RISC-V

Introduction

RISC-V is a specification of an instruction set architecture (ISA) for 32-bit, 64-bit, and 128-bit microprocessors. This ISA gives a common base of machine code language for processors. It describes the standard that a RISC-V processor has to comply with.

RISC stands for reduced instruction set computer, and RISC-V appends the suffix V for the Latin number five. With this, the 5th version of RISC, developed at the University of California, Berkeley, is denoted.

The specification(s) of RISC-V are published under the "Creative Commons Attribution" license. Thus, RISC-V is an open ISA that allows everyone to build processors conforming to RISC-V without license fees.

For processing an instruction, a typical RISC processor performs the classic five-stage RISC pipeline.

This pipeline can be executed as a single cycle for every instruction or pipelined. Using pipelining, parallel execution of the stages can be achieved.

This pipeline includes:

- instruction fetch (IF)
- instruction decode (ID)
- instruction execute (EX)
- memory access (MEM)
- write back (WB)

Pipeline Explanation

- The instruction is fetched from the memory pointed by the program counter.
- As the instruction is encoded as binary code, it is decoded, and it is determined what further information is required and fetched.
- Then the instruction is executed, and a computation using the ALU happens at this stage.
- If the instruction includes memory access, it is the next stage.
- Finally, if there is a result, it is written back to register at the last stage.

Assembly Programming

Source code (.s) --> Cross-Assembler --> Object file (.o) --> Cross-Linker (+ Linker script) --> Executable (.out)

RISC-V Simulator/Emulators:

1. Ripes
2. QEmu
3. Spike

Ripes Simulator Installation

1. Download AppImage file from https://github.com/mortbopet/Ripes/releases
2. cmd> chmod a+x /path/of/AppImageFile
3. cmd> ./AppImageFile

QEmu Installation for RISC-V

1. cmd> sudo apt install qemu-system-misc qemu-user-static binfmt-support opensbi u-boot-qemu
2. cmd> sudo apt install gcc-riscv64-linux-gnu gdb-multiarch
3. Implement "Hello World Program" using any editor.

```
.text

.global _start

_start:
    addi a0, zero, 7
    addi a7, zero, 93
    ecall
```

4. cmd> riscv64-linux-gnu-as -o hello.o hello.s
5. cmd> riscv64-linux-gnu-ld -o hello.out hello.o
6. cmd> qemu-riscv64-static ./hello.out

7. cmd> echo $?
8. cmd> riscv64-linux-gnu-objdump -s -d hello.out
9. cmd> qemu-riscv64-static -g 1234 ./hello.out
10. For debugging, give these commands on another terminal.
    - cmd> gdb-multiarch ./hello.out

```
gdb> target remote :1234
gdb> display /3i $pc       -- see next 3 instructions
gdb> si                    -- step into (or ni)
gdb> info registers x1 x2
gdb> q                     -- quit
```

RISC-V ISA

- The RISC-V unprivileged ISA describes four base ISAs:
  - RV32I (32-bit integer)
  - RV32E (32-bit embedded)
  - RV64I (64-bit integer)
  - RV128I (128-bit integer)
- The numbers 32, 64, and 128 stand for the size of a register and the address space.
- For instance, a register of size 32 can hold values up to 2^32. The RV32I base instruction set contains only 40 instructions.

RISC-V ISA Extensions

Abbreviation	Standard extension (subset)
M	integer multiplication and division
A	atomic instructions
F	single-precision floating point
D	double-precision floating point

Abbreviation	Standard extension (subset)
Q	quad-precision floating point
C	compressed instructions
V	vector operations

- Examples:
  - An RV64IMAFDC supports 64-bit base instructions (RV64I), integer multiplication and division (M), atomic instructions (A), floating point with single (F) and double precision (D), and compressed instructions (C).
  - Sometimes you may see processor information that is built after reading a special control register. Such information may end with SU as RV64IMAFDCSU. This means the processor supports supervisor mode (S) and user mode (U).

RISC-V Registers

- The base ISAs specify 32 registers and the program counter.
- The registers are named x0 to x31. The register's width corresponds to the base ISA, e.g., the registers are each 32 bits wide for RV32I.
- Extensions can have further registers. For example, the floating point extensions have further registers for floating point arithmetics.
- The use of the registers from higher-level languages is specified by the application binary interface (ABI).
- The ABI contains a convention on how the registers should be used when a compiler translates a program in a higher-level language into machine language. e.g. The register x0 holds the value 0, which cannot be changed.

Register	ABI Name	Description
x0	zero	Zero constant
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporaries

Register	ABI Name	Description
x8	s0/fp	Saved / Frame pointer
x9	s1	Saved register
x10-x11	a0-a1	Function args. / return values
x12-x17	a2-a7	Function arguments
x18-x27	s2-s11	Saved registers
x28-x31	t3-t6	Temporaries
pc	-	Program counter

- RISC-V is a load-store architecture, so arithmetic operations are done within the registers.
- The RISC-V base integer instructions are divided into five types with different encoding schemes. * R-type (register)
    - I-type (immediate)
    - S-type (store)
    - B-type (branch)
    - U-type (upper immediate)
    - J-type (jump).

- The table shows the encoding:

Type/Bit	31:25	24:20	19:15	14:12	11:7	6:0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12,10:5]	rs2	rs1	funct3	imm[4:1, 11]	opcode
U	imm[31:12]				rd	opcode
J	imm[20, 10:1, 11, 19:12]				rd	opcode

- Op-code + funct3 specifies operation to be performed.
- The machine code of "addi x1, x0, 1" is composed as |0000 0000 0001|0000 0|000|0000 1|001 0011|
    - In hex: 0x00100093
    - Because: addi Op-code=0x13, funct3=0x00

Arithmetic and Logical operations (Immediate)

- The following table shows the instructions which use immediate values.

instruction	name	format	opcode	funct3	description
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
slli	Shift Left Logical Imm.	I	0010011	0x1	imm[11:5]=0×00, rd = rs1 << imm[4:0]
srli	Shift Right Logical Imm.	I	0010011	0x5	imm[11:5]=0×00, rd = rs1 << imm[4:0]
srai	Shift Right Arith. Imm.	I	0010011	0x5	imm[11:5]=0×20, rd = rs1 >> imm[4:0]
slti	Set Less Than Imm.	I	0010011	0x2	rd = (rs1 < imm)? 0:1
sltiu	Set Less Than Imm. Un.	I	0010011	0x3	rd = (rs1 < imm)? 0:1

Arithmetic and Logical operations (Registers)

- The next category of the R-format comprises arithmetic and logical operations that use two registers as source and one register as destination.
- Most of the instructions are a counterpart to the operations with immediate values.

- The following table shows these instructions:

instruction	name	format	opcode	funct3	funct7	description
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
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2
sra	Set Right Arith.	R	0110011	0x5	0x20	rd = rs1 >> rs2
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)? 0:1
sltu	Set Less Than Un.	R	0110011	0x3	0x00	rd = (rs1 < rs2)? 0:1

Load and Store instructions

- To access memory, there are instructions of the I-format for loading data values from memory into a register and of the S-type for storing a register value into memory. This table shows the instructions:

instruction	name	format	opcode	funct3	description
`lb`	Load Byte	I	0000011	0x0	rd = M[rs1+imm][7:0]
`lh`	Load Half	I	0000011	0x1	rd = M[rs1+imm][15:0]
`lw`	Load Word	I	0000011	0x2	rd = M[rs1+imm][31:0]
`lbu`	Load Byte Un.	I	0000011	0x0	rd = M[rs1+imm][7:0]
`lhu`	Load Half Un.	I	0010011	0x0	rd = M[rs1+imm][15:0]
`sb`	Store Byte	I	0010011	0x0	M[rs1+imm][7:0] = rs2[7:0]
`sh`	Store Half	I	0100011	0x1	M[rs1+imm][15:0] = rs2[15:0]
`sw`	Store Word	I	0100011	0x2	M[rs1+imm][31:0] = rs2[31:0]

- We consider the following example and that the value 0x2 is at the memory address 0x10000000:

Address	Machine code	Meaning
0x0	0x00100093	addi x1, x0, 0x1

Address	Machine code	Meaning
0x4	0x01c09093	slli x1, x1, 28
0x8	0x00008103	lb x2, 0(x1)

- The first two instructions set the value in the register x1 to 0x10000000: The value 1 is written into x1 and shifted to the left by 28 positions, which yields 0x10000000 in the register x1.
- The third instruction loads a byte from the address given by x1 and added by the immediate value 0, thus, from the address 0x10000000
- As you see, setting an address in a register with the instructions discussed so far is less convenient.
- To provide an easier way, there are the instructions of the U-format.

instruction	name	format	opcode	funct3	description
`lui`	Load Upper Imm.	U	0110111	-	rd = imm << 12
`auipc`	Add Upper Imm. to PC	U	0010111	-	rd = PC + (imm << 12)

- The instruction lui rd, imm sets the value of the register rd to the immediate value shifted 12 bits to the left. And the instruction auipc rd, imm sets the value of the register rd to the result of the sum of the current program counter's value, and the immediate value shifted 12 bits to the left.

Address	Machine code	instruction	Meaning
0x0	0x100000b7	lui x1, 0x10000	loads 0x10000000 into x1
0x4	0x10000117	auipc x2, 0x10000	loads 0x10000004 into x2

- The instruction lui x1, 0x10000 gives the register x1 the value 0x10000 << 12, thus, x1 = 0x10000000 as the 12-bit shift 'appends' three zeros in hexadecimal.

- The instruction auipc x2, 0x10000 sets the register x2 to the sum of the current PC's value (here 0x4) plus the immediate value 0x10000 << 12. Thus, the register x2 has the value 0x10000004 after executing this instruction.

Control instructions

- To allow control flow, the instructions of the B-format are for conditional branching. They are used to branching to another instruction than the directly following one if a condition is fulfilled.

instruction	name	format	opcode	funct3	description
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 < Un.	B	1100011	0x6	if (rs1 < rs2) pc+=imm
bgeu	Branch >= Un.	B	1100011	0x7	if (rs1 >= rs2) pc+=imm

- A branch instruction compares the values of the registers rs1 and rs2 with the comparison specified by the instruction. If the result of the comparison, the condition, is true, the program counter is incremented by the immediate value. E.g., the instruction blt rs1, rs2, imm checks if the content of rs1 is less than the content of rs2. If this is the case, the pc is changed by pc = pc + imm. Otherwise, it continues with the next instruction in memory (pc = pc +4).

- The instructions blt and bge consider the signess of the values; the instructions bltu and bgeu do not.
- Besides the conditional branches, there are unconditional jump instructions jalr and jal.
- These instructions allow jumping to a specific address and, thus, modify the program counter but write the (possible) return address into a register.
- The return address is the instruction at the address after the jump instruction.
- Return addresses allow jumping back (return) from the code segment jumped to.
- The modularity of programs becomes possible this way. The instructions use different formats:

instruction	name	format	opcode	funct3	description
`jal`	Jump and Link	J	1101111	-	rd = PC+4; PC += imm
`jalr`	Jump and Link Register	I	1100111	0x0	rd = PC+4; PC = rs + imm

System Interface

- The two instructions, ecall and ebreak are in the I-format. Both instructions are for accessing system functionality.
- System functionality usually requires privileged access and instructions which are outside the unprivileged user-mode.

instruction	name	format	opcode	funct3	description
ecall	Environment Call	I	1110011	0x0	imm = 0, rd = rs1 = 0, transfer control to system
ebreak	Environment Break	I	1110011	0x0	imm = 1, rd = rs1 = 0, transfer control to debugger

- The instruction ecall requests a service operation of the system, a system call.
- According to an ABI of the system, an identifier and further parameters are provided in registers, whereby the identifier specifies the kind of the system call.
- Example: The following example calls the exit function on Linux. This system call exit instructs the Linux operating system to finish and terminate the calling program.

```
addi x10 x0 0       ; a0=0 - arg of exit()
addi x17 x0 93      ; a7=93 i.e. syscall number of exit()
ecall
```

- The ebreak instruction is used for debugging programs.

Memory Ordering

- The fence instruction is for synchronizing memory access between multiple processors using the same memory.
- The fence instruction divides a program part into instructions before it (predecessor set) and instructions behind it (successor set).
- Thus, it is useful for systems with more than one RISC-V hart, hardware thread (processor core).
- The fence instruction takes care that every instruction of the predecessor set is done by the executing hart and, thus, observed by other harts before any instruction of the successor set.

- The fence instruction is important for synchronizing processes in operating systems, e.g., mechanisms like locks.

instruction	name	format	opcode	funct3	description
fence	Fence	I	0001111	0x0	rd, rs1 reserved. Normal fence for all memory access types has imm = 0b000011111111.

Extension 'M'

- The M-extension comprises instructions for multiplication and division. The instructions are given in this table:

instruction	name	format	opcode	funct3	description	instruction
mul	Multiply	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
mulh	Multiply High	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
mulhsu	Multiply High Sign/Uns.	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
mulhu	Multiply Unsigned	R	0110011	0x3	0x01	rd = (rs1 * rs2)[63:32]
div	Divide	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	Divide Unsigned	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	Remainder Unsigned	R	0110011	0x7	0x01	rd = rs1 % rs2

- A division by zero, which is undefined, does result in the value -1 or all bits set in the result register. Thus, the divisor has to be checked, and a branch may be taken to handle this case in the case of zero.

Assembler Directives

- Assembler directives are for the assembler. They provide the assembler with information on how the text following a directive should be treated, e.g., if the next part contains instructions or data. Directives begin with a dot. The following table shows a list of frequently used directives.

Directive	Arguments	Description
`.text`		change to .text section
`.data`		change to .data section
`.rodata`		change to .rodata section
`.bss`		change to .bss section
`.section`	.text, .data, .rodata, .bss	change to section given by arguments
`.equ`	name, value	define name for value
`.ascii`	“string”	begin string without null terminator
`.asciz`	“string”	begin string with null terminator
`.string`	“string”	same as .asciz
`.byte`	expression [,expression]*	8-bit comma separated words
`.half`	expression [,expression]*	16-bit comma separated words
`.word`	expression [,expression]*	32-bit comma separated words
`.dword`	expression [,expression]*	64-bit comma separated words
`.zero`	integer	zero bytes
`.align`	integer	align to the power of 2
`.globl`	symbol_name	make symbol_name apparent in symbol table

Hide Notes

- These directives are important when you write a program that is linked for running in an environment, e.g., on an embedded system or an operating system.

- The directive .text provides the assembler the information that the following source code section is program/instruction code. The directive .data provides the information that a data section follows, which is initialized and modifiable. The directive .rodata is for an initialized read-only data section (constants). And the directive .bss is for an uninitialized modifiable data section. These sections can be defined by directive .section following with the kind of section.
- A name for a constant value can be defined by .equ. Data values for ASCII strings are set by the directives .ascii, .asciz, and .string. The directives .asciz and .string add a null byte after the text. The text string follows a directive.
- The directives .byte, .half, .word, and .dword are for setting one or more values that follow the directives. The directives result is different in the size of the value(s) following, e.g., .byte does only accept 8-bit values.
- The directive .zero specifies an array of bytes that are all zero. The following integer number after .zero specifies the number of zero values following.
- The directive .align aligns the memory values following the number given by 2 to the power of the parameter. The alignment is achieved by inserting zero bytes.
- The directive .globl symbol_name makes a symbol_name, which is usually defined by a label visible for the linker. The symbol _start is required to provide the program's entry point for the linker, though the linker script defines which symbol is the entry point. We do not cover linker scripts and use the default one in the considered Linux Gnu toolchain environment. Nevertheless, linker scripts are important when working in different environments or with embedded systems, as linker scripts specify the memory layout the program uses.
- The assembler also knows the dot (.) when processing a program. The dot stands for the current address.

Pseudo Instructions

- An assembler translates an assembly program into machine code. In this course, we mainly use the GNU assembler. The assembler does not only understand the RISC-V instructions specified by the ISA, but it also understands pseudo instructions and directives.
- Pseudo instructions are part of the assembly language and are translated into machine code. In contrast to the ISA instructions, pseudo instructions are easier for the programmer to use. A pseudo instruction can be translated into more than one ISA base instruction. Further, pseudo instructions resolve the issue of using manually relocation functions. The next tables show pseudo instructions and how they are translated into basic instructions.

Pseudo instruction	Base instruction(s)	Description
la rd, symbol	auipc rd, symbol[31:12]	Load address (non position independent code

`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, symbol[11:0]`	Load address (non position independent code - non-PIC)
`la rd, symbol`	`auipc rd, symbol@GOT[31:12]` `l{w|d} rd, symbol[11:0](rd)`	Load address (position independent code PIC)
`lla ra, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load local address
`lga rd, symbol`	`auipc rd, symbol@GOT[31:12]` `l{w|d} rd, symbol@GOT[11:0](rd)`	Load global address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rs, symbol, rd`	`auipc rd, symbol[31:12]` `s{b|h|w|d} rs, symbol[11:0](rd)`	Store global
`nop`	`addi x0, x0, 0`	No operation
`li rd, imm`	`Different instructions`	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	1's complement
`neg rd, rs`	`sub rd, x0, rs`	2's complement
`negw rd, rs`	`subw rd, x0, rs`	2's complement word

- The next table shows the pseudo instructions for conditional branching. One part of them simplifies branching using comparisons with zero by omitting the register x0. The other part offers comfort functions for more comparisons by rearranging the parameters for base instructions, e.g., branch less or equal ble rs, rt, imm is the same as bge rt, rs, imm.

Pseudo instruction	Base instruction(s)	Description
`beqz rs, imm`	`beq rs, x0, imm`	if (rs == 0) PC+=imm
`bnez rs, imm`	`bne rs, x0, imm`	if (rs != 0) PC+=imm
`blez rs, imm`	`bge x0, rs, imm`	if (rs <= 0) PC+=imm
`bgez rs, imm`	`bge rs, x0, imm`	if (rs >= 0) PC+=imm
`bltz rs, imm`	`blt rs, x0, imm`	if (rs < 0) PC+=imm
`bgtz rs, imm`	`blt x0, rs, imm`	if (rs > 0) PC+=imm
`bgt rs, rt, imm`	`blt rt, rs, imm`	if (rs > rt) PC+=imm
`ble rs, rt, imm`	`bge rt, rs, imm`	if (rs <= rt) PC+=imm
`bgtu rs, rt, imm`	`bltu rt, rs, imm`	if (rs > rt) PC+=imm, unsign.
`bleu rs, rt, imm`	`bgeu rt, rs, imm`	if (rs <= rt) PC+=imm, unsign.

- The pseudo instructions for unconditional jumping simplify programming for the developer. Registers that are not required do not appear with pseudo instructions. And jumping to an address far away is provided by call and tail.

Pseudo instruction	Base instruction(s)	Description
`j imm`	`jal x0, imm`	PC += imm
`jal imm`	`jal x1, imm`	x1 = PC+4; PC += imm
`jr rs`	`jalr x0, rs, 0`	PC = rs
`jalr rs`	`jalr x1, rs, 0`	x1 = PC+4; PC = rs
`ret`	`jalr x0, x1, 0`	PC = x1
`call imm`	`auipc x6, imm[31:12]` `jalr x1, x6, imm[11:0]`	x1 = PC+4; PC = imm
`tail imm`	`auipc x6, imm[31:12]` `jalr x0, x6, imm[11:0]`	PC = imm

Stack operations

- The stack is a memory area that is used via the stack pointer (register sp). The stack pointer is usually provided by the system; thus, the register sp comes initialized for a user application. The stack grows from a high address to a low address in memory.
- The storing of data on the stack is done by the so-called push action. The value is taken from the stack by the pop action. Let us consider that the register sp has the value 0xff0:

```
li t0, 0xbeabdeaf   # t0 = 0xbeabdeaf
addi sp, sp, -4     # grow stack for four bytes
sw t0, 0(sp)        # push t0 (0xbeabdeaf) to the stack
```

```
lw t0, 0(sp)        # pop 0xbeabdeaf from the stack to t0
addi sp, sp, 4      # shrink stack back
```

Calling convention

- There should be a convention which register is used for return addresses when different programmers work together on a larger program.
- Further, a convention is important, which registers can be assumed as changed or not changed when a function returns.
- This table summarizes the aliases and storing convention, e.g., the register x1 has the alias ra which stands for return address, and the caller of a function has to store the value of register x1 before the function is called and restore it after the function's return.

Register	ABI alias	Description	Saver
x0	zero	zero constant	-
x1	ra	return address	caller
x2	sp	stack pointer	callee / function
x3	gp	global pointer	- / should not be used from user
x4	tp	thread pointer	- / should not be used from user
x5-x7	t0-t2	temporaries	caller

x8	s0 / fp	saved / Frame pointer	callee / function
x9	s1	saved register	callee / function
x10-x11	a0-a1	function args. / return values	caller
x12-x17	a2-a7	function arguments	caller
x18-x27	s2-s11	saved registers	callee / function
x28-x31	t3-t6	temporaries	caller
pc	-	program counter	-

- The stack pointer, global pointer, and thread pointer are used in a system's context. The global pointer and the thread pointer should not be used except from the (operating) system. The stack pointer can be used for storing/restoring values.
- We consider here only the unprivileged instructions of the ISA as the focus is on the basics of the RISC-V assembly language. Unprivileged instructions can be used in every privilege level. RISC-V comes with three privilege levels: user (U), supervisor (S), and machine (M). System functions, e.g., of an operating system, run in supervisor-mode or machine-mode. To use system functions from user-mode, the system functions can be called, but require that you stick to the application binary interface.

Function

- A function (or routine) is implemented by saving the return address, jumping to the function code and returning from the function. It has to be taken care to use the correct register for saving the return address. It is therefore best to stick to the ABI which provides a convention about the usage of registers for parameters and return addresses. For example:

```
li a0, 0            # parameter in a0
jal ra, func        # call of func, return address in ra
nop                 # instruction to be executed after function return
```

```
func:               # function code
# code for function, must not change ra
li a0, 1            # return results in a0, here the value 1
ret                 # return to address given by ra
```

- Instructions that push registers on the stack at the beginning of a function are called function prologue, and instructions that pop registers from the stack at the end of a function are called function epilogue.
- They serve to prepare the use of the registers inside the function’s main body.