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, you can access it offline by opening `docs/index.html` in a web browser such as Firefox, or online at `https://drup.github.io/cap-lab22/`.

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

EXERCISE #1 ► RISCV **Simulator - test**
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 testsuite

A few reminders and new features of the testsuite:

- 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 ran through `test_expect` (see below);
- Several tests can be run on each `.c` files:
  - `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 testsuite leaves generated `.s` files next to the `.c` source file.

4.2 First step: three-address code generation

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.

```
##Automatically generated RISCV code, MIF08 & CAP
##non executable 3-Address instructions version


##prelude
# [...] Some automatically generated code that will be explained in a future lab

##Generated Code
# [...] Some automatically generated code that will be explained in a future lab
        li temp_0, 0
        li temp_1, 0
        # (stat (assignment n = (expr (atom 1))) ;)
        li temp_2, 1
        mv temp_0, temp_2
        # (stat (assignment a = (expr (atom 7))) ;)
        li temp_3, 7
        mv temp_1, temp_3
        # (stat (while_stat while ( (expr (expr (atom n)) < (expr (atom a))) ) ( stat_block { (block (stat (assignment n = (expr (expr (atom n)) + (expr (atom 1))) ) ;)) })))
lbl_begin_while_1_main:
        li temp_4, 0
        bge temp_0, temp_1, lbl_end_relational_3_main
        li temp_4, 1
lbl_end_relational_3_main:
        beq temp_4, zero, lbl_end_while_2_main
        # (stat (assignment n = (expr (expr (atom n)) + (expr (atom 1)))) ;)
        li temp_5, 1
        add temp_6, temp_0, temp_5
        mv temp_0, temp_6
        j lbl_begin_while_1_main
lbl_end_while_2_main:
        # (stat (print_stat println_int ( (expr (atom n)) ) ;))
        mv a0, temp_0
        call println_int
# [...] Some automatically generated code that will be explained in a future lab

##postlude
# [...] Some automatically generated code that will be explained in a future lab
```

EXERCISE #2 ► **3-address code generation**

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

```
python3 MiniCC.py --mode codegen-linear TP04/tests/provided/step1/test00.c --reg-alloc=none
```

Don't forget to run `make` before if you need to regenerate the lexer and parser with ANTLR (i.e. if `python3` complains with `No module named 'MiniCLexer'`). Observe the generated code in `<samepath>/test00.s`[1]. You now have to implement the 3-address code generation rules seen in the course. Code and test incrementally[2]:

[1] We generated RISCV comments with MiniC statements for debug.

[2] Using files in the `TP04/tests/*` directories. All the test files you use will have to be in your archive.

- We give you the code generation for the `println_int` instruction. It basically produces a call to the proper function in the library.
- Implement numerical expressions without variables (constants are expected to hold on 64 bits, no boolean expression for now). We advise you to postpone the implementation of MultiplicativeExpr, and first finish this Lab without them (details are given section 4.6).
- Then check that (numerical) expressions with variables work (assignment and usage of variables in expressions are given);

At this step, the code generation is not finished, but we will do some allocation to be able to test properly. All examples in `tests/provided/step1` directory should generate code without any error at this point:

```
for i in TP04/tests/provided/step1/*.c; do
  echo "file="$i; python3 MiniCC.py --mode codegen-linear --reg-alloc=none $i >/dev/null;
done
```

4.3 Testing with the trivial allocator (and real RISCV instructions)

The code generated at this point is not executable since it uses temporaries. 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.

EXERCISE #3 ► **Testing the trivial allocator**
Open, read, understand the `NaiveAllocator` implementation in `Lib/Allocator.py` (`https://drup.github.io/cap-lab22/api/Lib.Allocator.html#Lib.Allocator.NaiveAllocator`) and how it is used to perform the actual RISCV code generation [3]. Then, intensively test your former code generation with this allocator [4]:

```
make TEST_FILES="TP04/tests/provided/step1/*.c" test-naive
```

This script tests all files specified in `TEST_FILES` (or, if not specified, all files in the `*/tests/*` directories except those whose name start with a special character):

- if the pragma `// EXPECTED` is present in the file, it compares the actual output after assembling and simulating with the list of expected values. For instance:

```
int main(){
  int x, y;
  x = 42;
  println_int(x);
  y = x + 8;
  println_int(y);
  return 0;
}
// EXPECTED
// 42
// 50
```

  is an example test case to test assignments.

- If the `AllocationError` exception is raised by the naive allocator, the test is considered “skipped” (i.e. it’s not a failure, but not really a success either, we can’t conclude; the same test case will be used for other allocation strategies).

[3] All available registers are in a list named `GP_REGS`.
[4] Be careful, this allocator crashes if there is more than a certain number of temporaries!

- If the compilation succeeded, it compares the actual output after assembling and simulating to the `// EXPECTED` statements given in the file (which are themselves compared to the output given by `riscv64-unknown-elf-gcc`).
- For debugging, you can obviously launch your compiler manually with e.g.

  ```
  pyright &&
  python3 MiniCC.py --mode codegen-linear --reg-alloc naive --stdout <file>
  ```

  Run `python3 MiniCC.py --help` or see `MiniCC.py` for more options. The `--debug` option allows getting some debug output. Alternatively, you can run the testsuite on a single test file with:

  ```
  make TEST_FILES=TP04/tests/provided/test_while2b.c test-naive
  ```

- When making tests with `make test-naive`, a coverage of your code is created in a folder `htmlcov`. You can look at the file `TP04_MiniCCodeGen3AVisitor_py.html` to check which part of your code has been executed during the tests. If some lines of code you wrote have been missed during the tests, then you must write your own tests for these parts!

At this step, the tests should be OK or SKIPPED for all files given in directory `tests/step1/`:

```
make test-naive
[...]
============================ xx passed, xx skipped in xx seconds ========
```

Now that we have a way to test our code generation for tiny MiniC codes, we can come back to it.

4.4 Finish 3 address code generation

Now that you know how to test your code using the naive allocator, go back to code generation and finish it.

EXERCISE #4 ► **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 `RiscV.print_code()`), you need to generate the instruction to jump to this label (we get the label 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
```

EXERCISE #5 ▶ **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 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 testsuite with `make test-naive` 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) [5], and `label` is a label to jump to if the condition evaluates to true.

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

Tests Up to now, you used `make test-naive` 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-lab4` 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-lab4` does fail.

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).

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.

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

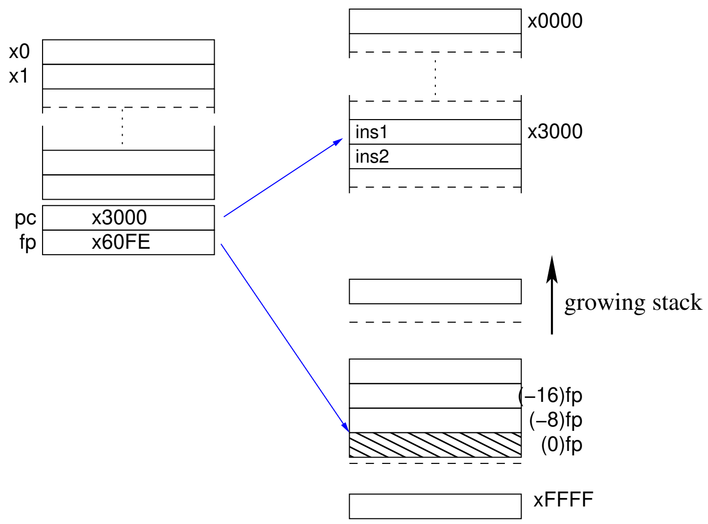

Figure 4.1: Memory model for RISCV

EXERCISE #6 ► **Manual translation**

Complete the expected output for the following two statements (13/15 lines of RISCV code). 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'

```
##Generated code without prelude and postlude
        # (stat (assignment x = (expr (atom 4))) ;)
        # li temp_2, 4
        li s2, 4
        sd s2, -24(fp)
        # end li temp_2, 4
        # mv temp_1, temp_2
        ld s1, -24(fp)
        mv s2, s1
        sd s2, -16(fp)
        # end mv temp_1, temp_2
        # (stat (assignment y = (expr (expr (atom 12)) + (expr (atom x)))) ;)
        # li temp_3, 12
        # TODO 2 lines
        # end li temp_3, 12
        # add temp_4, temp_3, temp_1
        # TODO 4 lines
        # end add temp_4, temp_3, temp_1
        # mv temp_0, temp_4
        # NOT TODO
```

EXERCISE #7 ► **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. This method only maps each temporary to a new offset in memory (in a PYTHON `dict`), allowing you to use the method `get_alloced_loc()` on an Temporary used in the code.
2. Complete the method `AllInMemAllocator.replace(old_instr)` that takes 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 between 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))
```

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 any more.**

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

4.6 Multiplicative Expressions (multiplication, division, modulo)

EXERCISE #8 ► **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 $(a/b)*b+a\%b = a$.

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

4.7 Extensions (bonus)

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.

EXERCISE #9 ► **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.

4.8 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` (including your own) should pass.

c	dest <- fresh_tmp() code.add("li *dest*, *c*") return dest

x	# get the temporary associated to x.
	reg <- symbol_table[x]
	return reg

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 <- new_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

<table>
<tr><td>x = e</td><td><pre>dest <- GenCodeExpr(e)
loc <- symbol_table[x]
code.add("mv loc, dest")</pre></td></tr>
<tr><td>S1; S2</td><td><pre># Just concatenate codes
GenCodeSmt(S1)
GenCodeSmt(S2)</pre></td></tr>
<tr><td>if b then S1 else S2</td><td><pre>lelse <- new_label()
lendif <- new_label()
t1 <- GenCodeExpr(b)
#if the condition is false, jump to else
code.add("beq lelse, t1, 0")
GenCodeSmt(S1) # then
code.add("j lendif")
code.addLabel(lelse)
GenCodeSmt(S2) # else
code.addLabel(lendif)</pre></td></tr>
<tr><td>while b do S done</td><td><pre>ltest <- new_label()
lendwhile <- new_label()
code.addLabel(ltest)
t1 <- GenCodeExpr(b)
code.add("beq lendwhile, t1, 0")
GenCodeSmt(S) # execute S
code.add("j ltest") # and jump to the test
code.addLabel(lendwhile) # else it is done.</pre></td></tr>
</table>

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):

```
  addi a0, zero, 17  # initialisation of  a register to 17
loop:
  addi a0, a0, -1    # subtraction of an immediate
  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 RISC-V Assembly Programmer's Manual - adapted for CAP and MIF08

A.3.1 Copyright and License Information - Documents

- 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.

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 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 a0=a1, and jumps to 'end' if its the case.

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".

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`