tiny-asm: an assembler for riscv

jacob navia

Contents

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 assembler. 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.

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:

   `https://github.com/bminor/binutils-gdb.git`.

   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
   `https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf` but there are apparently more recent ones. There is a depository in github at
   `https://github.com/riscv/riscv-isa-manual` 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

   `https://dwarfstd.org/doc/DWARF5.pdf`. 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 generated 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
   `https://pine64.com/product-category/star64/`. This is the machine I am using, for around 110 US$. You can buy similar ones directly from the chinese, for instance
   `https://www.waveshare.com/visionfive2.htm`, 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`.

1.2 Building tiny-asm

The build process runs as follows:

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

```
$ gcc -o asm 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 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 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.

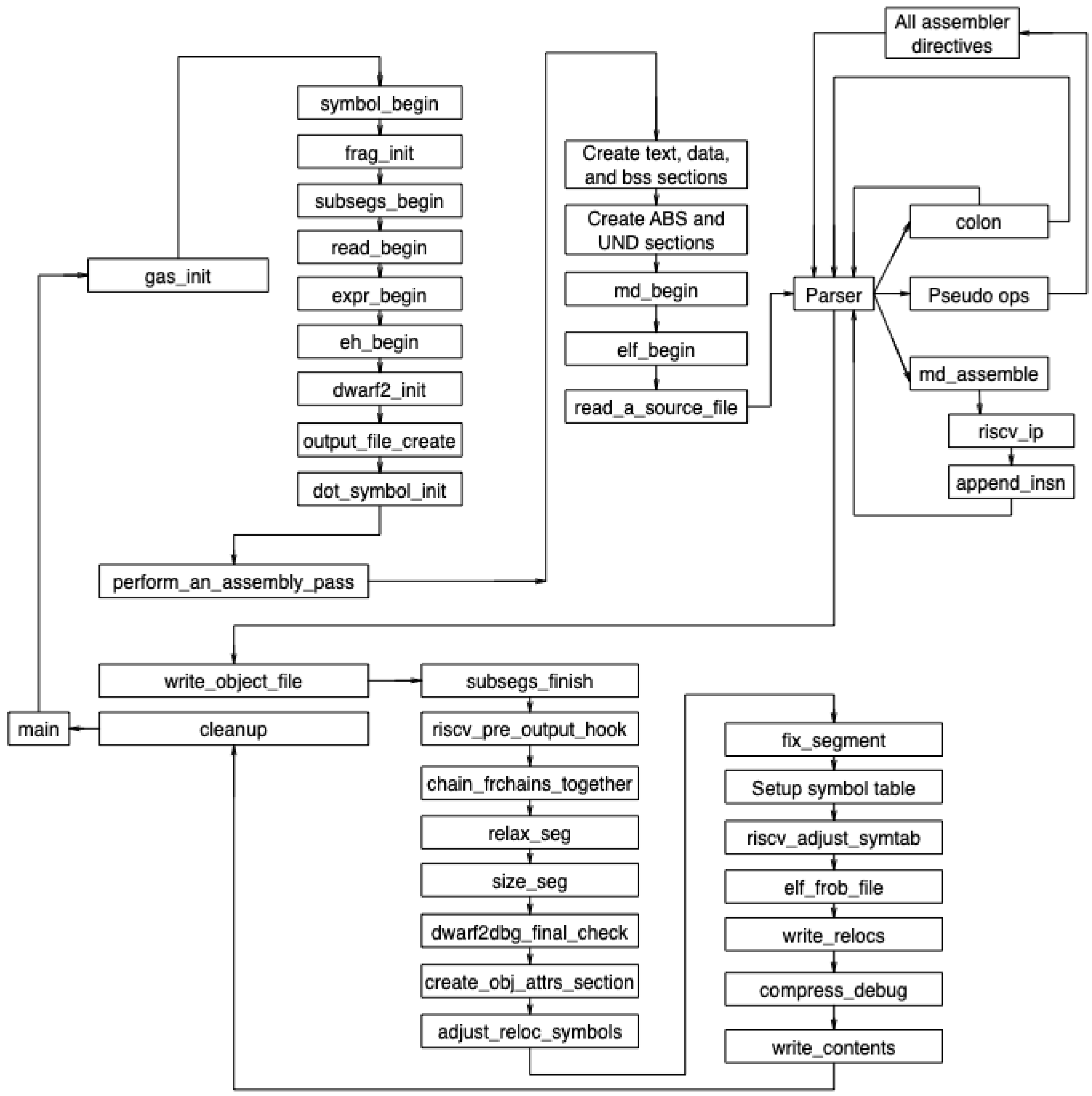


Figure 1.1: Overview of the assembler control flow

In figure 1.1 (page 8) 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 cram 40 000 lines into a diagram... too much.

We start with `main` that organizes all three parts [2]. 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

[2] 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.

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 run-time 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.

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`.

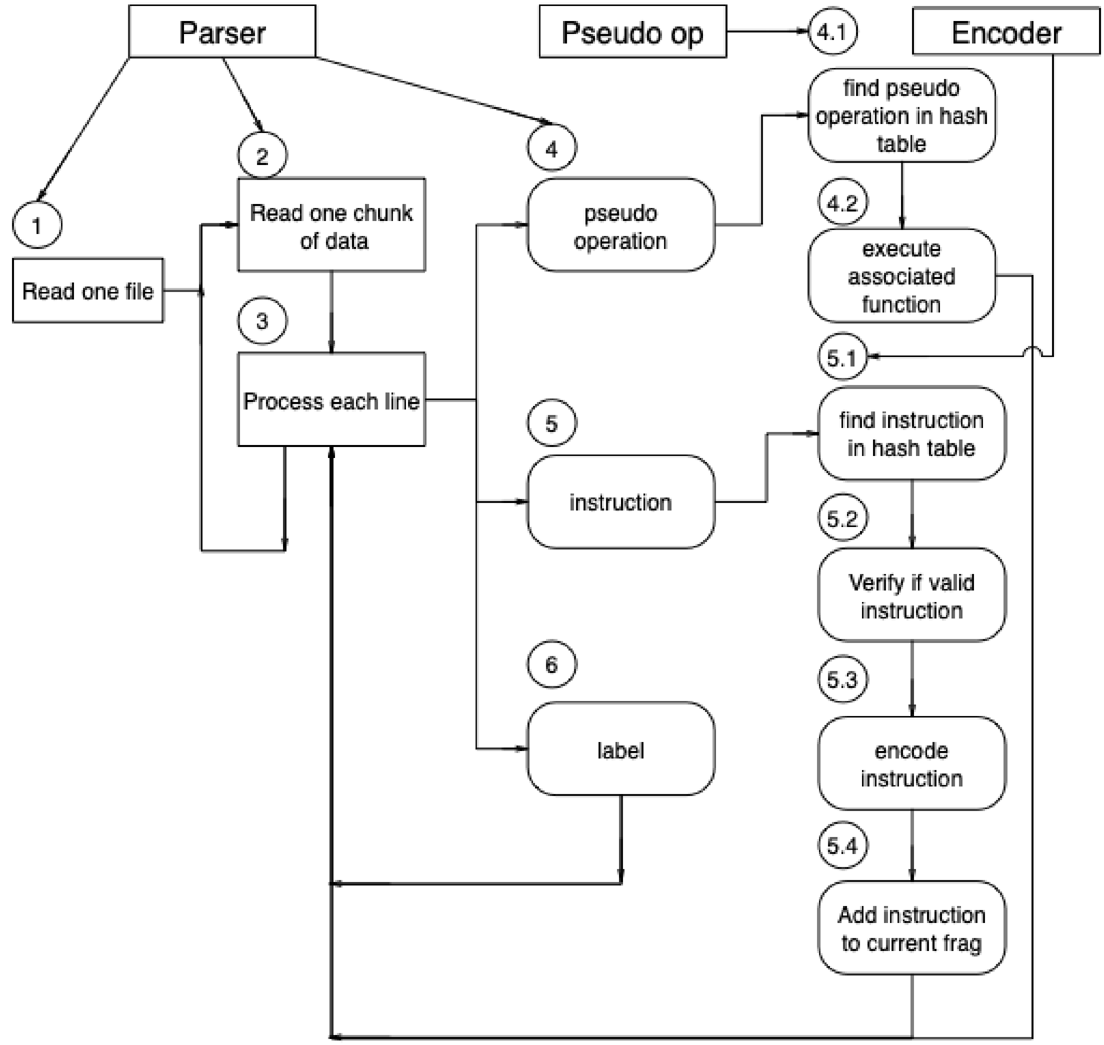

Parser
Pseudo op
4.1
Encoder
find pseudo operation in hash table
2
4
4.2
1
Read one chunk of data
pseudo operation
execute associated function
Read one file
3
5.1
Process each line
5
find instruction in hash table
instruction
5.2
Verify if valid instruction
6
5.3
label
encode instruction
5.4
Add instruction to current frag

Figure 1.2: A more detailed view of the parser

We assume that the assembler input is a single file containing instructions, data, and 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[3].

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.

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

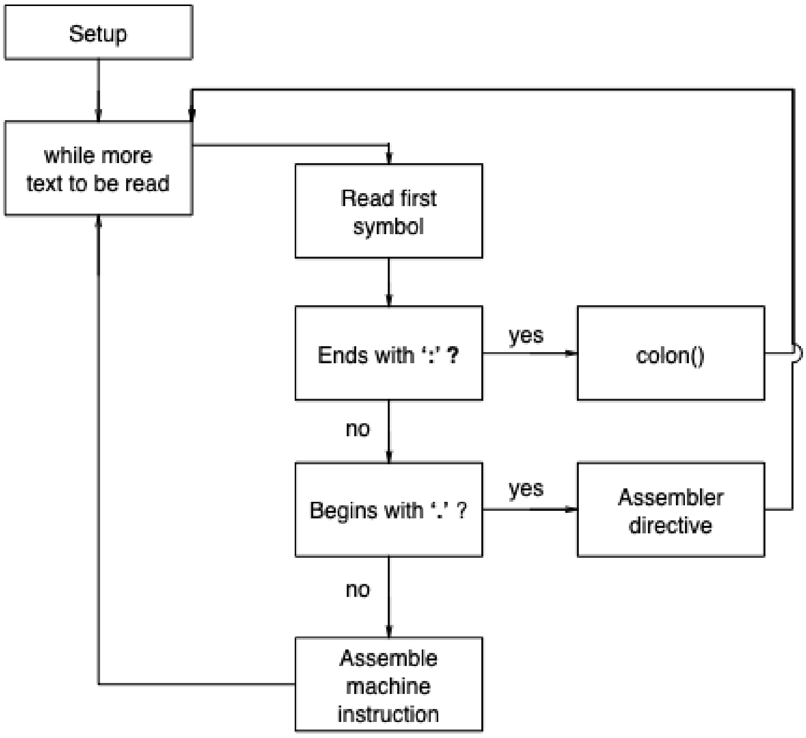


Figure 1.3: `read_a_source_file function`

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 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.

1.4 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.

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

Table 1.1: RISCV symbolic register names

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.

1.5 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

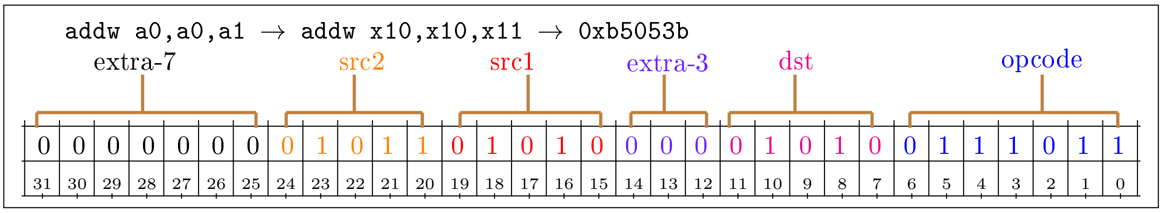
> The RISC-V ISA keeps the source (rs1 and rs2) and destination (rd) registers at the same position in all formats to simplify decoding. Except for the 5-bit immediates used in CSR instructions, immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction 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. [4]

1.5.1 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`

[4] 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



0	0	0	0	0	0	0	0	1	0	1	1	0	1	0	1	0	0	0	0	0	1	0	1	0	0	1	1	1	0	1	1
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

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.
- 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.

1.5.2 The "I" format

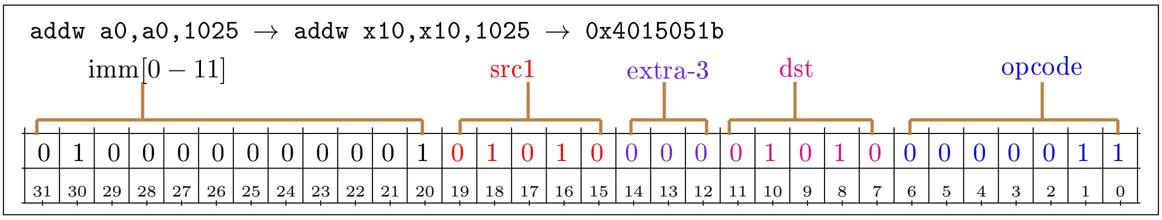
This format changes the `"R"` format by merging `src2` with `extra-7` to give a 12 bit field where an immediate integer value caan be stored (up to $2^{12} - 1$ → 4095 values can be stored).

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.

Figure 1.5: **I** Instruction layout



`addw a0,a0,1025 → addw x10,x10,1025 → 0x4015051b`

0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

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` → 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 [5]

1.5.3 The "U" format

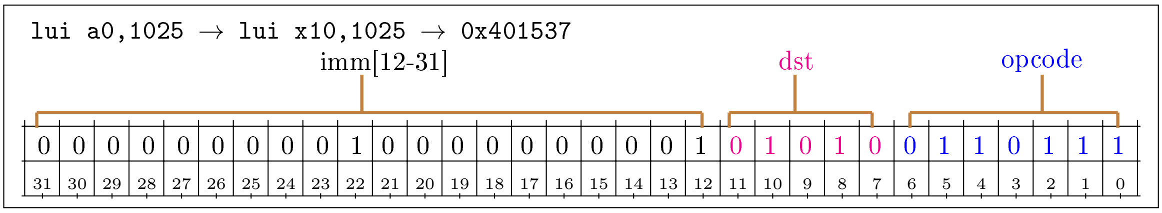
A variant of the **I** format featuring more space for immediate constants is the **U** format, that can hold immediate constants with 20 bits.

The `lui`[6]. 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:

[5] 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.

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

Figure 1.6: **U** Instruction layout



`lui a0,1025 → lui x10,1025 → 0x401537`

> 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.[7]

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 16.

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:

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

1.5.4 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.

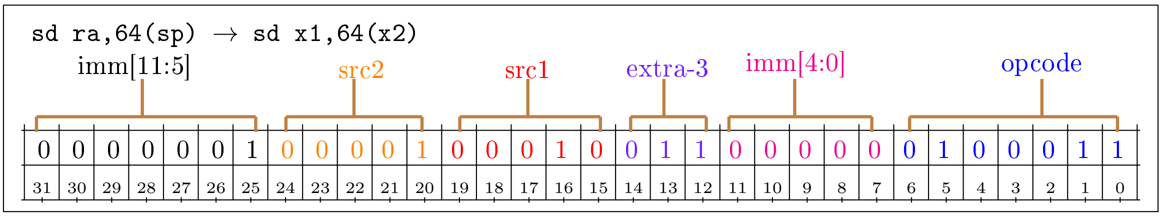
We use the instruction `sd ra,64(sp)` as example. This instruction means: Store the contents of the return address register `(ra)` at the memory address obtained by adding 64 to the contents of the sp register. We have here an address that is obtained by adding the contents of a register and a *displacement* 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.[8]

[7] Riscv ISA Architecture §2.2

[8] 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 or 64.

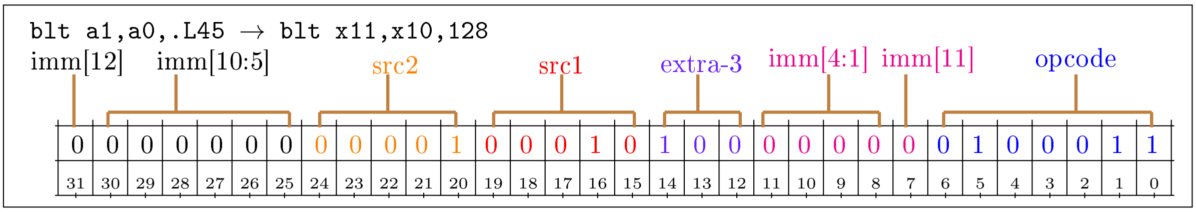
For extracting the immediate from the instruction we use the macro

```
#define EXTRACT_STYPE_IMM(x) \
(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 1.5.2 page 15.

1.5.5 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

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

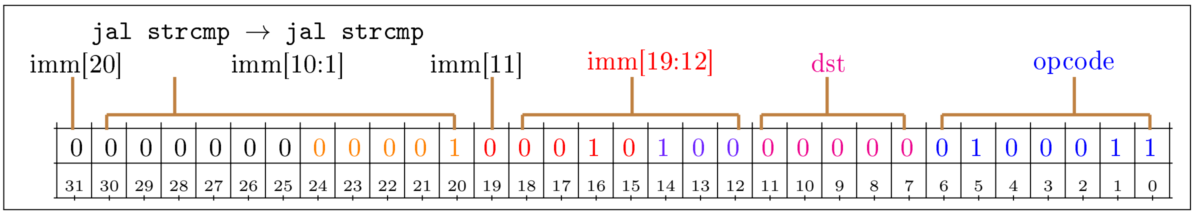
The macro to access the immediate value is way more complicated due to the bit scrambling...

```
#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))
```

1.5.6 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



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	1

Why this scrambled layout? The Risc-v manual tells us:

> 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))
```

1.6 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[9].

Table 1.3: 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 '.

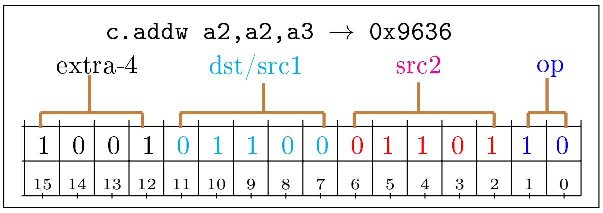
<table>
<tr><th>Meaning</th><th>Code</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>Register</td><td>CR</td><td colspan="4">Extra-4</td><td colspan="5">dst/src1</td><td colspan="5">src2</td><td colspan="2">op</td></tr>
<tr><td>Immediate</td><td>CI</td><td colspan="3">Extra-3</td><td>S</td><td colspan="5">rd/rs1</td><td colspan="5">immediate</td><td colspan="2">op</td></tr>
<tr><td>Store local</td><td>CSS</td><td colspan="3">Extra-3</td><td colspan="6">imm</td><td colspan="5">rs2</td><td colspan="2">op</td></tr>
<tr><td>Wide imm</td><td>CIW</td><td colspan="3">Extra-3</td><td colspan="8">imm</td><td colspan="3">rd'</td><td colspan="2">op</td></tr>
<tr><td>Load</td><td>CL</td><td colspan="3">Extra-3</td><td colspan="3">imm</td><td colspan="3">rs1'</td><td colspan="2">imm</td><td colspan="3">rd'</td><td colspan="2">op</td></tr>
<tr><td>Store</td><td>CS</td><td colspan="3">Extra-3</td><td colspan="3">imm</td><td colspan="3">rs1'</td><td colspan="2">imm</td><td colspan="3">rs2'</td><td colspan="2">op</td></tr>
<tr><td>Arithmetic</td><td>CA</td><td colspan="6">Extra-6</td><td colspan="3">rd'/rs1'</td><td colspan="2">Extra-2</td><td colspan="3">rs2'</td><td colspan="2">op</td></tr>
<tr><td>Branch</td><td>CB</td><td colspan="3">Extra-3</td><td colspan="3">offset</td><td colspan="3">rs1'</td><td colspan="5">offset</td><td colspan="2">op</td></tr>
</table>

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

Jump	CJ	Extra-3	jump target	op

1.6.1 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:

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

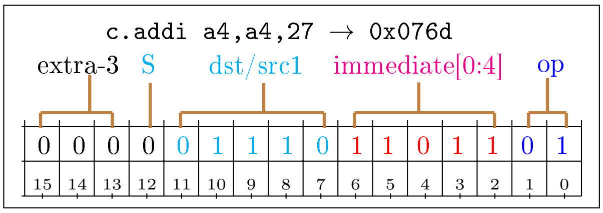
In the first one we set to zero all bits in the field that will be written. The second one introduces the bits into the right position. The *or* operation joins those parts into a single value.

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.

1.6.2 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 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:

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



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

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

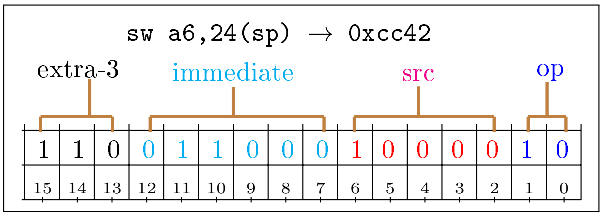
The first macro uses the same technique for sign extending that our `RV_IMM_SIGN` uses (see 1.5.2 page 15). We just need another expression since the other was fixed for 32 bits.

1.6.3 The stack relative store (CSS) format

Five instructions use the CSS format:

1. `c.swsp` or store word to an offset from sp, the stack pointer.
2. `c.sdsp` or store double word (64 bits) to an offset from sp.
3. `c.fswsp` or store single precision (32 bits) to an offset from sp.
4. `c.fsdsp` or store double precision (64 bits) to an sp offset.

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:

```
#define EXTRACT_CSSTYPE_IMM(x) (RV_X(x, 7, 6) << 0)
#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. [10]

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.

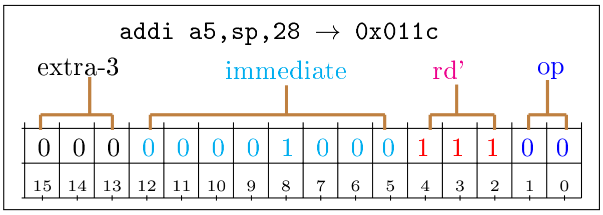
And... "one more thing" as Steve Jobs liked to say, there is a problem with zero offsets from the stack pointer. Normally a zero offset is omitted, i.e. you do NOT write `sw a6,0(sp)`, you just write `sw a6,(sp)`. The handling of the `CM` directive tests for this with the function `riscv_handle_implicit_zero_offset`.

[10] 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.

1.6.4 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. [11]. 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.[12]

   The "simpliying" above refers to hardware simplification.

4. The Extra-3 field is zero.

The macros used to access the immediate are:

```
#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"`

1.6.5 The compressed load (CL) format

1. The OP field is zero.

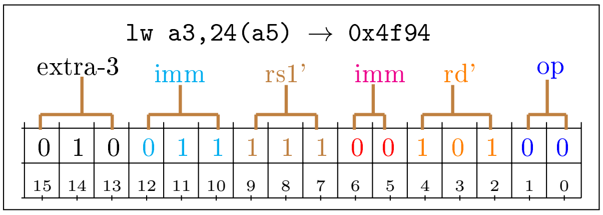
2. The destination register is 5 (a3).[13]

[11] 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

[12] Risc-V Unprivileged ISA V20191213 §16.2

[13] These values are in table 1.3

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



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.

1.7 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.

```
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:

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[14].

Table 1.6: Argument descriptions

Char	Description
`\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.
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 1.8 page 27.
V	Vector instructions. This leads to a nested switch statement too.
,	Synchronization. Arguments are separated by commas. The software tests this and ignores the separators.

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

Table 1.6: Argument descriptions

`()` `[]`	Displacement or index. Same behavior as for commas.
$<$	Shift amount for shifts less than 32.
$>$	Shift amount for 0 to word length - 1.
Z	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.
E	Control register number. This is used only in privileged instructions.
m	Rounding mode. This argument expects a character string representing the rounding mode. It can be one of `"rne"`,`"rtz"`, `"rdn"`,`"rup"`, `"rmm"`, `0`,`0`,`"dyn"`. See table 1.7 page 27.
PQ	Fence predecessor or successor
d	Destination register.
s	First source register. Also called src1 in the documentation.
t	Second source register. The `'t'` is for `target`. It is also called src2 in the documentation.
r	RS3
D	Floating point destination register
S	Floating point source 1.
T	Floating point source 2
U	Floating point source 1 and 2
R	Floating point RS3.
F	Expects a bit field, that is defined by the following character
I	`M_LI` macro. Immediate value.
A	Requests a symbol
B	Requests a symbol or a constant.
j	Sign extended immediate.
q	Expects a register store displacement.
o	Expects a load displacement.
1	Used for thread local storage.
p	PC relative offset
0	Expects a zero displacement. For instance: `lr.w a5,0(sp)`.
u	Expects a 20 bit immediate
a	20 bit relative offset.
c	Call using the global object table
O	Opcode field
y	bs immediate for branch offsets.
Y	rnum immediate
z	Expects a zero
W	Various operands
X	Integer immediate

Below is the set of rounding modes for the `m` parameter. It has been taken from the Sifive site[15]. Edited in May 27th 2020.

Table 1.7: Accepted rounding modes for the 'm' parameter

Binary Value	Mnemonic	Meaning

[15] https://observablehq.com/@nschwass/riscv-f-extension-single-precision-floating-point-instruction The URL seems truncated but it is not...

Table 1.7: Accepted rounding modes for the 'm' parameter

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.

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

Table 1.8: Compressed instruction types

Char	Description	Char	Description
s	Source register 1 (x8-x15)	w	Source 1 and destination when they are the same
t	Source 2 with x8-x15	x	Source 2 and destination are the same. x8-x15 only.
U	Source 1 and destination the same.	v	Source 2
c	Source 1 constrained to be sp	z	Source 2 should be the zero register
>	Shift amount between 0 and word length - 1	5	Five bit field
6	Six bit numeric field	8	Eight bit field
j	Non-zero immediate	k	Immediate (possibly zero)
l	Load immediate (64 bits)	m	Load immediate
n	Immediate offset from SP	o	`C.addiw`, `c.li`, and `c.andi` allow zero immediate. C.addi allows zero immediate as hint. Otherwise this is same as 'j'.
K	scaled by 4 stack addend	L	Stack offset scaled by 16
M	Scaled by 4 stack displacement(32 bits store)	N	Data reference with offset from stack scaled by 4(64 bits store)
u	Immediate for jumps	v	Immediate for jumps
S	Floating point source 1 x8-x15	D	Floating point source 2 x8-x15
T	Floating point source 2	F	Field of 6, 4, 3, or 2 bits

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.

1.8 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.)[16] 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.

1.8.1 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.

[16] 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`.

- `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.

```
    rsi.pass = 0;
    while (1) {
        rsi.changed = 0;
        map_over_sections(relax_seg,&rsi);
        rsi.pass++;
        if (!rsi.changed)
        break;
    }
```

  `rsi` is a variable of type `struct relax_seg_info`[17]. 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.

- `dwarf2dbg_final_check`. This is interesting stuff. There is a proposal from Alexandre Oliva[18] 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.[19]

- `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

```
    if (symbol_rootP) {
        symbolS      *symp;

```

[17] 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.

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

[19] 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, 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.

```
        for (symp = symbol_rootP; symp; symp = symbol_next(symp))
            resolve_symbol_value(symp);
    }
    resolve_local_symbol_values();
    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.
    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.

1.9 Assembler directives

1.9.1 .align, .p2align, p2alignw, p2alignl

Entries in the table:

```
{"align",s_align_ptwo,0},
{"p2align",s_align_ptwo,0},
{"p2alignw",s_align_ptwo,-2},
{"p2alignl",s_align_ptwo,-4},
```

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

```
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:

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

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

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

The actual call sequence looks like this:

```
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.

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

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

```
{"ascii",stringer,8 + 0},
{"asciz",stringer,8 + 1},
{"string8",stringer,8 + 1},
{"string16",stringer,16 + 1},
{"string32",stringer,32 + 1},
{"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.

1.9.3 .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`. [20]

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:

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

```
1    do {
2        TC_PARSE_CONS_RETURN_TYPE ret = TC_PARSE_CONS_RETURN_NONE;
3        ret = TC_PARSE_CONS_EXPRESSION(&exp,(unsigned int)nbytes);
4
5        if (rva) {
6            if (exp.X_op == O_symbol)
7                exp.X_op = O_symbol_rva;
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.

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

All those directives have only *one* thing in common: they are completely **ignored** by the 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.

1.9.5 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},
```

1.9.6 reloc

The documentation of GAS says about this directive:

`.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.[21]

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

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

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

```
    .file  "tld.c"
    .size  mm, 16
mm:
    .word  0
    .word  -1610612736
    .word  -1253836416
    .word  1073779231
    .text
    .globl main
    .type  main, @function
main:
    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>
```

[21] 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

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>
```

Nothing changed. Weird.

Next thing: 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.

😜

1.9.7 globl

```
.global symbol, .globl 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.

Index