Lab 4
Syntax-Directed Code Generation

Objective

During the previous lab, you have written your own interpreter of the MiniC language. In this lab the objective is to generate *valid* RISCV codes from MiniC programs:

- Generate 3-address code for the MiniC language.
- Generate executable "dummy" RISCV from programs in MiniC via two simple allocation algorithms.
- **Please follow instructions and COMMENT YOUR CODE!**

Student files are in the Git repository.

Make sure your Git repository is up-to-date, using `git pull`.

4.1 Preliminaries

This section must be read **carefully**.

Important remark From now on, we add the following restriction to the MiniC language: Values (variables, argument of `println_int`) are of type (signed) `int` or `bool` only (no float, no string, no char). Thus all values can be stored in regular registers or in one cell (64 bits) in memory. You can let your program crash (`raise MiniCUnsupportedError(...)`) if another type of variable is provided.

Note that real compilers would perform the code generation from a decorated AST (with type annotations attached to nodes). For simplicity, we will work on the non-decorated AST: our language is simple enough to generate code without decorations.

Structure of the compiler's code In the `MiniC/Lib` folder, we provide you with many utility functions. A detailed documentation of the library is given in the repository, and can be accessed at the root of the git repository by opening `docs/index.html` in a web browser such as Firefox.

As for other files in the MiniC directory:

- `TP04/MiniCCodeGen3AVisitor.py` is the code generation algorithm, implemented as a visitor.

- The file `TP03/MiniCTypingVisitor.py` is reused from lab3. If your typechecker is buggy, you can use the compiler's `--disable-typecheck` to run the code generation without typechecking, and give the value True to `DISABLE_TYPECHECK` in `test_codegen.py`.

- The main Python file, `MiniCC.py` as in lab3, now accepts new options related to code generation (check `python3 MiniCC.py --help` for a full list). Running

  `python3 MiniCC.py --mode codegen-linear <file>`

  launches the chain: production of 3-address code with temporaries, allocation, replacement, print.

- The script `test_codegen.py` will help you test your code. We will use it in Section 4.4 through Makefile targets.

- The `README-codegen.md` file is to be completed progressively during the lab.

Re-test the command-line version of the RISCV simulator, for example with code from TP01:

```
cd ../TP01/riscv/
riscv64-unknown-elf-gcc libprint.s test_print.s -o test_print.riscv
spike -m100 pk test_print.riscv
cd ../../MiniC/
```

4.1.1 Conventions used in the assembly code

- All data items are stored on 64 bits (double-words, 8 bytes).

- Registers s1, s2, and s3 are reserved for temporary computations (e.g. to compute an address before or after an sd or a ld, or to store a value between a memory access and an arithmetic operation). Note that s0 is an alias for fp, hence s0 must not be used as a general purpose register either.

- Registers s4, ..., s11, t0, ..., t6 are general purpose registers, that can be used freely by the code generator. In your Python code, you can access the list of general-purpose registers with Operands.GP_REGS. s*i* and t*i* registers will behave differently in presence of function calls, but are considered equivalent for now.

- To store properly in memory, it is mandatory to compute offsets from the "reserved" register fp. To be compatible with the RISCV ecosystem, we will use a stack **growing with decreasing addresses**. Thus data in the stack is accessed by adding a **negative offset** (multiple of 8) to fp. In other words, we use the memory locations –8(fp), –16(fp), ... The sp register points to the first data contained in the stack. It is always 16-byte (2 double-words) aligned.

- Registers a1 to a7 are not used at all for the moment.

4.1.2 Conventions used in the test suite

A few reminders and new features of the test suite:

- Test files should contain directives giving the expected behavior:

  - // EXPECTED and the following lines to give the expected output;
  - // EXITCODE *n* gives the expected return code of the compiler, i.e. // EXITCODE 1 when the code should be rejected by your typechecker (see previous lab for the specification of different exit codes);
  - // SKIP TEST EXPECTED to specify that this test should not be run through test_expect (see below);

- Several tests are run on each .c files when launching make test-*something* (make test-naive, make test-lab4, etc.):

  - test_expect, that compiles the file using riscv64-unknown-elf-gcc. It checks that EXPECTED directives are correct, but doesn't test your compiler.
  - test_naive_alloc, test_alloc_mem, test_smart_alloc that compiles the file using your compiler, using the corresponding register allocation algorithm. The test suite leaves generated .s files next to the .c source file.

- make test FILTER=... can be used to restrict the set of files on which to run the test. Specify either a single file or an extended wildcard like TP04/tests/**/*while*.c.

4.2 First step: get familiar with the code and test suite

In this section you have to implement the course rules in order to produce RISCV code with temporaries. These rules are given in Figure 4.2 on page 9 and Figure 4.3 on page 10.

Here is an example of the expected output of this part. From the following MiniC program:

```
#include "printlib.h"

int main() {
    int a,n;
    n = 1;
    a = 7;
```

```
    while (n < a) {
      n = n+1;
    }
    println_int(n);
    return 0;
}
```

the following code is supposed to be generated.

```
1  ##Automatically generated RISCV code, MIF08 & CAP
2  ##non executable 3-Address instructions version
3
4
5  ##prelude
6  # [...] Some automatically generated code that will be explained in a future lab
7
8  ##Generated Code
9  # [...] Some automatically generated code that will be explained in a future lab
10        li temp_0, 0
11        li temp_1, 0
12        # (stat (assignment n = (expr (atom 1))) ;)
13        li temp_2, 1
14        mv temp_0, temp_2
15        # (stat (assignment a = (expr (atom 7))) ;)
16        li temp_3, 7
17        mv temp_1, temp_3
18        # (stat (while_stat while ( (expr (expr (atom n)) < (expr (atom a))) ) (
      stat_block { (block (stat (assignment n = (expr (expr (atom n)) + (expr (atom 1))))
      ) ;)) })))
19 lbl_begin_while_1_main:
20        li temp_4, 0
21        bge temp_0, temp_1, lbl_end_relational_3_main
22        li temp_4, 1
23 lbl_end_relational_3_main:
24        beq temp_4, zero, lbl_end_while_2_main
25        # (stat (assignment n = (expr (expr (atom n)) + (expr (atom 1)))) ;)
26        li temp_5, 1
27        add temp_6, temp_0, temp_5
28        mv temp_0, temp_6
29        j lbl_begin_while_1_main
30 lbl_end_while_2_main:
31        # (stat (print_stat println_int ( (expr (atom n)) ) ;))
32        mv a0, temp_0
33        call println_int
34 # [...] Some automatically generated code that will be explained in a future lab
35
36 ##postlude
37 # [...] Some automatically generated code that will be explained in a future lab
```

4.2.1 3-address code generation on supported C file

In the skeleton, we provide you an incomplete `MiniCCodeGen3AVisitor.py`. To run it, type

```
make  # to generate the lexer and parser
python3 MiniCC.py --mode codegen-linear TP04/tests/provided/step1/test00.c \
               --reg-alloc=none
```

```
cat TP04/tests/provided/step1/test00.s
```

Observe the generated code, it is complete (the skeleton supports all features used in this input file). Since we used `--reg-alloc=none`, register allocation wasn't performed, so we still get temporaries in the output, which is not executable. We generated RISCV comments with MiniC statements to help debugging.

To get executable code, we provide you the naive (everything in registers) allocation:

```
python3 MiniCC.py --mode codegen-linear TP04/tests/provided/step1/test00.c \
                --reg-alloc=naive --output TP04/tests/provided/step1/test00-naive.s
cat TP04/tests/provided/step1/test00-naive.s
riscv64-unknown-elf-gcc TP04/tests/provided/step1/test00-naive.s libprint.s \
                -o TP04/tests/provided/step1/test00-naive.riscv
spike pk TP04/tests/provided/step1/test00-naive.riscv
```

This should ouptut 42 (plus the usual `bbl loader` line).

You can automate this, plus the run of `test_expect` (see above) with:

```
make FILTER=TP04/tests/provided/step1/test00.c test-naive
```

4.2.2 3-address code generation on C file not supported by the skeleton

Now, run the same command on an input file not yet supported by the skeleton:

```
python3 MiniCC.py --mode codegen-linear TP04/tests/provided/step1/test00b.c \
                --reg-alloc=none
```

You should get a `NotImplementedError` exception, with a backtrace pointing to the location in the generator where the feature (here, additive expressions) need to be implemented. To get an idea of what you'll need to implement, you may run:

```
git grep NotImplementedError
```

Here also, you can automate the test with:

```
make FILTER=TP04/tests/provided/step1/test00b.c test-naive
```

Off course, you can also run the whole test suite without `FILTER`:

```
make test-naive
```

A coverage report of the test suite is generated in `htmlcov`, it can help you identify untested parts of your code.

It is strongly advised to adopt a "test driven" methodology, i.e. run the test suite, see what fails, fix it, and iterate. When implementing a feature for which the skeleton has no test, write a test first.

The rest of the lab boils down to "make sure all tests pass and the test suite covers 100% of the generator and allocators".

4.3 3 address code generation

EXERCISE #1 ► **Basic cases for 3 address code generation**
Implement the 3-address code generation corresponding to test cases in TP04/`tests/provided/step1/`, i.e. make the following command succeeds:

```
make FILTER="TP04/tests/provided/step1/*" test-naive
```

EXERCISE #2 ► **A few corner-cases**
Some points may require extra care, in the implementation or in the tests:

- Don't forget the automatic initialization (in MiniC, unlike real C). Unlike the interpreter, initialization cannot be done by initializing a Python dictionary. Make sure the initialization code is properly generated.

- Don't forget the explicit errors for division by zero. We provide you a piece of assembly code raising the error (see `print_code()` given in the library of the skeleton), you need to generate the instruction to jump to this label (we get it with `self._current_function.fdata.get_label_div_by_zero()`) when the right operand of a division or modulo is 0.

- `float` and `string` are unsupported. The compiler raises `MiniCUnsupportedError` when encountering any of them. Tests are provided for this.

Note that testing the division by 0 requires a bit of attention. We need to check that the executable exits with code 1 at runtime, that the output is correct, but we can't check that GCC gives the same behavior because GCC doesn't give a clean error message. A test case may therefore be:

```
#include "printlib.h"

int main(){
        println_int(1 / 0);
        return 0;
}
// SKIP TEST EXPECTED
// EXECCODE 1
// EXPECTED
// Division by 0
```

Test and implement all these features.

EXERCISE #3 ► **End of 3-address code generation for MiniC**
Implement the 3-address code generation rules:
- for boolean expressions and numerical comparison: compute 1 (true) or 0 (false) in the destination register; be careful the `not` boolean instruction is not as easy as you could wish;
- while loops;
- if then else.

At this point all the tests should be ok for all files in directory `TP04/tests/provided/step2/`. However these tests are not sufficient, you should add some other ones (in the directory `TP04/tests/students/`). Run the test suite with `make test-naive MODE=codegen-linear` to use all the test files.

About `if` and `while`   For tests (and boolean expressions), make sure you generate "conditional jumps" with:

```
self._current_function.add_instruction(
        RiscV.conditional_jump(label, op1, cond, op2))
```

where `op1` (resp `op2`) is the left operand (resp right operand or the numerical constant 0, nothing else), i.e. a register or a value of the boolean condition `cond` (`Condition('beq')` for equality, for instance) [1], and `label` is a label to jump to if the condition evaluates to true.

Multiplicative expressions

EXERCISE #4 ► **3-address code generation for multiplicative expressions**
If not already done, extend your work to multiplicative expressions. Conventions for division and multiplication should be the same as in C: division is truncated toward zero, and modulo is such that $(a/b) * b + a\%b = a$.

$$
\begin{array}{rclrcl}
4/3 & = & 1 & 4\%3 & = & 1 \\
(-4)/3 & = & -1 & (-4)\%3 & = & -1 \\
4/(-3) & = & -1 & 4\%(-3) & = & 1 \\
(-4)/(-3) & = & 1 & (-4)\%(-3) & = & -1
\end{array}
$$

[1]We suggest to use `git grep` and find this class definition and this method somewhere in the library we provide.

4.4   More on the naive allocator

We provide you with an allocation method which allocates temporaries in registers as long as possible, and fails if there is no more available registers. The process takes as input the former 3-address code and transforms each instruction according to the allocation function. When there are not enough registers available, the allocator raises an `AllocationError` exception. The test suite is programmed to skip tests raising this exception (i.e. the test is neither considered as a success nor as a failure).

Open, read, understand the `NaiveAllocator` implementation in `Lib/Allocator.py` and how it is used to perform the actual RISCV code generation.

4.5   RISCV **code with "all-in-mem" allocation of temporaries**

Tests   Up to now, you used `make test-naive MODE=codegen-linear` to test your code, and at this point all tests should pass, or be skipped (do not forget to make a test where the naive allocation uses too many registers!). From now on, you should use the more complete `make test-mem MODE=codegen-linear` command, that tests everything with the provided naive allocator, and the all-in-memory allocator you have to write. If you use `MiniCC.py` directly, the corresponding option is `--reg-alloc=all-in-mem`.

Check that `make test-mem MODE=codegen-linear` does fail. You can also run `make test-lab4` to run the tests for all allocators in this lab.

Implementation   As the number of registers for allocation is bounded by the number of available general purpose registers, i.e. `len(Operands.GP_REGS)`, the naive allocator cannot deal with more temporaries than general-purpose registers: we have to find a way to store the results elsewhere. In this particular lab, we will use the following solution:
  • The generated code will use memory locations in the stack.
  • All values that are propagated from one rule to another (sub-expressions, . . .) must be stored in the stack, whose address will be stored in FP.
  • $s1, s2, s3$ will be used to compute the value to store or as a destination register for the value(s) to read. Technically, only 2 of these registers are mandatory, but you should be cautious if you try a 2-registers-only solution.
  • In order to know if a given (temporary) operand should be read and/or written, use the `is_read_only` method of the `Instruction` class.
Figure 4.1 depicts the stack implementation for the RISCV machine, that follows the RISC-V calling convention (stack growing downwards, stack-pointer always 16-bytes aligned).
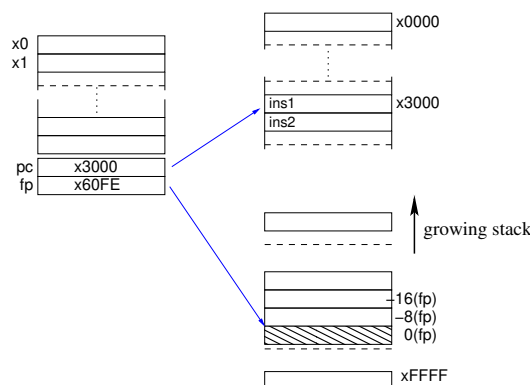

Figure 4.1: Memory model for RISCV

Following the convention that `fp` always stores the "begining of stack address", pushing the content of register $s3$ in the stack will be done following the steps:
  • compute a new offset (call to the `fresh_offset` method).
  • generate the following instruction:

```
sd s3, -offset*8(fp)
# sd = store double = 64-bits store
# -offset*8(fp) = memory location at address fp-offset*8
```

Getting back the value is similar.

To understand the principle, complete manually the expected output for the following two statements. The temporary `temp_3` is located at $-32(fp)$ and `temp_4` is located at $-40(fp)$:

```
int x, y;
x=4;
y=12+x
```

Listing 4.1: 'all in mem alloc for test_while2b.c'

```
1  ##Generated code without prelude and postlude
2         # (stat (assignment x = (expr (atom 4))) ;)
3         # li temp_2, 4
4         li s2, 4
5         sd s2, -24(fp)
6         # end li temp_2, 4
7         # mv temp_1, temp_2
8         ld s1, -24(fp)
9         mv s2, s1
10        sd s2, -16(fp)
11        # end mv temp_1, temp_2
12        # (stat (assignment y = (expr (expr (atom 12)) + (expr (atom x)))) ;)
13        # li temp_3, 12
14        # TODO 2 lines
15
16
17        # end li temp_3, 12
18        # add temp_4, temp_3, temp_1
19        # TODO 4 lines
20
21
22
23
24        # end add temp_4, temp_3, temp_1
25        # mv temp_0, temp_4
26        # NOT TODO
```

<u>EXERCISE #5</u> ▶ **Implement**
Now you are on your own to implement this code generation. The relevant file is `TP04/AllInMemAllocator.py`. Here are the main steps (less than 50 locs of PYTHON):

1. We have implemented for you an `AllInMemAllocator.prepare()` method. It only maps each temporary to a new offset in memory (in a PYTHON `dict`), allowing the use of the method `get_alloced_loc()` on a temporary used in the code.

2. Complete the method `AllInMemAllocator.replace(old_instr)` taking as input a "3-address with temporaries" RISCV code and outputs a list of instructions as a replacement. For instance, each time we access a source operand, we have to load it from memory before, thus the `replace` should contain something like

   ```
   # regxxx is the register used to hold the value betwen the load and
   # the operation itself (one of s1, s2, s3).
   ```

```
    # loc is the place in memory where the temporary is allocated (of
    # the form Offset(..., fp), obtained with get_alloced_loc().
    before.append(RiscV.ld(regxxx, loc))
```

Be careful to not add useless `ld` or `sd` instructions!

The files you generate have to be tested with the RISCV simulator with the same script as before. **Of course, with "all-in-mem" allocation, tests that were "skipped" due to the lack of registers with the naive allocation should not be skipped for `test_alloc_mem`.**

More tests   Now that your compiler can deal with a large number of temporaries, make sure all features are heavily tested (the test suite we provide is in no way sufficient).

4.6 Extensions

You may need to write tests that are accepted by your compiler but not by GCC. If you do so, add a `// SKIP TEST EXPECTED` directive in your tests, to disable the `test_expect` that would otherwise check your file using GCC.

<u>EXERCISE #6</u> ► **C- or Fortran-like for loops code generation**
If you implemented one of the extensions in Lab 3, you can add it to code generation.
    Note that the semantics of fortran-like loops when the loop counter is assigned within the loop makes the code generation harder than C-like loops, where the loop counter is a variable like any other.

4.7 Delivery

This lab will be graded, but we will only ask you to upload it along with its second part (Lab4b), which takes place next week. We highly recommend you to finish this part at least up to the all-in-mem allocator before, in particular all tests from `make test-lab4 MODE=codegen-linear` (including your own) should pass.

c	
| | ```
dest <- fresh_tmp()
code.add("li dest, c")
return dest
``` |
| x | |
| | ```
# get the temporary associated to x.
tmp <- symbol_table[x]
return tmp
``` |
| e_1+e_2 | |
| | ```
t1 <- GenCodeExpr(e_1)
t2 <- GenCodeExpr(e_2)
dest <- fresh_tmp()
code.add("add dest, t1, t2")
return dest
``` |
| e_1-e_2 | |
| | ```
t1 <- GenCodeExpr(e_1)
t2 <- GenCodeExpr(e_2)
dest <- fresh_tmp()
code.add("sub dest, t1, t2")
return dest
``` |
| true | |
| | ```
dest <-fresh_tmp()
code.add("li dest, 1")
return dest
``` |
| $e_1 < e_2$ | |
| | ```
dest <- fresh_tmp()
t1 <- GenCodeExpr(e1)
t2 <- GenCodeExpr(e2)
endrel <- fresh_label()
code.add("li dest, 0")
# if t1>=t2 jump to endrel
code.add("bge endrel, t1, t2")
code.add("li dest, 1")
code.addLabel(endrel)
return dest
``` |

Figure 4.2: 3@ Code generation for numerical or Boolean expressions

x = e	
	```\n tmp <- GenCodeExpr(e)\n loc <- symbol_table[x]\n code.add("mv loc, tmp")\n```
S1; S2	
	```\n# Just concatenate codes\nGenCodeSmt(S1)\nGenCodeSmt(S2)\n```
if *b* then *S*1 else *S*2	
	```\nlelse <- fresh_label()\nlendif <- fresh_label()\nt1 <- GenCodeExpr(b)\n#if the condition is false, jump to else\ncode.add("beq lelse, t1, 0")\nGenCodeSmt(S1) # then\ncode.add("j lendif")\ncode.addLabel(lelse)\nGenCodeSmt(S2) # else\ncode.addLabel(lendif)\n```
while *b* do *S* done	
	```\nltest <- fresh_label()\nlendwhile <- fresh_label()\ncode.addLabel(ltest)\nt1 <- GenCodeExpr(b)\ncode.add("beq lendwhile, t1, 0")\nGenCodeSmt(S) # execute S\ncode.add("j ltest") # and jump to the test\ncode.addLabel(lendwhile) # else it is done.\n```

Figure 4.3: 3@ Code generation for Statements

Appendix A
RISCV **Assembly Documentation (ISA), rv64g**

About

- RISCV is an open instruction set initially developed by Berkeley University, used among others by Western Digital, Alibaba and Nvidia.

- We are using the rv64g instruction set: **R**isc-**V**, 64 bits, **G**eneral purpose (base instruction set, and extensions for floating point, atomic and multiplications), without compressed instructions. In practice, we will use only 32 bits instructions (and very few of floating point instructions).

- Document: Laure Gonnord and Matthieu Moy, for CAP and MIF08.

This is a simplified version of the machine, which is (hopefully) conform to the chosen simulator.

A.1   Installing the simulator and getting started

To get the RISCV assembler and simulator, follow instructions of the first lab (git pull on the course lab repository).

A.2   The RISCV architecture

Here is an example of RISCV assembly code snippet (a proper `main` function would be needed to execute it, cf. course and lab):

```
1   addi a0, zero, 17  # initialisation of  a register to 17
2 loop:
3   addi a0, a0, -1    # subtraction of an immediate
4   j  loop           # equivalent to jump xx
```

The rest of the documentation is adapted from `https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md` and `https://github.com/jameslzhu/riscv-card/blob/master/riscv-card.pdf`

A.3   (Incomplete) RISC-V Assembly Programmer's Manual

A.3.1   Copyright and License Information - Documents

The RISC-V Assembly Programmer's Manual is

© 2017 Palmer Dabbelt `palmer@dabbelt.com` © 2017 Michael Clark `michaeljclark@mac.com` © 2017 Alex Bradbury `asb@lowrisc.org`

It is licensed under the Creative Commons Attribution 4.0 International License (CC-BY 4.0). The full license text is available at https://creativecommons.org/licenses/by/4.0/.

- Official Specifications webpage: https://riscv.org/specifications/

- Latest Specifications draft repository: https://github.com/riscv/riscv-isa-manual

This document has been modified by Laure Gonnord & Matthieu Moy, in 2019 for teaching purpose (MIF08 and CAP).

A.3.2 Registers

Registers are the most important part of any processor. RISC-V defines various types, depending on which extensions are included: The general registers (with the program counter), control registers, floating point registers (F extension), and vector registers (V extension). We won't use control nor F or V registers.

General registers

The RV32I base integer ISA includes 32 registers, named x0 to x31. The program counter PC is separate from these registers, in contrast to other processors such as the ARM-32. The first register, x0, has a special function: Reading it always returns 0 and writes to it are ignored.

In practice, the programmer doesn't use this notation for the registers. Though x1 to x31 are all equally general-use registers as far as the processor is concerned, by convention certain registers are used for special tasks. In assembler, they are given standardized names as part of the RISC-V **application binary interface** (ABI). This is what you will usually see in code listings. If you really want to see the numeric register names, the -M argument to objdump will provide them.

Register	ABI	Use by convention	Preserved?
x0	zero	hardwired to 0, ignores writes	*n/a*
x1	ra	return address for jumps	no
x2	sp	stack pointer	yes
x3	gp	global pointer	*n/a*
x4	tp	thread pointer	*n/a*
x5	t0	temporary register 0	no
x6	t1	temporary register 1	no
x7	t2	temporary register 2	no
x8	s0 *or* fp	saved register 0 *or* frame pointer	yes
x9	s1	saved register 1	yes
x10	a0	return value *or* function argument 0	no
x11	a1	return value *or* function argument 1	no
x12	a2	function argument 2	no
x13	a3	function argument 3	no
x14	a4	function argument 4	no
x15	a5	function argument 5	no
x16	a6	function argument 6	no
x17	a7	function argument 7	no
x18	s2	saved register 2	yes
x19	s3	saved register 3	yes
x20	s4	saved register 4	yes
x21	s5	saved register 5	yes
x22	s6	saved register 6	yes
x23	s7	saved register 6	yes
x24	s8	saved register 8	yes
x25	s9	saved register 9	yes
x26	s10	saved register 10	yes
x27	s11	saved register 11	yes
x28	t3	temporary register 3	no
x29	t4	temporary register 4	no
x30	t5	temporary register 5	no
x31	t6	temporary register 6	no
pc	*(none)*	program counter	*n/a*

Registers of the RV32I. Based on RISC-V documentation and Patterson and Waterman "The RISC-V Reader" (2017)

As a general rule, the **saved registers** `s0` to `s11` are preserved across function calls, while the **argument registers** `a0` to `a7` and the **temporary registers** `t0` to `t6` are not. The use of the various specialized registers such as `sp` by convention will be discussed later in more detail.

A.3.3 Instructions

Arithmetic

add, addi, sub, classically.

```
addi a0, zero, 42
```

initialises `a0` to $0 + 42 = 42$.

Labels

Text labels are used as branch, unconditional jump targets and symbol offsets. Text labels are added to the symbol table of the compiled module.

```
loop:
        j loop
```

Jumps and branches target is encoded with a relative offset in bytes. It is relative to the beginning of the current instruction. For example, the self-loop above corresponds to an offset of 0 bytes.

Branching

Test and jump, within the same instruction:

```
  beq a0, a1, end
```

tests whether the values stored in `a0` and `a1` are equal, and jumps to 'end' if so.

Absolute addressing

The following example shows how to load an absolute address:

```
.section .text
.globl _start
_start:
        lui a0,      %hi(msg)        # load msg(hi)
        addi a0, a0, %lo(msg)        # load msg(lo)
        jal ra, puts
2:      j 2b

.section .rodata
msg:
        .string "Hello World\n"
```

which generates the following assembler output and relocations as seen by objdump:

```
0000000000000000 <_start>:
   0:   000005b7            lui a1,0x0
           0: R_RISCV_HI20 msg
   4:   00858593            addi    a1,a1,8 # 8 <.L21>
           4: R_RISCV_LO12_I   msg
```

Relative addressing

The following example shows how to load a PC-relative address:

```
.section .text
.globl _start
_start:
1:      auipc a0,     %pcrel_hi(msg) # load msg(hi)
        addi  a0, a0, %pcrel_lo(1b)  # load msg(lo)
        jal ra, puts
2:      j 2b

.section .rodata
msg:
        .string "Hello World\n"
```

which generates the following assembler output and relocations as seen by objdump:

```
0000000000000000 <_start>:
   0:   00000597            auipc   a1,0x0
            0: R_RISCV_PCREL_HI20   msg
   4:   00858593            addi    a1,a1,8 # 8 <.L21>
            4: R_RISCV_PCREL_LO12_I .L11
```

Load Immediate

The following example shows the `li` pseudo instruction which is used to load immediate values:

```
        li a0, 0x76543210
```

which generates the following assembler output as seen by objdump (generated code will be different depending on the constant):

```
   0:   76543537            lui     a0,0x76543
   4:   2105051b            addiw   a0,a0,528
```

Load Address

The following example shows the `la` pseudo instruction which is used to load symbol addresses:

```
.section .text
.globl _start
_start:

        la a0, msg

.section .rodata
msg:
        .string "Hello World\n"
```

A.3.4 Assembler directives for CAP and MIF08

Both the RISC-V-specific and GNU .-prefixed options.
    The following table lists assembler directives:

Directive	Arguments	Description
.align	integer	align to power of 2 (alias for .p2align)

Directive	Arguments	Description
.file	"filename"	emit filename FILE LOCAL symbol table
.globl	symbol_name	emit symbol_name to symbol table (scope GLOBAL)
.local	symbol_name	emit symbol_name to symbol table (scope LOCAL)
.section	[{.text,.data,.rodata,.bss}]	emit section (if not present, default .text) and make current
.size	symbol, symbol	accepted for source compatibility
.text		emit .text section (if not present) and make current
.data		emit .data section (if not present) and make current
.rodata		emit .rodata section (if not present) and make current
.string	"string"	emit string
.equ	name, value	constant definition
.word	expression [, expression]*	32-bit comma separated words
.balign	b,[pad_val=0]	byte align
.zero	integer	zero bytes

A.3.5 Assembler Relocation Functions

The following table lists assembler relocation expansions:

Assembler Notation	Description	Instruction / Macro
%hi(symbol)	Absolute (HI20)	lui
%lo(symbol)	Absolute (LO12)	load, store, add
%pcrel_hi(symbol)	PC-relative (HI20)	auipc
%pcrel_lo(label)	PC-relative (LO12)	load, store, add

A.3.6 Instruction encoding

Credit This is a subset of the RISC-V greencard, by James Izhu, licence CC by SA, `https://github.com/jameslzhu/riscv-card`

Core Instruction Formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

"imm[x:y]" means "bits x to y from binary representation of imm". "imm[y|x]" means "bits y, then x of imm". Negative immediate values are stored using two's complement (e.g. -1 is 1111...1).

RV32I Base Integer Instructions - CAP subset

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	

Pseudo Instructions

Pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address
`{lb|lh|lw|ld} rd, symbol`	`auipc rd, symbol[31:12]` `{lb|lh|lw|ld} rd, symbol[11:0](rd)`	Load global
`{sb|sh|sw|sd} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`{flw|fld} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`{fsw|fsd} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O

RV32M Multiply Extension (basic instructions)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
`mul`	MUL	R	`0110011`	`0x0`	`0x01`	`rd = (rs1 * rs2)[31:0]`
`div`	DIV	R	`0110011`	`0x4`	`0x01`	`rd = rs1 / rs2`
`rem`	Remainder	R	`0110011`	`0x6`	`0x01`	`rd = rs1 % rs2`