sfence.w.inval	svinval	--	
sgt	I	d,t,s	
sgtu	I	d,t,s	
sgtz	I	d,t	
sh rs1,symb,rs2	I	t,A,s	∇
sh	I	t,q(s)	
sh1add	Zba	d,s,t	
sh1add.uw	Zba	d,s,t	
sh2add	Zba	d,s,t	
sh2add.uw	Zba	d,s,t	
sh3add	Zba	d,s,t	
sh3add.uw	Zba	d,s,t	
sha256sig0 rd,rs1	Zknh	d,s	
sha256sig1 rd,rs1	Zknh	d,s	
sha256sum0 rd,rs1	Zknh	d,s	
sha256sum1 rd,rs1	Zknh	d,s	
sha512sig0 rd,rs1,rs2	Zknh	d,s	
sha512sig1 rd,rs1,rs2	Zknh	d,s	
sha512sum0 rd,rs1	Zknh	d,s	
sha512sum0r rd,rs1,rs2	Zknh	d,s,t	
sha512sum1 rd,rs1,rs2	Zknh	d,s	$rd = ror64(rs1,28) \oplus ror64(rs1,34) \oplus ror64(rs1,39)$;[59]
sinval.vma	svinval	s,t	
sll rd,rd, shamt	C	d,CU,C>	$rd \leftarrow rd << shamt$
sll rd,rs1,shamt	I	d,s,>	$rd \leftarrow rs1 << shamt$
sll rd,rs1,rs2	I	d,s,t	$rd \leftarrow rs1 << rs2$
slli rd,rs1,shamt	C	d,CU,C>	$rd \leftarrow rd << shamt$
slli rd,rs,shamt	I	d,s,>	$rd \leftarrow rs << shamt$
slli.uw rd,rs1,shamt	Zba	d,s,>	$rd \leftarrow (rs1_{0..31}) << shamt$
slliw rd,rs1,shamt	I	d,s,<	$rd \leftarrow rs1 << shamt$

[59] $\oplus$ = xor

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
sllw rd,rs1,im5	I	d,s,<	$rd \leftarrow \pm rs1_{0-31} << im5$
sllw	I	d,s,t	
slt	I	d,s,j	
slt	I	d,s,t	
slti	I	d,s,j	
sltiu	I	d,s,j	
sltu	I	d,s,j	
sltu	I	d,s,t	
sltz	I	d,s	
sm3p0 rd,rs1	Zksh	d,s	
sm3p1 rd,rs1,rs2,bs	Zksh	d,s	
sm4ed	Zksed	d,s,t,y	
sm4ks rd,rs1,rs2,bs	Zksed	d,s,t,y	
snez	I	d,t	
sra rd,rsrc1,rsrc2	C	Cs,Cw,C>	$rd \leftarrow rsrc1 >> rsrc2$
sra	I	d,s,>	
sra	I	d,s,t	
srai	C	Cs,Cw,C>	
srai	I	d,s,>	
sraiw	I	d,s,<	
sraw	I	d,s,<	
sraw	I	d,s,t	
sret	I	--	
srl	C	Cs,Cw,C>	
srl	I	d,s,>	
srl	I	d,s,t	
srli	C	Cs,Cw,C>	
srli	I	d,s,>	
srliw	I	d,s,<	
srlw	I	d,s,<	
srlw	I	d,s,t	
sub	C	Cs,Cw,Ct	
sub	I	d,s,t	
subw	C	Cs,Cw,Ct	
subw	I	d,s,t	
sw	C	CV,CM(Cc)	
sw	C	Ct,Ck(Cs)	
sw	I	t,A,s	
sw	I	t,q(s)	
tail	I	c	
th.addsl	ba	d,s,t, Xu2@25	
th.dcache.call	cmo	--	
th.dcache.ciall	cmo	--	
th.dcache.cipa	cmo	s	
th.dcache.cisw	cmo	s	
th.dcache.civa	cmo	s	
th.dcache.cpa	cmo	s	
th.dcache.cpal1	cmo	s	
th.dcache.csw	cmo	s	
th.dcache.cva	cmo	s	
th.dcache.cval1	cmo	s	
th.dcache.iall	cmo	--	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
th.dcache.ipa	cmo	s	
th.dcache.isw	cmo	s	
th.dcache.iva	cmo	s	
th.ext rd, rs1, im1,im2	bb	d,s,Xu6@26, Xu6@2a0	$rd \leftarrow \pm rs1[im1..im2]$
th.extu rd,rs1,im1,im2	bb	d,s,Xu6@26, Xu6@20	$rd \leftarrow rs1[im1..im2]$
th.ff0 rd,rs1	bb	d,s	rd←index first bit 0 in rs1
th.ff1 rd,rs1	bb	d,s	rd←index first bit 1 in rs1
th.flrd	fmemidx	D,s,t,Xu2@25	
th.flrw	fmemidx	D,s,t,Xu2@25	
th.flurd	fmemidx	D,s,t,Xu2@25	
th.flurw	fmemidx	D,s,t,Xu2@25	
th.fmv.hw.x	fmv	d,S	
th.fmv.x.hw	fmv	d,S	
th.fsrd fd, r1, r2, im2	fmemidx	D,s,t,Xu2@25	$mem[r1+(r2 << im2)] \leftarrow fd$
th.fsrw fd, r1, r2, im2	fmemidx	D,s,t,Xu2@25	$mem[r1 + (r2 << im2)]_{0..31} \leftarrow fd_{0..31}$
th.fsurd	fmemidx	D,s,t,Xu2@25	
th.fsurw	fmemidx	D,s,t,Xu2@25	
th.icache.iall	cmo	--	
th.icache.ialls	cmo	--	
th.icache.ipa	cmo	s	
th.icache.iva	cmo	s	
th.ipop	int	--	
th.ipush	int	--	
th.l2cache.call	cmo	--	
th.l2cache.ciall	cmo	--	
th.l2cache.iall	cmo	--	
th.lbia rd, (rs1), im5,im2 Post-increment.	memidx	d,(s), Xs5@20, Xu2@25	$rd \leftarrow \pm(mem[rs1])$ $rs1 \leftarrow rs1 \pm (im5 << im2)$ $rd \neq rs1$
th.lbib rd, (rs1), im5,im2 Pre-increment	memidx	d,(s), Xs5@20, Xu2@25	$rs1 \leftarrow rs1 \pm (im5 << im2)$ $rd \leftarrow \pm(mem[rs1])$ $rd \neq rs1$
th.lbuia rd,(rs1) Post-increment	memidx	d,(s), Xs5@20, Xu2@25	$rd \leftarrow (mem[rs1])$ $rs1 \leftarrow rs1(im5 << im2)$ $rd \neq rs1$
th.lbuibrd, (rs1), im5,im2 Pre-increment	memidx	d,(s), Xs5@20, Xu2@25	$rs1 \leftarrow rs1 + (im5 << im2)$ $rd \leftarrow +(mem[rs1])$ $rd \neq rs1$
th.ldd d1,d2,(s3),im2	mempair	d,t,(s), Xu2@25, Xl4	$d1 \leftarrow mem[r3 + (im2 << 4)]$ $d2 \leftarrow mem[r3+8+im2 << 4]$ $rd1 \neq rd2 \neq r3$
th.ldia rd,(rs1),im5,im2 Post-increment	memidx	d,(s), Xs5@20, Xu2@25	$rd \leftarrow mem[rs1]$ $rs1 \leftarrow rs1 \pm (im5 << im2)$
th.ldib rd,(r1),im5,im2 Pre-increment	memidx	d,(s), Xs5@20, Xu2@25	$r1 \leftarrow r1 \pm (im5 << im2)$ $rd \leftarrow mem[r1]$

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
th.lhia rd,(rs1),im5,im2 Post-increment.	memidx	d,(s), Xs5@20, Xu2@25	$rd \leftarrow \pm mem[rs1]_{0..16}$ $rs1 \leftarrow rs1 \pm (im5 << im2)$
th.lhib rd,(rs1),im5,im2 Pre-increment.	memidx	d,(s), Xs5@20, Xu2@25	$rs1 \leftarrow rs1 \pm (im5 << im2)$ $rd \leftarrow \pm mem[rs1]_{0..16}$
th.lhuia rd,(rs1),im5,im2 Post-increment	memidx	d,(s),Xs5@20, Xu2@25	$rd \leftarrow mem[rs1]_{0..16}$ $rs1 \leftarrow rs1 + (im5 << im2)$
th.lhuib rd,(r1),im5,im2 Pre-increment	memidx	d,(s),Xs5@20, Xu2@25	$r1 \leftarrow r1 + (im5 << im2)$ $rd \leftarrow +mem[r1]_{0..16}$
th.lrb	memidx	d,s,t,Xu2@25	
th.lrbu	memidx	d,s,t,Xu2@25	
th.lrd	memidx	d,s,t,Xu2@25	
th.lrh	memidx	d,s,t,Xu2@25	
th.lrhu	memidx	d,s,t,Xu2@25	
th.lrw	memidx	d,s,t,Xu2@25	
th.lrwu	memidx	d,s,t,Xu2@25	
th.lurb	memidx	d,s,t,Xu2@25	
th.lurbu	memidx	d,s,t,Xu2@25	
th.lurd	memidx	d,s,t,Xu2@25	
th.lurh	memidx	d,s,t,Xu2@25	
th.lurhu	memidx	d,s,t,Xu2@25	
th.lurw	memidx	d,s,t,Xu2@25	
th.lurwu	memidx	d,s,t,Xu2@25	
th.lwd rd,imm7(rs1)	mempair	d,t,(s), Xu2@25, Xl3	$rd \leftarrow \pm mem[rs1 \pm imm7]_{0..31}$ $rd + 1 \leftarrow \pm mem[rs1 \pm imm7]_{32..63}$
th.lwia	memidx	d,(s), Xs5@20, Xu2@25	
th.lwib	memidx	d,(s), Xs5@20 Xu2@25	
th.lwud	mempair	d,t,(s),Xu2@25, Xl3	
th.lwuia	memidx	d,(s),Xs5@20, Xu2@25	
th.lwuib	memidx	d,(s),Xs5@20, Xu2@25	
th.mula	mac	d,s,t	
th.mulah	mac	d,s,t	
th.mulaw	mac	d,s,t	
th.muls rd,r1,r2	mac	d,s,t	$rd_{[0..15]} \leftarrow rd_{-}(r1 \times r2)$
th.mulsh rd,r1,r2	mac	d,s,t	$rd \leftarrow \pm rd_{[0..15]} - (r1_{[0..15]} \times r2_{[0..15]})$
th.mulsw rd,r1,r2	mac	d,s,t	$rd \leftarrow \pm rd_{[0..31]} - (r1_{[0..31]} \times r2_{[0..31]})$
th.mveqz rd,r1,r2	condmov	d,s,t	$rd \leftarrow (r2 = 0)\ ?\ r1\ :rd$
th.mvnez rd,r1,r2	condmov	d,s,t	$rd \leftarrow (r2 \neq 0)\ ?\ r1\ :rd$
th.rev	bb	d,s	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
th.revw	bb	d,s	
th.sbia	memidx	d,(s),Xs5@20, Xu2@25	
th.sbib	memidx	d,(s),Xs5@20, Xu2@25	
th.sdd	mempair	d,t,(s),Xu2@25, X14	
th.sdia	memidx	d,(s),Xs5@20, Xu2@25	
th.sdib	memidx	d,(s),Xs5@20, Xu2@25	
th.sfence.vmas	sync	s,t	
th.shia	memidx	d,(s),Xs5@20, Xu2@25	
th.shib r2,(r1),im5,im2 8 bit store	memidx	d,(s),Xs5@20 Xu2@25	$r1 \leftarrow \pm r1 + (im5 << im2)$ $mem[r1][0..15] \leftarrow r2[0..15]$
th.srb rd,r1,r2,im2	memidx	d,s,t,Xu2@25	$r1 \leftarrow r1 \pm (im5 << im2)$ $mem[r1][0..7] \leftarrow r2[0..7]$
th.srd rd,r1,r2,im2 64 bits store	memidx	d,s,t,Xu2@25	$mem[rs1+rs2 << im2] \leftarrow rd$
th.srh rd,r1,r2,imm2 16bits store	memidx	d,s,t,Xu2@25	$mem[r1+r2 << imm2] \leftarrow rd$
th.srri rd,rs,im6	bb	d,s,Xu6@20	$rd \leftarrow\curvearrowleft (rs >> im6)$
th.srriw rd,rs,im5	bb	d,s,Xu5@20	$rd \leftarrow\curvearrowleft (rs >> im5)[0..31]$
th.srw rd,r1,r2,im2	memidx	d,s,t,Xu2@25	$mem[r1 + r2 << im2] \leftarrow rd[0..31]$
th.surb rd,r1,r2,im2	memidx	d,s,t,Xu2@25	$mem[r1 + r2 << im2] \leftarrow rd[0..7]$
th.surd rd,r1,r2,im2	memidx	d,s,t,Xu2@25	
th.surh rd,r1,r2,im2	memidx	d,s,t,Xu2@25	$mem[r1 + r2 << im2] \leftarrow rd[0..15]$
th.surw	memidx	d,s,t,Xu2@25	$mem[r1 + r2 << im2] \leftarrow rd[0..31]$
th.swd	mempair	d,t,(s),Xu2@25, X13	
th.swia	memidx	d,(s),Xs5@20, Xu2@25	
th.swib	memidx	d,(s),Xs5@20, Xu2@25	
th.sync	sync	--	
th.sync.i	sync	--	
th.sync.is	sync	--	
th.sync.s	sync	--	
th.tst	bs	d,s,Xu6@20	
th.tstnbz	bb	d,s	
unimp	C	--	
unimp	I	--	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
uret	I	--	
vaadd.vv	V	Vd,Vt,VsVm	
vaadd.vx	V	Vd,Vt,sVm	
vaaddu.vv	V	Vd,Vt,VsVm	
vaaddu.vx	V	Vd,Vt,sVm	
vadc.vim	V	Vd,Vt,Vi,V0	
vadc.vvm	V	Vd,Vt,Vs,V0	
vadc.vxm	V	Vd,Vt,s,V0	
vadd.vi vd, vs2, imm, vm	V	Vd,Vt,ViVm	$vd_e \leftarrow \forall vs2_e + imm$
vadd.vv vd, vs2, vs1, vm	V	Vd,Vt,VsVm	$vd_e \leftarrow vs2_e + vs1_e$
vadd.vx vd,vs2,rs1,vm	V	Vd,Vt,sVm	$vd_e \leftarrow vs2_e + rs1$
vand.vi	V	Vd,Vt,ViVm	
vand.vv	V	Vd,Vt,VsVm	
vand.vx	V	Vd,Vt,sVm	
vasub.vv	V	Vd,Vt,VsVm	
vasub.vx	V	Vd,Vt,sVm	
vasubu.vv	V	Vd,Vt,VsVm	
vasubu.vx	V	Vd,Vt,sVm	
vcompress.vm	V	Vd,Vt,Vs	
vcpop.m	V	d,VtVm	
vdiv.vv	V	Vd,Vt,VsVm	
vdiv.vx	V	Vd,Vt,sVm	
vdivu.vv	V	Vd,Vt,VsVm	
vdivu.vx	V	Vd,Vt,sVm	
vfabs.v	Zvef	Vd,VuVm	
vfadd.vf	Zvef	Vd,Vt,SVm	
vfadd.vv	Zvef	Vd,Vt,VsVm	
vfclass.v	Zvef	Vd,VtVm	
vfcvt.f.x.v	Zvef	Vd,VtVm	
vfcvt.f.xu.v	Zvef	Vd,VtVm	
vfcvt.rtz.x.f.v	Zvef	Vd,VtVm	
vfcvt.rtz.xu.f.v	Zvef	Vd,VtVm	
vfcvt.x.f.v	Zvef	Vd,VtVm	
vfcvt.xu.f.v	Zvef	Vd,VtVm	
vfdiv.vf	Zvef	Vd,Vt,SVm	
vfdiv.vv	Zvef	Vd,Vt,VsVm	
vfirst.m	V	d,VtVm	
vfmacc.vf	Zvef	Vd,S,VtVm	
vfmacc.vv	Zvef	Vd,Vs,VtVm	
vfmadd.vf	Zvef	Vd,S,VtVm	
vfmadd.vv	Zvef	Vd,Vs,VtVm	
vfmax.vf	Zvef	Vd,Vt,SVm	
vfmax.vv	Zvef	Vd,Vt,VsVm	
vfmerge.vfm	Zvef	Vd,Vt,S,V0	
vfmin.vf	Zvef	Vd,Vt,SVm	
vfmin.vv	Zvef	Vd,Vt,VsVm	
vfmsac.vf	Zvef	Vd,S,VtVm	
vfmsac.vv	Zvef	Vd,Vs,VtVm	
vfmsub.vf	Zvef	Vd,S,VtVm	
vfmsub.vv	Zvef	Vd,Vs,VtVm	
vfmul.vf	Zvef	Vd,Vt,SVm	
vfmul.vv	Zvef	Vd,Vt,VsVm	
vfmv.f.s	Zvef	D,Vt	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
vfmv.s.f	Zvef	Vd,S	
vfmv.v.f	Zvef	Vd,S	
vfncvt.f.f.w	Zvef	Vd,VtVm	
vfncvt.f.x.w	Zvef	Vd,VtVm	
vfncvt.f.xu.w	Zvef	Vd,VtVm	
vfncvt.rod.f.f.w	Zvef	Vd,VtVm	
vfncvt.rtz.x.f.w	Zvef	Vd,VtVm	
vfncvt.rtz.xu.f.w	Zvef	Vd,VtVm	
vfncvt.x.f.w	Zvef	Vd,VtVm	
vfncvt.xu.f.w	Zvef	Vd,VtVm	
vfneg.v	Zvef	Vd,VuVm	
vfnmacc.vf	Zvef	Vd,S,VtVm	
vfnmacc.vv	Zvef	Vd,Vs,VtVm	
vfnmadd.vf	Zvef	Vd,S,VtVm	
vfnmadd.vv	Zvef	Vd,Vs,VtVm	
vfnmsac.vf	Zvef	Vd,S,VtVm	
vfnmsac.vv	Zvef	Vd,Vs,VtVm	
vfnmsub.vf	Zvef	Vd,S,VtVm	
vfnmsub.vv	Zvef	Vd,Vs,VtVm	
vfrdiv.vf	Zvef	Vd,Vt,SVm	
vfrec7.v	Zvef	Vd,VtVm	
vfrece7.v	Zvef	Vd,VtVm	
vfredmax.vs	Zvef	Vd,Vt,VsVm	
vfredmin.vs	Zvef	Vd,Vt,VsVm	
vfredosum.vs	Zvef	Vd,Vt,VsVm	
vfredsum.vs	Zvef	Vd,Vt,VsVm	
vfredusum.vs	Zvef	Vd,Vt,VsVm	
vfrsqrt7.v	Zvef	Vd,VtVm	
vfrsqrte7.v	Zvef	Vd,VtVm	
vfrsub.vf	Zvef	Vd,Vt,SVm	
vfsgnj.vf	Zvef	Vd,Vt,SVm	
vfsgnj.vv	Zvef	Vd,Vt,VsVm	
vfsgnjn.vf	Zvef	Vd,Vt,SVm	
vfsgnjn.vv	Zvef	Vd,Vt,VsVm	
vfsgnjx.vf	Zvef	Vd,Vt,SVm	
vfsgnjx.vv	Zvef	Vd,Vt,VsVm	
vfslide1down.vf	Zvef	Vd,Vt,SVm	
vfslide1up.vf	Zvef	Vd,Vt,SVm	
vfsqrt.v	Zvef	Vd,VtVm	
vfsub.vf	Zvef	Vd,Vt,SVm	
vfsub.vv	Zvef	Vd,Vt,VsVm	
vfwadd.vf	Zvef	Vd,Vt,SVm	
vfwadd.vv	Zvef	Vd,Vt,VsVm	
vfwadd.wf	Zvef	Vd,Vt,SVm	
vfwadd.wv	Zvef	Vd,Vt,VsVm	
vfwcvt.f.f.v	Zvef	Vd,VtVm	
vfwcvt.f.x.v	Zvef	Vd,VtVm	
vfwcvt.f.xu.v	Zvef	Vd,VtVm	
vfwcvt.rtz.x.f.v	Zvef	Vd,VtVm	
vfwcvt.rtz.xu.f.v	Zvef	Vd,VtVm	
vfwcvt.x.f.v	Zvef	Vd,VtVm	
vfwcvt.xu.f.v	Zvef	Vd,VtVm	
vfwmacc.vf	Zvef	Vd,S,VtVm	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
vfwmacc.vv	Zvef	Vd,Vs,VtVm	
vfwmsac.vf	Zvef	Vd,S,VtVm	
vfwmsac.vv	Zvef	Vd,Vs,VtVm	
vfwmul.vf	Zvef	Vd,Vt,SVm	
vfwmul.vv	Zvef	Vd,Vt,VsVm	
vfwnmacc.vf	Zvef	Vd,S,VtVm	
vfwnmacc.vv	Zvef	Vd,Vs,VtVm	
vfwnmsac.vf	Zvef	Vd,S,VtVm	
vfwnmsac.vv	Zvef	Vd,Vs,VtVm	
vfwredosum.vs	Zvef	Vd,Vt,VsVm	
vfwredsum.vs	Zvef	Vd,Vt,VsVm	
vfwredusum.vs	Zvef	Vd,Vt,VsVm	
vfwsub.vf	Zvef	Vd,Vt,SVm	
vfwsub.vv	Zvef	Vd,Vt,VsVm	
vfwsub.wf	Zvef	Vd,Vt,SVm	
vfwsub.wv	Zvef	Vd,Vt,VsVm	
vid.v	V	VdVm	
viota.m	V	Vd,VtVm	
vl1r.v	V	Vd,0(s)	
vl1re16.v	V	Vd,0(s)	
vl1re32.v	V	Vd,0(s)	
vl1re64.v	V	Vd,0(s)	
vl1re8.v	V	Vd,0(s)	
vl2r.v	V	Vd,0(s)	
vl2re16.v	V	Vd,0(s)	
vl2re32.v	V	Vd,0(s)	
vl2re64.v	V	Vd,0(s)	
vl2re8.v	V	Vd,0(s)	
vl4r.v	V	Vd,0(s)	
vl4re16.v	V	Vd,0(s)	
vl4re32.v	V	Vd,0(s)	
vl4re64.v	V	Vd,0(s)	
vl4re8.v	V	Vd,0(s)	
vl8r.v	V	Vd,0(s)	
vl8re16.v	V	Vd,0(s)	
vl8re32.v	V	Vd,0(s)	
vl8re64.v	V	Vd,0(s)	
vl8re8.v	V	Vd,0(s)	
vle1.v	V	Vd,0(s)	
vle16.v	V	Vd,0(s)Vm	
vle16ff.v	V	Vd,0(s)Vm	
vle32.v	V	Vd,0(s)Vm	
vle32ff.v	V	Vd,0(s)Vm	
vle64.v	V	Vd,0(s)Vm	
vle64ff.v	V	Vd,0(s)Vm	
vle8.v	V	Vd,0(s)Vm	
vle8ff.v	V	Vd,0(s)Vm	
vlm.v	V	Vd,0(s)	
vloxei16.v	V	Vd,0(s),VtVm	
vloxei32.v	V	Vd,0(s),VtVm	
vloxei64.v	V	Vd,0(s),VtVm	
vloxei8.v	V	Vd,0(s),VtVm	
vloxseg2ei16.v	V	Vd,0(s),VtVm	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
vloxseg2ei32.v	V	Vd,0(s),VtVm	
vloxseg2ei64.v	V	Vd,0(s),VtVm	
vloxseg2ei8.v	V	Vd,0(s),VtVm	
vloxseg3ei16.v	V	Vd,0(s),VtVm	
vloxseg3ei32.v	V	Vd,0(s),VtVm	
vloxseg3ei64.v	V	Vd,0(s),VtVm	
vloxseg3ei8.v	V	Vd,0(s),VtVm	
vloxseg4ei16.v	V	Vd,0(s),VtVm	
vloxseg4ei32.v	V	Vd,0(s),VtVm	
vloxseg4ei64.v	V	Vd,0(s),VtVm	
vloxseg4ei8.v	V	Vd,0(s),VtVm	
vloxseg5ei16.v	V	Vd,0(s),VtVm	
vloxseg5ei32.v	V	Vd,0(s),VtVm	
vloxseg5ei64.v	V	Vd,0(s),VtVm	
vloxseg5ei8.v	V	Vd,0(s),VtVm	
vloxseg6ei16.v	V	Vd,0(s),VtVm	
vloxseg6ei32.v	V	Vd,0(s),VtVm	
vloxseg6ei64.v	V	Vd,0(s),VtVm	
vloxseg6ei8.v	V	Vd,0(s),VtVm	
vloxseg7ei16.v	V	Vd,0(s),VtVm	
vloxseg7ei32.v	V	Vd,0(s),VtVm	
vloxseg7ei64.v	V	Vd,0(s),VtVm	
vloxseg7ei8.v	V	Vd,0(s),VtVm	
vloxseg8ei16.v	V	Vd,0(s),VtVm	
vloxseg8ei32.v	V	Vd,0(s),VtVm	
vloxseg8ei64.v	V	Vd,0(s),VtVm	
vloxseg8ei8.v	V	Vd,0(s),VtVm	
vlse16.v	V	Vd,0(s),tVm	
vlse32.v	V	Vd,0(s),tVm	
vlse64.v	V	Vd,0(s),tVm	
vlse8.v	V	Vd,0(s),tVm	
vlseg2e16.v	V	Vd,0(s)Vm	
vlseg2e16ff.v	V	Vd,0(s)Vm	
vlseg2e32.v	V	Vd,0(s)Vm	
vlseg2e32ff.v	V	Vd,0(s)Vm	
vlseg2e64.v	V	Vd,0(s)Vm	
vlseg2e64ff.v	V	Vd,0(s)Vm	
vlseg2e8.v	V	Vd,0(s)Vm	
vlseg2e8ff.v	V	Vd,0(s)Vm	
vlseg3e16.v	V	Vd,0(s)Vm	
vlseg3e16ff.v	V	Vd,0(s)Vm	
vlseg3e32.v	V	Vd,0(s)Vm	
vlseg3e32ff.v	V	Vd,0(s)Vm	
vlseg3e64.v	V	Vd,0(s)Vm	
vlseg3e64ff.v	V	Vd,0(s)Vm	
vlseg3e8.v	V	Vd,0(s)Vm	
vlseg3e8ff.v	V	Vd,0(s)Vm	
vlseg4e16.v	V	Vd,0(s)Vm	
vlseg4e16ff.v	V	Vd,0(s)Vm	
vlseg4e32.v	V	Vd,0(s)Vm	
vlseg4e32ff.v	V	Vd,0(s)Vm	
vlseg4e64.v	V	Vd,0(s)Vm	
vlseg4e64ff.v	V	Vd,0(s)Vm	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
vlseg4e8.v	V	Vd,0(s)Vm	
vlseg4e8ff.v	V	Vd,0(s)Vm	
vlseg5e16.v	V	Vd,0(s)Vm	
vlseg5e16ff.v	V	Vd,0(s)Vm	
vlseg5e32.v	V	Vd,0(s)Vm	
vlseg5e32ff.v	V	Vd,0(s)Vm	
vlseg5e64.v	V	Vd,0(s)Vm	
vlseg5e64ff.v	V	Vd,0(s)Vm	
vlseg5e8.v	V	Vd,0(s)Vm	
vlseg5e8ff.v	V	Vd,0(s)Vm	
vlseg6e16.v	V	Vd,0(s)Vm	
vlseg6e16ff.v	V	Vd,0(s)Vm	
vlseg6e32.v	V	Vd,0(s)Vm	
vlseg6e32ff.v	V	Vd,0(s)Vm	
vlseg6e64.v	V	Vd,0(s)Vm	
vlseg6e64ff.v	V	Vd,0(s)Vm	
vlseg6e8.v	V	Vd,0(s)Vm	
vlseg6e8ff.v	V	Vd,0(s)Vm	
vlseg7e16.v	V	Vd,0(s)Vm	
vlseg7e16ff.v	V	Vd,0(s)Vm	
vlseg7e32.v	V	Vd,0(s)Vm	
vlseg7e32ff.v	V	Vd,0(s)Vm	
vlseg7e64.v	V	Vd,0(s)Vm	
vlseg7e64ff.v	V	Vd,0(s)Vm	
vlseg7e8.v	V	Vd,0(s)Vm	
vlseg7e8ff.v	V	Vd,0(s)Vm	
vlseg8e16.v	V	Vd,0(s)Vm	
vlseg8e16ff.v	V	Vd,0(s)Vm	
vlseg8e32.v	V	Vd,0(s)Vm	
vlseg8e32ff.v	V	Vd,0(s)Vm	
vlseg8e64.v	V	Vd,0(s)Vm	
vlseg8e64ff.v	V	Vd,0(s)Vm	
vlseg8e8.v	V	Vd,0(s)Vm	
vlseg8e8ff.v	V	Vd,0(s)Vm	
vlsseg2e16.v	V	Vd,0(s),tVm	
vlsseg2e32.v	V	Vd,0(s),tVm	
vlsseg2e64.v	V	Vd,0(s),tVm	
vlsseg2e8.v	V	Vd,0(s),tVm	
vlsseg3e16.v	V	Vd,0(s),tVm	
vlsseg3e32.v	V	Vd,0(s),tVm	
vlsseg3e64.v	V	Vd,0(s),tVm	
vlsseg3e8.v	V	Vd,0(s),tVm	
vlsseg4e16.v	V	Vd,0(s),tVm	
vlsseg4e32.v	V	Vd,0(s),tVm	
vlsseg4e64.v	V	Vd,0(s),tVm	
vlsseg4e8.v	V	Vd,0(s),tVm	
vlsseg5e16.v	V	Vd,0(s),tVm	
vlsseg5e32.v	V	Vd,0(s),tVm	
vlsseg5e64.v	V	Vd,0(s),tVm	
vlsseg5e8.v	V	Vd,0(s),tVm	
vlsseg6e16.v	V	Vd,0(s),tVm	
vlsseg6e32.v	V	Vd,0(s),tVm	
vlsseg6e64.v	V	Vd,0(s),tVm	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
vlsseg6e8.v	V	Vd,0(s),tVm	
vlsseg7e16.v	V	Vd,0(s),tVm	
vlsseg7e32.v	V	Vd,0(s),tVm	
vlsseg7e64.v	V	Vd,0(s),tVm	
vlsseg7e8.v	V	Vd,0(s),tVm	
vlsseg8e16.v	V	Vd,0(s),tVm	
vlsseg8e32.v	V	Vd,0(s),tVm	
vlsseg8e64.v	V	Vd,0(s),tVm	
vlsseg8e8.v	V	Vd,0(s),tVm	
vluxei16.v	V	Vd,0(s),VtVm	
vluxei32.v	V	Vd,0(s),VtVm	
vluxei64.v	V	Vd,0(s),VtVm	
vluxei8.v	V	Vd,0(s),VtVm	
vluxseg2ei16.v	V	Vd,0(s),VtVm	
vluxseg2ei32.v	V	Vd,0(s),VtVm	
vluxseg2ei64.v	V	Vd,0(s),VtVm	
vluxseg2ei8.v	V	Vd,0(s),VtVm	
vluxseg3ei16.v	V	Vd,0(s),VtVm	
vluxseg3ei32.v	V	Vd,0(s),VtVm	
vluxseg3ei64.v	V	Vd,0(s),VtVm	
vluxseg3ei8.v	V	Vd,0(s),VtVm	
vluxseg4ei16.v	V	Vd,0(s),VtVm	
vluxseg4ei32.v	V	Vd,0(s),VtVm	
vluxseg4ei64.v	V	Vd,0(s),VtVm	
vluxseg4ei8.v	V	Vd,0(s),VtVm	
vluxseg5ei16.v	V	Vd,0(s),VtVm	
vluxseg5ei32.v	V	Vd,0(s),VtVm	
vluxseg5ei64.v	V	Vd,0(s),VtVm	
vluxseg5ei8.v	V	Vd,0(s),VtVm	
vluxseg6ei16.v	V	Vd,0(s),VtVm	
vluxseg6ei32.v	V	Vd,0(s),VtVm	
vluxseg6ei64.v	V	Vd,0(s),VtVm	
vluxseg6ei8.v	V	Vd,0(s),VtVm	
vluxseg7ei16.v	V	Vd,0(s),VtVm	
vluxseg7ei32.v	V	Vd,0(s),VtVm	
vluxseg7ei64.v	V	Vd,0(s),VtVm	
vluxseg7ei8.v	V	Vd,0(s),VtVm	
vluxseg8ei16.v	V	Vd,0(s),VtVm	
vluxseg8ei32.v	V	Vd,0(s),VtVm	
vluxseg8ei64.v	V	Vd,0(s),VtVm	
vluxseg8ei8.v	V	Vd,0(s),VtVm	
vmacc.vv	V	Vd,Vs,VtVm	
vmacc.vx	V	Vd,s,VtVm	
vmadc.vi	V	Vd,Vt,Vi	
vmadc.vim	V	Vd,Vt,Vi,V0	
vmadc.vv	V	Vd,Vt,Vs	
vmadc.vvm	V	Vd,Vt,Vs,V0	
vmadc.vx	V	Vd,Vt,s	
vmadc.vxm	V	Vd,Vt,s,V0	
vmadd.vv	V	Vd,Vs,VtVm	
vmadd.vx	V	Vd,s,VtVm	
vmand.mm	V	Vd,Vt,Vs	
vmandn.mm	V	Vd,Vt,Vs	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
vmandnot.mm	V	Vd,Vt,Vs	
vmax.vv	V	Vd,Vt,VsVm	
vmax.vx	V	Vd,Vt,sVm	
vmaxu.vv	V	Vd,Vt,VsVm	
vmaxu.vx	V	Vd,Vt,sVm	
vmclr.m	V	Vv	
vmcpy.m	V	Vd,Vu	
vmerge.vim	V	Vd,Vt,Vi,V0	
vmerge.vvm	V	Vd,Vt,Vs,V0	
vmerge.vxm	V	Vd,Vt,s,V0	
vmfeq.vf	Zvef	Vd,Vt,SVm	
vmfeq.vv	Zvef	Vd,Vt,VsVm	
vmfge.vf	Zvef	Vd,Vt,SVm	
vmfge.vv	Zvef	Vd,Vs,VtVm	
vmfgt.vf	Zvef	Vd,Vt,SVm	
vmfgt.vv	Zvef	Vd,Vs,VtVm	
vmfle.vf	Zvef	Vd,Vt,SVm	
vmfle.vv	Zvef	Vd,Vt,VsVm	
vmflt.vf	Zvef	Vd,Vt,SVm	
vmflt.vv	Zvef	Vd,Vt,VsVm	
vmfne.vf	Zvef	Vd,Vt,SVm	
vmfne.vv	Zvef	Vd,Vt,VsVm	
vmin.vv	V	Vd,Vt,VsVm	
vmin.vx	V	Vd,Vt,sVm	
vminu.vv	V	Vd,Vt,VsVm	
vminu.vx	V	Vd,Vt,sVm	
vmmv.m	V	Vd,Vu	
vmnand.mm	V	Vd,Vt,Vs	
vmnor.mm	V	Vd,Vt,Vs	
vmnot.m	V	Vd,Vu	
vmor.mm	V	Vd,Vt,Vs	
vmorn.mm	V	Vd,Vt,Vs	
vmornot.mm	V	Vd,Vt,Vs	
vmsbc.vv	V	Vd,Vt,Vs	
vmsbc.vvm	V	Vd,Vt,Vs,V0	
vmsbc.vx	V	Vd,Vt,s	
vmsbc.vxm	V	Vd,Vt,s,V0	
vmsbf.m	V	Vd,VtVm	
vmseq.vi	V	Vd,Vt,ViVm	
vmseq.vv	V	Vd,Vt,VsVm	
vmseq.vx	V	Vd,Vt,sVm	
vmset.m	V	Vv	
vmsge.vi	V	Vd,Vt,VkVm	
vmsge.vv	V	Vd,Vs,VtVm	
vmsge.vx	V	Vd,Vt,s,VM,VT	
vmsge.vx	V	Vd,Vt,sVm	
vmsgeu.vi	V	Vd,Vt,VkVm	
vmsgeu.vi	V	Vd,Vu,OVm	
vmsgeu.vv	V	Vd,Vs,VtVm	
vmsgeu.vx	V	Vd,Vt,s,VM,VT	
vmsgeu.vx	V	Vd,Vt,sVm	
vmsgt.vi	V	Vd,Vt,ViVm	
vmsgt.vv	V	Vd,Vs,VtVm	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
vmsgt.vx	V	Vd,Vt,sVm	
vmsgtu.vi	V	Vd,Vt,ViVm	
vmsgtu.vv	V	Vd,Vs,VtVm	
vmsgtu.vx	V	Vd,Vt,sVm	
vmsif.m	V	Vd,VtVm	
vmsle.vi	V	Vd,Vt,ViVm	
vmsle.vv	V	Vd,Vt,VsVm	
vmsle.vx	V	Vd,Vt,sVm	
vmsleu.vi	V	Vd,Vt,ViVm	
vmsleu.vv	V	Vd,Vt,VsVm	
vmsleu.vx	V	Vd,Vt,sVm	
vmslt.vi	V	Vd,Vt,VkVm	
vmslt.vv	V	Vd,Vt,VsVm	
vmslt.vx	V	Vd,Vt,sVm	
vmsltu.vi	V	Vd,Vt,VkVm	
vmsltu.vi	V	Vd,Vu,OVm	
vmsltu.vv	V	Vd,Vt,VsVm	
vmsltu.vx	V	Vd,Vt,sVm	
vmsne.vi	V	Vd,Vt,ViVm	
vmsne.vv	V	Vd,Vt,VsVm	
vmsne.vx	V	Vd,Vt,sVm	
vmsof.m	V	Vd,VtVm	
vmul.vv	V	Vd,Vt,VsVm	
vmul.vx	V	Vd,Vt,sVm	
vmulh.vv	V	Vd,Vt,VsVm	
vmulh.vx	V	Vd,Vt,sVm	
vmulhsu.vv	V	Vd,Vt,VsVm	
vmulhsu.vx	V	Vd,Vt,sVm	
vmulhu.vv	V	Vd,Vt,VsVm	
vmulhu.vx	V	Vd,Vt,sVm	
vmv.s.x	V	Vd,s	
vmv.v.i	V	Vd,Vi	
vmv.v.v	V	Vd,Vs	
vmv.v.x	V	Vd,s	
vmv.x.s	V	d,Vt	
vmv1r.v	V	Vd,Vt	
vmv2r.v	V	Vd,Vt	
vmv4r.v	V	Vd,Vt	
vmv8r.v	V	Vd,Vt	
vmxnor.mm	V	Vd,Vt,Vs	
vmxor.mm	V	Vd,Vt,Vs	
vnclip.wi	V	Vd,Vt,VjVm	
vnclip.wv	V	Vd,Vt,VsVm	
vnclip.wx	V	Vd,Vt,sVm	
vnclipu.wi	V	Vd,Vt,VjVm	
vnclipu.wv	V	Vd,Vt,VsVm	
vnclipu.wx	V	Vd,Vt,sVm	
vncvt.x.x.w	V	Vd,VtVm	
vneg.v	V	Vd,VtVm	
vnmsac.vv	V	Vd,Vs,VtVm	
vnmsac.vx	V	Vd,s,VtVm	
vnmsub.vv	V	Vd,Vs,VtVm	
vnmsub.vx	V	Vd,s,VtVm	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
vnot.v	V	Vd,VtVm	
vnsra.wi	V	Vd,Vt,VjVm	
vnsra.wv	V	Vd,Vt,VsVm	
vnsra.wx	V	Vd,Vt,sVm	
vnsrl.wi	V	Vd,Vt,VjVm	
vnsrl.wv	V	Vd,Vt,VsVm	
vnsrl.wx	V	Vd,Vt,sVm	
vor.vi	V	Vd,Vt,ViVm	
vor.vv	V	Vd,Vt,VsVm	
vor.vx	V	Vd,Vt,sVm	
vpopc.m	V	d,VtVm	
vredand.vs	V	Vd,Vt,VsVm	
vredmax.vs	V	Vd,Vt,VsVm	
vredmaxu.vs	V	Vd,Vt,VsVm	
vredmin.vs	V	Vd,Vt,VsVm	
vredminu.vs	V	Vd,Vt,VsVm	
vredor.vs	V	Vd,Vt,VsVm	
vredsum.vs	V	Vd,Vt,VsVm	
vredxor.vs	V	Vd,Vt,VsVm	
vrem.vv	V	Vd,Vt,VsVm	
vrem.vx	V	Vd,Vt,sVm	
vremu.vv	V	Vd,Vt,VsVm	
vremu.vx	V	Vd,Vt,sVm	
vrgather.vi	V	Vd,Vt,VjVm	
vrgather.vv	V	Vd,Vt,VsVm	
vrgather.vx	V	Vd,Vt,sVm	
vrgatherei16.vv	V	Vd,Vt,VsVm	
vrsub.vi	V	Vd,Vt,ViVm	
vrsub.vx	V	Vd,Vt,sVm	
vs1r.v	V	Vd,0(s)	
vs2r.v	V	Vd,0(s)	
vs4r.v	V	Vd,0(s)	
vs8r.v	V	Vd,0(s)	
vsadd.vi	V	Vd,Vt,ViVm	
vsadd.vv	V	Vd,Vt,VsVm	
vsadd.vx	V	Vd,Vt,sVm	
vsaddu.vi	V	Vd,Vt,ViVm	
vsaddu.vv	V	Vd,Vt,VsVm	
vsaddu.vx	V	Vd,Vt,sVm	
vsbc.vvm	V	Vd,Vt,Vs,V0	
vsbc.vxm	V	Vd,Vt,s,V0	
vse1.v	V	Vd,0(s)	
vse16.v	V	Vd,0(s)Vm	
vse32.v	V	Vd,0(s)Vm	
vse64.v	V	Vd,0(s)Vm	
vse8.v	V	Vd,0(s)Vm	
vsetivli	V	d,Z,Vb	
vsetvl	V	d,s,t	
vsetvli	V	d,s,Vc	
vsext.vf2	V	Vd,VtVm	
vsext.vf4	V	Vd,VtVm	
vsext.vf8	V	Vd,VtVm	
vslide1down.vx	V	Vd,Vt,sVm	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
vslide1up.vx	V	Vd,Vt,sVm	
vslidedown.vi	V	Vd,Vt,VjVm	
vslidedown.vx	V	Vd,Vt,sVm	
vslideup.vi	V	Vd,Vt,VjVm	
vslideup.vx	V	Vd,Vt,sVm	
vsll.vi	V	Vd,Vt,VjVm	
vsll.vv	V	Vd,Vt,VsVm	
vsll.vx	V	Vd,Vt,sVm	
vsm.v	V	Vd,0(s)	
vsmul.vv	V	Vd,Vt,VsVm	
vsmul.vx	V	Vd,Vt,sVm	
vsoxei16.v	V	Vd,0(s),VtVm	
vsoxei32.v	V	Vd,0(s),VtVm	
vsoxei64.v	V	Vd,0(s),VtVm	
vsoxei8.v	V	Vd,0(s),VtVm	
vsoxseg2ei16.v	V	Vd,0(s),VtVm	
vsoxseg2ei32.v	V	Vd,0(s),VtVm	
vsoxseg2ei64.v	V	Vd,0(s),VtVm	
vsoxseg2ei8.v	V	Vd,0(s),VtVm	
vsoxseg3ei16.v	V	Vd,0(s),VtVm	
vsoxseg3ei32.v	V	Vd,0(s),VtVm	
vsoxseg3ei64.v	V	Vd,0(s),VtVm	
vsoxseg3ei8.v	V	Vd,0(s),VtVm	
vsoxseg4ei16.v	V	Vd,0(s),VtVm	
vsoxseg4ei32.v	V	Vd,0(s),VtVm	
vsoxseg4ei64.v	V	Vd,0(s),VtVm	
vsoxseg4ei8.v	V	Vd,0(s),VtVm	
vsoxseg5ei16.v	V	Vd,0(s),VtVm	
vsoxseg5ei32.v	V	Vd,0(s),VtVm	
vsoxseg5ei64.v	V	Vd,0(s),VtVm	
vsoxseg5ei8.v	V	Vd,0(s),VtVm	
vsoxseg6ei16.v	V	Vd,0(s),VtVm	
vsoxseg6ei32.v	V	Vd,0(s),VtVm	
vsoxseg6ei64.v	V	Vd,0(s),VtVm	
vsoxseg6ei8.v	V	Vd,0(s),VtVm	
vsoxseg7ei16.v	V	Vd,0(s),VtVm	
vsoxseg7ei32.v	V	Vd,0(s),VtVm	
vsoxseg7ei64.v	V	Vd,0(s),VtVm	
vsoxseg7ei8.v	V	Vd,0(s),VtVm	
vsoxseg8ei16.v	V	Vd,0(s),VtVm	
vsoxseg8ei32.v	V	Vd,0(s),VtVm	
vsoxseg8ei64.v	V	Vd,0(s),VtVm	
vsoxseg8ei8.v	V	Vd,0(s),VtVm	
vsra.vi	V	Vd,Vt,VjVm	
vsra.vv	V	Vd,Vt,VsVm	
vsra.vx	V	Vd,Vt,sVm	
vsrl.vi	V	Vd,Vt,VjVm	
vsrl.vv	V	Vd,Vt,VsVm	
vsrl.vx	V	Vd,Vt,sVm	
vsse16.v	V	Vd,0(s),tVm	
vsse32.v	V	Vd,0(s),tVm	
vsse64.v	V	Vd,0(s),tVm	
vsse8.v	V	Vd,0(s),tVm	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
vsseg2e16.v	V	Vd,0(s)Vm	
vsseg2e32.v	V	Vd,0(s)Vm	
vsseg2e64.v	V	Vd,0(s)Vm	
vsseg2e8.v	V	Vd,0(s)Vm	
vsseg3e16.v	V	Vd,0(s)Vm	
vsseg3e32.v	V	Vd,0(s)Vm	
vsseg3e64.v	V	Vd,0(s)Vm	
vsseg3e8.v	V	Vd,0(s)Vm	
vsseg4e16.v	V	Vd,0(s)Vm	
vsseg4e32.v	V	Vd,0(s)Vm	
vsseg4e64.v	V	Vd,0(s)Vm	
vsseg4e8.v	V	Vd,0(s)Vm	
vsseg5e16.v	V	Vd,0(s)Vm	
vsseg5e32.v	V	Vd,0(s)Vm	
vsseg5e64.v	V	Vd,0(s)Vm	
vsseg5e8.v	V	Vd,0(s)Vm	
vsseg6e16.v	V	Vd,0(s)Vm	
vsseg6e32.v	V	Vd,0(s)Vm	
vsseg6e64.v	V	Vd,0(s)Vm	
vsseg6e8.v	V	Vd,0(s)Vm	
vsseg7e16.v	V	Vd,0(s)Vm	
vsseg7e32.v	V	Vd,0(s)Vm	
vsseg7e64.v	V	Vd,0(s)Vm	
vsseg7e8.v	V	Vd,0(s)Vm	
vsseg8e16.v	V	Vd,0(s)Vm	
vsseg8e32.v	V	Vd,0(s)Vm	
vsseg8e64.v	V	Vd,0(s)Vm	
vsseg8e8.v	V	Vd,0(s)Vm	
vssra.vi	V	Vd,Vt,VjVm	
vssra.vv	V	Vd,Vt,VsVm	
vssra.vx	V	Vd,Vt,sVm	
vssrl.vi	V	Vd,Vt,VjVm	
vssrl.vv	V	Vd,Vt,VsVm	
vssrl.vx	V	Vd,Vt,sVm	
vssseg2e16.v	V	Vd,0(s),tVm	
vssseg2e32.v	V	Vd,0(s),tVm	
vssseg2e64.v	V	Vd,0(s),tVm	
vssseg2e8.v	V	Vd,0(s),tVm	
vssseg3e16.v	V	Vd,0(s),tVm	
vssseg3e32.v	V	Vd,0(s),tVm	
vssseg3e64.v	V	Vd,0(s),tVm	
vssseg3e8.v	V	Vd,0(s),tVm	
vssseg4e16.v	V	Vd,0(s),tVm	
vssseg4e32.v	V	Vd,0(s),tVm	
vssseg4e64.v	V	Vd,0(s),tVm	
vssseg4e8.v	V	Vd,0(s),tVm	
vssseg5e16.v	V	Vd,0(s),tVm	
vssseg5e32.v	V	Vd,0(s),tVm	
vssseg5e64.v	V	Vd,0(s),tVm	
vssseg5e8.v	V	Vd,0(s),tVm	
vssseg6e16.v	V	Vd,0(s),tVm	
vssseg6e32.v	V	Vd,0(s),tVm	
vssseg6e64.v	V	Vd,0(s),tVm	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
vssseg6e8.v	V	Vd,0(s),tVm	
vssseg7e16.v	V	Vd,0(s),tVm	
vssseg7e32.v	V	Vd,0(s),tVm	
vssseg7e64.v	V	Vd,0(s),tVm	
vssseg7e8.v	V	Vd,0(s),tVm	
vssseg8e16.v	V	Vd,0(s),tVm	
vssseg8e32.v	V	Vd,0(s),tVm	
vssseg8e64.v	V	Vd,0(s),tVm	
vssseg8e8.v	V	Vd,0(s),tVm	
vssub.vv	V	Vd,Vt,VsVm	
vssub.vx	V	Vd,Vt,sVm	
vssubu.vv	V	Vd,Vt,VsVm	
vssubu.vx	V	Vd,Vt,sVm	
vsub.vv	V	Vd,Vt,VsVm	
vsub.vx	V	Vd,Vt,sVm	
vsuxei16.v	V	Vd,0(s),VtVm	
vsuxei32.v	V	Vd,0(s),VtVm	
vsuxei64.v	V	Vd,0(s),VtVm	
vsuxei8.v	V	Vd,0(s),VtVm	
vsuxseg2ei16.v	V	Vd,0(s),VtVm	
vsuxseg2ei32.v	V	Vd,0(s),VtVm	
vsuxseg2ei64.v	V	Vd,0(s),VtVm	
vsuxseg2ei8.v	V	Vd,0(s),VtVm	
vsuxseg3ei16.v	V	Vd,0(s),VtVm	
vsuxseg3ei32.v	V	Vd,0(s),VtVm	
vsuxseg3ei64.v	V	Vd,0(s),VtVm	
vsuxseg3ei8.v	V	Vd,0(s),VtVm	
vsuxseg4ei16.v	V	Vd,0(s),VtVm	
vsuxseg4ei32.v	V	Vd,0(s),VtVm	
vsuxseg4ei64.v	V	Vd,0(s),VtVm	
vsuxseg4ei8.v	V	Vd,0(s),VtVm	
vsuxseg5ei16.v	V	Vd,0(s),VtVm	
vsuxseg5ei32.v	V	Vd,0(s),VtVm	
vsuxseg5ei64.v	V	Vd,0(s),VtVm	
vsuxseg5ei8.v	V	Vd,0(s),VtVm	
vsuxseg6ei16.v	V	Vd,0(s),VtVm	
vsuxseg6ei32.v	V	Vd,0(s),VtVm	
vsuxseg6ei64.v	V	Vd,0(s),VtVm	
vsuxseg6ei8.v	V	Vd,0(s),VtVm	
vsuxseg7ei16.v	V	Vd,0(s),VtVm	
vsuxseg7ei32.v	V	Vd,0(s),VtVm	
vsuxseg7ei64.v	V	Vd,0(s),VtVm	
vsuxseg7ei8.v	V	Vd,0(s),VtVm	
vsuxseg8ei16.v	V	Vd,0(s),VtVm	
vsuxseg8ei32.v	V	Vd,0(s),VtVm	
vsuxseg8ei64.v	V	Vd,0(s),VtVm	
vsuxseg8ei8.v	V	Vd,0(s),VtVm	
vt.maskc	ventana condops	d,s,t	
vt.maskcn	ventana condops	d,s,t	
vwadd.vv	V	Vd,Vt,VsVm	
vwadd.vx	V	Vd,Vt,sVm	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
vwadd.wv	V	Vd,Vt,VsVm	
vwadd.wx	V	Vd,Vt,sVm	
vwaddu.vv	V	Vd,Vt,VsVm	
vwaddu.vx	V	Vd,Vt,sVm	
vwaddu.wv	V	Vd,Vt,VsVm	
vwaddu.wx	V	Vd,Vt,sVm	
vwcvt.x.x.v	V	Vd,VtVm	
vwcvtu.x.x.v	V	Vd,VtVm	
vwmacc.vv	V	Vd,Vs,VtVm	
vwmacc.vx	V	Vd,s,VtVm	
vwmaccsu.vv	V	Vd,Vs,VtVm	
vwmaccsu.vx	V	Vd,s,VtVm	
vwmaccu.vv	V	Vd,Vs,VtVm	
vwmaccu.vx	V	Vd,s,VtVm	
vwmaccus.vx	V	Vd,s,VtVm	
vwmul.vv	V	Vd,Vt,VsVm	
vwmul.vx	V	Vd,Vt,sVm	
vwmulsu.vv	V	Vd,Vt,VsVm	
vwmulsu.vx	V	Vd,Vt,sVm	
vwmulu.vv	V	Vd,Vt,VsVm	
vwmulu.vx	V	Vd,Vt,sVm	
vwredsum.vs	V	Vd,Vt,VsVm	
vwredsumu.vs	V	Vd,Vt,VsVm	
vwsub.vv	V	Vd,Vt,VsVm	
vwsub.vx	V	Vd,Vt,sVm	
vwsub.wv	V	Vd,Vt,VsVm	
vwsub.wx	V	Vd,Vt,sVm	
vwsubu.vv	V	Vd,Vt,VsVm	
vwsubu.vx	V	Vd,Vt,sVm	
vwsubu.wv	V	Vd,Vt,VsVm	
vwsubu.wx	V	Vd,Vt,sVm	
vxor.vi	V	Vd,Vt,ViVm	
vxor.vv	V	Vd,Vt,VsVm	
vxor.vx	V	Vd,Vt,sVm	
vzext.vf2	V	Vd,VtVm	
vzext.vf4	V	Vd,VtVm	
vzext.vf8	V	Vd,VtVm	
wfi	I	--	
wrs.nto	Zawrs	--	
wrs.sto	Zawrs	--	
xnor	Zbb|Zbkb	d,s,t	
xor	C	Cs,Ct,Cw	
xor	C	Cs,Cw,Ct	
xor	I	d,s,j	
xor	I	d,s,t	
xori	I	d,s,j	
xperm4	Zbkx	d,s,t	
xperm8	Zbkx	d,s,t	
zext.b	I	d,s	
zext.h	I	d,s	
zext.h	Zbb	d,s	
zext.h	Zbb	d,s	
zext.w	I	d,s	

Instruction name and syntax	Extens. required	Abstract parameters	Very short description
zext.w	Zba	d,s	

1.20 Further reading

1. The latest release of the Instruction Set Architecture (ISA): github/riscv
2. Linux Standard Base Core Specification, Generic Part. This is the official documentation for the ELF file format, for the debug frame machinery, etc. Download it from linuxfoundation.org.
3. The MaskRay blog. This is a very interesting blog full of references to low level stuff both for ARM, RISCV and other stuff, even windows. See: maskray.me
4. Improving DWARF. This is a very good and very readable critique of DWARF tables, presenting a DWARF table verifier, and, in general, a new perspective in debug tables and stack unwinding. (Slides) inria/france
5. This is the research article for the above slides. acm.org
6. A complete description of riscv relocations: sifive-blog
7. The specifications of the "Zbb" (bit manipulation) extension. github/riscv/bitmanip
8. The official specifications for the assembler: github/riscv/asm
9. Specifications for the Thead processor. www.t-head.cn
10. Want to write an hypervisor? Here you will find all about it: H extension

1.21 Answers to all exercises

Exercise 1: *The instruction* `bgt` *is an alias. How would you build it from the other instructions?*
Answer:
Just invert the arguments. `bgt rs1,rs2,label` → `blt rs2,rs1,label`

Exercise 2: *Write a small program that uses a conditional branch.*
Answer:

```
1    .globl main
2    main:
3    addi sp,sp,-16     // Build a stack frame
4    sd ra,8(sp)
5    sd s0,0(sp)
6    li t1,1            // t1 ← 1
7    li t2,2            // t2 ← 2
8    bgt t1,t2,.L1      // is t1 bigger than t2 ?
9    la a0,.LC1         // We did not branch. Load string address of LC1
10   j .L2              // Branch to call instruction
11   .L1:
12   la a0,.LC2         // We did branch. Load LC2 string
13   .L2:
14   call printf        // Do the call
15   ld ra,8(sp)        // Restore stack frame
16   ld s0,(sp)
17   jr ra              // Return
18   .LC1:
19   .string "Branch not taken\n"
20   .p2align 2
21   .LC2:
22   .string "Branch taken\n"
```

Executing:

```
1    star64:~/tiny-asm$ ./asm -o bgt.o bgt.s
2    star64:~/tiny-asm$ gcc bgt.o
3    star64:~/tiny-asm$ ./a.out
4    Branch not taken
```

Exercise 3: *Disassemble the program. What you see instead of bgt?*
Answer:

```
14: 0063c863            blt t2,t1,24 <.L1>
```

The assembler changed source and destination, using `blt`.

Exercise 4: *Change bgt into blt line 8. Does the output change?*
Answer:

```
star64:~/tiny-asm$ ./a.out
Branch taken
start64:~/tiny-asm$
```

Exercise 5: *How is the change achieved? Look at the source asm.c.*

Answer:
Looking at the opcode table we have:

```
{"bgt",0,INSN_CLASS_I,"t,s,p",MATCH_BLT,MASK_BLT,match_opcode,INSN_ALIAS|
    INSN_CONDBRANCH}
{"blt",0,INSN_CLASS_I,"s,t,p",MATCH_BLT,MASK_BLT,match_opcode,INSN_CONDBRANCH}
```

We can see that in the case of `bgt`, the instruction is marked as an *alias*. We see also that the match and the mask are identical of `bgt` and `blt`. The essential difference is in the argument string: bgt has `"t,s,p"`, and in the mask of `blt` we have `"s,t,p"`.

Exercise 6: *Use the XOR instruction to invert all bits in an integer register*
Answer:
Since $1 \oplus 1$ is zero, and $0 \oplus 1$ is 1, it suffices to have a right hand side of all ones (the number -1) and we are all set.

The instruction `not` (invert all bits) is `xor rs1,-1`. You can see this in the opcode table:

```
{"not",0,INSN_CLASS_I,"d,s",MATCH_XORI|MASK_IMM,MASK_XORI|MASK_IMM,match_opcode,
    INSN_ALIAS},
```

It has the `INSN_ALIAS` bit set, and the match is `MATCH_XORI`.

Exercise 7: *Write a program in assembler to print these 3 counters.*
Answer:

```
1   .globl main
2   main:                   Use of this name allows us to use C runtime
3   addi sp,sp,-16      Make room to establish a stack frame
4   sd ra,8(sp)         Save return address
5   sd s0,0(sp)         Save old stack frame
6   addi s0,sp,16       Establish a new frame. This is not actually needed.
7   rdtime a1           Read time into a1, that is the second argument
8   lla a0,.LC0         to printf. The first is the LC0 string
9   call printf         Let printf do the job
10  rdcycle a1          The same thing for the cycles. Use LC1.
11  lla a0,.LC1         String into a0
12  call printf
13  rdinstret a1        And the same for instructions returned. Use LC2
14  lla a0,.LC2         String into a0
15  call printf
16  ld ra,8(sp)         Restore return address
17  ld s0,0(sp)         Restore old frame pointer
18  jr  ra              Return to the startup code
19  .LC0:
20  .string "Time=0x%x\n"
21  .LC1:
22  .string "Cycles=0x%x\n"
23  .LC2:
24  .string "Instructions executed=%d\n"
```

Executing this yields:

```
Time=133663617138
Cycles=164249353990
Instructions executed=85266785454
```

Exercise 8: *Try to verify that time corresponds to a time measure*

Answer:
One way to do that is to call our program, then do something, then call it again. This should be a measure of how much time this "do something" takes. If we repeat that, we should arrive at similar results.

We will use "uptime" a utility that prints the time since startup.

```
star64:~/tiny-asm$ ./a.out;uptime;./a.out
Time=139156277825
Cycles=81886690065
Instructions executed=52342547651
15:22:18 up  9:39,  2 users,  load average: 0.00, 0.00, 0.00
Time=139156334252
Cycles=81891385713
Instructions executed=52345512910

star64:~/tiny-asm$ ./a.out;uptime;./a.out
Time=139235738528
Cycles=81923228282
Instructions executed=52376705932
15:22:38 up  9:40,  2 users,  load average: 0.00, 0.00, 0.00
Time=139235795755
Cycles=81927968778
Instructions executed=52379666534
star64:~/tiny-asm$
```

Exercise 9: *Use the "max" instruction to calculate the absolute value of a signed integer.*
Answer:
Use:

```
neg rd,rs1
max rd,rs1,rd
```

Exercise 10: *Write an assembler program to show the CSR flags*
Answer:

```
1   .globl main
2   main:
3   addi sp,sp,-16      Establish stack frame
4   sd ra,8(sp)         Save return address
5   sd s0,0(sp)         Save frame pointer
6   frcsr a1            Read the control register into the fisrt arg (a1)
7   lla  a0,.LC0        Read the string into the first argument (a0)
8   call printf         Call printf
9   ld ra,8(sp)         Restore return address
10  ld s0,0(sp)         Restore frame pointer
11  add sp,sp,16        Destroy stack frame
12  ret                 Bye bye
13  .LC0:
14  .string "CSR= 0x%lx\n"
```

This whole things makes just `printf("CSR=0x%x\n",csr);` But... wait, there is a bug.

```
$ asm -o rcsr.o rcsr.s
$ gcc rcsr.o
```

```
$ ./a.out
CSR= 0x0
$ ???
```

Well, of course. There wasn't any motives to set any of those flags above, and the rounding mode is zero (RNE). To see that we are really reading the csr let's provoke a division by zero, so at least we have something in there. We add following lines:

```
5     sd s0,0(sp)         Save frame pointer
6     li t1,12            Put 12 in register t1
7     fcvt.d.l f20,t1     Convert it to 12.0 in register f20
8     fcvt.d.l f21,x0     Put zero into register f21
9     fdiv.d f10,f20,f21  Divide 12.0/0.0
10    frcsr a1            Read the control register into the fisrt arg (a1)
11    etc                 Rest is the same
```

Now, it should show the Division by zero bit as ON.

```
$ asm -o rcsr.o rcsr.s
$ gcc rcsr.o
$ ./a.out
CSR= 0x8
$
```

That was it!

Exercise 11: *Write a subroutine that returns the flags of the CSR*
Answer:

```
1     # C Interface: int readcsr(void);
2     .globl readcsr
3     readcsr:
4     frcsr a0          Read the control and status register into a0
5     andi  a0,a0,15    Select the 4 lower bits and leave result in a0 (ABI)
6     jr ra             return
```

Here it is not necessary to build a stack frame since we do not make any calls, and we do not use any local variables. We build our result in the established register for returning results (a0).

Exercise 12: *Change the amoadd by amoswap. What are the values of t2, t3 and t4 after the operation?*
Answer:
`t2` stays unchanged at 47, `t3` contains 123, the value we stored at the bottom of the stack, and the bottom of the stack (`t4`) contains now 47.

Exercise 13: *Calculate ceil(log2(x))*
Answer:
The key here is to use the instruction `th.ff1`: it finds the first one bit starting at the most significant bit.

```
1     .globl bfd_log2
2  bfd_log2:
3     th.ff1 a0,a0
4     addi a0,a0,-64
5     sub a0,zero,a0
6     ret
```

The instructions in lines 4 and 5 calculate 64 - `ff1`. An alternative would be to put 64 in some register and then subtract the result of the ff1 operation from it, but that would destroy one register and would have the same number of instructions.

Another alternative would be:

```
1     .globl ff1
2  ff1:
3     th.ff1 a0,a0
4     xori a0,a0,63
5     ret
```

This is *almost* right, but calculates $\lfloor log_2(x) \rfloor$, so we have to add one to the result, what makes for the same number of instructions than the first one presented. This version is much less clear than the first one.

A more serious objection to our assembler versions is that our function returns 1 for an input of one, what is wrong. A serious version would test for that condition and branch accordingly.

Exercise 14: *Mismatch between source and disassembly. Explain*
Answer:

The explanation: actually, the instruction at address 4 is `addi a0,0`, what is actually a mov instruction. But the zero is not zero, since it is just a placeholder for a relocation. Looking at the relocations we see:

```
1     Disassembly of section .text:
2
3     0000000000000000 <main>:
4     0:  00000517          auipc  a0,0x0
5     0: R_RISCV_PCREL_HI20 .L2
6     0: R_RISCV_RELAX  *ABS*
7     4:  00050513          mv  a0,a0
8     4: R_RISCV_PCREL_LO12_I .L0
9     4: R_RISCV_RELAX  *ABS*
10    8:  00008067          ret
```

We see that a relocation points to address 4, indicating to put the lower 12 bits of the main address into an immediate and add them to a0.

Index