

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: **Risc-V**, 64 bits, **General 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/jameslzh/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

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.

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

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_L012_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
<code>.align</code>	integer	align to power of 2 (alias for <code>.p2align</code>)
<code>.file</code>	"filename"	emit filename FILE LOCAL symbol table
<code>.globl</code>	symbol_name	emit symbol_name to symbol table (scope GLOBAL)
<code>.local</code>	symbol_name	emit symbol_name to symbol table (scope LOCAL)
<code>.section</code>	[{.text,.data,.rodata,.bss}]	emit section (if not present, default <code>.text</code>) and make current

Directive	Arguments	Description
.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	zero-extends
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	
addi	ADD Immediate	I	0010011	0x0	0x00	rd = rs1 + imm	zero-extends
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]	
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	zero-extends zero-extends
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	
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 ≥ rs2) PC += imm	
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	<i>Myriad sequences</i>	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	fsgnjd.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

Appendix B

A bit of PYTHON 3 & ANTLR4

B.1 PYTHON

<https://docs.python.org/fr/3.5/tutorial/>
http://perso.limsi.fr/poinal/_media/python:cours:mementopython3.pdf

Coding Style :

<https://www.python.org/dev/peps/pep-0008/>

We strongly recommend to use:

`flake8 filename.py`

on each file.

Exceptions in PYTHON Recall that in PYTHON errors can be declared, thrown and caught as depicts Figure B.1

```
# declare !
class MyError(Exception):
    pass

# catch!
try:
    ...
except MyError:
    ...

# launch !
raise MyError("Error_Message")
```

Figure B.1: Exceptions in PYTHON

B.2 ANTLR4

A nice book:

<https://pragprog.com/book/tpantlr2/the-definitive-antlr-4-reference>

A nice tutorial:

<https://tomassetti.me/antlr-mega-tutorial/>