
# Compilation de Programmes (MIF08)

---

## Cahier de TD, janvier 2020

# Contents

# TD 1

# RISCV Architecture, Lexical Analysis and Grammars

## 1.1 The RISCV architecture

We give you the "RISCV cheat sheet". The objective is to recall concepts from the architecture course and manipulate the assembly code of the architecture you will compile to.

Your teaching assistant will make a demo of the RISCV simulator during this session.

EXERCISE #1 ▶ **TD**
On paper, write (in RISCV assembly language) a program which initializes the $R0$ register to 1 and increments it until it becomes equal to 8.

EXERCISE #2 ▶ **C to** RISCV- **Skip if you are late**
Translate into RISCV code the following C-code:

```
x=5;
if (x>12) y=70; else y=x+12;
```

## 1.2 Lexical Analysis

A bit of ANTLR4 syntax is given as companion material (on the course website).

EXERCISE #3 ▶ **Regular expressions for lexing**
Use the ANTLR4 syntax to define ANTLR4 macros to define:

1. Identifiers : any sequence of letters, digits and _ that does not begin by a digit nor _.
2. Floats like $-3.96$ (the sign is optional, but the dot is not).
3. Scientific notation like $-1.6E-12$.

EXERCISE #4 ▶ **Romans numbers**
Write an ANTLR4 lexical file that reads and interprets Roman numerals : $IV \rightarrow 4 \ldots$ You can use the fact that the lexical analysis always takes the rule to match the longest subchain.

## 1.3 Grammars

All grammars will use the ANTLR4 syntax.

EXERCISE #5 ▶ **Well-founded parenthesis**
Write a grammar and files to accept any text with well-formed parenthesis ')' and '['.

# TD 2

# AST, Attributions, Types

## 2.1 Derivation trees and attributions

EXERCISE #1 ▶ **Arithmetic expressions**
Let us consider the following grammar (the end of an expression is a semicolon):

$$
\begin{aligned}
Z &\rightarrow E; \\
E &\rightarrow E + T \\
E &\rightarrow T \\
T &\rightarrow T * F \\
T &\rightarrow F \\
F &\rightarrow (E) \\
F &\rightarrow i
\end{aligned}
$$

- What are the derivation trees for $1+2+3;$, $1+2*3;$, $(1+2)*3;$?
- Attribute the grammar to evaluate arithmetic expressions.

EXERCISE #2 ▶ **Declarations of variables**
Write a grammar that accepts declarations of variables like:

```
int x=1;
float y,z;
int t;
float u,v=0;
```

and rejects:

```
int x, int y;
```

Then write an attribution that prints individual declarations (of the first case) like:

```
int x=1; float y; float z; int t; float u; float v=0;
```

EXERCISE #3 ▶ **Prefixed expressions**
Consider prefixed expressions like `* + * 3 4 5 7` (or `* + 1 2 * 3 4`) and assigments of such expressions to variables:
`a=* + * 3 4 5  7`. Identifiers are allowed in expressions.

- Give a grammar that recognizes lists of such assigments.
- Write derivations trees.
- Write grammar rules to compute the values of the expressions.
- Write grammar rules to construct infix assigments during parsing: the former assigment will be transformed into the *string* `a=(3 * 4 + 5)*7`. Be careful to avoid useless parentheses.
- Modify the attribution to verify that the use of each identifier is done after his first definition.

## 2.2 The MiniC language

The objective here is to be familiar with the grammar of the language we will compile.

EXERCISE #4 ▶ **MiniC-grammar**

Here is the (simplified) grammar for the MiniC language (expr are numerical or boolean expressions):

```
grammar MiniC;

prog: function* EOF #progRule;

function: INTTYPE ID OPAR CPAR OBRACE vardecl_l block RETURN INT SCOL CBRACE #funcDecl;

vardecl_l: vardecl* #varDeclList;

vardecl: typee id_l SCOL #varDecl;

id_l
    : ID              #idListBase
    | ID COM id_l     #idList
    ;

block: stat*   #statList;

stat
    : assignment SCOL
    | if_stat
    | while_stat
    | print_stat
    ;

assignment: ID ASSIGN expr #assignStat;

if_stat: IF condition_block (ELSE IF condition_block)* (ELSE stat_block)? #ifStat;

condition_block: OPAR expr CPAR stat_block  #condBlock;

stat_block
    : OBRACE block CBRACE
    | stat
    ;

while_stat: WHILE OPAR expr CPAR  stat_block #whileStat;


print_stat
    : PRINTINT OPAR expr CPAR SCOL         #printintStat
    | PRINTFLOAT OPAR expr CPAR SCOL       #printfloatStat
    | PRINTSTRING OPAR expr CPAR SCOL      #printstringStat
    ;

expr_l
    : /* Nothing */     #exprListEmpty
    | expr              #exprListBase
    | expr COM expr_l   #exprList
    ;


expr
    : MINUS expr                           #unaryMinusExpr
    | NOT expr                             #notExpr
    | expr myop=(MULT|DIV|MOD)  expr       #multiplicativeExpr
    | expr myop=(PLUS|MINUS) expr          #additiveExpr
    | expr myop=(GT|LT|GTEQ|LTEQ)  expr    #relationalExpr
    | expr myop=(EQ|NEQ) expr              #equalityExpr
    | expr AND expr                        #andExpr
    | expr OR expr                         #orExpr
    | atom                                 #atomExpr
    ;

atom
    : OPAR expr CPAR #parExpr
    | (INT | FLOAT)  #numberAtom
    | (TRUE | FALSE) #booleanAtom
    | ID             #idAtom
    | STRING         #stringAtom
    ;
```

Write a valid program for this grammar.

## 2.3 Typing

We recall (some of) the rules of the course for Typing:

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma(x) = t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x : t} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e : t \quad \Gamma \vdash x : t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x = e : \texttt{void}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while}(\, b\,)\{\, S\,\} : \texttt{void}}$$

EXERCISE #5 ► **Typing under given environment**
Considering $\Gamma = \{x_1 \mapsto \texttt{int}\}$, prove that the given sequence of instructions is well typed:

```
x1 = 3 ;
while (x1 < 15) {
  x1 = x1 + 1 ;
}
```

EXERCISE #6 ► **Construction of Gamma with variable declarations**

Using the following rules:

$$\frac{}{t \;\; vlist; \rightarrow_d [v \mapsto t \text{ for v in } vlist]}$$

$$\frac{D_1 \rightarrow_d \Gamma_1 \quad D_2 \rightarrow_d \Gamma_2 \quad Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset}{D1; D_2 \rightarrow_d \Gamma_1 \cup \Gamma_2}$$

Compute $\Gamma$ the typing environnement for the given list of declarations:

```
int x;
bool b1, b2;
```

EXERCISE #7 ► **Boolean expressions**
Write all rules to type boolean expressions of MiniC.

# TD 3

# 3-address Code Generation, Liveness

## 3.1 Code generation with temporaries

The code we generate will have an unbounded number of temporaries (`tmp0`, `tmp1`, ...) but actual RISCV instructions (`add`, `and`, ...) except for the conditional Jump.

The instruction set and documentation for the RISCV machine can be found in Appendix A.

The code generation functions (see Appendix B) have the following signatures:

`GenCodeExpr` : AExp $\rightarrow$ Code$^*$ $\times$ $\mathbb{N}$
`GenCodeSmt` : Inst $\rightarrow$ Code$^*$

where $Code*$ is a sequence of 3-address instructions (RISCV with temporaries). As a side effect, the code generation for statements might update a map $Var \rightarrow \mathbb{N}$ (program variable to a temporary where to find its current value).

Auxiliary functions:

`newTemp()` : $\rightarrow \mathbb{N}$
`newLabel()` : $\rightarrow \mathbb{N}$

EXERCISE #1 ▶ **By hand!**
Using the code generation rules for the RISCV machine, generate the three-address RISCV code for the following (MiniC) program:

```
int main(){
    int a,n;

    n = 1;
    a = 7;
    while (n < a) {
      n = n+1;
    }

return 0;
}
```

EXERCISE #2 ▶ **A new operator for expressions**
Write a code generation rule for the `xor` boolean operator **without using the native** RISCV **operator.**

EXERCISE #3 ▶ **A new langage construction**
Write a code generation rule for the `repeat S until e` statement.

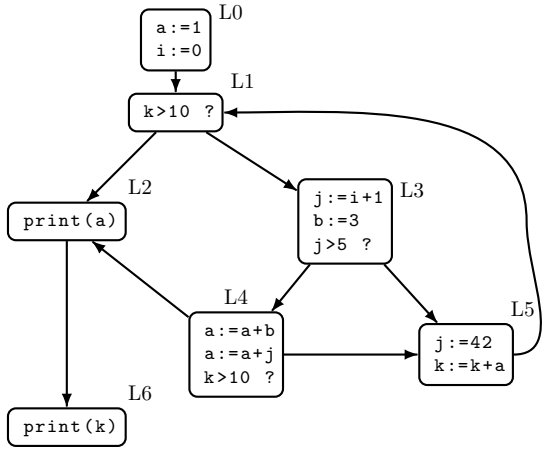
## 3.2 Liveness analysis

### 3.2.1 Liveness by hand

EXERCISE #4 ▶ **Liveness by hand - CC 2016**

In Figure 3.1, we give a CFG and we recall that a *variable is alive after a block if there exists a path from this block to one use of this variable that do not contain a definition of it.*

1. (by hand) Fill the array with "out"-alive variables for each block.
2. Remove dead code.




| bloc | variables vivantes en sortie du bloc |
|---|---|
| L0 | |
| L1 | |
| L2 | |
| L3 | |
| L4 | |
| L5 | |
| L6 | $\emptyset$ |

Figure 3.1: CFG and alive variables to complete

### 3.2.2 Liveness with fixpoint!

Let us recall the notations here: A variable at the left-hand side of an assignement is *killed* by the block. A variable whose value is used in this bloc (before any assignment) is *generated*.

$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = \text{final} \\ \bigcup\{LV_{entry}(\ell') | (\ell, \ell') \in flow(G)\} \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

The sets are initialised to $\emptyset$ and computed iteratively, until reaching a fixpoint.

EXERCISE #5 ▶ **Live variables**
Generate the CFG for the following program:

```
while d>0 then {
    a:=b+c;
    d:=d-b;
    e:=a+f;
    if e>0 then {
        f:=a+d;
        b:=d+f;
    }
    else{
        e:=a-c;
        }
    b:=a+c;
}
```

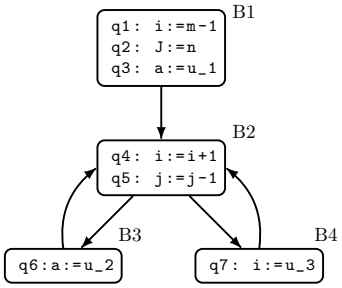
On this CFG:
- Compute $Gen$, $Kill$ for each block $\ell$
- Compute $In(\ell) = LV_{entry}(\ell)$ and $Out(\ell) = LV_{exit}(\ell)$ iteratively.
- Suppress the dead code.

| | | | Step | | Step | | Step | | Step | |
|---|---|---|---|---|---|---|---|---|---|---|
| $\ell$ | $kill(\ell)$ | $gen(\ell)$ | $In(\ell)$ | $Out(\ell)$ | $In(\ell)$ | $Out(\ell)$ | $In(\ell)$ | $Out(\ell)$ | $In(\ell)$ | $Out(\ell)$ |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |

| | | | Step | | Step | | Step | | Step | |
|---|---|---|---|---|---|---|---|---|---|---|
| $\ell$ | $kill(\ell)$ | $gen(\ell)$ | $In(\ell)$ | $Out(\ell)$ | $In(\ell)$ | $Out(\ell)$ | $In(\ell)$ | $Out(\ell)$ | $In(\ell)$ | $Out(\ell)$ |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |

EXERCISE #6 ▶ **Live Variables**

After code generation, we obtain the following graph:




On this graph, perform liveness analysis and suppress the dead code.

| | | | Step | | Step | | Step | | Step | |
|---|---|---|---|---|---|---|---|---|---|---|
| $\ell$ | $kill(\ell)$ | $gen(\ell)$ | $In(\ell)$ | $Out(\ell)$ | $In(\ell)$ | $Out(\ell)$ | $In(\ell)$ | $Out(\ell)$ | $In(\ell)$ | $Out(\ell)$ |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |

| | | | Step | | Step | | Step | | Step | |
|---|---|---|---|---|---|---|---|---|---|---|
| $\ell$ | $kill(\ell)$ | $gen(\ell)$ | $In(\ell)$ | $Out(\ell)$ | $In(\ell)$ | $Out(\ell)$ | $In(\ell)$ | $Out(\ell)$ | $In(\ell)$ | $Out(\ell)$ |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |

# TD 4

# Register allocation and final code generation

In the following exercises we are looking for compiler independant optimisations (on the 3-address code). The goal here is to allocate registers with as few "spilled variables" as possible.

## 4.1 Register Allocation

EXERCISE #1 ▶ **Code production and register allocation**
Consider the expression $E = ((n * (n+1)) + (2 * n)$. We assume that we have:

- The variable $n$ is stored in the stack slot referred as $[n]$ in the load (ld) instruction.

1. Generate a 3 address-code with temporaries and $rmem$ instruction to load $n$. Do it as blindly as possible (no temporary recycling).
2. (Without applying liveness analysis) Draw the liveness intervals. How many registers are sufficient to compute this expression?
3. Draw the interference graph (nodes are variables, edges are liveness conflicts).
4. Color this graph using the algorithm seen in the course (unbounded number of colors).
5. Give a register allocation with $K = 2$ registers, and rewrite code.

EXERCISE #2 ▶ **Register allocation, adapted from Exam, 2016**

We consider (in two columns) the following RISCV code. The $tmp_i$ are temporaries to be allocated (in registers, in memory). For this exercise, we consider that we have two instructions that are capable to directly read/write at memory labels (`ld , sd`).

```
[...]
ld  tmp_1, label1
ld  tmp_2, label2
sub tmp_3, tmp_1, tmp_2
ld  tmp_4, label3
ld  tmp_5, label4
sub tmp_6, tmp_4, tmp_5
add tmp_7, tmp_6, 0
add tmp_8, tmp_3, tmp_7
sd  tmp_8, label5
ret
```

```
;;données/résultats (.dword = 8 bytes)
label1 : .dword 2
label2 : .dword 3
label3 : .dword -1
label4 : .dword 7
label5 : .dword 0
```

1. What is the computed expression? Where will it be stored?
2. Fill the following table with stars: put a star for a given temporary at a given line if and only if it is alive *at the entry of the instruction*. After the last store, all temporaries are supposed to be dead.

| code | tmp_1 | tmp_2 | tmp_3 | tmp_4 | tmp_5 | tmp_6 | tmp_7 | tmp_8 |
|---|---|---|---|---|---|---|---|---|
| `ld tmp_1, label1` | | | | | | | | |
| `ld tmp_2, label2` | | | | | | | | |
| `sub tmp_3, tmp_1, tmp_2` | | | | | | | | |
| `ld tmp_4, label3` | | | | | | | | |
| `ld tmp_5, label4` | | | | | | | | |
| `sub tmp_6, tmp_4, tmp_5` | | | | | | | | |
| `add tmp_7, tmp_6, 0` | | | | | | | | |
| `add tmp_8, tmp_3, tmp_7` | | | | | | | | |
| `sd tmp_8, label5` | | | | | | | | |

3. Draw the interference graph.

4. Color the graph with the algorithm from the course with an infinite number of color (green, blue, red, black , ... in this order).

5. (We make as it we had only three available registers). We decide to spill the $tmp_3$ register and place it in memory.

   Generate the final code with two registers ($t_3$,$t_4$), sp and fp for the stack, $t_0, t_1, t_2$ for the spill management.

# 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. It is relative to the beginning of the current instruction. For example, the self-loop above corresponds to an offset of 0.

#### 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:

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

.equ CONSTANT, 0xcafebabe

        li a0, CONSTANT
```

which generates the following assembler output as seen by objdump:

```
0000000000000000 <_start>:
   0:   00032537            lui     a0,0x32
   4:   bfb50513            addi    a0,a0,-1029
   8:   00e51513            slli    a0,a0,0xe
   c:   abe50513            addi    a0,a0,-1346
```

**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) |
| .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 |

## 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 | | rd = (rs1 < rs2)?1:0 | |
| sltu | Set Less Than (U) | R | 0110011 | 0x3 | | rd = (rs1 < rs2)?1:0 | zero-extends |
| addi | ADD Immediate | I | 0010011 | 0x0 | 0x00 | rd = rs1 + imm | |
| xori | XOR Immediate | I | 0010011 | 0x4 | 0x00 | rd = rs1 ^ imm | |
| ori | OR Immediate | I | 0010011 | 0x6 | 0x00 | rd = rs1 \| imm | |
| andi | AND Immediate | I | 0010011 | 0x7 | 0x00 | 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]`<br>`addi rd, rd, symbol[11:0]` | Load address |
| `{lb\|lh\|lw\|ld} rd, symbol` | `auipc rd, symbol[31:12]`<br>`{lb\|lh\|lw\|ld} rd, symbol[11:0](rd)` | Load global |
| `{sb\|sh\|sw\|sd} rd, symbol, rt` | `auipc rt, symbol[31:12]`<br>`s{b\|h\|w\|d} rd, symbol[11:0](rt)` | Store global |
| `{flw\|fld} rd, symbol, rt` | `auipc rt, symbol[31:12]`<br>`fl{w\|d} rd, symbol[11:0](rt)` | Floating-point load global |
| `{fsw\|fsd} rd, symbol, rt` | `auipc rt, symbol[31:12]`<br>`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]`<br>`jalr x1, x1, offset[11:0]` | Call far-away subroutine |
| `tail offset` | `auipc x6, offset[31:12]`<br>`jalr x0, x6, offset[11:0]` | Tail call far-away subroutine |
| `fence` | `fence iorw, iorw` | Fence on all memory and I/O |

# Appendix B
## Code Generation Rules

| | |
|---|---|
| c | `dr <-newTemp()`<br>`code.add(InstructionLI(dr, c))`<br>`return dr` |
| x | `#get the place associated to x.`<br>`regval<-getTemp(x)`<br>`return regval` |
| $e_1$+$e_2$ | `t1 <- GenCodeExpr(e_1)`<br>`t2 <- GenCodeExpr(e_2)`<br>`dr <- newTemp()`<br>`code.add(InstructionADD(dr, t1, t2))`<br>`return dr` |
| $e_1$-$e_2$ | `t1 <- GenCodeExpr(e_1)`<br>`t2 <- GenCodeExpr(e_2)`<br>`dr <- newTemp()`<br>`code.add(InstructionSUB(dr, t1, t2))`<br>`return dr` |
| true | `dr <-newTemp()`<br>`code.add(InstructionLI(dr, 1))`<br>`return dr` |
| $e_1 < e_2$ | `dr <- newTemp()`<br>`t1 <- GenCodeExpr(e1)`<br>`t2 <- GenCodeExpr(e2)`<br>`endrel <- newLabel()`<br>`code.add(InstructionLI(dr, 0))`<br>`#if t1>=t2 jump to endrel`<br>`code.add(InstructionCondJUMP(endrel, t1, '>=' , t2)`<br>`code.add(InstructionLI(dr, 1))`<br>`code.addLabel(endrel)`<br>`return dr` |

Figure B.1: 3@ Code generation for numerical or Boolean expressions (t1 and t2 are already defined)

| | |
|---|---|
| x = e | `dr <- GenCodeExpr(e)`<br>`#a code to compute e has been generated`<br>`find loc the location for var x`<br>`code.add(instructionMV(loc,dr))` |
| S1; S2 | `#concat codes`<br>`GenCodeSmt(S1)`<br>`GenCodeSmt(S2)` |
| `if` *b* `then` *S*1 `else` *S*2 | `lelse,lendif <-newLabels()`<br>`t1 <- GenCodeExpr(b)`<br>`#if the condition is false, jump to else`<br>`code.add(InstructionCondJUMP(lelse, t1, "=", 0))`<br>`GenCodeSmt(S1) #then`<br>`code.add(InstructionJUMP(lendif))`<br>`code.addLabel(lelse)`<br>`GenCodeSmt(S2) #else`<br>`code.addLabel(lendif)` |
| `while(` *b* `){` *S* `}` | `ltest,lendwhile <-newLabels()`<br>`code.addLabel(ltest)`<br>`t1 <- GenCodeExpr(b)`<br>`code.add(InstructionCondJUMP(lendwhile, t1, "=", 0))`<br>`GenCodeSmt(S)` `#execute S`<br>`code.add(InstructionJUMP(ltest))` `#and jump to the test`<br>`code.addLabel(lendwhile)` `#else it is done.` |

Figure B.2: 3@ Code generation for Statements