Integrated development environment for teaching and research on RISC-V processors
(PDC2023-145832-I00)

Integrated development environment for teaching and research on RISC-V processors

CREATOR

D 2.1
User manual, use cases, and materials for CREATOR and Sail specification

Universidad Carlos III de Madrid

January, 2026

CONTENTS

USER MANUAL AND USE CASES

Architecture guides

RISC-V architecture

CREATOR supports RISC-V 32-bits and 64-bits full specification through Sail, but also has a 32/64-bits standalone build-in partial specification with the I, M, F, and D extensions. A quick reference guide of this partial specification is available at: https://creatorsim.github.io/creator/guides/riscv.pdf.

RISC-V Reference Guide (CREATOR Simulator)

System Calls (ecall)			
Service	**Call Code (a7)**	**Arguments**	**Result**
Print_int	1	a0 = integer	
Print_float	2	fa0 = float	
Print_double	3	fa0 = double	
Print_string	4	a0 = string addr	
Read_int	5		Integer in a0
Read_float	6		Float in fa0
Read_double	7		Double in fa0
Read_string	8	a0 = string addr a1 = length	
Sbrk	9	a0 = length	Address in a0
Exit	10		
Print_char	11	a0 = ASCII code	
Read_char	12		Char in a0

Integer Registers	
Register Name	**Usage**
zero	Constant 0
ra	Return address (routines/functions)
sp	Stack pointer
gp	Global pointer
tp	Thread pointer
t0..t6	Temporary (NOT preserved across calls)
s0..s11	Saved temporary (preserved across calls)
a0, a1	Arguments for functions / return value
a2..a7	Arguments for functions
Floating-point registers	
ft0..ft11	Temporary (NOT preserved across calls)
fs0..fs11	Saved temporary (preserved across calls)
fa0, fa1	Arguments for functions / return value
fa2..fa7	Arguments for functions

Data transfer		Arithmetic (floating-point, .s/.d)	
li rd, n	rd = n (PseudoInst, n-> 32 bits)	fmv.s fd, fs1	fd = fs1
mv rd, rs	rd = rs	fadd.s fd, fs1, fs2	fd = fs1+fs2
lui rd, inm	rd = inm[31:12] <<12 (extend the sign)	fsub.s fd, fs1, fs2	fd = fs1-fs2
Arithmetic (integer)		fmul.s fd, fs1, fs2	fd = fs1*fs2
add rd, rs1, rs2	rd = rs1+rs2	fdiv.s fd, fs1, fs2	fd = fs1/fs2
addi rd, rs1, n	rd = rs1 + n (n-> 12 bits)	fmin.s fd, fs1, fs2	fd = min(fs1,fs2)
sub rd, rs1, rs2	rd = rs1- rs2	fmax.s fd, fs1, fs2	fd = max(fs1,fs2)
mul rd, rs1, rs2	rd = rs1* rs2	fsqrt.s fd, fs1	fd = sqrt(fs1)
div rd, rs1, rs2	rd = rs1/rs2	fmadd.s fd, fs1, fs2, fs3	fd = fs1*fs2+fs3
rem rd, rs1, rs2	rd = rs1% rs2	fmsub.s fd, ff1, fs2, fs3	fd = fs1*fs2-fs3
Logical (integer)		fabs.s fd, fs1	fd = |fs|
and rd, rs1, rs2	rd = rs1 AND rs2	fneg.s fd, fs1	fd = -fs
andi rd, rs1, n	rd = rs1 AND n (n-> 12 bits)	**Integer ↔ Floating point**	
or rd, rs1, rs2	rd = rs1 OR rs2	fmv.w.x fd, rs	fd = rs single = integer
ori rd, rs1, n	rd = rs1 OR n (n-> 12 bits)	fmv.x.w rd, fs	rd = fs integer = single
not rd, rs1	rd = !rs1 (one's complement)	**Comparison (integer), n-> 12 bits**	
neg rd, rs1	rd = !rs1 + 1 (two's complement)	slt rd, rs1, rs2	if (s(rs1) < s(rs2)) rd = 1; else rd = 0
xor rd, rs1, rs2	rd = rs1 XOT rs2	sltu rd, rs1, rs2	if (u(rs1) < u(rs2)) rd = 1; else rd = 0
srli rd, rs1, n	rd = rs1 >> n logical, n-> 5 bits	slti rd, rs1, n	if (s(rs1) < s(n)) rd = 1; else rd = 0
slli rd, rs1, n	rd = rs1 << n n-> 5 bits	sltiu rd, rs1, n	if (u(rs1) < u(5)) rd = 1; else rd = 0
srai rd, rs1, n	rd = rs1 >> n arithmetic, n-> 5 bits	seqz rd, rs1	if (rs1 == 0) rd = 1; else rd = 0
sra rd, rs1, rs2	rd = rs1 >> rs2 arithmetic	snez rd, rs1	if (rs1 != 0) rd = 1; else rd = 0
sll rd, rs1, rs2	rd = rs1 << rs2	sgtz rd, rs1	if (rs1 > 0) rd = 1; else rd = 0
srl rd, rs1, rs2	rd = rs1 >> rs2 logical	sltz rd, rs1	if (rs1 < 0) rd = 1; else rd = 0
Branch Instructions (integer registers)		**Comparison (floating point) (rd=int register, fs1 and fs2 floating point register)**	
beq t0 t1 etiq	Jump to etiq if t0==t1	feq.s rd, fs1, fs2	if (fs1== fs2) rd= 1;else rd = 0 (float)
bne t0 t1 etiq	Jump to etiq if t0!=t1	fle.s rd, fs1, fs2	if (fs1<= fs2) rd= 1;else rd = 0 (float)
blt t0 t1 etiq	Jump to etiq if t0<t1	flt.s rd, fs1, fs2	if (fs1< fs2) rd= 1;else rd = 0 (float)
bltu t0 t1 etiq	Jump to etiq if t0<t1 (unsigned)	feq.d rd, fs1, fs2	if (fs1== fs2) rd= 1;else rd = 0 (double)
bge t0 t1 etiq	Jump to etiq if t0>=t1	fle.d rd, fs1, fs2	if (fs1<= fs2) rd= 1;else rd = 0 (double)
bgeu t0 t1 etiq	Jump to etiq if t0>=t1 (unsigned)	flt.d rd, fs1, fs2	if (fs1< fs2) rd= 1;else rd = 0 (double)
bgt t0 t1 etiq	Jump to etiq if t0>t1	**Function Calls**	
bgtu t0 t1 etiq	Jump to etiq if t0>t1 (unsigned)	jal ra, address	ra = PC; PC = address
ble t0 t1 etiq	Jump to etiq if t0<=t1	jr ra	PC = ra
bleu t0 t1 etiq	Jump to etiq if t0<=t1 (unsigned)	**Hardware Counter**	
j etiq	PC = PC + etiq	rdcycle rd	rd = number of elapsed cycles
Memory Access (integer registers), n->12 bits		**Memory access (floating point), n->12bits**	
la rd, address	rd = address address->32 bits	flw fd, n(rs1)	fd = Memory[n+rs1] load float
lb rd, n(rs1)	rd = Memory[n+rs1] load byte	fsw fd, n(rs1)	Memory[n+rs1] = fd store float
lbu rd, n(rs1)	rd = Memory[n+rs1] load byte unsigned	fld fd, n(rs1)	fd = Memory[n+rs1] load double
lw rd, n(rs1)	rd = Memory[n+rs1] load word	fsd fd, n(rs1)	Memory[n+rs1] = fd store double
sb rd, n(rs1)	Memory[n+rs1] = rd store byte		
sw rd, n(rs1)	Memory[n+rs1] = rd store word		

Conversion Operations		Floating-pont Clasification		
fcvt.w.s rd, fs1	From single precision (fs1) to integer (rd) with sign	fclass.s rd, fs1	Classify single precision	
fcvt.wu.s rd, fs1	From single precision (fs1) to integer (rd) without sign	fclass.d rd, fs1	Classify double precision	
fcvt.s.w fd, rs1	From integer with sign (rs1) to single precision (fd)	**Bit position in rd**	**Meaning**	
fcvt.s.wu fd, rs1	From integer without sign (rs1) to single precision (fd)	0, 7	-Inf, +Inf	
fcvt.w.d rd, fs1	From rom double precision (fs1) to integer (rd) with sign	1	Normalized negative	
fcvt.wu.d rd, fs1	From double precision (fs1) to integer (rd) without sign	2	Not normalized negative	
fcvt.d.w fd, rs1	From integer with sign (rs1) to double precision (fd)	3, 4	-0, +0	
fcvt.d.wu fd, rs1	From integer without sign (rs1) to double precision (fd)	5	Non normalized positive	
fcvt.s.d fd, fs1	From double (fs1) to single precision (fd)	6	Normalized positive	
fcvt.d.s fd, fs1	From single (fs1) to double precision (fd)	8, 9	Signaling NaN, Quiet NaN	

System calls

The supported system calls are the following:

Service	Trap Code	Input	Output	Notes
print_int	a7 = 1	a0 = int to be printed	Print a0 to display	
print_float	a7 = 2	fa0 = float to be printed	Print fa0 to display	
print_double	a7 = 3	fa0 = double to be printed	Print fa0 to display	
print_string	a7 = 4	a0 = 1st char's address	Print string in the display	
read_int	a7 = 5		Read integer in a0	
read_float	a7 = 6		Read float to fa0	
read_double	a7 = 7		Read double to fa0	
read_string	a7 = 8	a0 = buffer address, a1= buffer length	Read string	
sbrk	a7 = 9	a0 = number of bytes	a0 points to the allocated memory	Allocation from heap
exit	a7 = 10			End of execution
print_char	a7 = 11	a0 = ASCII code	Print a0 to display	
read_char	a7 = 12		Read char to a0	

Interrupts

The RISC-V-32 ISA defines different privilege modes that determine the level of access and control a process has over the system's resources levels, in particular, access to *Control Status Registers* (CSRs), the privileged instruction set, and privileged ISA extensions. The instruction subset to control CSRs is provided in the *Zicsr* extension.

The ISA defines three levels: User/Application (U), Supervisor (S), and Machine (M), allowing implementations to provide one, two, or three of them (There are only three supported combinations: M-level, M-level and U-level, or all three). M-level is the highest privilege level, while U-level is intended for conventional applications and S-level for operating systems.

RISC-V handles exceptions and interrupts by generating traps. As a trap involves elevating the privilege level, e.g. requesting an OS system call from a user program, both M and S privilege levels include a set of CSRs for handling them [1]. The `mstatus` (*Machine Status*) register keeps track of and controls the CPU's current operating state. In this register, there are several interrupt-related fields: field `MIE` (*Machine Interrupt Enable*) controls whether interrupts are globally enabled or disabled for that privilege mode, field `MPP` (*Machine Previous Privilege Mode*) holds the previous privilege mode, and, to allow nesting of interrupts, field `MPIE` stores the previous value of `MIE` when an interrupt occurs.

[1] As they are virtually the same, only the M-level set will be described.

The `mie` (*Machine Interrupt Enable*) register allows for a finer control of interrupts, allowing the programmer to set which types of interrupts are enabled: field `MSIE` (*Machine Software Interrupt Enable*), `MTIE` (*Machine Timer Interrupt Enable*), and `MEIE` (*Machine External Interrupt Enable*) control software, timer, and external interrupts, respectively.

The `mip` (*Machine Interrupt Pending*) register indicates the interrupts that are currently pending. Similarly to register `mie`, it specifies the type of interrupt on specific fields: `MSIP` (*Machine Software Interrupt Pending*), `MTIP` (*Machine Timer Interrupt Pending*), and `MEIP` (*Machine External Interrupt Pending*). Each of these fields may be writable or may be read-only. Register `mcause` (*Machine Cause*) provides information about the event that caused the trap. This register contains a field `I` specifying if the cause was an interrupt or an exception, while the rest of the bits are reserved for the exception code. RISC-V defines some of these exception codes, while others are left for the implementation to use.

Register `mtvec` (*Machine Trap-Vector Base-Address*) holds the trap vector configuration. Depending on the value of the `MODE` field, RISC-V allows for polled interrupts (*direct mode*, with a value of `0`) or vectored interrupts (*vectored mode*, with a value of `1`).

In direct mode, all traps cause the PC to be set to the address in the `BASE` field, while on vectored mode, traps set the PC to address *BASE + 4×cause*, *cause* being the exception code found in `mcause`. Finally, register `mepc` (*Machine Exception Program Counter*) holds the address of the instruction that generated the exception while it is handled.

When a trap is taken to M-mode, the corresponding flag in `mip` is set, and the `MIE` field in `mstatus` and the corresponding flag in `mie` are checked to determine if that interrupt is enabled. If it's enabled, the `MIE` field is copied into the `MPIE` field of `mstatus` and `MIE` is cleared, disabling further interrupts. Then, the previous privilege mode is stored in the `MPP` field of `mstatus`, the cause of the exception is encoded into `mcause`, the current PC is stored into the `mepc` register, and the PC is set to the address specified by `mtvec`. When the interrupt handler finishes, it calls `mret`, which resets the privilege mode (reading the `MPP` field in `mstatus`), re-enables interrupts (by copying back field `MPIE` to `MIE` in `mstatus`), and sets the PC to the value of `mepc`.

With regard to the `mip` register, the ISA states that if the field is writable, a pending interrupt can be cleared by clearing the field, but if the field is read-only, the implementation must provide some other mechanism for clearing the pending interrupt.

More details are available at the Master Thesis "Implementing Interrupts, Timers, and Memory-Mapped I/O in CREATOR" by Luis Daniel Casais Mezquida, and the RISC-V's Specification.

MIPS architecture

Overview

CREATOR has a 32-bits build-in specification of the MIPS32 instruction set. A quick reference guide of this specification is available at: https://creatorsim.github.io/creator/guides/mips32.pdf.

Registers

Name	Number	Use
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Evaluation of expressions and function results
v1	3	Evaluation of expressions and function results
a0	4	Argument 1
a1	5	Argument 2
a2	6	Argument 3
a3	7	Argument 4
t0..t7	8..15	Temporal (no value is saved between calls)
s0..s7	16..23	Temporal (value is saved between calls)
t8, t9	24, 25	Temporal (no value is saved between calls)
k0, k1	26, 27	Reserved for the operating system kernel
gp	28	Pointer to the global area
sp	29	Stack pointer
fp	30	Stack frame pointer
ra	31	Return address, used by function calls

System calls

Service	Call code	Arguments	Results
print_integer	$v0 = 1	$a0 = integer	
print_float	$v0 = 2	$f12 = real (32 bits)	
print_double	$v0 = 3	$f12 = real (64 bits)	
print_string	$v0 = 4	$a0 = string	
read_integer	$v0 = 5		Integer ($v0)
read_float	$v0 = 6		Real 32-bits ($f0)
read_double	$v0 = 7		Real 64-bits ($f0)
read_string	$v0 = 8	$a0 = buffer, $a1 = length	
sbrk	$v0 = 9	$a0 = quantity	Address ($v0)
exit	$v0 = 10		
print_char	$v0 = 11	$a0 = byte	
read_char	$v0 = 12		$v0 (ASCII code)

Assembly directives

.data	The following data definitions that appear are stored in the data segment. It can include an argument that indicates the address from where the data will begin to be stored.
.text	The instructions that follow this directive are placed in the code segment. It can include a parameter that indicates where the code zone begins.
.globl symbol	Declares a global symbol that can be referenced from other programs.
.extern label n	Declares that data stored from label occupies N bytes and that label is a global symbol. This directive allows the assembler to store data in an area of the data segment that can be accessed through the $gp register.
.ascii string	Store the string in memory, but it does not end with NULL ('\ 0')
.asciiz string	Store the string in memory and place a NULL ('\ 0') at the end.
.byte b1, ..., bn	Stores N values in successive bytes of memory.
.half h1, ..., hn	Stores N 16-bit numbers in consecutive half-words.
.word w1, ..., wn	Stores N 32-bit elements (words) in consecutive memory locations.
.float f1, ..., fn	Stores N real values of simple precision in consecutive memory positions.
.double d1, ..., dn	Stores N real double precision values in consecutive memory addresses.

Constant Handling Instructions

li Rdest, inmediate	Load immediate value
lui Rdest, inmediate	Load the 16 bits of the lower part of the immediate value in the upper part of the register. The bits of the lower part are set to 0

Data transfer instructions

move Rdest, Rsrc	Move the contents of the Rsrc register to the Rdest register.
mfhi Rdest	Move the contents of the HI register to the Rdest register.
mflo Rdest	Move the contents of the LO register to the Rdest register.
mthi Rsrc	Move the contents of the Rsrc register to the HI register.
mtlo Rsrc	Move the contents of the Rsrc register to the LO register.

Load instructions

la Rdest, address	Load address in Rdest (the address value, not the content)
lb Rdest, address	Load the byte of the specified address and extend the sign
lbu Rdest, address	Load the byte of the specified address, do not extend the sign
lh Rdest, address	Load 16 bits of the specified address, sign is extended
lhu Rdest, address	Load 16 bits of the specified address, no sign is extended
lw Rdest, address	Load a word from the specified address.

Storing instructions

sb Rsrc, address	Stores the lowest Rsrc byte in the indicated address.
sh Rsrc, address	Stores the low half word (16 bits) of a register in the indicated memory address.
sw Rsrc, address	Store the Rsrc in the indicated address.

Arithmetic and logical instructions

In all the following instructions, *Src2* can be both a register and an immediate value (a 16-bit integer) and in those where it puts *inm* it only accepts an immediate value

add Rdest, Rsrc1, Src2	Sum with overflow
addi Rdest, Rsrc1, inm	Add an immediate number with overflow
addu Rdest, Rsrc1, Src2	Sum without overflow
addiu Rdest, Rsrc1, inm	Add an immediate number without overflow
and Rdest, Rsrc1, Src2	AND logical operation
andi Rdest, Rsrc1, inm	AND logical operation with an immediate number
div Rsrc1, Rsrc2	Divide with overflow. Leave the quotient in the register lo and the rest in the register hi
divu Rsrc1, Rsrc2	Divide without overflow. Leave quotient in the register lo and the rest in the register hi
div Rdest, Rsrc1, Rrc2	Divide with overflow
divu Rdest, Rsrc1, Rrc2	Divide without overflow
mul Rdest, Rsrc1, Src2	Multiply without overflow
mult Rsrc1, Rsrc2	Multiply, the low part of the result is left in the lo register and the high part in the hi register
multu Rsrc1, Rsrc2	Multiply without overflow, the low part of the result goes to LO and the high part to HI
mod Rdest, Rsrc1, Rsrc2	Division module with overflow
modu Rdest, Rsrc1, Rsrc2	Division module without overflow
nop	It does not perform any operation
nor Rdest, Rsrc1, Src2	NOR Logic Operation
or Rdest, Rsrc1, Src2	OR Logic Operation
ori Rdest, Rsrc1, inm	OR Logic Operation with immediate
rem Rdest, Rsrc1, Rsrc2	Division module with overflow
rotr rdest, rsrc1, inm	Right rotation of Src2 number of bits
sll Rdest, Rsrc1, inm	Logical bit shift to the left
srl Rdest, Rsrc1, inm	Logical bit shift to the right
sra Rdest, Rsrc1, inm	Arithmetic bit shift to the right
sub Rdest, Rsrc1, Src2	Subtraction (with overflow)
subu Rdest, Rsrc1, Src2	Subtraction (without overflow)
xor Rdest, Rsrc1, Src2	XOR Logic Operation

Branch and jump instructions

In all the following instructions, Src2 can be a record or an immediate value. Branch instructions use a signed 16-bit offset; So you can skip 215-1 instructions forward or 215 instructions backward. The jump instructions contain a 26-bit address field.

b label	Unconditional branch to the instruction that is on label.
beq Rsrc1, Src2, label	Conditional branch if Rsrc1 is equal to Src2.
beqz Rsrc, label	Conditional branch if the Rsrc register is equal to 0.
bge Rsrc1, Src2, label	Conditional branch if the Rsrc1 register is greater than or equal to Src2 (signed).
bgeu Rsrc1, Src2, label	Conditional branch if the Rsrc1 register is greater than or equal to Src2 (unsigned).
bgez Rsrc, label	Conditional branch if the Rsrc register is greater than or equal to 0.
bgezal Rsrc, label	Conditional branch if the Rsrc register is greater than or equal to 0. Save the current address in the $ra register ($31)
bgt Rsrc1, Src2, label	Conditional branch if the Rsrc1 register is greater than Src2 (signed).
bgtu Rsrc1, Src2, label	Conditional branch if the Rsrc1 register is greater than Src2 (unsigned).
bgtz Rsrc, label	Conditional branch if Rsrc is greater than 0.
ble Rsrc1, Src2, label	Conditional branch if Rsrc1 is less than or equal to Src2 (signed).
bleu Rsrc1, Src2, label	Conditional branch if Rsrc1 is less than or equal to Src2 (unsigned).
blez Rsrc, label	Conditional branch if Rsrc is less than or equal to 0.
blt Rsrc1, Src2, label	Conditional branch if Rsrc1 is less than Src2 (signed).
bltu Rsrc1, Src2, label	Conditional branch if Rsrc1 is less than Src2 (unsigned).
bltz Rsrc, label	Conditional branch if Rsrc is less than 0.
bne Rsrc1, Src2, label	Conditional branch if Rsrc1 is not equal to Src2.
bnez Rsrc, label	Conditional branch if Rsrc is not equal to 0.
j label	Unconditional jump.
jal label	Unconditional jump, stores the current address at $ ra ($31).
jalr Rsrc	Unconditional jump, stores the current address at $ ra ($31).
jalr Rsrc1, Rsrc2	Unconditional jump, stores the current address in Rsrc1.
jr Rsrc	Unconditional jump.

Comparison instructions

In all the following instructions, Src2 can be both a register and an immediate value (a 16-bit integer).

seq Rdest, Rsrc1, Src2	Set Rdest to 1 if Rsrc1 is equal to Src2, 0 in other case.
sge Rdest, Rsrc1, Src2	Set Rdest to 1 if Rsrc1 is greater or equal to Src2, and 0 in other case (with sign).
sgeu Rdest, Rsrc1, Src2	Set Rdest to 1 if Rsrc1 is greater or equal to Src2, and 0 in other case (without sign).
sgt Rdest, Rsrc1, Src2	Set Rdest to 1 if Rsrc1 is greater than Src2, and 0 in other case (with sign).
sgtu Rdest, Rsrc1, Src2	Set Rdest to 1 if Rsrc1 is greater than Src2, y 0 in other case (without sign).
sle Rdest, Rsrc1, Src2	Set Rdest to 1 if Rsrc1 is little or equal to Src2, 0 in other case (with sign).
sleu Rdest, Rsrc1, Src2	Set Rdest to 1 if Rsrc1 is little or equal to Src2, 0 in other case (without sign).
slt Rdest, Rsrc1, Src2	Set Rdest to 1 if Rsrc1 is little than Src2, 0 in other case (with sign).
sltu Rdest, Rsrc1, Src2	Set Rdest to 1 if Rsrc1 is little than Src2, 0 in other case (without sign).
sne Rdest, Rsrc1, Src2	Set Rdest to 1 if Rsrc1 not equal to Src2, and 0 in other case.

(Floating point) Constant Handling Instructions

li.s fs, value	Load the value (float) into the fs register of the mathematical coprocessor
li.d fd, value	Load the value (double) into the fd register of the mathematical coprocessor

(Floating point) Data transfer instructions

mfc1 Rdest, CPsrc	Copy contents of the CPsrc register of the floating-point coprocessor to the Rdest CPU register.
mtc1 Rsrc, CPdest	Copy contents of the Rsrc register of the CPU to the CPdest register of the floating-point coprocessor.
mov.s fd, fs	Move the contents of the fs record to the fd record. (float)
mov.d fd, fs	Move the contents of the fs record to the fd record. (double)

(Floating point) Arithmetic and logical instructions

In all the following instructions, Src2 can be both a register and an immediate value (a 16-bit integer) and in those where it puts inm it only accepts an immediate value

abs.s fd, fs	Absolute value of a real 32-bit number.
abs.d fd, fs	Absolute value of a real 64-bit number.
add.s fd, fs, ft	Add the register fs and ft and store the result in fd (float)
add.d fd, fs, ft	Add the register fs and ft and store the result in fd (double)
div.s fd, fs, ft	Divide fs by ft and leave the result in fd (float)
div.d fd, fs, ft	Divide fs by ft and leave the result in fd (double)
mul.s fd, fs, ft	Multiply the register fs and ft and leave your result in fd. (float)
mul.d fd, fs, ft	Multiply the register fs and ft and leave your result in fd. (double)
rsqrt.s fd, fs	Reciprocal Square Root fd = 1.0/sqrt.s(fs) (float)
rsqrt.d fd, fs	Reciprocal Square Root fd = 1.0/sqrt.s(fs) (double)
sqrt.s fd, fs	Square root of fs (float): fd=sqrt(fs)
sqrt.d fd, fs	Square root of fs (double)
sub.s fd, fs, ft	Subtraction (float): fd = fs-ft
sub.d fd, fs, ft	Subtraction (double)

(Floating point) Load instructions

l.s fs, address	Load in fs the value of the float (32 bits) found in the specified address.
l.d fd, address	Load fd with the value of double (64 bits) found in the specified address.

(Floating point) Store instructions

s.s fs, address	Stores the fs register in the indicated address. (float)
s.d fd, address	Store a double (64 bits) in the indicated address, the value of 64 bits comes from fd.

Conversion instructions

cvt.d.s fd, fs	Turn a float into a double, the result is saved in fd
cvt.d.w fd, Rsrc	Convert an integer to a double, the result is saved in fd
cvt.s.d fd, fs	Turn a double into a float, the result is saved in fd
cvt.s.w fd, Rsrc	Convert an integer into a float, the result is saved in fd
cvt.w.s Rdest, fs	Convert a float into an integer, the result is saved in Rdest
cvt.w.d Rdest, fs	Convert a double into an integer, the result is saved in Rdest

System calls

The supported system calls are the following ones:

Service	Trap Code	Input	Output	Notes
print_int	$v0 = 1	$a0 = int to be printed	Print $a0 to display	
print_float	$v0 = 2	$f12 = float to be printed	Print $f12 to display	
print_double	$v0 = 3	$f12 = double to be printed	Print $f12 to display	
print_string	$v0 = 4	$a0 = 1st char's address	Print string in stardard output	
read_int	$v0 = 5		Read integer to $v0	
read_float	$v0 = 6		Read float to $v0	
read_double	$v0 = 7		Read double to $v0	
read_string	$v0 = 8	$a0 = buffer address, $a1= buffer length	Read string	
sbrk	$v0 = 9	$a0 = number of bytes	$v0 points to the allocated memory	Allocation from heap
exit	$v0 = 10			End of execution

CLI User Guide

The CREATOR Command-Line Interface (CLI) provides a powerful text-based environment for assembly programming, debugging, and testing. It's designed for users who prefer working in a terminal and offers advanced features for development and automation.

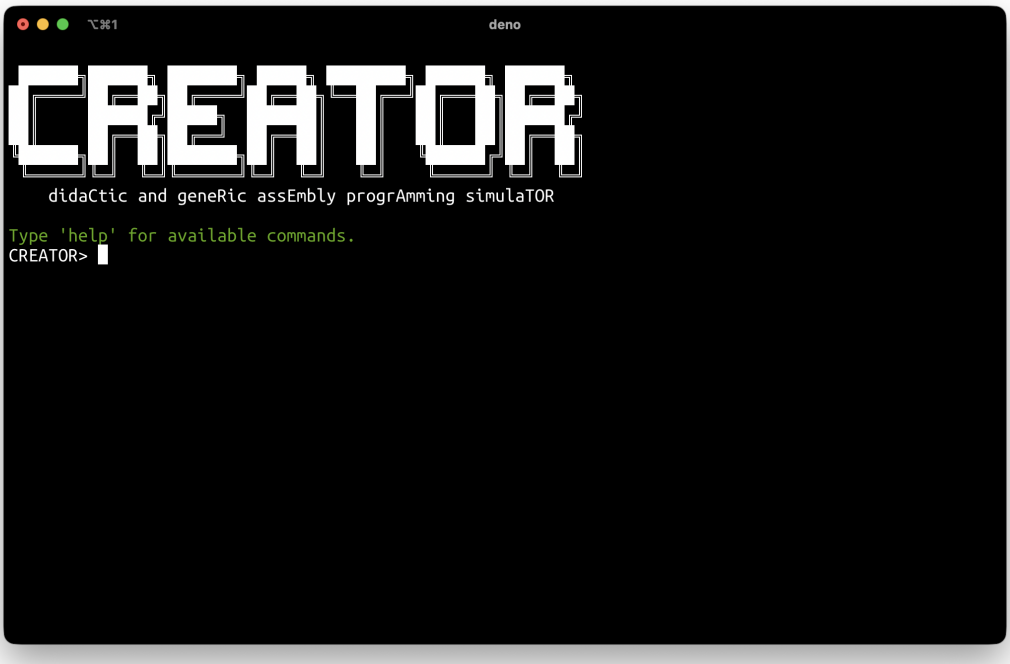

Figure 1: CREATOR CLI interactive prompt.

The CLI version offers several key features:

- **Interactive Mode**: Full-featured REPL with command history
- **State Management**: Save and restore complete simulator snapshots
- **Configuration System**: Customize aliases, shortcuts, and preferences via YAML
- **Accessible Mode**: Screen reader support for visually impaired users

Installation

Installing CREATOR CLI

Installation via precompiled Binary

The easiest way to get started with the CLI version is to use the precompiled binaries. They include everything you need to run CREATOR without additional dependencies. Download the latest precompiled binary for your OS from the releases page and place it in a directory included in your

system's PATH as `creator-cli`.

You're all set! You can verify the installation by running:

```
creator-cli --help
```

To access the CLI from any terminal, ensure the binary is in your PATH or create a symbolic link to it in a directory that is. See below for instructions on creating an alias.

Building from Source

Alternatively, you can build the CREATOR CLI locally.

Prerequisites

- Git is required to download the project.
- Deno must be installed on your system. Follow the instructions on the Deno website to install it.
- Bun is the recommended package manager to install dependencies.

Steps

1. Clone the CREATOR repository:

```
git clone https://github.com/creatorsim/creator-beta.git --recurse-submodules --
  depth=1
cd creator-beta

```

2. Install dependencies:

```
bun install

```

3. Build the CLI:

```
bun build:cli

```

This build script **requires** having Bun installed, but if you don't want to, you can use another package manager (e.g., NPM) and do:

```
npm wasm:cli && npm build:cli:native

```

4. Test your installation:

```
creator-cli --help

```

You should see the CREATOR help message with available options.

You can also just run the CLI without building with Deno:

```
deno run --allow-all src/cli/creator6.mts --help
```

Getting Architecture Files

CREATOR requires architecture definition files to simulate different processors. These files are in YAML format and define the instruction set, registers, memory layout, and other architecture-specific details.

Default Architectures

The repository includes several architecture files in the `architecture/` directory. The releases also include an `architectures.zip` file with the same contents.

For example, the following architectures are provided:

- `RV32IMFD.yml` - RISC-V 32-bit
- `RV64IMFD.yml` - RISC-V 64-bit
- `MIPS32.yml` - MIPS 32-bit
- `Z80.yml` - Z80 8-bit processor (with interrupts)

Configuration

The CLI version supports user configuration via a YAML file located at:

- **macOS/Linux**: `~/.config/creator/config.yml`
- **Windows**: `%USERPROFILE%\.config\creator\config.yml`

This file is automatically created with defaults on first run. See the Configuration section for more details.

Updating

If you installed via precompiled binaries, download the latest version from the releases page.

To update the CLI version, pull the latest changes from the repository and reinstall dependencies:

```
git pull origin main && bun install
```

Command-Line Options

The CREATOR CLI accepts various command-line options to configure architecture, input files, and behavior.

Basic Usage

```
creator-cli [options]
```

Required Options

`--architecture, -a`

Required: Yes
Type: String (path to YAML file)
Description: Path to the architecture definition file. Specifies which processor architecture to simulate.
Examples:

```
: RISC-V architecture
creator-cli --architecture architecture/riscv32.yml --asembly program.s
: MIPS architecture
creator-cli --architecture architecture/mips32.yml  -s program.s
```

Input Options

`--assembly, -s`

Type: String (path to assembly file)
Description: Assembly source file to assemble and load. Assembles the specified file and loads it into memory. **Example**:

```
creator-cli -a architecture/riscv32.yml --asembly hello.s
```

`--bin, -b`

Type: String (path to binary file)
Description: Binary file to load directly into memory. Loads a pre-assembled binary file without compilation. **Example**:

```
creator-cli -a architecture/riscv32.yml --bin programlbin
```

Note: Binary and assembly options are mutually exclusive. If both are provided, binary takes precedence.

`--library, -l`

Type: String (path to library file)
Description: Library file to load before assembly. Loads a library of pre-assembled code that your program can reference. This is only supported when using the default CREATOR assembler. **Example**:

```
creator-cli -a architecture/riscv32.yml -l stdlib.yml -s program.s
```

Assembler Options

`--assembler, -C`

Type: String
Default: `default`
Description: Assembler backend to use. Selects which assembler to use for compilation.

Available Options:

- `default` - CREATOR native assembler
- `sjasmplus` - Z80 assembler with macro support
- `rasm` - Alternative Z80 assembler

Examples:

```
: Use default CREATOR assembler
creator-cli -a architecture/riscv32.yml -s program.s

: Use sjasmplus for Z80
creator-cli -a architecture/z80.yml -s program.s --assembler sjasmplus

: Use rasm for Z80
creator-cli -a architecture/z80.yml -s program.s -C rasm

```

`--isa, -i`

Type: Array of strings
Default: `[]`
Description: ISA extensions to load. In supported architectures, specifies which ISA extensions to enable. If unspecified, the full ISA defined in the architecture file is used.

Examples:

```
: RISC-V with M extension (multiply/divide)
creator-cli -a architecture/riscv32.yml --isa M -s program.s

: RISC-V with multiple extensions
```

```
  creator-cli -a architecture/riscv32.yml --isa M F D -s program.s

  : RISC-V base only (no extensions)
  creator-cli -a architecture/riscv32.yml -s program.s

```

Other Options

`--accessible, -A`

Type: Boolean
Default: `false`
Description: Enable accessible mode for screen readers. Disables colors, ASCII art, and fancy formatting for compatibility with screen readers.

Example:

```
creator-cli --accessible -a riscv32.yml -s program.s
```

Changes in Accessible Mode:

- - No ANSI color codes
- - Plain text output only
- - No ASCII art
- - Descriptive text for all information
- - Structured table layouts

Note: Can also be set in configuration file. Command-line flag overrides config.

`--config, -c`

Type: String (path to YAML file)
Default: `~/.config/creator/config.yml`
Description: Path to configuration file. Specifies a custom configuration file instead of the default location.

Example:

```
creator-cli -a riscv32.yml -s program.s --config custom-config.yml
```

See Configuration for config file format.

`--state`

Type: String (path to JSON file)
Default: None
Description: File to save execution state on exit. Saves the current simulator state to the specified file.

Example:

```
creator-cli -a riscv32.yml -s program.s --state mystate.json
```

Note: Use `restore` command in interactive mode to load saved states.

`--reference, -r`

Type: String (path to file)
Default: None
Description: Reference file for validation. Used for grading and validation of student exercises. Not typically used by end users. See Grading Student Exercises for details.

`--interrupt-handler`

Possible values: `"default"`, `"custom"`
Default: `"default"`
Description: Interrupt handler to use, either CREATOR's default handler or a custom architecture-defined one. Selects the desired interrupt handler to use: CREATOR's default handler, with support for architecture-defined system calls, or a fully custom architecture-defined one.

Example:

```
creator-cli -a riscv32.yml -s program.s --interrupt-handler custom
```

See Interrupt Support for a guide on how to use these features.

Getting Help

Show all available options:

```
creator-cli --help
```

Commands Reference

Complete reference for all CREATOR CLI commands. Commands are case-insensitive and support aliases defined in your configuration.

Execution Control

`step`

Execute one instruction and display the result.

Syntax: `step` or just press Enter. **Alias**: Configured in config file (commonly `s`)

Output:

```
0x00400000 (0x00000293) addi t0,zero,0
```

Shows: `<address> (<machine_code>) <assembly>`

Behavior:

- Executes the instruction at current PC
- PC advances to next instruction - Registers and memory update accordingly
- State is saved for `unstep`

When to Use:

- Step-by-step debugging
- Observe instruction-level behavior
- Verify each operation

`unstep`

Undo the last executed instruction (reverse step).

Syntax: `unstep`
Alias: Configured in config file

Requirements:

- `max_states` must be set in config (>0)
- At least one instruction must have been executed

Behavior:

- Restores simulator to state before last instruction
- PC moves backward

- Register and memory changes reverted
- Can unstep multiple times up to `max_states` limit

Example:

```
CREATOR> step
0x00400000 (0x00000293) addi t0,zero,0

CREATOR> unstep
# PC back at 0x00400000, t0 unchanged
```

When to Use:

- Went too far in debugging
- Want to re-examine instruction
- Study before/after states

`run [count]`

Execute multiple instructions continuously.

Syntax:

- `run` - Run until completion or breakpoint
- `run <count>` - Run exactly N instructions

Alias: Configured in configuration file (commonly `r`)

Examples:

```
CREATOR> run           # Run to completion
CREATOR> run 100       # Run 100 instructions
CREATOR> run 1000000   # Run 1 million instructions
```

Stops When: - Program completes (execution_index = -2) - Breakpoint is hit - Error occurs - Pause shortcut issued

Output:

- Each instruction displayed as it executes
- “Program execution completed.” when done
- “Breakpoint hit at” when breakpoint reached

Performance:

- Executes in chunks for responsive UI
- Can pause mid-execution

`silent [count]`

Run instructions without displaying output.

Syntax:

- `silent` - Run silently until completion
- `silent <count>` - Run N instructions silently

Use Cases:

- Fast execution without visual clutter
- Performance testing
- Running initialization code
- Getting to a specific point quickly

Example:

```
CREATOR> silent 1000   # Skip first 1000 instructions
CREATOR> list          # Now see where we are
```

`continue`

Resume execution from paused or stepped state.

Syntax: `continue`
Alias: Configured in config file (commonly `c`)

Behavior:

- If paused mid-run: Resumes from current point
- If at breakpoint: Steps once then continues running
- Otherwise: Steps once then runs

Example Flow:

```
CREATOR> run
# ... executes many instructions ...
CREATOR> pause
Execution paused.

CREATOR> continue
# Resumes execution
```

`nur`

uN-Run (reverse run) - Step backwards until breakpoint or program start.

Syntax: `nur`

Requirements:

- `max_states` must be set in config
- Sufficient state history available

Behavior:

- Repeatedly unsteps
- Stops at breakpoints (going backwards)
- Stops when no more history available

Example:

```
CREATOR> run          # Execute past several breakpoints
CREATOR> nur          # Go back to last breakpoint
```

When to Use:

- Overshot target during debugging
- Want to re-examine earlier state
- Find when something changed

`until <address|label>`

Run until specified address is reached.

Syntax: `until <address>` or `until <label>`

Examples:

```
CREATOR> until 0x400010    # Run until address
CREATOR> until loop_end    # Run until label
```

Behavior:

1. Sets temporary breakpoint at address/label
2. Runs program
3. When breakpoint hit, removes it

Equivalent To:

```
break <address>
run
break <address>  # Toggle off
```

`reset`

Reset simulator to initial state.

Syntax: `reset`

What Gets Reset:

- Program counter to entry point
- All registers to initial values
- Memory to loaded state (ROM + initialized data)
- Execution history cleared
- All temporary state cleared

What Persists:

- Loaded program
- Architecture definition
- Breakpoints (optional: can clear manually)

Example:

```
CREATOR> run
Program execution completed.

CREATOR> reset
```

```
Program reset.

CREATOR> list
# Back at start
```

Breakpoint Management

`break [address|label]`

Set, remove, or list breakpoints.

Syntax:

- `break` - List all breakpoints
- `break <address>` - Toggle breakpoint at hex address
- `break <label>` - Toggle breakpoint at label
- `break <hex_number>` - Toggle at hex (without 0x)

Alias: Commonly `b`

Examples:

```
CREATOR> break                    # List breakpoints
CREATOR> break 0x400008           # Toggle at address
CREATOR> break main               # Toggle at label
CREATOR> break 10                 # Toggle at address 0x10
```

Output:

```
Breakpoint set at 0x00400008 (loop): addi t0,t0,1
# or
Breakpoint removed at 0x00400008 (loop): addi t0,t0,1
```

Label Resolution: - If input matches a label, uses label's address - Otherwise, treats as hex address (with or without 0x prefix)

Visual Indicators:

- - Red `*` in `list` output
- - Red highlighting of instruction line
- - Mentioned in breakpoint list

Inspection Commands

`list [limit]`

Display loaded instructions with addresses, labels, and code.

Syntax:

- `list` - Show all instructions
- `list <n>` - Show first N instructions
- `list --limit <n>` - Show first N instructions (explicit)

Output Format (When loading an Assembly Source):

```
   B | Address | Label    | Loaded Instruction    | User Instruction
  ---|---------|----------|-----------------------|--------------------
>    | 0x400000| main     | addi t0,zero,0x123    | li t0, 0x123
     | 0x400004|          | lui a0,0x10000        | la a0, myword
```

Output Format (When loading a Binary File):

```
   B | Address | Label    | Decoded Instruction   | Machine Code (hex)
  ---|---------|----------|-----------------------|--------------------
>    | 0x400000| main     | addi t0,zero,0x123    | 0x12300293
```

Columns:

- **B**: Breakpoint indicator (`*` if set)
- **>**: Current PC indicator (green)
- **Address**: Instruction memory address
- **Label**: Label at this address (if any)
- **Loaded/Decoded**: The actual instruction in memory
- **User/Machine**: Original source or machine code hex

Colors (in normal mode):

- Green: Current instruction (at PC)
- Yellow: Previously executed instruction
- Red: Instructions with breakpoints

`insn`

Display the current instruction at PC.

Syntax: `insn`

Output:

```
0x00400000 (0x00000293) addi t0,zero,0x123
```

Shows the instruction about to be executed.

`reg <registerBank|name> [format]`

Display register values.

Syntax:

- `reg list` - List available register banks
- `reg <registerBank>` - Show all registers of register bank
- `reg <registerBank> <format>` - Show registers in specific format
- `reg <name>` - Show specific register
- `reg <name> <format>` - Show specific register in format

Formats:

- - `raw` - Hexadecimal (default)
- - `decimal` or `dec` - Decimal (signed)
- - `number` - Same as decimal

Examples:

```
CREATOR> reg list
Register types:
  int_registers
  fp_registers

CREATOR> reg int_registers
int_registers:
zero(x0): 0x00000000  ra(x1): 0x00000000  sp(x2): 0x7FFFFFFC  gp(x3): 0x10008000
t0(x5):   0x00000123  t1(x6): 0x00000000  t2(x7): 0x00000000  fp(x8): 0x00000000
...

CREATOR> reg int_registers dec
```

```
int_registers:
zero(x0): 0        ra(x1): 0     sp(x2): 2147483644  gp(x3): 268468224
t0(x5):   291      t1(x6): 0     t2(x7): 0           fp(x8): 0
...

CREATOR> reg t0
t0: 0x00000123 | 291

CREATOR> reg pc
PC: 0x00400000 | 4194304
```

`mem <address> [count]`

Display memory contents with hints and annotations.

Syntax:

- - `mem <address>` - Show one word at address
- - `mem <address> <count>` - Show count bytes starting at address

Address Format:

- - Hexadecimal with 0x prefix: `0x10000000`
- - Hexadecimal without prefix: `10000000`

Examples:

```
CREATOR> mem 0x10000000
0x10000000: 0x12345678

CREATOR> mem 0x10000000 16
0x10000000: 0x12345678  // myvar:int (32b)
0x10000004: 0x00000064  // count:int (32b)
0x10000008: 0x00000001
0x1000000C: 0x00000002
```

Features:

- - **Hints**: Shows variable names and types if available
- - **Colors** (normal mode): Different colors for different hints
- - **Word-aligned**: Displays in architecture word sizes
- - **Annotations**: Comments show semantic information

Hints Format:

```
0x10000000: 0x12345678  // tag:type (size in bits) @+offset
```

Multiple Hints Per Word:

```
0x10000000: 0x12FF00AB  // a:char (8b) @+0, b:char (8b) @+1, c:short (16b) @+2
```

Each hint is color-coded differently.

`hexview <address> [count] [bytes_per_line]`

Display memory in hexdump format with ASCII representation.

Syntax:

- - `hexview <address>` - Default: 16 bytes, 16 per line
- - `hexview <address> <count>` - Specified bytes, 16 per line
- - `hexview <address> <count> <bytes_per_line>` - Custom layout

Examples:

```
CREATOR> hexview 0x10000000
0x10000000: 78 56 34 12 48 65 6C 6C  6F 20 57 6F 72 6C 64 00  |xV4.Hello World.|

CREATOR> hexview 0x10000000 32 8
0x10000000: 78 56 34 12 48 65 6C 6C  |xV4.Hell|
0x10000008: 6F 20 57 6F 72 6C 64 00  |o World.|
0x10000010: 00 00 00 00 00 00 00 00  |........|
0x10000018: 00 00 00 00 00 00 00 00  |........|
```

Format:

```
<address>: <hex bytes...>  |<ASCII>|
```

ASCII Column:

- - Printable characters shown as-is
- - Non-printable shown as `.` - Useful for finding strings in memory

---

`stack [max_bytes]`

Display call stack information and stack memory contents.

Syntax:

- - `stack` - Show stack with default limit (256 bytes)
- - `stack <max_bytes>` - Show up to specified bytes

Output Sections:

1. **Call Stack Hierarchy**:

```
Call Stack:
  > main (0x7FFFFFFC - 0x7FFFFFF0, 12 bytes)
    * calculate (0x7FFFFFF0 - 0x7FFFFFE4, 12 bytes)
      * helper (0x7FFFFFE4 - 0x7FFFFFDC, 8 bytes)
```

2. **Current Frame Details**:

```
Current Frame Details:
Function: helper
Frame: 0x7FFFFFE4 - 0x7FFFFFDC
Size: 8 bytes
Caller: calculate
Caller frame: 0x7FFFFFF0 - 0x7FFFFFE4
```

3. **Stack Memory Contents**:

```
Stack Memory Contents:
0x7FFFFFDC: 0x00000000  ← SP, helper frame start
0x7FFFFFE0: 0x00400020  // return address
0x7FFFFFE4: 0x00000005  ← helper frame end, calculate frame start
0x7FFFFFE8: 0x00400010  // saved ra
0x7FFFFFEC: 0x00000003  // local_var
0x7FFFFFF0: 0x00400000  ← calculate frame end, main frame start
```

Frame Colors (normal mode):

- - Each stack frame shown in different color
- - Current (top) frame in green
- - Older frames in cyan, magenta, blue, yellow

Annotations:

- - Frame boundaries marked
- - Stack pointer (SP) indicated
- - Variable names from hints
- - Return addresses identified

State Management

`snapshot [filename]`

Save complete simulator state to a JSON file.

Syntax:

- - `snapshot` - Auto-generate timestamped filename
- - `snapshot <filename>` - Save to specific file

Auto-Generated Format:

```
snapshot-2025-01-15T10-30-45-123Z.json
```

Saved State Includes:

- - All register values
- - All memory contents
- - Program counter
- - Execution state
- - Previous PC (for display)
- - Stack tracker state
- - Custom extra data

Example:

```
CREATOR> run 100
CREATOR> snapshot debug-state.json
Snapshot saved to debug-state.json

CREATOR> run
Program execution completed.

CREATOR> restore debug-state.json
State restored from debug-state.json

CREATOR> list
# Back at instruction 100
```

Use Cases:

- - Save progress before risky operations
- - Create checkpoints during debugging

- - Share reproducer for bugs
- - Test different execution paths from same state

`restore <filename>`

Restore simulator state from a snapshot file.

Syntax: `restore <filename>`

Requirements: - File must be valid snapshot JSON - Architecture must match

Behavior:

1. Resets current state
2. Loads snapshot from file
3. Restores all registers, memory, PC
4. Restores previous PC tracking
5. Ready to continue from restored point

Example:

```
CREATOR> restore checkpoint1.json
State restored from checkpoint1.json

CREATOR> reg pc
PC: 0x00400064 | 4194404

CREATOR> step
# Continue from restored state
```

Utility Commands

`clear`

Clear the terminal screen.

Syntax: `clear`

Effect: - Clears all terminal output - Prompt reappears at top - State unchanged

When to Use:

- - Declutter after lots of output
- - Fresh view for new debugging session
- - Before taking screenshot

`help`

Display all available commands with brief descriptions.

Syntax: `help`

Output:

- List of all commands
- Brief description of each
- Keyboard shortcuts (if enabled)
- Aliases (from configuration)

`about`

Display information about CREATOR.

Syntax: `about`

Shows:

- CREATOR version
- CLI version
- Runtime version (Deno/Node)
- Platform information
- Copyright and credits

`alias`

List defined command aliases.

Syntax: `alias`

Output:

```
Current command aliases:
  s   → step
  b   → break
  r   → run
  c   → continue
  reg → reg int_registers
```

Aliases can be defined in your config file at: */.config/creator/config.yml*

Note: Aliases are defined in Configuration file, not via command.

`quit`

Exit the simulator.

Syntax: `quit`

Behavior:

- Exits interactive mode
- Closes the simulator
- No confirmation prompt

Data Loss Warning:

- Unsaved state is lost
- Use `snapshot` before quitting to save progress

Command Aliases

Commands support aliasing via configuration file.

Common aliases are:

- `s` → `step`

- `b` → `break`
- `r` → `run`
- `c` → `continue`
- `u` → `unstep`

See Configuration for setting up aliases.

Keyboard Shortcuts

When `keyboard_shortcuts` is enabled in config, single key-presses can execute commands:

- `Ctrl+S` → `step` (example)
- `Ctrl+B` → `break` (example)
- `Ctrl+L` → `clear` (example)

See Configuration for setting up shortcuts.

Configuration

CREATOR supports extensive configuration through YAML files, allowing you to customize behavior, create command aliases, and define keyboard shortcuts.

Configuration File Location

Default location: `~/.config/creator/config.yml`

Custom location: Use `--config` option:

```
creator-cli --config ~/my-creator-config.yml program.s
```

If the configuration file doesn't exist, CREATOR creates it with default settings automatically.

Configuration Structure

The configuration file has three main sections:

```
settings:
  # General behavior settings
aliases:
```

```
  # Command aliases (shortcuts for long commands)
shortcuts:
  # Single-key keyboard shortcuts
```

Settings Section

Available Settings

max_states (number):

- - Maximum number of states saved for `unstep`/`nur` (reverse execution)
- - `0`: Reverse execution disabled
- - `-1`: Unlimited history (uses more memory)
- - Default: `100`

accessible (boolean):

- - Enable accessibility mode for screen readers - Provides verbose, structured output
- - Disables visual formatting
- - Default: `false`

keyboard_shortcuts (boolean):

- - Enable single-key shortcuts in interactive mode
- - Default: `true`

auto_list_after_shortcuts (boolean):

- - Automatically run `list` command after keyboard shortcuts
- - Helps see context after stepping
- - Default: `true`

Example Settings

```
settings:
  max_states: 50 # Keep last 50 states
  accessible: false # Standard visual mode
  keyboard_shortcuts: true # Enable shortcuts
  auto_list_after_shortcuts: false # Manual listing
```

Aliases Section

Create command shortcuts to save typing.

Syntax

```
aliases:
  alias_name: full_command
```

Built-in Examples

```
aliases:
  # Short names for common commands
  s: step
  r: run
  b: break

  # Register groups
  regs: reg int_registers
  fregs: reg fp_registers

  # Common inspections
  pc: reg pc
  sp: reg sp
  ra: reg ra

  # Memory dumps
  text: hexview 0x00400000 64 16
  data: hexview 0x10010000 64 16
  stack: mem sp 16
```

Custom Aliases

You can create aliases for any command with arguments:

```
aliases:
  # Debugging helpers
  showargs: reg a0 a1 a2 a3
  showtemps: reg t0 t1 t2 t3 t4 t5 t6

  # Quick breakpoints
  bmain: break main
```

```
  bloop: break main_loop

  # Memory views with hints
  datahint: mem 0x10010000 8 --hints

  # Step multiple times
  step5: step 5
  step10: step 10

  # Combined operations
  runshow: run; reg; stack
```

Using Aliases

Once defined, use them like regular commands:

```
creator> s        # Same as 'step'
creator> showargs # Same as 'reg a0 a1 a2 a3'
creator> datahint # Same as 'mem 0x10010000 8 --hints'
```

Shortcuts Section

Define single-key shortcuts for interactive mode.

Syntax

```
shortcuts:
   command: "key"
```

The key must be a single character (in quotes).

Default Shortcuts

```
shortcuts:
  step: " " # Spacebar
  run: "r"
  break: "b"
  snapshot: "s"
  unstep: "u"
  quit: "q"
  help: "h"
  list: "l"
```

Custom Shortcuts

```
shortcuts:
   # Execution control
   step: " " # Spacebar
   run: "r"
   continue: "c"
   reset: "x"

   # State inspection
   reg: "g" # 'g' for reGisters
   mem: "m"
   stack: "k"

   # Debugging
   break: "b"
   list: "l"
   unstep: "u"
   snapshot: "s"
   restore: "o" # 'o' for lOad

   # Utility
   help: "h"
   clear: "z"
   quit: "q"
```

Shortcut Behavior

- Press the key once to execute the command.
- If `auto_list_after_shortcuts` is enabled, instructions list automatically.
- Shortcuts work only in interactive mode.
- Disable with `keyboard_shortcuts: false`.

Complete Example Configuration

```
# ~/.config/creator/config.yml

settings:
  max_states: 50
  accessible: false
  keyboard_shortcuts: true
  auto_list_after_shortcuts: true
```

```
aliases:
  # Basic command shortcuts
  s: step
  r: run
  b: break
  c: continue
  # Register shortcuts
  pc: reg pc
  sp: reg sp
  ra: reg ra
  args: reg a0 a1 a2 a3 a4 a5 a6 a7
  temps: reg t0 t1 t2 t3 t4 t5 t6
  saved: reg s0 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11

  # Memory shortcuts
  text: hexview 0x00400000 64 16
  data: hexview 0x10010000 64 16
  stackview: mem sp 16
  stackhex: hexview sp 64 16

  # Debugging shortcuts
  here: insn
  where: reg pc
  showstate: reg; stack

  # Quick stepping
  s5: step 5
  s10: step 10
  u5: unstep 5

shortcuts:
  step: " "
  run: "r"
  break: "b"
  snapshot: "s"
  unstep: "u"
  quit: "q"
  help: "h"
  list: "l"
  clear: "z"
```

Configuration Loading

CREATOR loads configuration in this order:

1. **Default config**: Built-in settings
2. **User config**: `~/.config/creator/config.yml` (if exists)
3. **Custom config**: `--config` option (if specified)
4. **Command-line options**: Override config file settings

Teaching Resources

CREATOR is a powerful tool for teaching computer architecture and assembly programming. This document provides resources and guidance for educators looking to integrate CREATOR into their curriculum.

CREATOR's CLI also allows you to validate a program against an expected final state. This includes specifying the values of memory, registers (including floating point registers, within an error threshold), and the display buffer, and whether to error on calling convention errors (sentinel). See Validating Program Execution for more information.

CREATOR allows educators to create custom architectures tailored to their curriculum. This can be done by defining new architecture YAML files that specify instruction sets, registers, memory layout, and other architecture-specific details. See "Creating Custom Architectures" for more information.

Setting up the Remote Laboratory

The remote laboratory allows users to remotely execute their programs on real hardware through the use of multiple CREATOR Gateways.

Currently, only the ESP32 Gateway is supported.

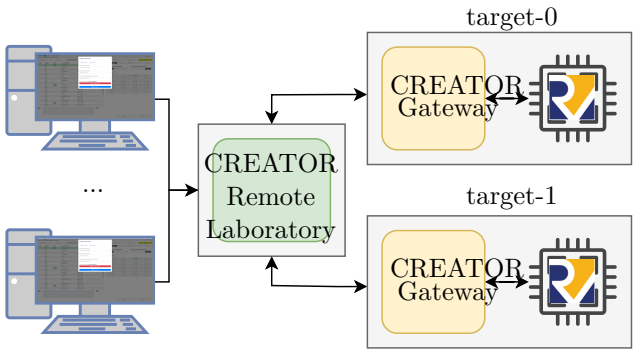

Figure 2: CREATOR Remote Laboratory diagram.

Setup and configuration

The recommended way of running the remote laboratory is through the Docker image.

For alternative ways of execution, check the source repository.

First, create a file structure such as this:

```
creator-remote-lab/
+-- compose.yaml
+-- results/
+-- config/
|   +-- deployment.json
```

On the remote lab container configuration: - `results/` must be mounted to `/app/results/` and `config/` to `/app/config/` - The server will listen on port `5000`, so we should expose that port - In order to send the results via email, you must set your Google Mail address in the `EMAIL` environment variable and your app password in the `PASSW` environment variable.

We should also deploy the gateways. More information on CREATOR Gateway.

We can do all this with Docker Compose, in the `compose.yaml` file:

```yaml
services:
  creator-remote-lab:
    image: creatorsim/creator-remote-lab:latest
    volumes:
      - ./config:/app/config
      - ./results:/app/results
    ports:
      - "5000:5000"
    environment:
      - EMAIL=test@arcos.inf.uc3m.es # google mail address
      - PASSW=abcdefghijklmnop # google app password

target-0:
    image: creatorsim/creator-gateway-esp32:latest
    devices:
      - /dev/ttyUSB0
    stdin_open: true
    tty: true

target-1:
    image: creatorsim/creator-gateway-esp32:latest
    devices:
      - /dev/ttyUSB1
    stdin_open: true
    tty: true
```

The configuration of the target boards is placed in `config/deployment.json`. You must provide the type of board (`target_board`), the USB port of the board (`target_port`), and the URL of the

gateway server (`target_url`). As we're doing everything with Docker Compose, we can just use their container names and prevent exposing the ports to `localhost`.

E.g.:

```
{
    "target-0": {
        "target_board": "esp32c3",
        "target_port": "/dev/ttyUSB0",
        "target_url": "http://target-0:8080"
    },
    "target-1": {
        "target_board": "esp32c3",
        "target_port": "/dev/ttyUSB1",
        "target_url": "http://target-1:8080"
    }
}
```

The execution results will be stored in the `results/` folder.

Then, we can deploy everything with:

```
docker compose up
```

CREATOR API (CAPI)

CAPI allows instruction definitions to interact with custom CREATOR functions.

`CAPI` is a global variable, so you can access it from your browser's developer console if you're using the web application.

Memory

Interaction with CREATOR's memory.

`CAPI.MEM.write`

```
CAPI.MEM.write(
  address: bigint,
  bytes: number,
  value: bigint,
  reg_name?: string,
  hint?: string,
```

```
  noSegFault: boolean = true,
): void { }
```

Writes a specific `value` to the specified `address` and number of `bytes`. You can provide extra information about which register used to hold the value (`reg_name`), or any hint about what that value might be (`hint`). To enable checking if the address is in a writable segment, set `noSegFault` to false.

E.g.:

```
CAPI.MEM.write(registers.t0, 1, registers.t0, "t0");
```

CAPI.MEM.read

```
CAPI.MEM.read(
  address: bigint,
  bytes: number,
  reg_name: string,
  noSegFault: boolean = true,
): bigint { }
```

Reads a specific number of `bytes` in the specified `address` and returns them. You can provide extra information about which register will hold the value (`reg_name`). To enable checking if the address is in a readable segment, set `noSegFault` to false.

E.g.:

```
registers.t0 = CAPI.MEM.read(registers.t1, 1, "t0");
```

CAPI.MEM.addHint

```
CAPI.MEM.addHint(address: bigint, hint: string, sizeInBits?: number): boolean { }
```

Adds a `hint` (description of what the address holds) for the specified memory `address`. If a hint already exists at the specified address, it replaces it. You can optionally specify the size of the stored type in bits (`sizeInBits`). Returns `true` if the hint was successfully added, else `false`.

E.g.:

```
CAPI.MEM.addHint(registers.f0, "float64", 64);
```

System calls

CREATOR's system calls.

CAPI.SYSCALL.exit

```
CAPI.SYSCALL.exit(): void { }
```

Terminates the execution of the program.

E.g.:

```
CAPI.SYSCALL.exit();
```

CAPI.SYSCALL.print

```
CAPI.SYSCALL.print(
  value: number | bigint,
  type: "int32" | "float" | "double" | "char" | "string",
): void { }
```

Prints the specified `value` of the specified `type` to the console.

Supported types are:

- - `"int32"`: signed 32-bit integer
- - `"float"` / `"double"`: JS's
- Number
- - `"char"`: UTF-16 character value
- - `"string"`: address of a null-terminated array of 1-byte chars

E.g.:

```
CAPI.SYSCALL.print(registers.a0, "char");
```

CAPI.SYSCALL.read

```
CAPI.SYSCALL.read(
  dest_reg_info: string,
  type: "int32" | "float" | "double" | "char" | "string",
  aux_info?: string,
): void { }
```

Reads the specified value `type` from the console and stores it in the specified register by name (`dest_reg_info`).

Supported types are:

- - `"int32"`: signed 32-bit integer
- - `"float"` / `"double"`: JS's
- Number
- - `"char"`: UTF-16 character value
- - `"string"`: `aux_info` is the name of the register that holds the address of a null-terminated array of 1-byte chars

E.g.:

```
CAPI.SYSCALL.read("a0", "char");
CAPI.SYSCALL.read("a0", "string", "a1");
```

`CAPI.SYSCALL.get_clk_cycles`

```
CAPI.SYSCALL.get_clk_cycles(): number { }
```

Returns the number of clock cycles that have passed since the program started.

E.g.:

```
CAPI.SYSCALL.get_clk_cycles();
```

Validation

`CAPI.SYSCALL.raise`

```
CAPI.SYSCALL.raise(msg: string): never { }
```

Raises an error with a specific `msg`.

E.g.:

```
CAPI.SYSCALL.raise("Help!");
```

`CAPI.SYSCALL.isOverflow`

```
CAPI.SYSCALL.isOverflow(op1: bigint, op2: bigint, res_u: bigint): boolean { }
```

Checks if the result `res_u` of operating two operands `op1` and `op2` caused an overflow.

E.g.:

```
CAPI.SYSCALL.isOverflow(registers.t0, registers.t1, registers.t0 + registers.t1);
```

Stack

These functions are used for the stack tracker and sentinel modules, and are a way to tell CREATOR when a new function frame begins and ends. They should be included in the instructions that jump to, o return from, a routine, as is the case of RISC-V's `jal` and `jr` instructions.

`CAPI.STACK.beginFrame`

```
CAPI.STACK.beginFrame(addr?: bigint): void { }
```

Marks the beginning of a function frame at `address`. If not specified, it takes the current (real) value of the program counter.

E.g.:

```
registers.pc = ...
CAPI.SYSCALL.beginFrame();
```

`CAPI.STACK.endFrame`

```
CAPI.STACK.endFrame(): void { }
```

Ends the current stack frame.

E.g.:

```
registers.pc = ...
CAPI.SYSCALL.endFrame();
```

Floating point

`CAPI.FP.split_double`

```
CAPI.FP.split_double(reg: bigint, index: 0 | 1): string { }
```

Given a double precision IEEE 754 value `reg`, gets the 32-bits most significant (`index=1`) bits or the least significant bits (`index=0`). It returns it as a string of bits.

E.g.:

```
const foo = CAPI.FP.split_double(registers.t0, 1);
```

CAPI.FP.uint2float32

```
CAPI.FP.uint2float32(value: number): number { }
```

Transforms the unsigned integer `value` to a single precision IEEE754 JS Number.

E.g.:

```
CAPI.FP.uint2float32(5);
```

CAPI.FP.float322uint

```
CAPI.FP.float322uint(value: number): bigint { }
```

Transforms the single precision floating point (JS Number) `value` into an unsigned integer.

E.g.:

```
CAPI.FP.uint2float32(5.0);
```

CAPI.FP.int2uint

```
CAPI.FP.int2uint(
  value: number | bigint,
  bits: number = 64,
): bigint { }
```

Transforms a signed integer (JS Number) `value` into an unsigned integer of `bits` number of bits.

E.g.:

```
CAPI.FP.int2uint(-5);
```

CAPI.FP.uint2int

```
CAPI.FP.uint2int(value: number | bigint): bigint { }
```

Transforms an unsigned integer `value` into a signed integer (JS Number) of `bits` number of bits.

E.g.:

```
CAPI.FP.uint2int(5);
```

CAPI.FP.uint2float64

```
CAPI.FP.uint2float64(value0: number | bigint, value1?: number): number { }
```

Transforms an unsigned integer `value0` into a signed integer to a 64-bit float (JS Number). Supports two calling conventions: 1. Single `value0` argument: converts a 64-bit integer directly 2. Two 32-bit arguments: converts low (`value0`) and high (`value1`) 32-bit parts.

E.g.:

```
CAPI.FP.uint2float64(5);
```

CAPI.FP.float642uint

```
CAPI.FP.float642uint(value: number): [number, number] { }
```

Transforms the double precision floating point (JS Number) `value` into an unsigned integer.

E.g.:

```
CAPI.FP.float642uint(5.0);
```

CAPI.FP.check_ieee

```
CAPI.FP.check_ieee(sign: string, exponent: string, mantissa: string): number { }
```

Check the type of number is in IEEE 754 format. Returns a 10-bit mask where the position of the set bit indicates the type of the IEEE 754 number:

- - 0 -> -inf
- - 1 -> -normalized number
- - 2 -> -non-normalized number
- - 3 -> -0
- - 4 -> +0
- - 5 -> +non-normalized number
- - 6 -> +normalized number
- - 7 -> +inf
- - 8 -> signaling NaN
- - 9 -> quiet NaN

E.g.:

```
CAPI.FP.check_iee(parseInt(a[0]), parseInt(a.slice(1,9), 2), parseInt(a.slice(10), 2));
```

`CAPI.FP.float2bin`

```
CAPI.FP.float2bin(f: number): string { }
```

Transforms the single precision floating point JS Number `f` into a binary string.

E.g.:

```
CAPI.FP.float2bin(5.0);
```

Registers

`CAPI.REG.read`

```
CAPI.REG.read(name: string): bigint { }
```

Returns the value stored in register `name`.

E.g.:

```
const foo = CAPI.REG.read("t0");
```

`CAPI.REG.write`

```
CAPI.REG.write(value: bigint, name: string): void { }
```

Stores `value` in register `name`.

E.g.:

```
CAPI.REG.write(0x69n, "t0");
```

Architecture

`CAPI.ARCH` exposes the interface of the currently loaded architecture plugin.

The supported plugins are `riscv`, `mips` and `z80`.

RISC-V

The RISC-V architecture plugin provides utilities for handling floating point operations and immediate value generation.

generateLoadImmediate

```
CAPI.ARCH.generateLoadImmediate(val: bigint | number, destReg: string): string { }
```

Generates a sequence of RISC-V instructions to load an immediate value into a register. Returns a semicolon-separated string of instructions that can load values of any size into the destination register.

E.g.:

```
const instructions = CAPI.ARCH.generateLoadImmediate(0x12345678n, "t0");
// Returns: "lui t0, 0x12345;addiw t0, t0, 1656"
```

toJSNumberD

```
CAPI.ARCH.toJSNumberD(bigIntValue: bigint): [number, string] { }
```

Converts a 64-bit register value to a JavaScript number when the D (double-precision floating point) extension is enabled. Returns a tuple containing the numeric value and a type string indicating the representation.

The function handles:

- - Canonical 64-bit NaN (`0x7ff8000000000000n`) → `[NaN, "NaN64"]`
- - NaN-boxed 32-bit floats (upper 32 bits all 1's) → `[value, "NaNBfloat32_64"]`
- - Valid 64-bit doubles → `[value, "float64"]` - Zero → `[0, "float32"]`

E.g.:

```
let value, type;
[value, type] = CAPI.ARCH.toJSNumberD(registers.ft0);
```

toJSNumberS

```
CAPI.ARCH.toJSNumberS(bigIntValue: bigint): [number, string] { }
```

Converts a 32-bit register value to a JavaScript number when only the S (single-precision floating point) extension is enabled. Returns a tuple containing the numeric value and a type string.

The function handles: - Canonical 32-bit NaN (`0x7fc00000n`) → `[NaN, "NaN32"]` - Valid 32-bit floats → `[value, "float32"]` - Zero → `[0, "float32"]`

E.g.:

```
let value, type;
[value, type] = CAPI.ARCH.toJSNumberS(registers.ft0);
```

toBigInt

```
CAPI.ARCH.toBigInt(number: number, type: string): bigint { }
```

Converts a JavaScript number to a bigint representation suitable for storing in a RISC-V register. The `type` parameter specifies the format.

Supported types:

- - `"float32"`: Single-precision float
- - `"float64"`: Double-precision float
- - `"NaNBfloat32_64"`: NaN-boxed single-precision (upper 32 bits all 1's)
- - `"NaN64"`: Canonical 64-bit NaN
- - `"NaN32"`: Canonical 32-bit NaN

E.g.:

```
registers.ft0 = CAPI.ARCH.toBigInt(3.14, "float32");
```

NaNBox

```
CAPI.ARCH.NaNBox(bigIntValue: bigint): bigint { }
```

NaN-boxes a 32-bit floating point value for use in 64-bit floating point registers. Sets the upper 32 bits to all 1's as per the RISC-V specification (section 21.2).

E.g.:

```
const nanBoxed = CAPI.ARCH.NaNBox(registers.ft0);
```

MIPS

The MIPS architecture plugin provides utilities for handling double-precision floating point operations across register pairs.

validateEvenRegister

```
CAPI.ARCH.validateEvenRegister(regName: string): void { }
```

Validates that a register number is even. Required for double-precision operations in MIPS, which use register pairs. Throws an error if the register is not even.

E.g.:

```
CAPI.ARCH.validateEvenRegister("f0");  // OK
CAPI.ARCH.validateEvenRegister("f1");  // Throws error
```

readDouble

```
CAPI.ARCH.readDouble(regName: string): number { }
```

Reads a double-precision floating point value from a MIPS register pair. The `regName` must be even (e.g., `"f0"`, `"f2"`). The function reads from the specified register and the next sequential register to reconstruct a 64-bit double.

E.g.:

```
const value = CAPI.ARCH.readDouble("f0");  // Reads f0 and f1
```

writeDouble

```
CAPI.ARCH.writeDouble(value: number, regName: string): void { }
```

Writes a double-precision floating point value to a MIPS register pair. The `regName` must be even. The function splits the value across the specified register and the next sequential register.

E.g.:

```
CAPI.ARCH.writeDouble(3.14159, "f0");  // Writes to f0 and f1
```

readDoublePair

```
CAPI.ARCH.readDoublePair(reg1Name: string, reg2Name: string): number[] { }
```

Reads two double-precision values from register pairs and returns them as an array. Both register names must be even.

E.g.:

```
const [val1, val2] = CAPI.ARCH.readDoublePair("f0", "f2");
```

writeDoublePair

```
CAPI.ARCH.writeDoublePair(value1: number, value2: number, reg1Name: string, reg2Name: string): voi
```

Writes two double-precision values to register pairs. Both register names must be even.

E.g.:

```
CAPI.ARCH.writeDoublePair(1.5, 2.5, "f0", "f2");
```

binaryDoubleOperation

```
CAPI.ARCH.binaryDoubleOperation(
  destReg: string,
  src1Reg: string,
  src2Reg: string,
  operation: (val1: number, val2: number) => number
): void { }
```

Performs a binary operation on two double-precision values. Reads from source register pairs, applies the operation function, and writes the result to the destination register pair. All register names must be even.

E.g.:

```
CAPI.ARCH.binaryDoubleOperation("f0", "f2", "f4", (a, b) => a + b);
```

unaryDoubleOperation

```
CAPI.ARCH.unaryDoubleOperation(
  destReg: string,
  srcReg: string,
  operation: (val: number) => number
): void { }
```

Performs a unary operation on a double-precision value. Reads from the source register pair, applies the operation function, and writes the result to the destination register pair. Both register names must be even.

E.g.:

```
CAPI.ARCH.unaryDoubleOperation("f0", "f2", (x) => Math.sqrt(x));
```

Z80

The Z80 architecture plugin provides utilities for flag calculations, keyboard handling, and I/O operations.

Flag Constants The following flag bit masks are available:

```
CAPI.ARCH.S_FLAG   // 0x80 - Sign flag
CAPI.ARCH.Z_FLAG   // 0x40 - Zero flag
CAPI.ARCH.H_FLAG   // 0x10 - Half-carry flag
```

```
CAPI.ARCH.PV_FLAG  // 0x04 - Parity/Overflow flag
CAPI.ARCH.N_FLAG   // 0x02 - Add/Subtract flag
CAPI.ARCH.C_FLAG   // 0x01 - Carry flag
```

State Variables The following variables track the Z80 system state:

```
CAPI.ARCH.borderColor     // Border color (0-7) for ZX Spectrum
CAPI.ARCH.interruptMode   // Current interrupt mode (0, 1, or 2)
CAPI.ARCH.interruptPin    // Interrupt pin state (0 = low, 1 = high)
CAPI.ARCH.timerCounter    // Timer counter value (bigint)
CAPI.ARCH.keyState        // Object mapping event.code strings to pressed state (boolean)
CAPI.ARCH.keyMap          // Keyboard matrix mapping port high bytes to key codes
```

E.g.:

```
// Set interrupt mode 2
CAPI.ARCH.interruptMode = 2;

// Check if a key is pressed
if (CAPI.ARCH.keyState["KeyA"]) {
  // Key A is pressed
}

// Set border color to red
CAPI.ARCH.borderColor = 2n;
```

calculateFlags_INC

```
CAPI.ARCH.calculateFlags_INC(oldValue: bigint, initialF: bigint): bigint { }
```

Calculates flags for an 8-bit INC (increment) operation. Returns the new F register value with appropriate flags set. The C flag is preserved from `initialF`.

E.g.:

```
registers.F = CAPI.ARCH.calculateFlags_INC(registers.A, registers.F);
```

calculateFlags_DEC

```
CAPI.ARCH.calculateFlags_DEC(oldValue: bigint, initialF: bigint): bigint { }
```

Calculates flags for an 8-bit DEC (decrement) operation. Returns the new F register value. The C flag is preserved, and N flag is always set.

E.g.:

```
registers.F = CAPI.ARCH.calculateFlags_DEC(registers.A, registers.F);
```

calculateFlags_ADD

```
CAPI.ARCH.calculateFlags_ADD(val1: bigint, val2: bigint): bigint { }
```

Calculates flags for 8-bit addition (ADD). Returns the new F register value with all flags computed based on the operation.

E.g.:

```
registers.F = CAPI.ARCH.calculateFlags_ADD(registers.A, registers.B);
```

calculateFlags_ADC

```
CAPI.ARCH.calculateFlags_ADC(val1: bigint, val2: bigint, initialF: bigint): bigint { }
```

Calculates flags for 8-bit addition with carry (ADC). The carry bit is extracted from `initialF`.

E.g.:

```
registers.F = CAPI.ARCH.calculateFlags_ADC(registers.A, registers.B, registers.F);
```

calculateFlags_SUB

```
CAPI.ARCH.calculateFlags_SUB(val1: bigint, val2: bigint): bigint { }
```

Calculates flags for 8-bit subtraction (SUB). The N flag is always set for subtraction operations.

E.g.:

```
registers.F = CAPI.ARCH.calculateFlags_SUB(registers.A, registers.B);
```

calculateFlags_SBC

```
CAPI.ARCH.calculateFlags_SBC(val1: bigint, val2: bigint, initialF: bigint): bigint { }
```

Calculates flags for 8-bit subtraction with carry (SBC). The carry bit is extracted from `initialF`.

E.g.:

```
registers.F = CAPI.ARCH.calculateFlags_SBC(registers.A, registers.B, registers.F);
```

calculateFlags_LOGICAL

```
CAPI.ARCH.calculateFlags_LOGICAL(result: bigint, setH: number): bigint { }
```

Calculates flags for logical operations (AND, OR, XOR). Set `setH` to `1` for AND operations (sets H flag), or `0` for OR/XOR operations (resets H flag). N and C flags are always reset.

E.g.:

```
registers.F = CAPI.ARCH.calculateFlags_LOGICAL(registers.A & registers.B, 1);
```

calculateFlags_CP

```
CAPI.ARCH.calculateFlags_CP(val1: bigint, val2: bigint): bigint { }
```

Calculates flags for the compare (CP) operation. Performs a subtraction for flag calculation purposes without storing the result.

E.g.:

```
registers.F = CAPI.ARCH.calculateFlags_CP(registers.A, registers.B);
```

calculateFlags_ADD16

```
CAPI.ARCH.calculateFlags_ADD16(val1: bigint, val2: bigint, initialF: bigint): bigint { }
```

Calculates flags for 16-bit addition (ADD HL, rr). The S, Z, and P/V flags are preserved from `initialF`. N flag is reset.

E.g.:

```
registers.F = CAPI.ARCH.calculateFlags_ADD16(registers.HL, registers.BC, registers.F);
```

calculateFlags_SBC16

```
CAPI.ARCH.calculateFlags_SBC16(val1: bigint, val2: bigint, initialF: bigint): bigint { }
```

Calculates flags for 16-bit subtraction with carry (SBC HL, rr). The carry bit is extracted from `initialF`.

E.g.:

```
registers.F = CAPI.ARCH.calculateFlags_SBC16(registers.HL, registers.BC, registers.F);
```

calculateFlags_BIT

```
CAPI.ARCH.calculateFlags_BIT(value: bigint, bit: number, initialF: bigint): bigint { }
```

Calculates flags for the BIT instruction, which tests a specific bit (0-7) in a value. The C flag is preserved from `initialF`, and H flag is always set. N flag is always reset.

E.g.:

```
registers.F = CAPI.ARCH.calculateFlags_BIT(registers.A, 7, registers.F);
```

calculateFlags_ROTATE

```
CAPI.ARCH.calculateFlags_ROTATE(result: bigint, carry: bigint): bigint { }
```

Calculates flags for rotate and shift instructions (RLC, RRC, SLA, etc.). The `carry` parameter should be `0n` or `1n` representing the bit that was shifted out. H and N flags are always reset.

E.g.:

```
const carry = registers.A >> 7n;
const result = ((registers.A << 1n) | carry) & 0xffn;
registers.F = CAPI.ARCH.calculateFlags_ROTATE(result, carry);
```

calculateFlags_SRL

```
CAPI.ARCH.calculateFlags_SRL(result: bigint, carry: bigint): bigint { }
```

Calculates flags for the SRL (Shift Right Logical) instruction. The `carry` parameter is the original bit 0 that was shifted out. S, H, and N flags are always reset.

E.g.:

```
const carry = registers.A & 1n;
const result = registers.A >> 1n;
registers.F = CAPI.ARCH.calculateFlags_SRL(result, carry);
```

pressKey

```
CAPI.ARCH.pressKey(code: string): void { }
```

Registers a keyboard key as being pressed. The `code` parameter should be a JavaScript `event.code` string (e.g., `"KeyA"`, `"Digit1"`).

E.g.:

```
CAPI.ARCH.pressKey("KeyA");
```

releaseKey

```
CAPI.ARCH.releaseKey(code: string): void { }
```

Registers a keyboard key as being released.

E.g.:

```
CAPI.ARCH.releaseKey("KeyA");
```

readULAKeyboard

```
CAPI.ARCH.readULAKeyboard(port: bigint): bigint { }
```

Emulates a read from the ZX Spectrum ULA keyboard port. The high byte of the port address determines which half-row of keys to poll. Returns an 8-bit value where bits 0-4 represent key states (0 = pressed, 1 = not pressed), and bits 5-7 are typically high.

E.g.:

```
const keyState = CAPI.ARCH.readULAKeyboard(0xf7fen);  // Read row with keys 1-5
```

read

```
CAPI.ARCH.read(port: bigint): bigint { }
```

Reads a byte from an I/O port. Handles: - Port `0x01`: Keyboard buffer (non-blocking) - ULA ports (keyboard matrix): ZX Spectrum keyboard - Unhandled ports: Returns `0xFF`

E.g.:

```
const value = CAPI.ARCH.read(0x01n);  // Read from keyboard buffer
```

write

```
CAPI.ARCH.write(port: bigint, value: bigint): void { }
```

Writes a byte to an I/O port. Handles:

- - Port `0x02`: Screen output
- - Port `0xFE`: ZX Spectrum ULA (sets border color from bits 0-2)
- - Port `0x17`: CREATOR-specific port for ecall (exits on value `10n`)

E.g.:

```
CAPI.ARCH.write(0x02n, 65n);  // Write 'A' to screen
CAPI.ARCH.write(0xfen, 0x02n);  // Set border color to red
```

Interrupts

Functions to manage interrupts and privilege modes.

For more information, see Interrupt Support.

CAPI.INTERRUPTS.setUserMode

```
CAPI.INTERRUPTS.setUserMode(): void { }
```

Sets the privilege level to User.

E.g.:

```
CAPI.INTERRUPTS.setUserMode();
```

CAPI.INTERRUPTS.setKernelMode

```
CAPI.INTERRUPTS.setKernelMode(): void { }
```

Sets the privilege level to Kernel.

E.g.:

```
CAPI.INTERRUPTS.setKernelMode();
```

CAPI.INTERRUPTS.create

```
CAPI.INTERRUPTS.create(type: InterruptType): void { }
```

Creates an interrupt of the specified `type`.

E.g.:

```
CAPI.INTERRUPTS.create(InterruptType.Software);
```

CAPI.INTERRUPTS.enable

```
CAPI.INTERRUPTS.enable(type: InterruptType): void { }
```

Enables an interrupt `type`.

E.g.:

```
CAPI.INTERRUPTS.enable(InterruptType.Software);
```

CAPI.INTERRUPTS.globalEnable

```
CAPI.INTERRUPTS.globalEnable(): void { }
```

Globally enables interrupts.

E.g.:

```
CAPI.INTERRUPTS.globalEnable();
```

CAPI.INTERRUPTS.disable

```
CAPI.INTERRUPTS.disable(type: InterruptType): void { }
```

Disables an interrupt `type`.

E.g.:

```
CAPI.INTERRUPTS.disable(InterruptType.Software);
```

CAPI.INTERRUPTS.globalDisable

```
CAPI.INTERRUPTS.globalDisable(): void { }
```

Globally disables interrupts.

E.g.:

```
CAPI.INTERRUPTS.globalDisable();
```

CAPI.INTERRUPTS.isEnabled

```
CAPI.INTERRUPTS.isEnabled(type: InterruptType): boolean { }
```

Checks if an interrupt `type` is enabled.

E.g.:

```
CAPI.INTERRUPTS.isEnabled(InterruptType.Software);
```

CAPI.INTERRUPTS.isGlobalEnabled

```
CAPI.INTERRUPTS.isGlobalEnabled(): boolean { }
```

Checks if interrupts are globally enabled.

E.g.:

```
CAPI.INTERRUPTS.isGlobalEnabled();
```

CAPI.INTERRUPTS.clear

```
CAPI.INTERRUPTS.clear(type: InterruptType): void { }
```

Clears interrupts of the specified `type`.

E.g.:

```
CAPI.INTERRUPTS.clear(InterruptType.Software);
```

CAPI.INTERRUPTS.globalClear

```
CAPI.INTERRUPTS.globalClear(): void { }
```

Clears all interrupts.

E.g.:

```
CAPI.INTERRUPTS.globalClear();
```

CAPI.INTERRUPTS.setCustomHandler

```
CAPI.INTERRUPTS.setCustomHandler(): void { }
```

Sets the interrupt handler to the custom handler.

E.g.:

```
CAPI.INTERRUPTS.setCustomHandler();
```

CAPI.INTERRUPTS.setCREATORHandler

```
CAPI.INTERRUPTS.setCREATORHandler(): void { }
```

Sets the interrupt handler to the default CREATOR handler.

E.g.:

```
CAPI.INTERRUPTS.setCREATORHandler();
```

CAPI.INTERRUPTS.setHighlight

```
CAPI.INTERRUPTS.setHighlight(): void { }
```

Highlights the current instruction as "interrupted" in the UI.

E.g.:

```
CAPI.INTERRUPTS.setHighlight();
```

CAPI.INTERRUPTS.clearHighlight

```
CAPI.INTERRUPTS.setHighlight(): void { }
```

Removes the "interrupted" highlight in the UI. Typically used in instructions that return from the interrupt handler, such as RISC-V's `mret`.

E.g.:

```
CAPI.INTERRUPTS.clearHighlight();
```

Creating Custom Architectures

CREATOR supports defining custom architectures through YAML configuration files. This allows adding new instruction sets or modifying existing ones.

Creating an Architecture File

An architecture file is a YAML file that describes the architecture's properties, including its instruction set, registers, memory layout, and other relevant details. Instructions are defined with their binary encoding, assembly syntax, and semantics.

We provide a JSON schema for the architecure file at https://creatorsim.github.io/creator/schema/architecture.json.

The actual definition for an instruction is a simple JavaScript code block to manipulate the simulator state. Within this block, you have the `registers` variable to access the registers (e.g. `registers.PC`, or `registers[value]`), as well as `CAPI`, an API that allows you to interact with the simulator.

The values stored in the registers are BigInt. Take that into account when reading or writing values:

```
const foo = registers.PC; // 420n
registers.PC = foo + 1n;  // 421n
```

Let's define a simple 8-bit architecture with a few instructions.

The first step is to create a YAML file, e.g., `simplearch.yml`, and fill out the `config`. We want an architecture where the word size and byte size are both 8 bits. We'll also make it little-endian, although it doesn't matter in this specific case because a word contains only one byte. `pc_offset` will be `0`. The entry point will be a function named `main`, or address `0x0` if it doesn't exist; we'll use the `;` character to write comments, and the names of the registers won't be sensitive (`PC` == `pc`). We'll also enable memory alignment and passing convention checks.

The value of the program counter register (`program_counter`) inside the instruction definitions is affected by the `pc_offset`.
`pc_offset` is the offset that we'll add to the value of program counter the instruction "sees".
E.g. if `pc_offset` is `-4`, and we're executing an instruction at `0x0`, the "real" PC is `0x4` (because of the fetch performed at the start of the cycle), but the value of `registers.PC` (the "virtual" PC) will be `0x0`.

```yaml
version: 2.0.0
config:
  name: Simple8Bit
  description: A simple custom 8-bit architecture
  word_size: 8
  byte_size: 8
  endianness: little_endian
  pc_offset: 0
  main_function: main
  start_address: 0x0
  comment_prefix: ;
  sensitive_register_name: false
  memory_alignment: true
  passing_convention: true
```

For the registers, we'll make a control register bank with a `PC` program counter register (we'll mark that with the `program_counter` property), and another integer register bank with a `A` and `B` register, as well as a `SP` stack pointer register (`stack_pointer` property).

A floating point bank would be defined as:

```
# ...
- name: Floating point registers
  type: fp_registers
  double_precision: true # or 'false', if single-precision
```

All of these registers will be 8 bits, be initialized (`value`) and have a default value (`default_value`) of `0`, and will be both readable (`read` property) and writable (`write` property).

The `encoding` property will be used when decoding binary instructions, in this case we'll just make it sequential.
`name` is a list because a register can have multiple values, e.g. in RISC-V register `zero` can be also called `x0`, and so on. These names must all be unique.

```yaml
components:
  - name: Control registers
    type: ctrl_registers
    double_precision: false
    elements:
      - name:
          - PC
        nbits: 8
        encoding: 0
        value: 0
        default_value: 0
        properties:
          - read
          - write
          - program_counter
  - name: Integer registers
    type: int_registers
    double_precision: false
    elements:
      - name:
          - A
        encoding: 0
        nbits: 8
        value: 0
        default_value: 0
        properties:
          - read
```

```
        - write
    - name:
        - B
      encoding: 1
      nbits: 8
      value: 0
      default_value: 0
      properties:
        - read
        - write
    - name:
        - SP
      encoding: 2
      nbits: 8
      value: 0
      default_value: 0
      properties:
        - read
        - write
        - stack_pointer
```

For the memory layout, we'll use a simple `.text`, `.data`, `.stack` layout:

```
memory_layout:
  text:
    start: 0x0000
    end: 0x03FF
  data:
    start: 0x0400
    end: 0x7FFF
  stack:
    start: 0x8000
    end: 0xFFFF
```

Now it's time for the instructions. We want an architecture with two instructions: `NOP` and `ADD` in the `base` extension. The `NOP` instruction does nothing, while the `ADD` instruction adds the values of two registers and stores the result in a destination register.

As the instructions will have the same form, we can define an instruction template called `standard` defining that our instructions will be 1 word long (`nwords`), take 1 clock cycle (`clk_cycles`) and use the full word as an operation code (type `co`) field. We then override in each instruction the value of that field.

```
templates:
```

```yaml
  - name: standard
    nwords: 1
    clk_cycles: 1
    fields:
      - name: opcode
        type: co
        startbit: 7
        stopbit: 0
        order: 0

instructions:
  base:
    - name: nop
      template: standard
      fields:
        - field: opcode
          value: "0x00"
      definition: ""

    - name: add
      template: standard
      fields:
        - field: opcode
          value: "0x80"
      definition: |
        const oldValueA = registers.A;
        registers.A = (oldValueA + registers.B) & 0xFFn;
        registers.F = CAPI.ARCH.calculateFlags_ADD(oldValueA, registers.B);

directives:
  - name: .data
    action: data_segment
    size: null
  - name: .text
    action: code_segment
    size: null
  - name: .bss
    action: global_symbol
    size: null
  - name: .zero
    action: space
    size: 1
```

```yaml
- name: .space
  action: space
  size: 1
- name: .align
  action: align
  size: null
- name: .balign
  action: balign
  size: null
- name: .globl
  action: global_symbol
  size: null
- name: .string
  action: ascii_null_end
  size: null
- name: .asciz
  action: ascii_null_end
  size: null
- name: .ascii
  action: ascii_not_null_end
  size: null
- name: .byte
  action: byte
  size: 1
- name: .half
  action: half_word
  size: 2
- name: .word
  action: word
  size: 4
- name: .dword
  action: double_word
  size: 8
- name: .float
  action: float
  size: 4
- name: .double
  action: double
  size: 8
```

Plugins

Interrupt Support

Now, let's take our architecture and add support for some simple maskable and nonmaskable interrupts.

Custom handler

We'll define two new 1-bit integer registers `MIP` (*Maskable Interrupt Pending*) and `NIP` (*Nonmaskable Interrupt Pending*) that will be set to `1` when an interrupt of the type is pending. We'll also define another 1-bit integer register `IE` to enable (value of `1`) and disable (value of `0`) maskable interrupts.

We just need to add them to `simplearch.yml`:

```yaml
components:
  # ...
  - name: Integer registers
    # ...
    elements:
      # ...
      - name:
          - MIP
        encoding: 2
        nbits: 1
        value: 0
        default_value: 0
        properties:
          - read
          - write
      - name:
          - NIP
        encoding: 3
        nbits: 1
        value: 0
        default_value: 0
        properties:
          - read
          - write
      - name:
          - IE
        encoding: 4
        nbits: 1
        value: 1
```

```
      default_value: 1
      properties:
        - read
        - write
```

Now we have to define some functions to determine how interrupts work in this architecture.

First, we need to define how to determine if an interrupt happened. CREATOR has some predefined types of interrupts, and here we'll use `InterruptType.Maskable` and `InterruptType.Nonmaskable`. We have to write a function that returns the type of interrupt (`InterruptType`), or `null` if there is no interrupt:

You don't have to check if interrupts are enabled here, we'll define that later.

```
interrupts:
  check: |
    if (registers.NIP) return InterruptType.Nonmaskable;
    if (registers.MIP) return InterruptType.Maskable;
    return null;
```

Then, we must define how different types of interrupts can be created and cleared. We'll receive the desired type (`InterruptType`) inside the `type` variable:

```
interrupts:
  # ...
  create: |
    switch (type) {
      case InterruptType.Maskable:
        registers.MIP = 1n;
        break;
      case InterruptType.Nonmaskable:
        registers.NIP = 1n;
        break;
    }

  clear: |
    switch (type) {
      case InterruptType.Maskable:
        registers.MIP = 0n;
        break;
      case InterruptType.Nonmaskable:
        registers.NIP = 0n;
```

```
        break;
    }

  global_clear: |
    registers.MIP = 0n;
    registers.NIP = 0n;
```

`clear` is optional, it gets overriten by `global_clear` if it's not defined

Next, how they can be enabled and disabled, per type (and globally), as well as how to check if they are enabled. For the sake of simplicity, we'll assume nonmaskable interrupts can't be disabled.

```
# ...
interrupts:
  # ...
  is_enabled: |
    switch (type) {
      case InterruptType.Maskable:
        return registers.MIE === 1n;
      case InterruptType.Nonmaskable:
        // nonmaskable are always enabled
        return true;
    }
    return false;

  is_global_enabled: |
    return true;

  enable: |
    switch (type) {
      case InterruptType.Maskable:
        return registers.MIE = 1n;
        break;
      // we don't need to do anything for nonmaskable
    }

  disable: |
    switch (type) {
      case InterruptType.Maskable:
        registers.IE = 0n;
        break;
```

```
      // can't disable nonmaskable
    }

  global_enable: |
    registers.IE = 1n;

  global_disable: |
    registers.IE = 0n;
```

`enable` and `disable` are optional, they gets overriten by `global_` counterparts if it's not defined. `is_global_enabled` is optional, and defaults to `return true`.

Finally, we define the custom interrupt handler. This handler will disable interrupts, store `PC` in the stack, and jump to `0x0`. To disable interrupts, we do it "manually" by clearing `IE`, but we can also use CAPI to reuse the code we already defined.

Using these CAPI functions is recommended way of doing it, as it allows the application to (secretly) keep track of these interrupts.

```
interrupts:
  handlers:
    custom: |
      // disable interrupt
      CAPI.INTERRUPTS.disable(type);

      // store PC in stack
      registers.SP = (registers.SP - 1n) & 0xFFn;
      CAPI.MEM.write(registers.SP, 1, registers.PC);

      // jump to handler
      registers.PC = 0n;
```

Many architectures have a specific instruction to return from an interrupt, so let's make one, `reti`. This instruction will clear and enable interrupts and jump back to the address stored in the stack:

```
instructions:
  base:
    # ...
    - name: reti
      template: standard
      fields:
```

```
      - field: opcode
        value: "0x01"
    definition: |
      // enable interrupts
      CAPI.INTERRUPTS.globalEnable();

      // pop return address from stack
      registers.PC = CAPI.MEM.read(registers.SP, 1);
      registers.SP = (registers.SP + 1n) & 0xFFn;

      // notify UI that handler has finished
      CAPI.INTERRUPTS.clearHighlight();
```

CREATOR handler

As we mentioned in Interrupt Handling, CREATOR has two different interrupt handlers: the default "CREATOR" one, and a custom architecture-defined one. We also mentioned that the default handler is able to handle "architecture-defined system calls". Let's see a more concrete example, by implemening them in our architecture.

Why would we want this? Because we want to have our cake and eat it too.
Before interrupts were added to CREATOR, the definition of RISC-V's `ecall` function didn't create an interrupt, it just executed the desired system call depending on the value of register `a7`. But we wanted to have "real" interrupts and a "real" `ecall`. The problem is that this required an interrupt handler, and we didn't want to force our users to use it, we didn't want to silently include a kernel, and we didn't want to have two architectures: one with interrupts and one without.
So the solution (compromise) we found was this one, a second (default) interrupt handler that can be programmed in JS.

These system calls will generate a new type of interrupt (`InterruptType.EnvironmentCall`), so let's quickly modify the architecture to take them into account. We'll also add a new `EIP` register to signal that that type of interrupt is pending.

```
components:
  # ...
  - name: Integer registers
    # ...
    elements:
      # ...
      - name:
          - EIP
```

```
        encoding: 2
        nbits: 1
        value: 0
        default_value: 0
        properties:
          - read
          - write

# ...
interrupts:
  check: |
    if (registers.NIP) return InterruptType.Nonmaskable;
    if (registers.MIP) return InterruptType.Maskable;
    if (registers.EIP) return InterruptType.EnvironmentCall;
    return null;

  is_enabled: |
    switch (type) {
      case InterruptType.Maskable:
        return registers.MIE === 1n;
      case InterruptType.EnvironmentCall:
        return registers.EIE === 1n;
      case InterruptType.Nonmaskable:
        // nonmaskable are always enabled
        return true;
    }
    return false;

  enable: |
    switch (type) {
      case InterruptType.Maskable:
        // we don't need to do anything for nonmaskable
        break;
      default:
        registers.IE = 1n;
        break;
    }

  disable: |
    switch (type) {
      case InterruptType.Maskable:
        // can't disable nonmaskable
```

```
        break;
      default:
        registers.IE = 0n;
        break;
    }

  create: |
    switch (type) {
      case InterruptType.Maskable:
        registers.MIP = 1n;
        break;
      case InterruptType.Nonmaskable:
        registers.NIP = 1n;
        break;
      case InterruptType.EnvironmentCall:
        registers.EIP = 1n;
        break;
    }

  clear: |
    switch (type) {
      case InterruptType.Maskable:
        registers.MIP = 0n;
        break;
      case InterruptType.Nonmaskable:
        registers.NIP = 0n;
        break;
    }

  global_clear: |
    registers.MIP = 0n;
    registers.NIP = 0n;
    registers.EIP = 0n;

  # ...
```

Now we can define our `syscall` instruction:

```
# ...
instructions:
  base:
    # ...
```

```
    - name: syscall
      template: standard
      fields:
        - field: opcode
          value: "0x02"
      definition: CAPI.INTERRUPTS.create(InterruptType.EnvironmentCall);
```

The convention will be that the type of system call we want to use will be stored in register `A`, while register `B` will hold extra information. For example, a system call to print a number will print whatever `B` is holding.

To allow CREATOR's handler to handle them, as system calls depend on each architecture, we must define that in the architecture definition:

```
# ...
interrupts:
  handlers:
    # ...
    creator_syscall: |
      switch (registers.A) {
        case 1n:
          CAPI.SYSCALL.print(registers.B, 'int32');
          break;
      }

      // notify UI that handler has finished
      CAPI.INTERRUPTS.clearHighlight();
```

Privileged instructions

CREATOR also supports having privileged instructions that can only be executed in kernel mode, by adding the `privileged` property. This is the reason for having system calls in the first place, we allow the user to ask doing things that require a higher privilege (e.g. accessing I/O) without giving them that privilege itself.

Let's say that in our architecture, an interrupt always triggers an execution mode change. To achieve that, we should modify the custom handler so that it sets kernel mode (`CAPI.INTERRUPTS.setKernelMode();`) and modify the `reti` instruction so that it goes back to user mode (`CAPI.INTERRUPTS.setUserMode();`). As `reti` should only be used while dealing with interrupts, we'll make it a privileged instruction.

```
instructions:
  base:
```

```
    # ...
    - name: reti
      # ...
      properties:
        - privileged
      definition: |
        // enable interrupts
        CAPI.INTERRUPTS.globalEnable();

        // pop return address from stack
        registers.PC = CAPI.MEM.read(registers.SP, 1);
        registers.SP = (registers.SP + 1n) & 0xFFn;

        // notify UI that handler has finished
        CAPI.INTERRUPTS.clearHighlight();

        // set user mode
        CAPI.INTERRUPTS.setUserMode();

# ...

interrupts:
  handlers:
    custom: |
      // disable interrupt
      CAPI.INTERRUPTS.disable(type);

      // set kernel mode
      CAPI.INTERRUPTS.setKernelMode();

      // store PC in stack
      registers.SP = (registers.SP - 1n) & 0xFFn;
      CAPI.MEM.write(registers.SP, 1, registers.PC);

      // jump to handler
      registers.PC = 0n;
```

Validating program execution

The CREATOR CLI allows to validate the execution of a program.

For this, you first need to define a YAML validator file. This file must contain the expected final

state of the program, as well as any additional options you want to set for the validation.

Validator File

We provide a JSON schema for the validator file at https://creator-sim.github.io/creator/schema/validator-file.json.

The validator file is a YAML file that can contain the following fields: - `maxCycles`: The maximum number of cycles to run the program for validation. If the program exceeds this number of cycles, it will be considered invalid. - `floatThreshold`: The maximum allowed error for floating point register comparisons. - `sentinel`: A boolean indicating whether to enable sentinel checking (calling convention errors). - `state`: An object defining the expected final state of the program, which can contain: - `registers`: A mapping of integer register names to their expected values. - `floatRegisters`: A mapping of floating point register names to their expected values. - `memory`: A mapping of memory addresses (as strings) to their expected byte values. - `display`: The expected contents of the display buffer as a string.

The validator will return, following POSIX conventions, `0` if its valid and `1` if it's not, writting the error message to the standard error stream (stderr).

Example

Quick example of a validator file:

```yaml
maxCycles: 100
floatThreshold: 10e-10
sentinel: true
state:
  floatRegisters:
    f0: 0x40866666
  registers:
    sp: 0x0FFFFFFC
    a1: 0x0000005A
  memory:
    "0x200000": 0x45
  display: "-144"
```

Using the Validator

To use the validator file with CREATOR CLI, use the `--validate` option followed by the path to the validator YAML file.

For example:

```
creator-cli -a RV32IMFD.yml -s assembly.s --validate config.yml
```

Web Version Overview

CREATOR is a browser-based assembly programming environment that requires no installation. It provides a complete development and debugging environment for multiple architectures directly in your web browser.

Figure 3: CREATOR main interface.

Access

Visit https://creatorsim.github.io/creator to use CREATOR online.

Features

- **Multi-Architecture Support**: Work with RISC-V, MIPS, Z80, and custom architectures.
- **Integrated Code Editor**: Syntax highlighting, IntelliSense, and code navigation.
- **Simulator**: Step through code, set breakpoints, and inspect memory/registers.
- **Modular Assemblers**: Support for multiple assemblers like CREATOR Assembler and RASM.
- **User-Friendly Interface**: Intuitive layout with architecture selection, code editor, and simulator panels.

User Interface

The CREATOR web interface is designed for efficiency and ease of use. This guide covers all UI components and their functions.

CREATOR uses a familiar global menu bar at the top of the window for accessing various features and settings, which change based on the current view. There are four main views: Architecture Select, Editor, Simulator, and Architecture. To change between the views, use the "View" menu in the top-left corner.

Architecture Select View

The Architecture Select view is the initial screen you see when you open CREATOR. It allows you to choose the processor architecture you want to work with. You can select from a list of predefined architectures or load a custom architecture YAML file.

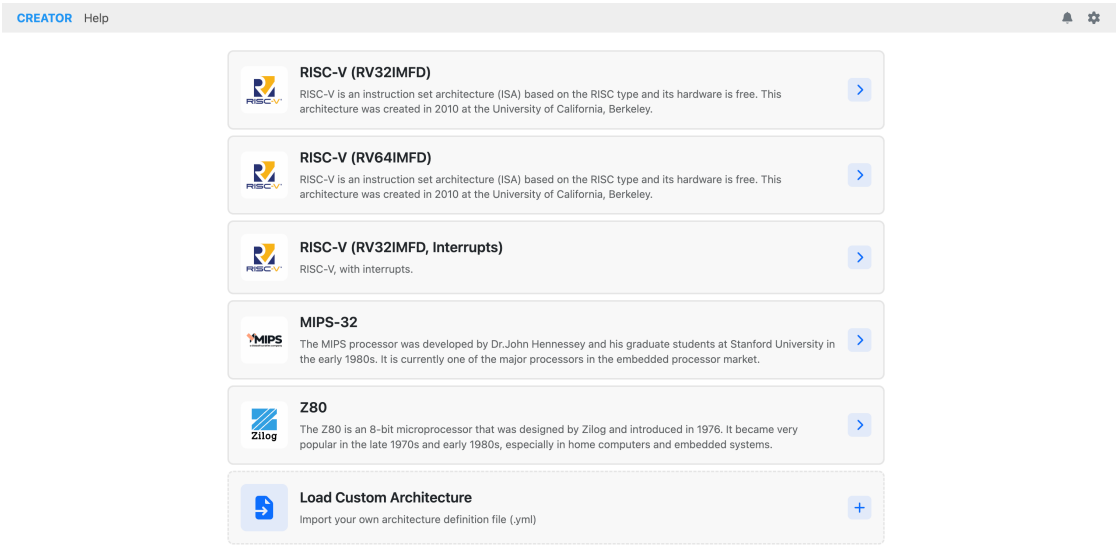

Figure 4: Architecture selection view for choosing processor architecture.

Editor View

CREATOR features a powerful code editor based on the Monaco Editor (the same editor used in VS Code). The editor supports syntax highlighting, IntelliSense, line numbers, breakpoints, and more. For specific editor features, refer to the Editor Features chapter.

Figure 5: Default view of the code editor with syntax highlighting.

Editor Menu Actions

When in the editor view, the menu bar provides the following exclusive actions: #### File

- **New File...**: Clear editor and start fresh
- **Open File...**: Load file from disk
- **Save As**: Download current code
- **Examples...**: Load example code
- **Get Code as URI**: Generate shareable URL

Library

- **Load Library...**: Load assembly libraries
- **Remove Library...**: Remove the loaded library
- **Library Tags**: View tags from the loaded library

Simulator View

The simulator view is divided into two columns: the left column contains the loaded instructions, while the right column contains tabs for execution control, registers, memory, and statistics.

Figure 6: Default simulator panel showing execution controls and tabs.

Left Column: Instruction List

The left column displays the list of loaded instructions in a scrollable panel. Each instruction is shown with its address, user representation (which may include pseudo-instructions), and the underlying assembled instruction.

Right Column: Simulator Tabs

1. Registers Tab The Registers tab displays the register bank(s) for the loaded architecture. You can view values in hexadecimal (the default), decimal (signed or unsigned) Clicking on a register opens a modal that shows even more display options, including binary, char, and IEEE 754 formats. This modal also allows you to edit the register value directly.

2. Memory Tab The Memory tab provides a full-featured hex editor. You can navigate to specific addresses, jump to memory sections (data, code, stack, heap), and toggle the tag display for easier debugging. The tag display shows the name of the variable or section associated with each memory address, if available. Hints are also shown for memory locations that have been tagged with type information. These addresses are highlighted in a different color for easy identification.

The hex editor also provides a memory layout diagram to help visualize different memory regions and their purposes.

To jump to a specific address, use the “Go to Address” input field. You can also quickly navigate

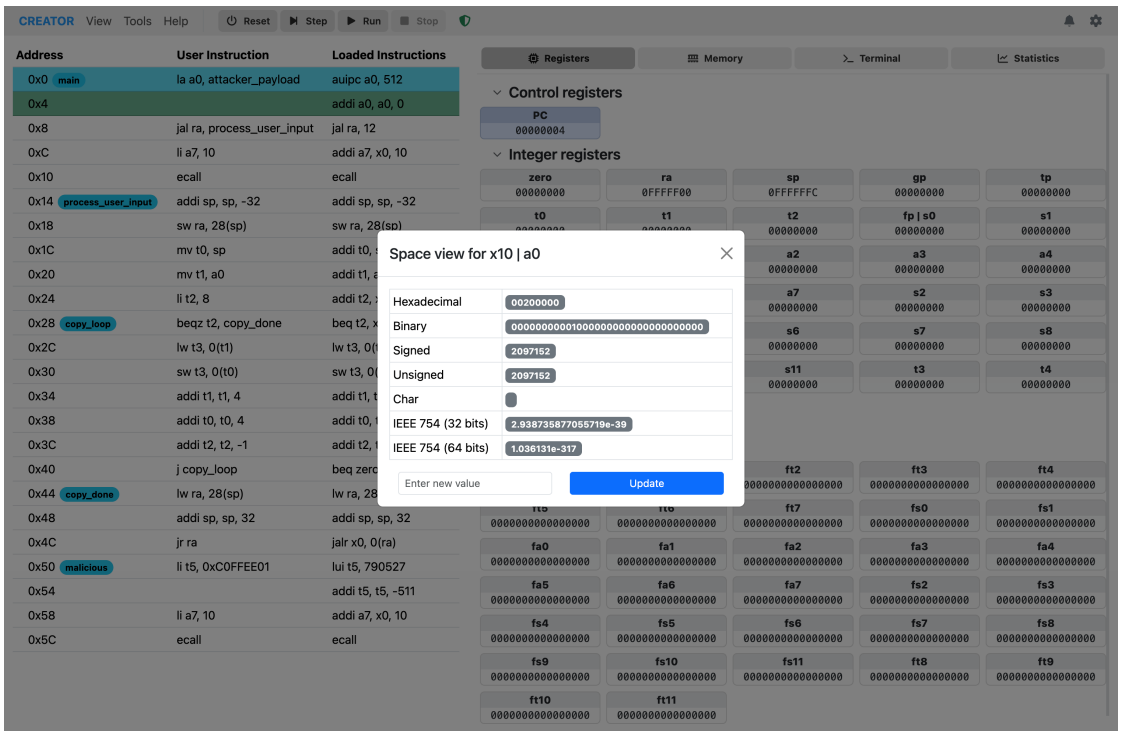

Figure 7: Registers tab displaying register values in space view mode.

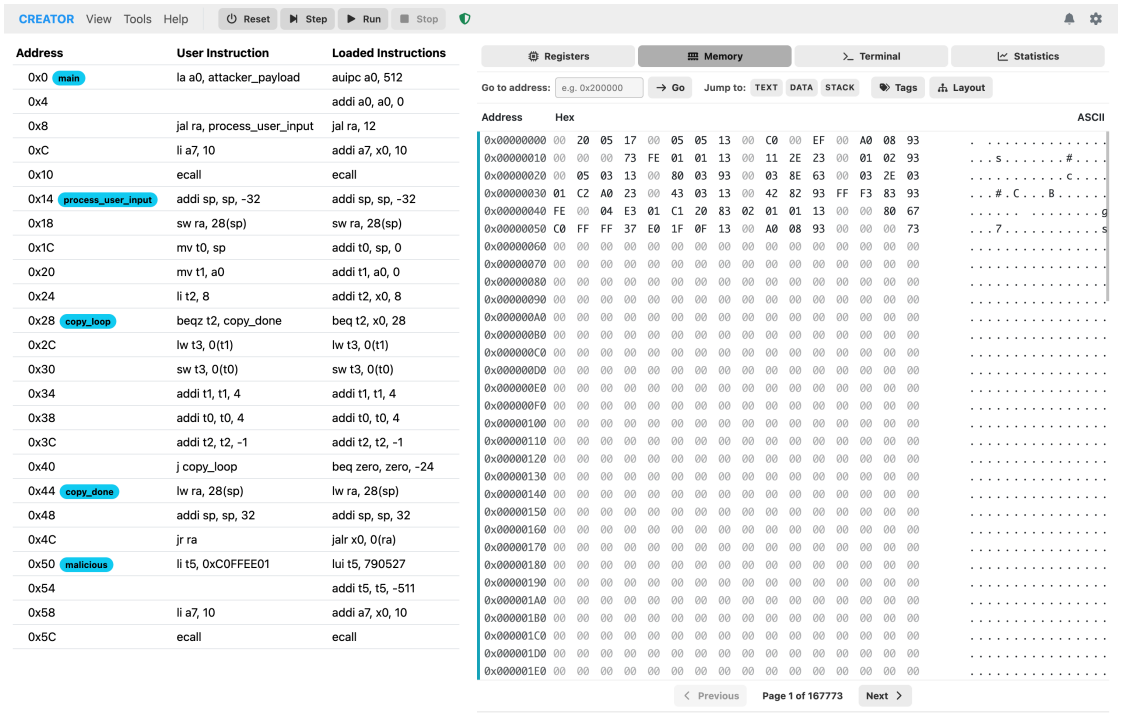

Figure 8: Memory tab showing hexadecimal memory contents.

to predefined sections like data, code, stack, and heap using the corresponding buttons.

You can also edit the memory contents directly by clicking on a byte and typing a new value. The

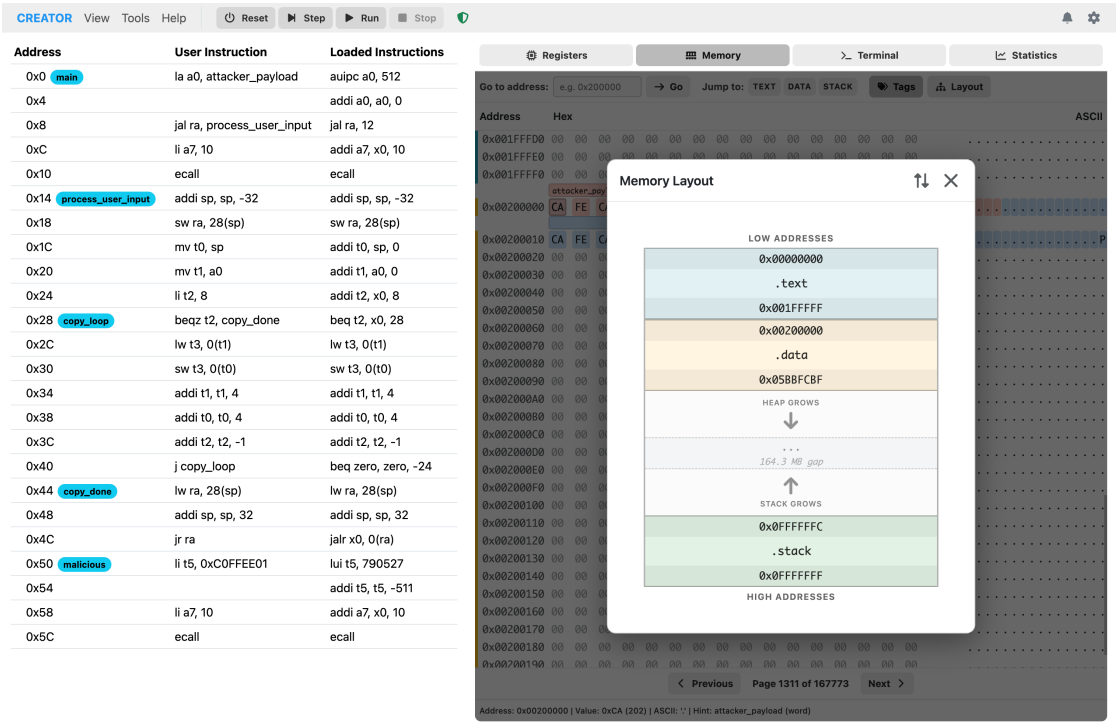

Figure 9: Memory layout diagram showing different memory regions.

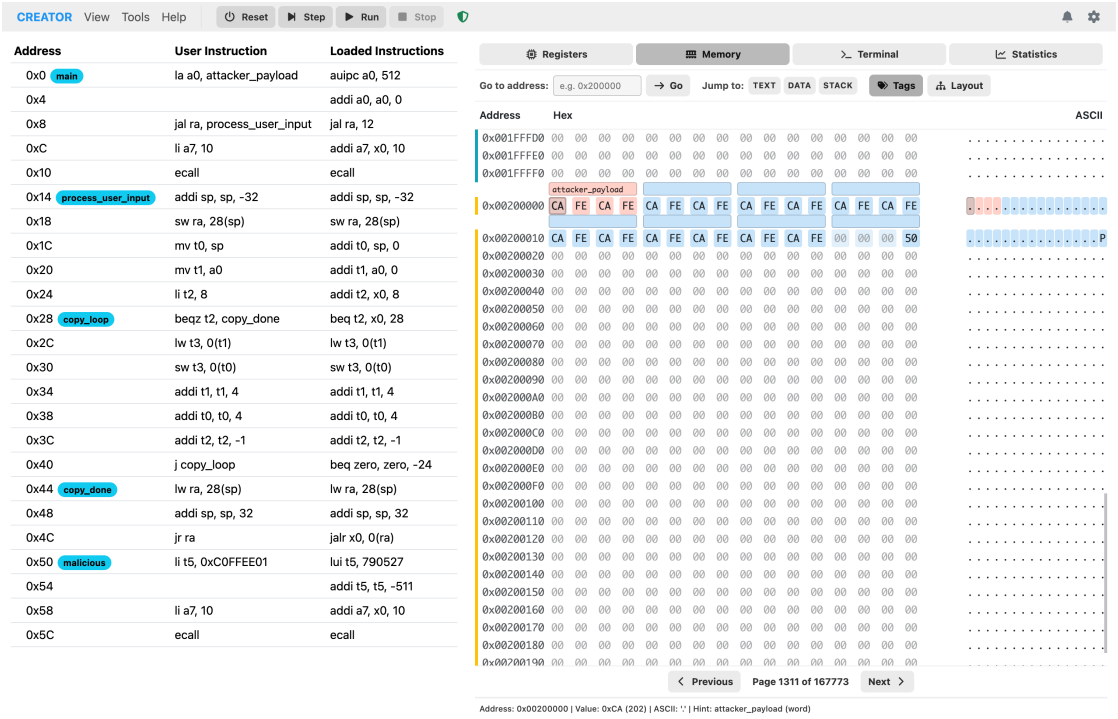

Figure 10: Memory viewer focused on data section.

editing mode allows you to modify memory on-the-fly during program execution or while paused at a breakpoint. You can edit multiple bytes in a row, simply by typing the new values consecutively.

3. Terminal Tab The terminal tab provides an interface for program input and output, simulating a console environment.

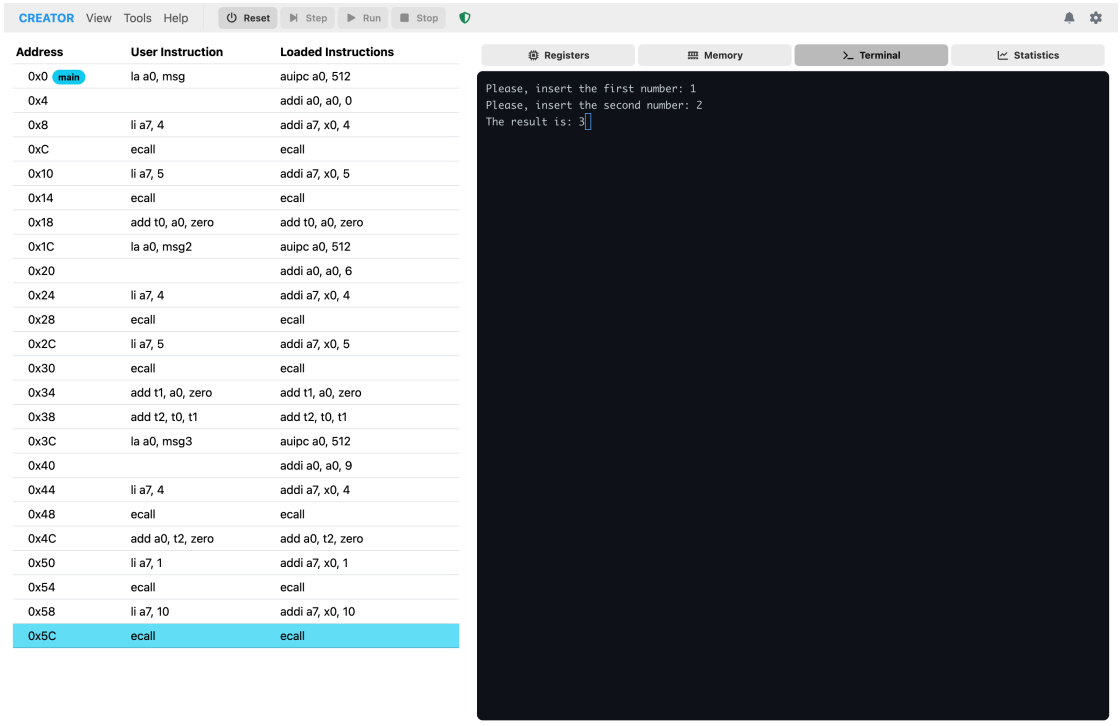

Figure 11: Terminal tab showing a simple sum program interaction.

4. Statistics Tab The Statistics tab shows execution metrics including the number of instructions executed, clock cycles, CPI (Cycles Per Instruction), and simulated execution time.

Sentinel

CREATOR includes a Sentinel feature that monitors the execution status of your program. It provides feedback on whether the parameter passing conventions are being followed correctly. The sentinel button changes color based on the status: - **Green (OK)**: All parameters passed correctly - **Red (Error)**: Parameter passing conventions violated

Architecture View

The architecture panel provides detailed information about the currently loaded architecture, including registers, instructions, directives, and pseudo-instructions. The left sidebar contains general architecture information such as the name, word size, endianness, and other relevant details.

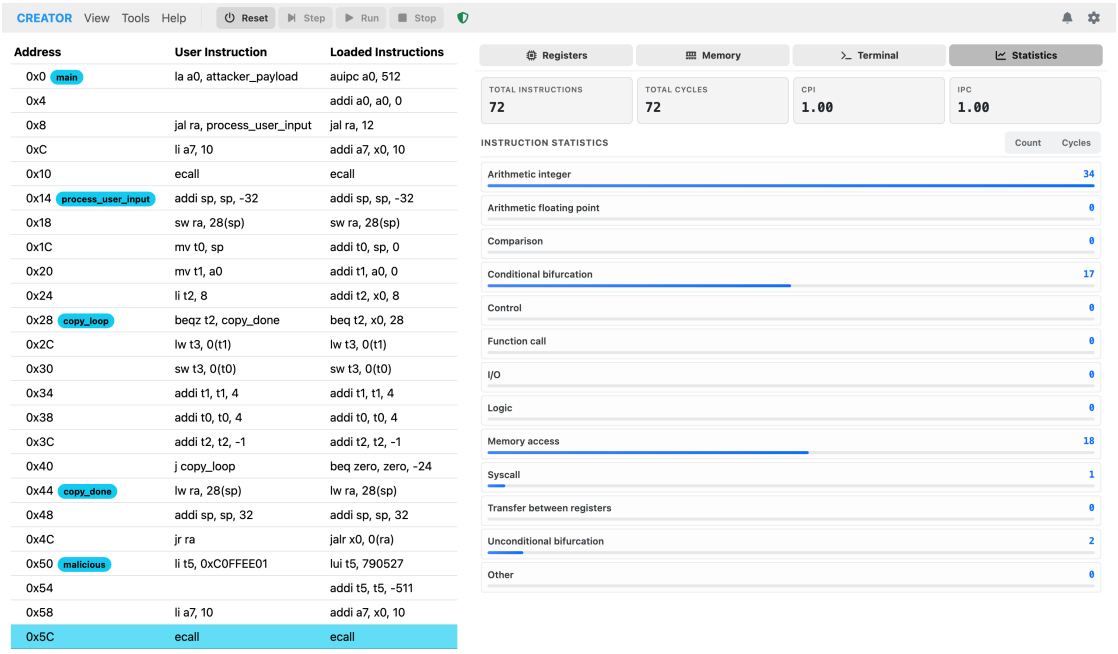

Figure 12: Statistics tab displaying execution metrics and performance data.

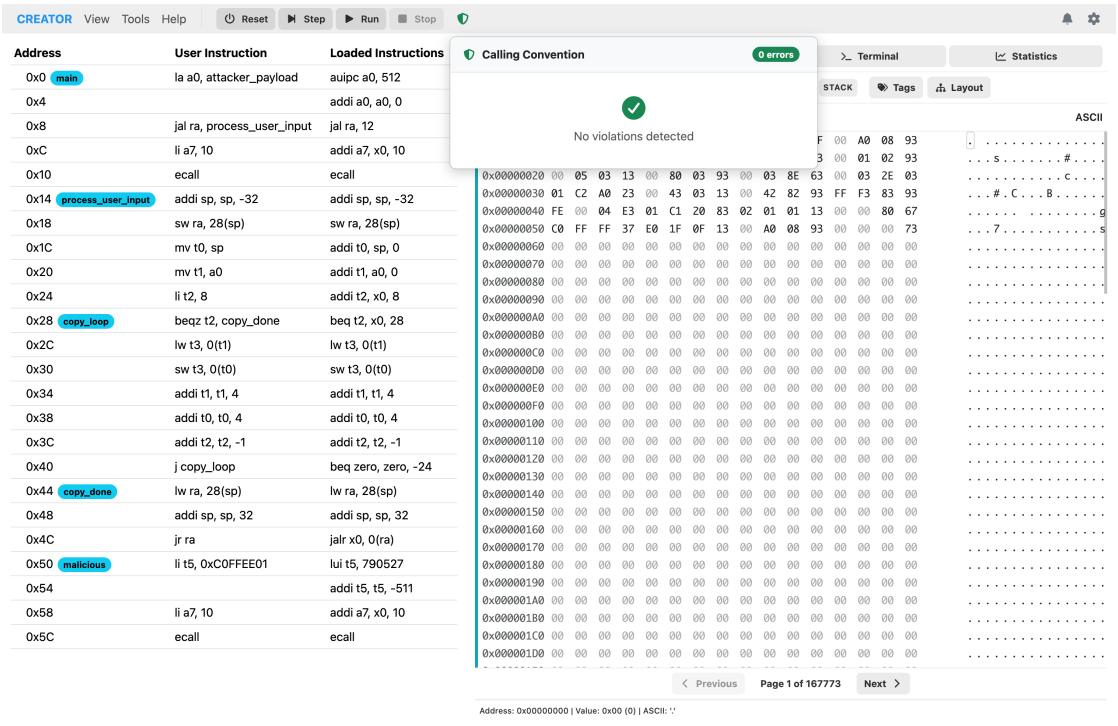

Figure 13: Simulator panel with sentinel status OK, indicating successful execution.

Architecture Instructions

The instructions section lists all supported assembly instructions for the selected architecture, along with their syntax, help, and encoding details. The diagrams are dynamically generated based on the

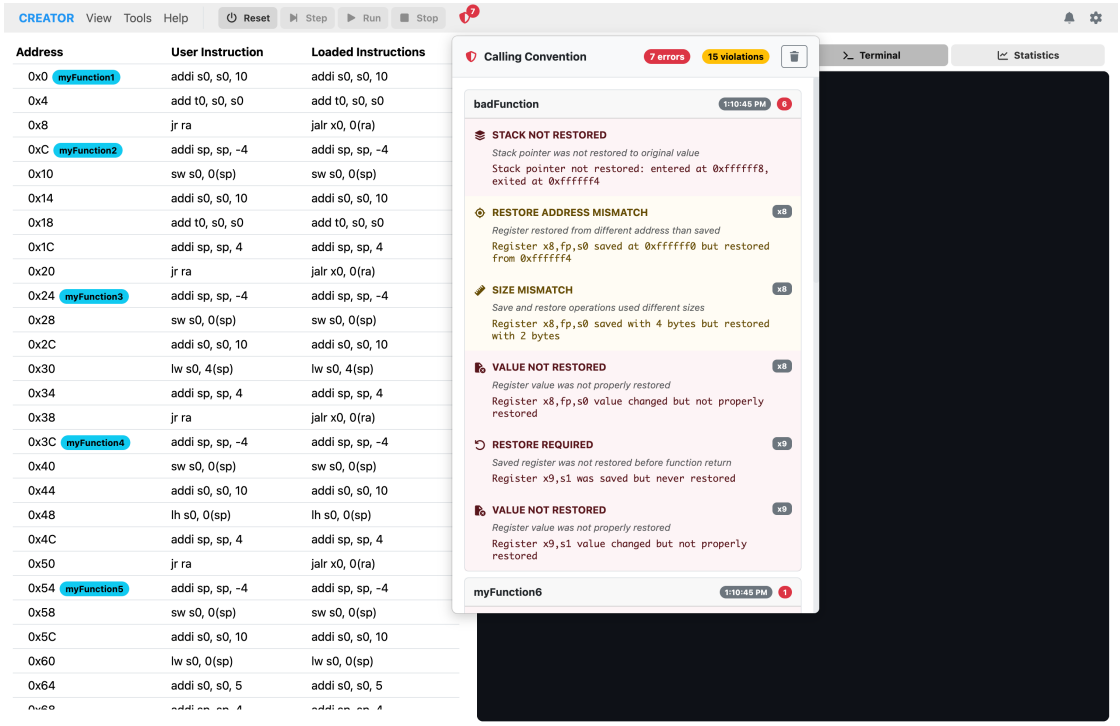

Figure 14: Simulator panel with sentinel status Error, indicating a parameter passing violation.

architecture YAML definition.

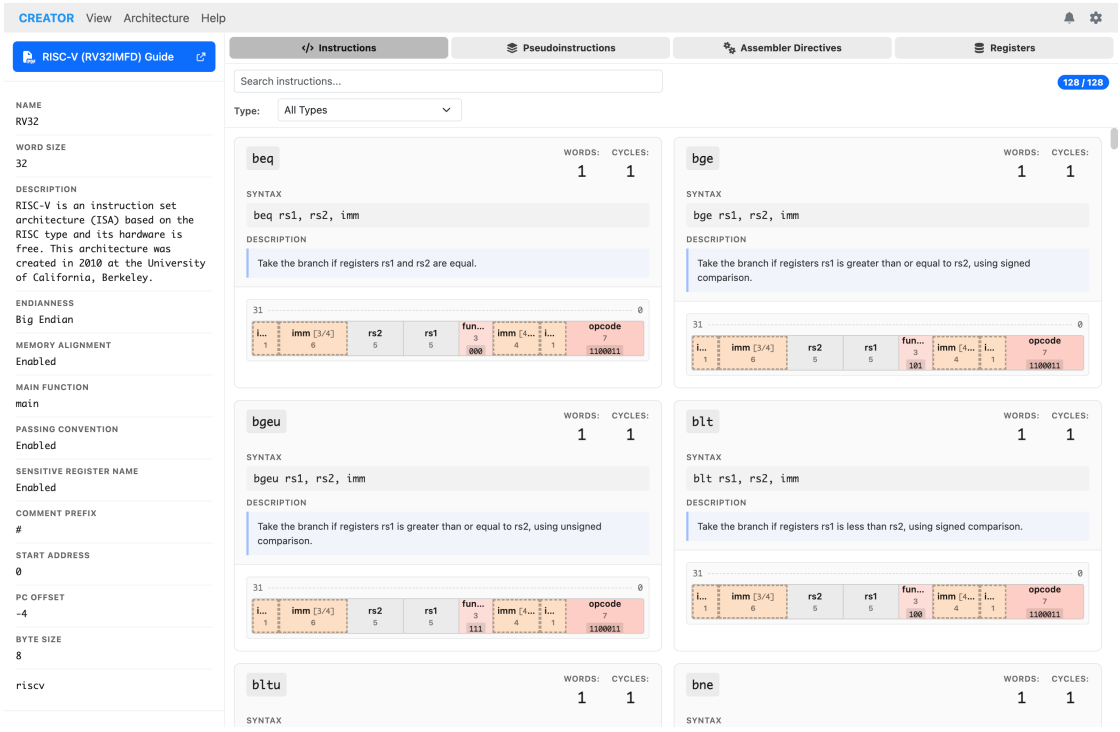

Figure 15: List of supported assembly instructions.

Architecture Pseudo-Instructions

The pseudo-instructions section displays all pseudo-instructions available for the architecture. Pseudo-instructions are higher-level assembly commands that expand into one or more real instructions during assembly.

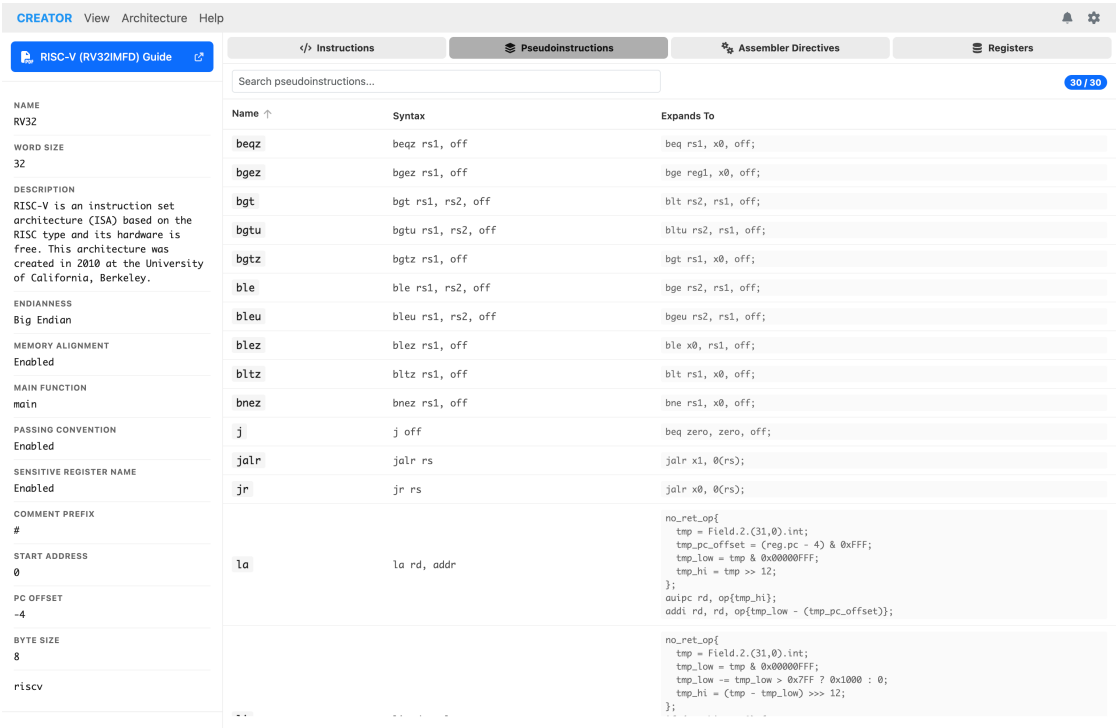

Figure 16: Pseudo-instructions that expand to multiple real instructions.

Architecture Directives

The directives section shows all assembler directives supported by the architecture, along with their syntax and descriptions.

Architecture Registers

The registers section provides a comprehensive list of all registers defined in the architecture, including their names, descriptions, and types.

Settings Menu

The setting menu is always accessible from the top-right corner of the interface. It provides options to customize the appearance and behavior of the editor and simulator.

The settings menu allows you to switch from light and dark mode. You can also adjust editor preferences and customize keyboard shortcuts.

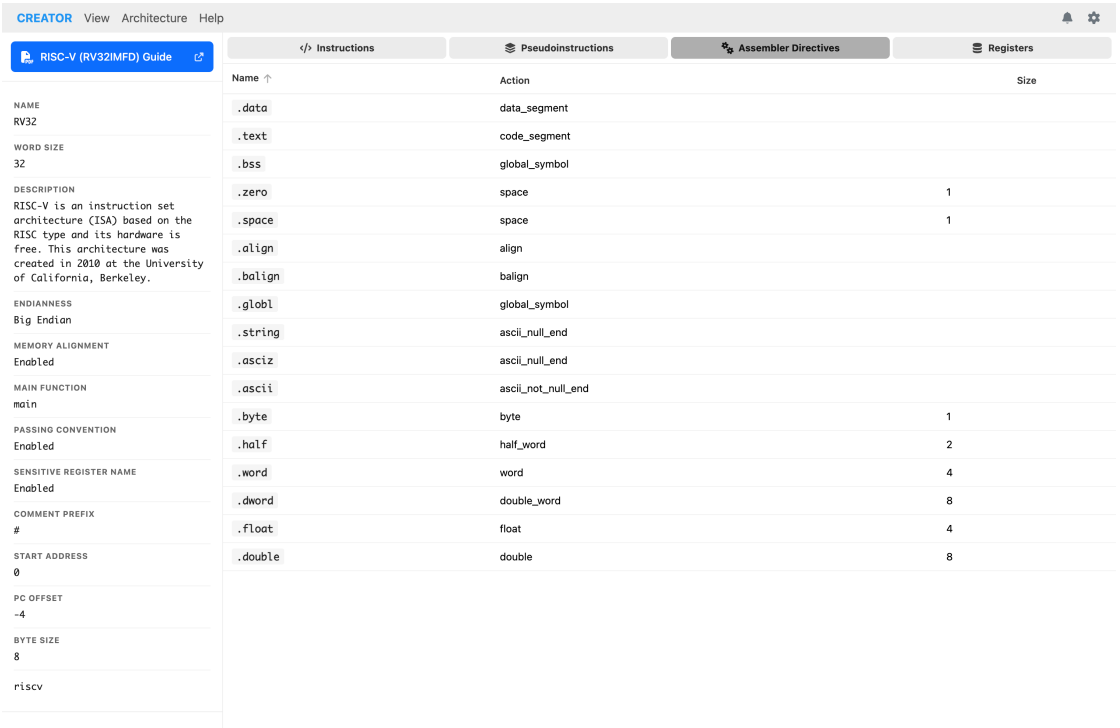

Figure 17: Assembler directives available for the architecture.

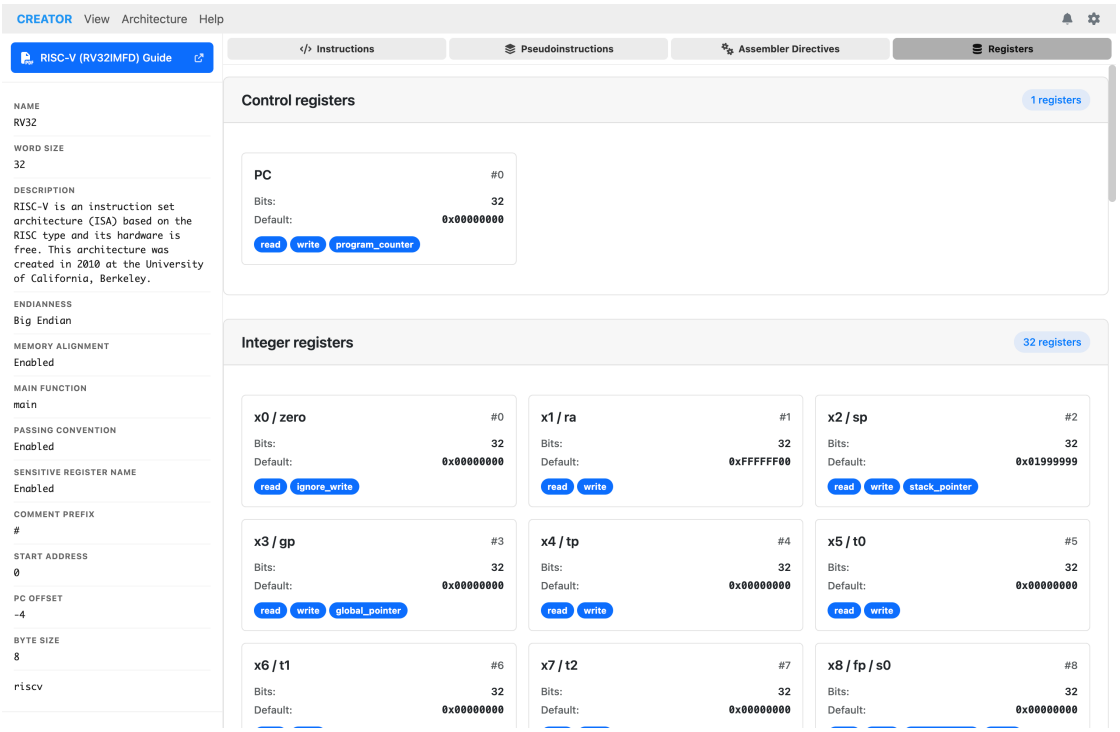

Figure 18: Architecture panel showing available registers for the selected processor.

CREATOR Remote Laboratory

CREATOR's Remote Laboratory allows students to execute their programs in remote ESP32 hardware provided by their teachers.

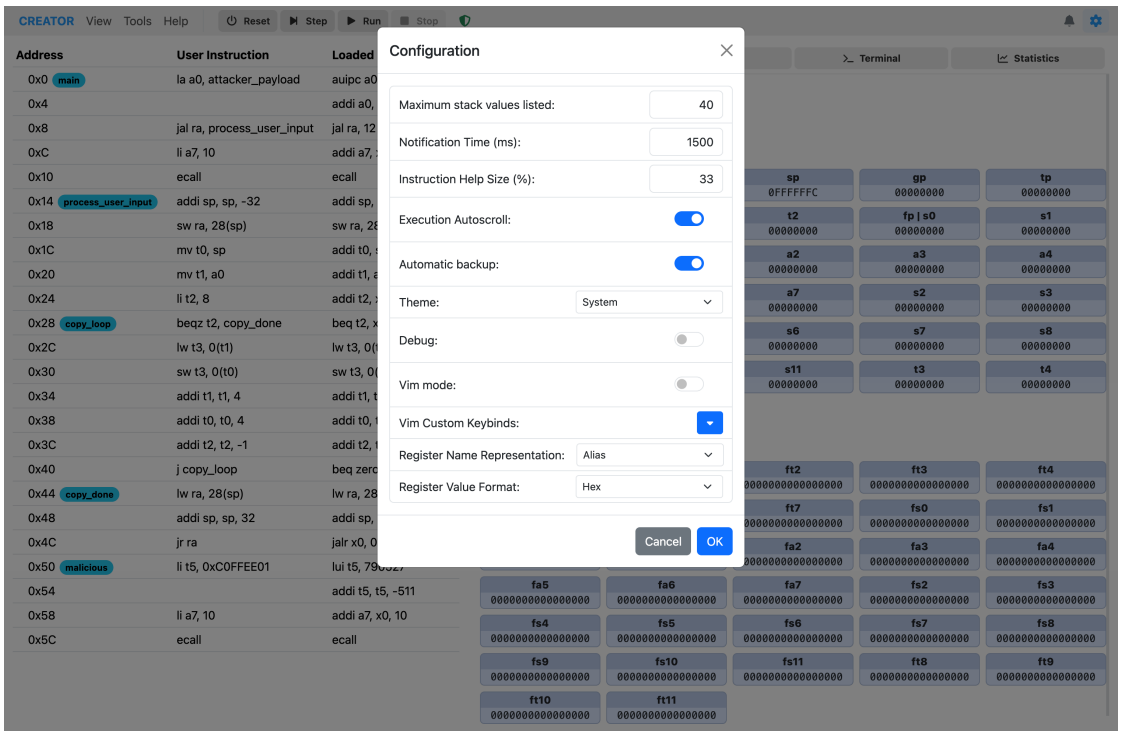

Figure 19: Settings menu drop-down showing theme selection and other options

CREATOR 6.0.x does not support keyboard (STDIN) inputs.

If you want to deploy your own remote laboratory, see Setting up the Remote Laboratory.

User Interface

The Remote Device Target Flash menu can be accessed from *Tools* → *Flash* → *Remote Device* in the simulator view.

1. Connect to the Remote Device URL where the server is deployed, e.g. `http://remote.arcos.inf.uc3m.es:5000`.
2. Select one of the available board types that the server supports
3. Enter your e-mail address to receive the execution results
4. Execute your program by using the *Send program* button.

The status of your program will be displayed. You can cancel the execution by using the *Cancel last program* button.

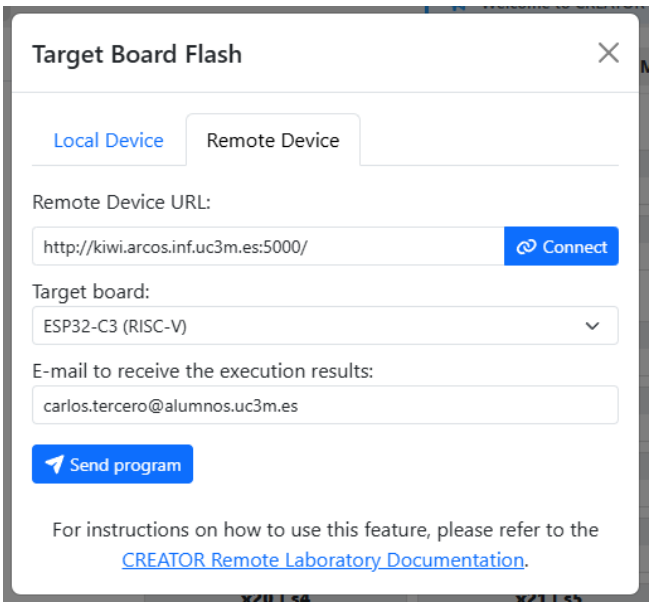

Figure 20: Remote Device Target

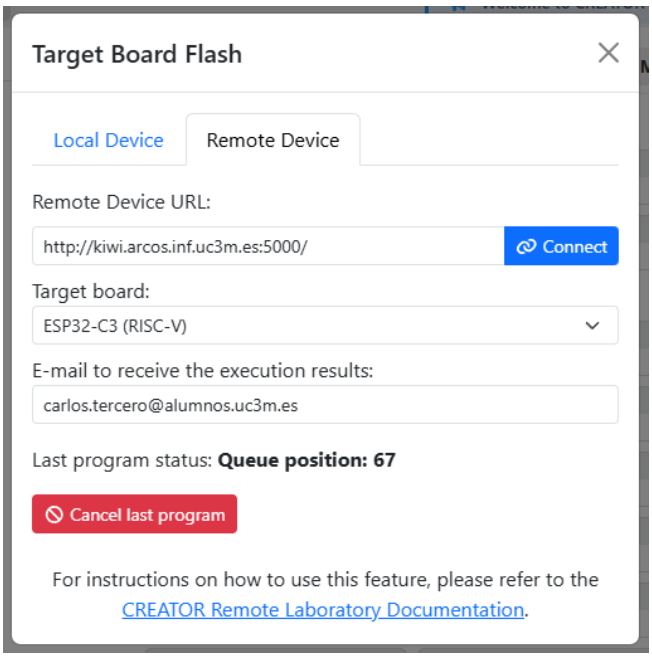

Figure 21: Remote UI (queued)

Editor Features

The CREATOR web editor is based on Monaco Editor, the same editor that powers Visual Studio Code.

Syntax Highlighting

Assembly language syntax is highlighted for better readability. This supports any architecture defined in CREATOR and requires no additional configuration.

Developers can extend or override the default syntax highlighting by creating their own language definitions. See the Development chapter for details.

Auto-Completion (IntelliSense)

As you type, the editor suggests completions for instructions, registers, and labels. These suggestions adapt to the currently selected architecture.

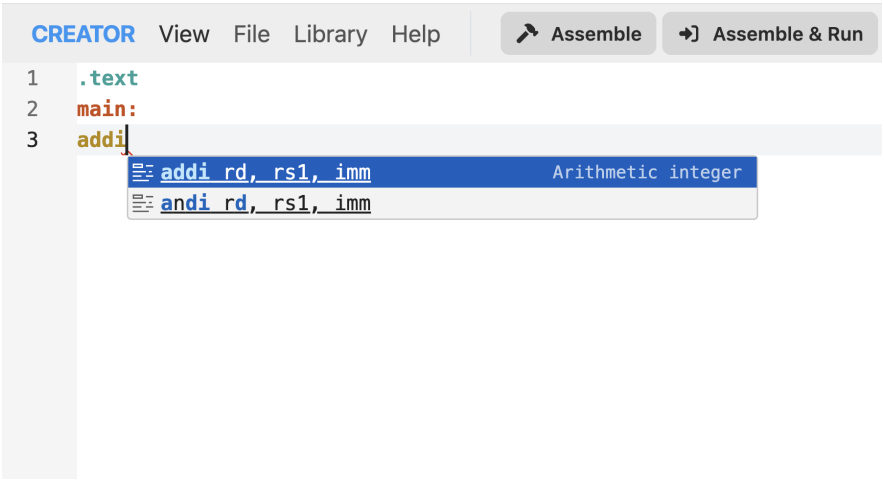

Figure 22: Auto-completion suggestions for RISC-V instructions.

Help Tooltips

Hover over instructions to see detailed help tooltips, including syntax, description, and usage examples.

Go to Definition/References

Right-click a label and select “Go to Definition” to jump to its declaration. Right-click and select “Find All References” to see all usages of that label in your code.

Code Comments

Single-Line Comments

Comments are architecture dependent. For RISC-V and MIPS, use `#` for single-line comments. The comment prefix for any given architecture can be consulted in the ”Architecture View” chapter.

Figure 23: Help tooltip for the `ADD` instruction in RISC-V.

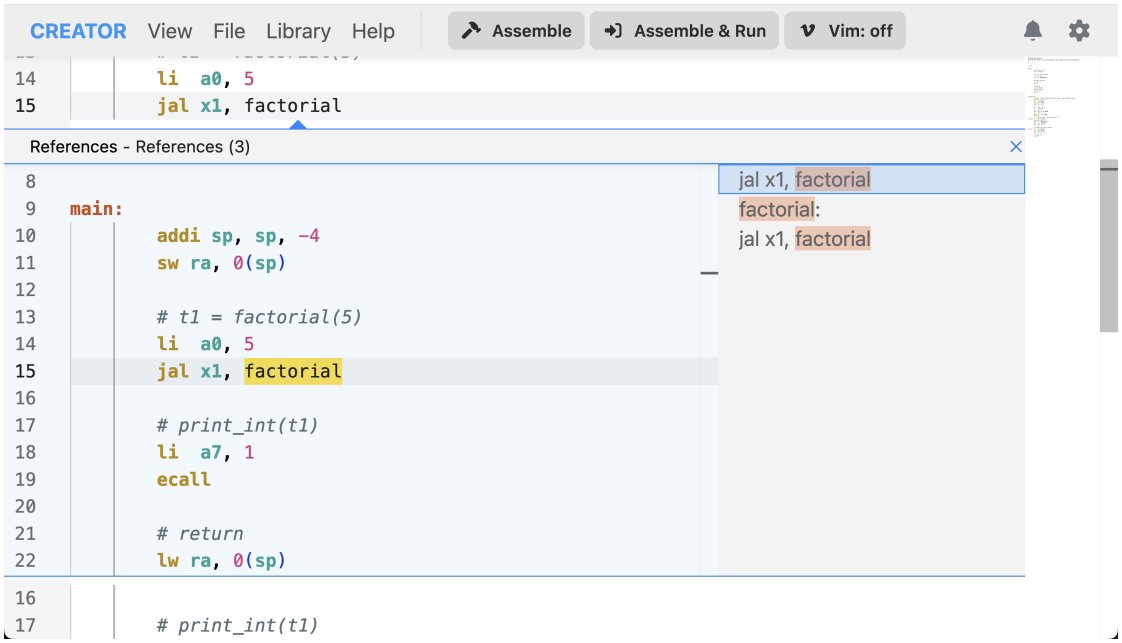

Figure 24: Finding all references to a label in the code.

Minimap

The minimap shows a bird's-eye view of your code: - Located on the right side of the editor - Shows entire file structure - Click to jump to sections - Highlights current viewport

Error and Warning Indicators

In supported architectures, the editor provides real-time error and warning indicators as you type. These errors are displayed as squiggly lines under the relevant code sections.

Common errors include: - Syntax errors - Unknown instructions - Invalid operands - Missing labels

Hover over the squiggly lines to see detailed error messages.

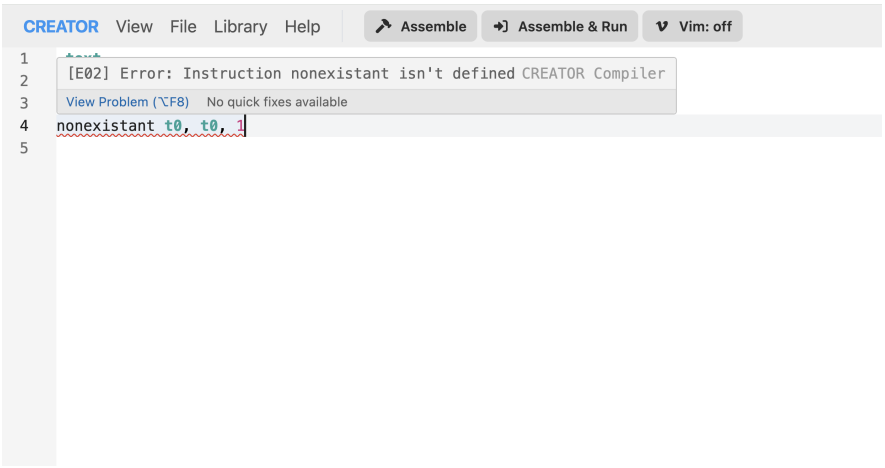

Figure 25: Error and warning indicators in the editor.

Vim Mode

Advanced users can enable Vim keybindings by clicking the “Vim” button in the editor toolbar. This activates Vim mode, allowing users to navigate and edit code using Vim commands. Custom keybinding can also be configured in the settings.

CREATOR Gateway

CREATOR supports the execution of RISC-V programs in real hardware devices.

We also support our predefined RISC-V system calls

Supported devices

CREATOR supports Espressif ESP32 development boards and RISC-V SBCs.

Espressif ESP32

Espressif’s family of ESP32 MCUs.

Figure 26: Vim mode enabled in the editor.

- `esp32c3`
  - ESP32-C3-DevKitC-02 (includes JTAG)
  - ESP32-C3-DevKitM-1 (no JTAG detected)
- `esp32c6`
  - ESP32-C6-DevKitC-1 (JTAG + port included)
  - ESP32-C6-DevKitM-1 (JTAG + port included)
- `esp32h2`
  - ESP32-H2-DevKitM-1 (JTAG + port)

These use the creator-gateway-esp32.

Single-Board Computers

RISC-V SBC boards with Linux that can run SSH and GDBGUI. At the moment, SBC RISC-V boards with Ubuntu 24.04.3 for RISC-V fulfill these requirements.

- OrangePi RV2: 8 RISC-V cores, Wi-Fi and Bluetooth connection
- Nezha D1-H 64 bit RISC-V: Single-core RISC-V64. It does not offer Wi-Fi connection or a GUI

These use the creator-gateway-sbc.

Executing the ESP32 gateway

Docker execution

This is the recommended way of executing the gateway on Windows, Linux, or macOS.

For more information, see the IDF documentation.

Windows

1. Install Docker Desktop
2. Connect your device and check which port it belongs to. You can do this with the `mode` terminal command, or in the Device Manager. The default port is `COM3`.
3. Set up the Remote Serial Port:
4. Download and unzip esptool
5. Run `esp_ffc2217_server` in the device's port (e.g. `COM3`):

```
powershell> esp_rfc2217_server -v -p 4000 COM3

```

6. Run the container:

```
docker run --rm --name creator-gateway-esp32 -it  \
           --init \
           -p 8080:8080 -p 5000:5000 \
           --add-host=host.docker.internal:host-gateway \
           creatorsim/creator-gateway-esp32:latest

```

You can also use a Docker compose file (`compose.yaml`):

1. Create the following `compose.yaml` file:

```yaml
services:
  creator-gateway-esp32:
    image: creatorsim/creator-gateway-esp32:latest
    ports:
      - "8080:8080" # gateway
      - "5000:5000" # gdbgui
    stdin_open: true
    tty: true
    init: true
    # for debug
    network_mode: bridge
    extra_hosts:
      - "host.docker.internal:host-gateway"
```

2. Deploy the docker compose in the directory of the YAML file:

```
docker compose up
```

Take into account that `docker compose up` is not interactive, therefore the program won't be able to read the keyboard inputs. You can run `docker compose up -d` and then attach to the specific container (check its name/id with `docker ps`) with `docker attach <container>`.

Linux/macOS

1. Install Docker engine (or Docker Desktop)
2. Connect your device and check which port it belongs to. It typically resides in `/dev/`, e.g. `/dev/ttyUSB0`. You can quickly check it with `ls /dev/ttyUSB*` (Linux) or `ls /dev/cu.usbserial-*` (macOS).
3. Run the container:

```
docker run --rm --name creator-gateway-esp32 -it \
           --init \
           -p 8080:8080 -p 5000:5000 \
           --device=/dev/ttyUSB0 \
           --add-host=host.docker.internal:host-gateway \
           creatorsim/creator-gateway-esp32

```

You can also use a Docker compose file (`compose.yaml`):

1. Create the following `compose.yaml` file:

```
services:
  creator-gateway-esp32:
    image: creatorsim/creator-gateway-esp32:latest
    ports:
      - "8080:8080" # gateway
      - "5000:5000" # gdbgui
    stdin_open: true
    tty: true
    init: true
    devices:
      - /dev/ttyUSB0 # device port
    # for debug
    network_mode: bridge
    extra_hosts:
      - "host.docker.internal:host-gateway"
```

2. Deploy the docker compose in the directory of the YAML file:

```
docker compose up
```

Take into account that `docker compose up` is not interactive, therefore the program won't be able to read the keyboard inputs. You can run `docker compose up -d` and then attach to the specific container (check its name/id with `docker ps`) with `docker attach <container>`.

Setting up the debugger The debugger is only available in boards with JTAG, and both USB and SERIAL ports must be connected to the computer.

For boards without the secondary port on the board, but with JTAG support, you can wire a USB-to_Dip as such:

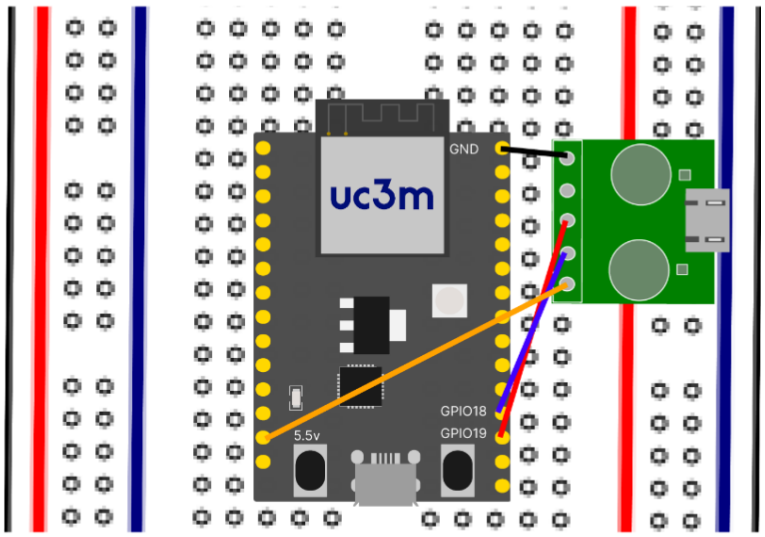

Figure 27: JTAG connection.

Linux/macOS

1. Setup OpenOCD
2. Download OpenOCD with ESP32 JTAG support v0.12.0-esp32-20241016 (for your OS and architecture) and unzip it
3. Add the `bin/` folder to your PATH, e.g.:

```
export PATH="/full/path/to/openocd-esp32/bin:$PATH"
```

4. Set the `OPENOCD_SCRIPTS` environment variable:

```
export OPENOCD_SCRIPTS="/full/path/to/openocd-esp32/share/openocd/scripts"
```

5. Execute the `openocd_start.sh` script (inside the `openocd_scripts` folder in our driver) with the type of device (e.g. `esp32c3`)

```
./openocd_start.sh esp32c3
```

6. After the container confirms that GDBGUI is up and running, access the web interface at http://localhost:5000

Windows

1. Install and setup Zadig
2. List all the devices in `Options` > `List All Devices` and select `USB Jtag/serial debug unit (Interface 2)`

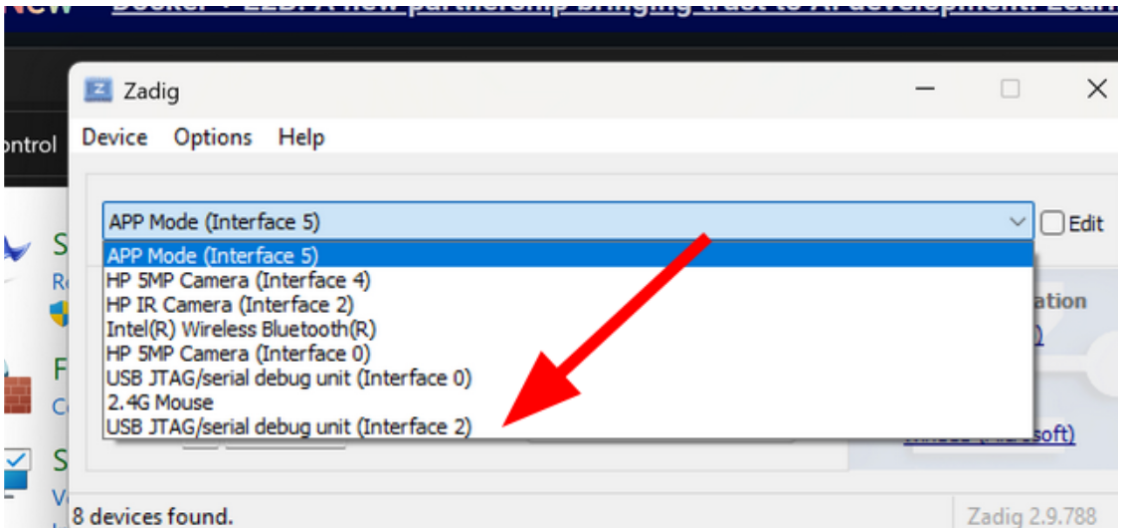

Figure 28: Zadig set device.

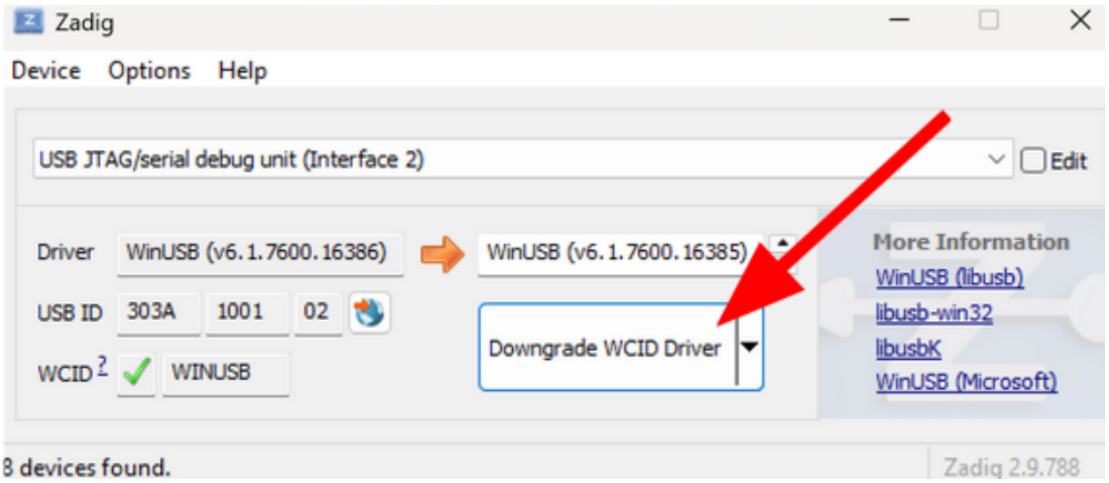

Figure 29: Zadig downgrade driver.

3. Downgrade the driver:
4. Setup OpenOCD
5. Download OpenOCD with ESP32 JTAG support v0.12.0-esp32-20241016 (for your OS and architecture) and unzip it.
6. Add the `bin\` folder to your PATH, e.g.:

```
set PATH=%PATH%;<openocd-esp32 path>\bin\
```

7. Execute the `openocd_start.bat` script (inside the `openocd_scripts` folder in our driver) with the type of device (e.g. `esp32c3`)

```
\openocd_start.bat esp32c3
```

8. After the container confirms that GDBGUI is up and running, access the web interface at http://localhost:5000

Native execution (Linux-only)

You can run the gateway natively on your Linux device.

1. Install Python 3.9

- With uv:

```
uv python install 3.9
```

- In Ubuntu:

```
sudo apt install software-properties-common sudo add-apt-repository ppa:deadsnakes/ppa
     sudo apt install python3.9
```

2. Install the ESP-IDF framework v5.3.2

- Follow the instructions from Espressif's documentation.
- To ensure Python 3.9 is used for the installation, first create a virtual environment in `~/.espressif/python_env/idf5.3_py3.9_en`, and activate it, before executing the `install.sh` script.

```
python3.9 -m venv ~/.espressif/python_env/idf5.3_py3.9_en source ~/.espressif/
    python_env/idf5.3_py3.9_env/bin/activate
```

3. Download and unzip the ESP32 gateway
4. Install the python dependencies
5. Use the `install.sh` script from ESP-IDF:

```
/path/tp/esp-idf-v5.3.2/install.sh
```

6. Install the Python dependencies with pip (move to the downloaded folder):

```
pip3 install -r requirements.txt
```

7. Load the ESP-IDF environment variables (`export.sh`)
8. Execute the gateway web service:

```
python3 gateway.py
```

Setting up the debugger You must set up **ports permissions** for the JTAG. Your user must be in the `plugdev` and `dialout` groups (in Ubuntu/Debian/Fedora), or `uucp` (in Arch Linux).

You can add yourself to the group with `usermod`, e.g.:

```
sudo usermod -a -G dialout $USER
sudo usermod -a -G plugdev $USER
```

Another error might occur: `gdb_exception_error – libusb_bulk_write error: LIBUSB_ERROR_NO_DEVICE`. This problem happens because the user doesn't have permission to write to the JTAG USB device, located in `/dev/bus/usb/003/XXX` (where `XXX` is a pseudo-random number).

You can check this by doing:

```
ls -lah /dev/bus/usb/003
```

And an example of the output might be:

```
total 0
drwxr-xr-x 2 root root 180 Oct 1 11:03 .
drwxr-xr-x 6 root root 120 Oct 1 10:36 ..
crw-rw-r-- 1 root root 189, 256 Oct 1 10:41 001
...
crw-rw-r-- 1 root root 189, 256 Oct 1 10:41 022
```

You can see the user and group are `root`.

To change this, we can configure udev so that, when it mounts the JTAG device it gives it the group permissions it typically gives to all other devices (`uucp` for Arch, `dialout` for Ubuntu).

To see the device's ID, run `lsusb`:

```
...
Bus 003 Device 018: ID 10c4:ea60 Silicon Labs CP210x UART Bridge
Bus 003 Device 022: ID 303a:1001 Espressif USB JTAG/serial debug unit
...
```

Therefore, the vendor ID for the JTAG is `303a`, and the product ID is `1001`, and for the UART `10c4` and `ea60`.

Create a new `/etc/udev/rules.d/99-Espressif.rules` file (with `sudo`!):

- For Ubuntu/Debian/Fedora:

```
: Set default permisions when mounting Espressif USB JTAG/serial debug unit and UART
SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device", ATTRS{idVendor}=="303a", ATTRS{idProduct
    }=="1001", GROUP="plugdev"
SUBSYSTEM=="tty", ATTRS{idVendor}=="10c4", ATTRS{idProduct}=="ea60", GROUP="dialout",
    MODE="0660"
```

- For Arch Linux:

```
: Set default permisions when mounting Espressif USB JTAG/serial debug unit
SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device", ATTRS{idVendor}=="303a", ATTRS{idProduct
    }=="1001", GROUP="uucp"
```

For more information, see the JTAG documentation.

Executing the SBC gateway

Recommendations for a correct SBC setup:

- Use the recommended power supply for your SBC
- Use a good Ethernet Cable
- Use a Class 10 MicroSD card from a recognizable brand from Amazon or another reliable retailer

1. Install the OS following the SBC manufacturer's instructions
2. Create the **default folder** where your CREATOR projects will be saved, e.g. `~/creator`
3. Ensure you provide the **correct rights** to the directory

```
sudo chown $USER:$USER ~/creator
sudo chmod u+rwx ~/creator
```

4. **Connect the SBC to the Internet** via Ethernet or Wi-Fi if possible. >

> Depending on the SBC's configuration, its IP may change from time to time. Check its private IP with `ip a` before each use. > An example IP would be `10.117.129.219`.

5. Check the SSH service status:

```
systemctl status ssh
```

6. Check your username with `whoami`. Typically, it's `ubuntu`.
7. Connect to the SBC from your computer via SSH:

```
ssh <user>@<IP>
```

Every time an SSH connection is made, the system will ask for a password. This can be overridden by copying your computer's SSH keys to the SBC.

1. Create a new SSH key in your computer:

```
ssh-keygen -t rsa -b 4096
```

2. Copy the key to the SBC:

```
ssh-copy-id <user>@<IP>
```

8. Download and unzip the SBC gateway in the SBC
9. Set up the gateway (inside the gateway folder):

10. Create a new Python virtual environment:

```
python3 -m venv .venv source .venv/bin/activate

```

11. Install the dependencies:

```
pip3 install -r requirements.txt

```

Currently, there is a small issue in GDBGUI that needs to be patched as such:

```
sed -i "/extra_files=get_extra_files()/a\        allow_unsafe_werkzeug=True," .
    venv/lib/python3.12/site-packages/gdbgui/server/server.py
```

To validate the changes, run:

```
grep -n "allow_unsafe_werkzeug" .venv/lib/python3.12/site-packages/gdbgui/
    server/server.py
```

10. Install the RISC-V Cross-Compiler on your macOS, Linux or Windows with WSL device
11. Execute the gateway:

```
python3 gateway.py
```

User Interface

The Target Flash menu can be accessed from *Tools* → *Flash* in the simulator view.

First, select your **device type** (ESP32 or SBC) and **target board**.

Then, provide the **target information**:

- ESP32
  - **Target port:** Port of the device's UART connection. The default values are:
    - **Linux:** `/dev/ttyUSB0`
    - **macOS:** `/dev/cu.usbserial-10`
    - **Windows:** `rfc2217://host.docker.internal:4000?ign_set_control`
- SBC
  - **Target user**: User and IP address of the SBC (`<user>@<ip>`)
  - **Target location**: Location of the project folder (e.g. `~/creator`)

Finally, provide the **flash URL**, the URL address of the gateway. By default, `https://localhost:8080`.

Buttons

- **Flash:** Builds and flashes CREATOR's program into your development board.

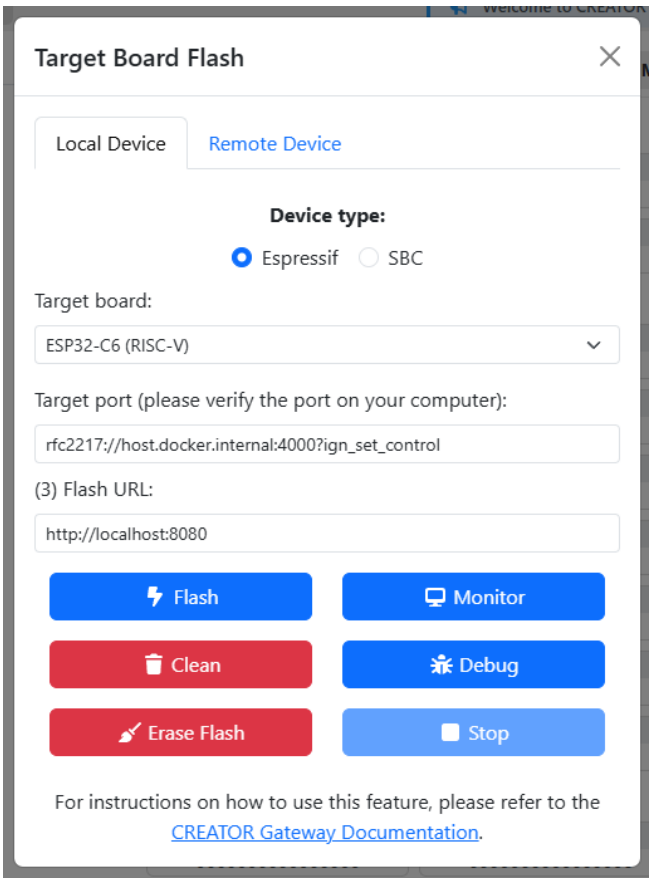

Figure 30: Target Board Flash dialog-box.

- **Monitor:** Executes development board's flashed program. It can be stopped by using the "Stop" button or the keyboard shortcuts `Ctrl` + `]` or `Ctrl` + `t` + `x`.
- **Debug**: If it's setup correctly, it will open another tab with an instance of GDBGUI ready to execute programs step-by-step.
- **Clean**: Erases gateway's copy of the program.
- **Erase-flash:** Erases the device's program.

Execution Control

The execution control panel allows you to run, step through, and reset your assembly programs. It provides essential controls for managing program execution within the simulator.

Execution Controls

- **Run**: Starts or resumes program execution until a breakpoint is hit or the program ends.
- **Step**: Executes the next instruction.
- **Reset**: Stops execution and resets the program state to the beginning.
- **Pause**: Temporarily halts execution, allowing you to inspect the current state.

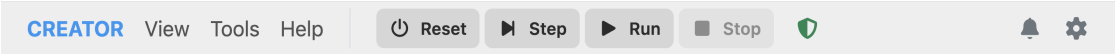

Figure 31: Execution control buttons in the simulator panel.

Breakpoints

CREATOR supports breakpoints to help with debugging. To set a breakpoint, click on any instruction in the instruction list. A red dot will appear next to the instruction, indicating an active breakpoint. When the program execution reaches a breakpoint, it will pause, allowing you to inspect registers, memory, and other state information.

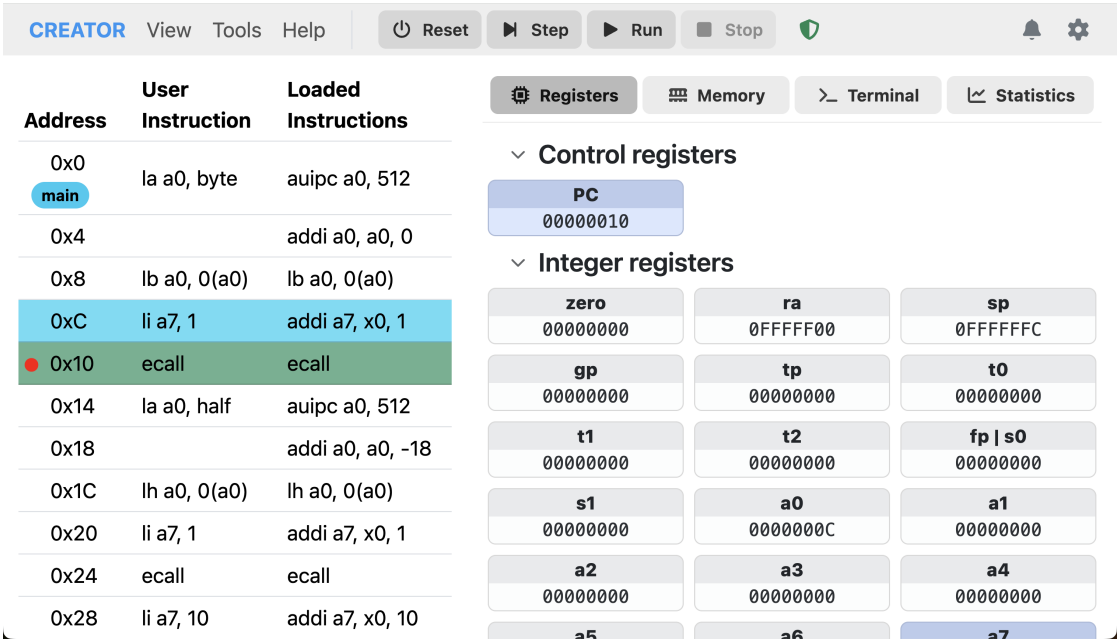

Figure 32: Execution paused at a breakpoint.

Execution Modes

- **User Mode**: Standard execution mode with full access to user-level instructions.
- **Kernel Mode**: Elevated execution mode for system-level instructions (if supported by the architecture).

For more information, see ”Privileged Instructions.

Interrupt Handling

Some architectures support interrupts. In such cases, CREATOR reacts according to the architecture's interrupt handling mechanisms. RISC-V and Z80 have their own interrupt models that are simulated accordingly.

CREATOR has two interrupt handlers: the default "CREATOR" handler, and a custom architecture-defined one. The CREATOR handler is the simplest of the two, it only handles architecture-defined system calls (see CREATOR handler), and treats all other interrupts as errors. On the other hand, the custom handler allows full control over interrupts, which requires writing a custom interrupt handler in the program.

You can modify the currently used handler in the configuration.

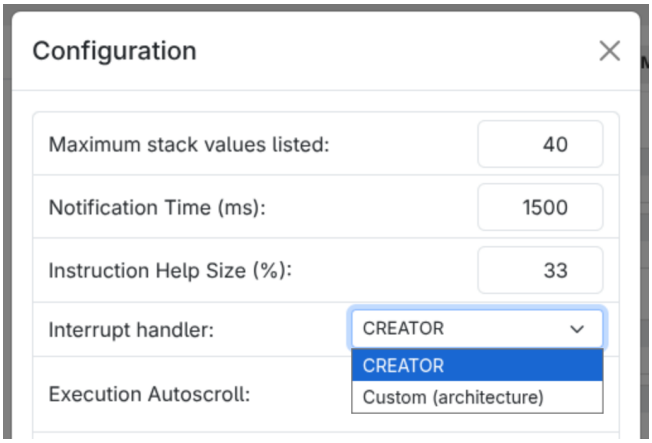

Figure 33: Modifying the interrupt handler.

For more information, see Interrupt Support.

Assemblers

CREATOR has support for a modular assembler system, allowing users to choose from multiple assemblers for different architectures. This chapter provides an overview of the available assemblers, their features, and how to use them within CREATOR.

CREATOR Assembler

The default assembler built into CREATOR. This assembler is designed to work seamlessly with the defined architectures. It is currently used for RISC-V, and MIPS. Any custom architectures defined by users will, by default, also use this assembler.

Limitations

Architectures with complex instruction sets may not be fully supported. One example is the Z80 architecture, which requires a more specialized assembler due to having multiple opcodes for the same instruction based on context. Instruction sets with multiple optional fields are not supported.

RASM

RASM is a Free and open-source Z80 assembler. When choosing the Z80 architecture in CREATOR, RASM is used as the default assembler.

Choosing an Assembler

By default, the assembler is chosen based on the architecture selected. From a user perspective, there is no need to choose. The appropriate assembler will always be used automatically.

Developers creating custom architectures can specify which assembler to use in the architecture definition file.

CREATOR Arduino module (CREATino)

CREATino allows the user to add custom Arduino libraries to their programs.

This library only works with the ESP32 gateway and the supported ESP32 boards. You must set up the gateway before using this module.

Using CREATino library functions

In the Editor view, go to *Library* → *Load Arduino Library.* The available functions will be displayed on the right.

Except the `printf` function (which is exclusive for Espressif Arduino devices) all the functions displayed can be found in Arduino's documentation and in *Help* → *Creatino Help.*

Creating your first program

As in the original Arduino sketches, CREATino programs must have a structure composed by a "setup" and a "loop" function. CREATOR provides an example template in *Help* → *Examples* → *Example 1: Template for new examples.*

Before executing your program in the device, make sure to select the *Arduino Support* checkbox to enable CREATino.

Aspects to consider when using this library

1. The supported ESP32 boards do not support have floating point operations. Arduino functions with `float` or `double` outputs will fail.
2. Avoid using the following GPIO pins:
   - **GPIO8:** BOOT MODE pin. It might show the following error:

     ```
     Serial port /dev/ttyUSB0  Connecting\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldot
     A fatal error occurred: Failed to connect to ESP32-C3: Wrong boot mode detected (0x0)! Th
     For troubleshooting steps visit: https://docs.espressif.com/projects/esptool/en/latest/tr
     ...
     ```

   - **GPIO 18 and 19:** Debug pins

Examples

Example 1: Internal LED Blink

This example is used to check if GPIO functions work correctly inside a ESP32-C3 board.

Components:

- * ESP32-C3-DevKitC-02 board

Step 1: Setup In the ESP32-C3-DevKitC-02, the internal LED pin corresponds to pin `30`, and it must be tagged as output. We use `pinMode` function for this.

The possible values for the mode are:

Mode	Value	Usage
`INPUT`	`0x01`	Digital input mode
`OUTPUT`	`0x03`	Digital output mode
`PULLUP`	`0x04`	Enables the internal pull-up resistor on an input pin (if the pin is disconnected, it will read HIGH). Buttons connected to ground
`INPUT_PULLUP`	`0x05`	Combines input configuration (`0x01`) with pull-up (`0x04`)
`PULLDOWN`	`0x08`	Ensures the pin reads LOW when disconnected
`INPUT_PULLDOWN`	`0x09`	Ensures the pin reads LOW when disconnected
`OPEN_DRAIN` (Data buses)	`0x10`	Output can only pull to ground; reading HIGH requires an external resistor or pull-up
`OUTPUT_OPEN_DRAIN` (Buses)	`0x13`	Combines digital output with open-drain mode
`ANALOG`	`0xC0`	Configures the pin for analog input (`ADC`)

In this case, our setup function will look like this:

```
setup:
  # pinMode(30, OUTPUT);
  li a0, 30
  li a1, 0x03
  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, pinMode
  lw ra, 0(sp)
  addi sp, sp, 4
  jr ra
```

Step 2: Loop The LED has been set up; now it is we can try to make it turn on and off, creating a blinking effect. For this, we'll use the `digitalWrite` function.

This function needs the number of the pin used (pin 30) and the state to write (in this case `0x1` or HIGH to turn on the light and `0x0` to turn it off)

We will also use a delay function to generate a delay, in milliseconds, between changes of state of the LED. For this example, we'll store the length of the delay in memory.

The result is:

```
.data
  time:
    .word 1000

.text
loop:
  # digitalWrite(LED_BUILTIN, HIGH);
  li a0,30
  li a1, 0x1
  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, digitalWrite
  lw ra, 0(sp)
  addi sp, sp, 4

  # delay(time);
  la a0, time
  lw a0, 0(a0)
  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, delay
  lw ra, 0(sp)
```

```
addi sp, sp, 4

# turn off

# digitalWrite(LED_BUILTIN, LOW);
li a0,30
li a1, 0x0
addi sp, sp, -4
sw ra, 0(sp)
jal ra, digitalWrite
lw ra, 0(sp)
addi sp, sp, 4

# delay(time);
la a0, time
lw a0, 0(a0)
addi sp, sp, -4
sw ra, 0(sp)
jal ra, delay
lw ra, 0(sp)
addi sp, sp, 4
j loop
```

Now the LED will start blinking.

Example 2: Button + LED

In this example we'll turn an LED on when the button is pressed.

Components:

- * ESP32-C3-DevKitC-02 board
- * Button (in GPIO 6)
- * LED (in GPIO 4)

There are two ways to do this: synchronously, using an infinite loop, and asynchronously, using interrupts.

Using an infinite loop

Step 1: Setup First we'll establish the LED as an output and the button as input. As shown in Use example 1, you need to configure the pin mode with `pinMode` before using the pins:

```
.data
  buttonPin: .word 6
  ledpin:    .word 4

.text

setup:
  # pinMode(buttonPin, INPUT_PULLUP);
  la a0, buttonPin
  lw a0, 0(a0)
  li a1, 0x05  # INPUT_PULLUP
  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, pinMode
  lw ra, 0(sp)
  addi sp, sp, 4

  # pinMode(ledpin, OUTPUT);
  la a0, ledpin
  lw a0, 0(a0)
  li a1, 0x03  # OUTPUT
  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, pinMode
  lw ra, 0(sp)
  addi sp, sp, 4

  jr ra
```

Step 2: Reading the button Once set up, we start the infinite loop by reading the button state using `digitalRead`, to which we only need to pass the button's pin (in this case, pin 6).

In our case, if it detects that the button is pressed, the `button_pressed` function will be called. Otherwise, the LED will remain off. A small `delay` is added to avoid overloading the system:

```
.data
  time:  .word 100

.text
loop:
  la a0, buttonPin
  lw a0, 0(a0)
  addi sp, sp, -4
```

```
  sw ra, 0(sp)
  jal ra, digitalRead
  lw ra, 0(sp)
  addi sp, sp, 4

  mv t0,a0

  li t1 ,0 # LOW

  beq t0, t1, button_pressed

  la a0, ledpin
  lw a0, 0(a0)
  li a1, 0x0
  jal ra, digitalWrite

  la a0, time
  lw a0, 0(a0)
  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, delay
  lw ra, 0(sp)
  addi sp, sp, 4

  j loop
```

Step 3: Turning the LED on As shown in Example 1, we will turn on the LED when button is pressed:

```
button_pressed:
  la a0, ledpin
  lw a0, 0(a0)
  li a1, 0x1
  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, digitalWrite
  lw ra, 0(sp)
  addi sp, 4

  la a0, time
  lw a0, 0(a0)
  addi sp, sp, -4
```

```
  sw ra, 0(sp)
  jal ra, delay
  lw ra, 0(sp)
  addi sp, 4

  jr ra
```

Using interrupts

Another way to achieve this project is by using GPIO interrupts, which are much more immediate but more complex to program.

In this case, we will use the `attachInterrupt` and `digitalPinToInterrupt` Arduino functions to obtain the interrupt number that can be assigned to the interrupt service routine for that pin.

Step 1: Setup As shown in Example 1, you need to configure the pin mode with `pinMode` before using the pins. Then we connect the button pin to an interrupt routine or ISR (which we will call `blink`) that runs automatically when the button is pressed. For this we will use `attachInterrupt` function.

The `attachInterrupt` function takes the following parameters: - The interrupt position corresponding to the pin for which we want to detect the interrupt (we use `digitalPinToInterrupt(pin)`) - The memory address where the interrupt service routine is located (in this case, we call it `blink`) - The interrupt mode, that can be one of the following:

Mode	Value	Usage
`DISABLED`	`0x00`	Interrupt disabled
`RISING`	`0x01`	Interrupt triggered on the rising edge (when the pin changes from LOW to HIGH)
`FALLING`	`0x02`	Interrupt triggered on the falling edge (when the pin changes from HIGH to LOW)
`CHANGE`	`0x03`	Interrupt triggered on any change of the pin state (both LOW→HIGH and HIGH→LOW)
`ONLOW`	`0x04`	Interrupt triggered while the pin remains LOW
`ONHIGH`	`0x05`	Interrupt triggered while the pin remains HIGH
`ONLOW_WE`	`0x06`	Same as `ONLOW`, but with write enable — allows modifications or writing to related registers while the pin is LOW
`ONHIGH_WE`	`0x07`	Same as `ONHIGH`, but with write enable

In this case, as we want the interruption when the button is pressed, we choose the `ON_LOW` mode.

This is how the `setup` function will look:

```
.data
  ledPin:       .byte 4
```

```
  interruptpin: .byte 6
  state:        .byte 0  #LOW
  on_low:       .byte 0x04

.text
setup:
  # pinMode(ledPin, OUTPUT);
  la t1, ledPin
  lb a0, 0(t1)
  li a1, 0x03  # OUTPUT
  addi sp, sp, -4
  sw ra,0(sp)
  jal ra, pinMode
  lw ra,0(sp)
  addi sp, sp, 4

  # pinMode(ledPin, INPUT_PULLUP);
  la t1, interruptpin
  lb a0, 0(t1)
  li a1, 0x05  # INPUT_PULLUP
  addi sp, sp, -4
  sw ra,0(sp)
  jal ra, pinMode
  lw ra,0(sp)
  addi sp, sp, 4

  # digitalPinToInterrupt(interruptpin);
  la t1, interruptpin
  lb a0, 0(t1)
  addi sp, sp, -4
  sw ra,0(sp)
  jal ra, digitalPinToInterrupt
  lw ra,0(sp)
  addi sp, sp, 4


  # attachInterrupt(digitalPinToInterrupt(interruptPin), blink, ON_LOW);
  la a1, blink
  la t1, on_low
  lb a2, 0(t1)
  addi sp, sp, -4
  sw ra,0(sp)
```

```
  jal ra, attachInterrupt
  lw ra,0(sp)
  addi sp, sp, 4

  jr ra
```

Step 2: ISR definition. We want to change the LED state when an interrupt is detected (the button's state changes). For that, we'll modify a variable stored in memory that indicates the LED's state, as follows:

```
blink:
  addi sp, sp, -8
  sw ra, 4(sp)
  sw t0, 0(sp)

  la t0, state
  lb a0, 0(t0)
  xori a0, a0, 1  # 0->1, 1->0
  sb a0, 0(t0)
  lw t0, 0(sp)
  lw ra, 4(sp)
  addi sp, sp, 8
  jr ra
```

Step 3: Loop The LED will be turned off until the button is pressed, as shown in the following code:

```
loop:

  # digitalWrite(ledPin, state);
  la t1, ledPin
  lb a0, 0(t1)
  li a1,0
  addi sp, sp, -4
  sw ra,0(sp)
  jal ra, digitalWrite
  lw ra,0(sp)
  addi sp, sp, 4

  # delay(100)
  li a0, 100
  addi sp, sp, -4
```

```
  sw ra,0(sp)
  jal ra, delay
  lw ra,0(sp)
  addi sp, sp, 4
  j loop

setup:

  # pinMode(ledPin, OUTPUT);
  la t1, ledPin
  lb a0, 0(t1)
  li a1, 0x03 #OUTPUT
  addi sp, sp, -4
  sw ra,0(sp)
  jal ra, pinMode
  lw ra,0(sp)
  addi sp, sp, 4

  # pinMode(ledPin, INPUT_PULLUP);
  la t1, interruptpin
  lb a0, 0(t1)
  li a1, 0x05  # INPUT_PULLUP
  addi sp, sp, -4
  sw ra,0(sp)
  jal ra, pinMode
  lw ra,0(sp)
  addi sp, sp, 4

  # digitalPinToInterrupt(interruptpin);
  la t1, interruptpin
  lb a0, 0(t1)
  addi sp, sp, -4
  sw ra,0(sp)
  jal ra, digitalPinToInterrupt
  lw ra,0(sp)
  addi sp, sp, 4

  # attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
  la a1, blink
  la t1, change
  lb a2, 0(t1)
  addi sp, sp, -4
```

```
sw ra,0(sp)
jal ra, attachInterrupt
lw ra,0(sp)
addi sp, sp, 4

jr ra
```

Example 3: Text input/output using serial output

For this use case, we are going to make a little use of basic functions from the Arduino Serial library. For more information, here is the official documentation (not all functions are available in this library, as they are not the most suitable for the environment in which we are working).

By definition, serial input functions are very fast (they do not wait for you to press Enter) and do not display a callback of what has been written. The purpose of this library is to familiarize yourself with these functions; therefore, **it is the student's responsibility to monitor each step** of what is being done.

Just as we previously used the `ecall` instruction to print messages in assembly CREATOR programs, here we will use the `serial_printf` function, which is part of Espressif's Arduino component.

Unlike higher-level functions like `Serial.print` and `Serial.println`, `serial_printf` requires explicit format specifiers to indicate the data types being printed; otherwise, the output is interpreted as a string.

On the other hand, we will use the `serial_readBytes` function to see how text input works. There is the `serial_read` variant (which only takes 1 character), `serial_readBytesUntil` (which stops storing characters when it finds the specified character), and those that parse, such as `serial_parseInt` (which only takes numbers).

Components:

- ESP32-C3-DevKitC-02 board

Step 1: Data Save messages to print in memory. For this use case we are going to declare in data section:

- The initial message to be displayed.
- The buffer allocated to store the input text.
- The message that includes a placeholder for outputting the entered text.
- An auxiliary callback used to display the entered number.

This would be reflected in the code as follows:

```
.data
    space:    .zero 100 # Buffer to place the string
    initial:  .string "Introduce number of letters:\n"
    aux:      .string "%d\nType your message\n"
    print:    .string "Your message: %s\n"
```

Step 2: Serial Start terminal output and input. On most boards, such as Arduino and Espressif, in order to use the terminal, it needs to be initialized with a specific frequency. In this case, we will use the `serial_begin` function with a baud rate of 115200.

If a value other than the one given is entered, strange characters may be printed or nothing may be printed at all.

It is declared in the code as follows:

```
setup: li a0, 115200
       addi sp, sp, -4
       sw ra, 0(sp)
       jal ra, serial_begin
       lw ra, 0(sp)
       addi sp, sp, 4
       jr ra
```

Step 3: Loop Check if the terminal is correctly open. A good practice that can be seen in Arduino is to use serial_available to see if the terminal has opened correctly.

If it has a value greater than 0, it means that it is operational and ready to use.

For simplicity's sake, you will often find that this check is not performed… but using `serial_available` is a very good error control.

So, we start our loop as follows:

```
loop: addi sp, sp, -4
      sw ra, 0(sp)
      jal ra, serial_available
      lw ra, 0(sp)
      addi sp, sp, 4
      beqz a0,aux_print
      j loop
```

'aux_print' is an auxiliary function to indicate to the user that they should enter the number only once. This is purely aesthetic but can provide clarity to the user.

```
aux_print: # serialPrintf
            la a0, initial
            addi sp, sp, -4
            sw ra, 0(sp)
            jal ra, serial_printf
            lw ra, 0(sp)
            addi sp, 4
            j read_num
```

'read_num': reads a number from terminal

In this case, we use the 'serial_parseInt 'function to get a number per terminal.

Why don't we use "serial_read"? Because "serial_read", although it returns a number, interprets the input in ASCII code.
If we enter the number '4' via the terminal, the values change:

- - In 'serial_read': 52
- - In ' serial_parseInt': 4

Since 'serial_read' is very fast, we will not move on to the next step if there is no number greater than 0 collected in a0, leaving the following code snippet:

```
read_num: addi sp, sp, -4
          sw   ra, 0(sp)
          jal  ra, serial_parseInt
          lw   ra, 0(sp)
          addi sp, sp,4
          mv   t0, a0
          bnez t0, print_int
          j    read_num
```

A small auxiliary function called `print_int` was created to view the callback of the value specified as length, as follows.

```
print_int:
  la   a0, aux
  mv   a1,t0
  addi sp, sp, -4
  sw   ra, 0(sp)
  jal  ra, serial_printf
  lw   ra, 0(sp)
  addi sp, 4
  j read_function
```

The `read_function` reads the text of the requested length. Once we have the length of our string, we collect the text we want to add with `serial_readBytes`, which is blocking until the requested length of characters has been reached.

We would end up with an implementation like this:

```
read_function:
    la a0, space
    mv a1, t0 # number of letters it will have
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, serial_readBytes
    lw ra, 0(sp)
    addi sp, sp, 4
    bne t0, a0, read_function
    la a0, print
    la a1, space
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, serial_printf
    lw ra, 0(sp)
    addi sp, 4
    jr ra
```

And this is the result!

Example 4: Daytime running lights with `analogRead`.

This time, we are going to make a small LED that lights up when it detects low light, using an LDR.

The sensitivity values of the sensor may vary depending on the location of the student and their device. We recommend having a flashlight or something that gives off light handy and using the debugger or prints.

Components:

- ESP32-C3-DevKitC-02 board
- LDR or photoresistor (in GPIO 2)
- LED (in GPIO 4)
- 10k Ω resistor

To find resistors with the correct value, look at the color code on their bands. You can either look at the color guide, use this calculator, or simply look for a resistor that matches the one in the photo.

Step 1: Setup Initialize LDR, pin, and serial output. To use the sensor, we must initialize it with `pinMode` to the value of INPUT (0x01) as we saw in Figure 2.1. The setup code would look like this:

```
.data
    lightSensorPin: .word 2
    ledPin: .word 4
    time: .word 100
    aux_msg: .string "LDR value: %d\n"

.text
setup:
  # Serial
  li a0,115200
  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, serial_begin
  lw ra, 0(sp)
  addi sp, sp,4

  # PinMode LED
  la t1, ledPin
  lw a0, 0(t1)
  li a1, 0x03 # OUTPUT
  addi sp, sp, -4
  sw ra,0(sp)
  jal ra, pinMode #pinMode(ledPin, OUTPUT);
  lw ra,0(sp)
  addi sp, sp, 4

  # Light sensor pin
  la t1, lightSensorPin
  lw a0, 0(t1)
  li a1, 0x01
  addi sp, sp, -4
  sw ra,0(sp)
  jal ra, pinMode# pinMode(lightSensorPin, INPUT);
  lw ra,0(sp)
  addi sp, sp, 4

  jr ra
```

Step 2: Loop Analog sensor reading Once everything is up and running, we read the light level in the room using `analogRead`.

In this tutorial, after a few tests, we found that the values reached were **between 1100 and 1400**. The darker it is, the higher the resistance value will be. **Therefore, a threshold of 1200 is set.**

- If the sensor reaches a value higher than 1200, the LED will turn on (`lightUp`).
- If, on the other hand, it does not reach that value, the LED remains off (`turnDown`).

In both cases, we add a small `delay`so that the readings are not so fast.

We end with this code:

```
    # delay
    la a0, time
    lw a0, 0(a0)
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, delay
    lw ra, 0(sp)
    addi sp, sp, 4

loop: # read LDR
     addi sp, sp, -4
     sw ra,0(sp)
     jal ra, analogRead
     # analogRead(lightSensorPin);
     lw ra,0(sp)
     addi sp, sp, 4

    # Print value
    mv t1, a0
    mv a1,a0
    la a0,aux_msg
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, serial_printf
    lw ra, 0(sp)
    addi sp, sp, 4

    # turn up led
    li t0, 1200 #Minumun value
    bgt t1, t0, lightUp
    blt t1, t0, turnDown
```

```
j loop
```

And this is the result:

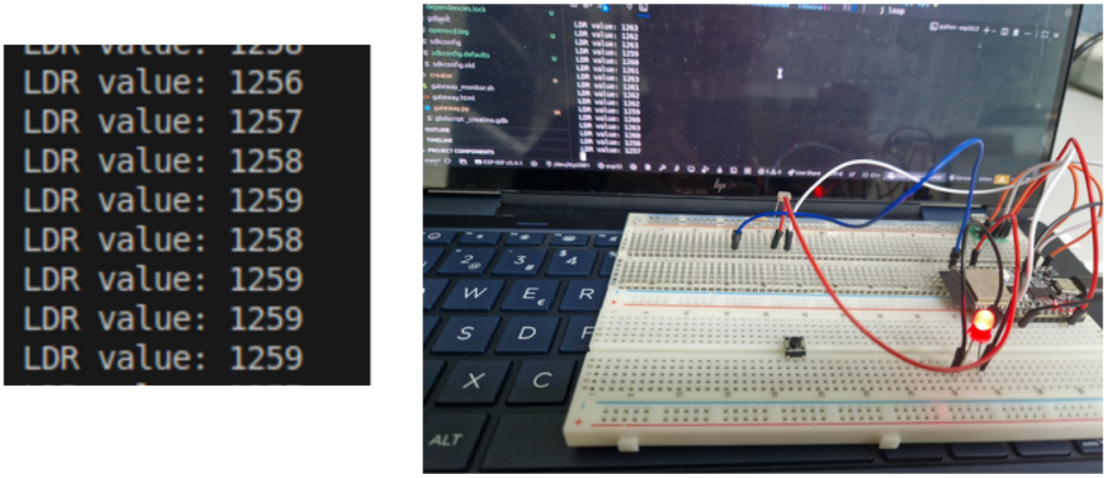

Figure 34: Arduino example 4.

Example 5: [Advanced] Piano

This case is a little more complex than the rest, but very creative. It is recommended to have a breadboard with plenty of space.

Each musical note is a frequency that blows air into the buzzer membrane. Consult [this page](https://www.tibotinspirationlab.com/reto/musica-con-arduino/) to learn how to calculate the other possible notes according to the desired tempo.

In this case, we are going to do it with interruptions. This example may cause a little more lag because there are continuous memory accesses and the “tone” function launches tasks.

Polling could be used.

Components:

- ESP32-C3-DevKitC-02 board
- 4 buttons
- A passive buzzer

Step 1: Data Declaration of pins and values of musical notes. We indicate in the data the pins we are going to use (avoiding ”dangerous” pins such as ‘GPIO8‘ and ‘GPIO1‘) and the frequency values of the musical notes.

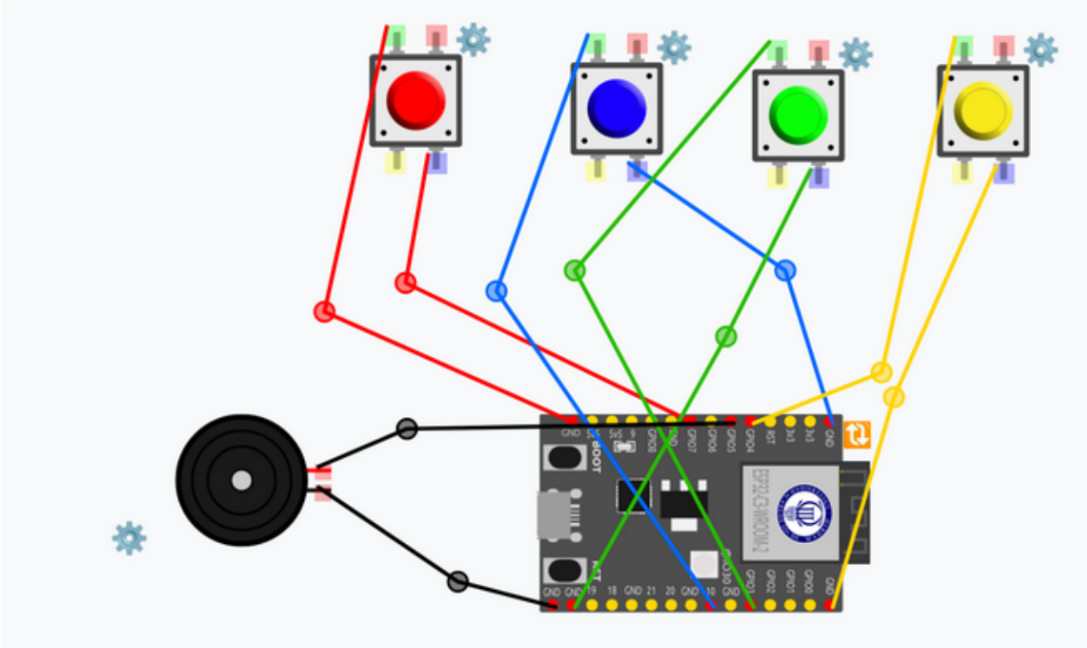

Figure 35: Arduino setup.

Care must be taken with the position of these values, as we are going to use these memory positions as arrays.

In this example, we have positioned them as follows:

```
.data
# GPIO
buzzerPin: .word 5
button_C4: .word 7
button_D4: .word 10
button_E4: .word 3
button_G4: .word 4

# Notes
note_C4: .word 262
note_D4: .word 294
note_E4: .word 330
note_G4: .word 392

#aux
button_count: .word 4
time_delay: .word 10
anyPressed: .word -1
.align 4
```

```
button_handlers:
    .word handleButton0
    .word handleButton1
    .word handleButton2
    .word handleButton3
```

Another way to position them is like this:

```
.data
# GPIO
buzzerPin: .word 5
buttons:   .word 7,10,3,4

# Notes
notes:     .word 262,294,330,392

# aux
button_count: .word  4
anyPressed:   .byte -1

.align 4
button_handlers: .word handleButton 0
.word handleButton 1
.word handleButton 2
.word handleButton 3
```

Step 2: Setup Configure all pins In this case, we are going to complicate things a little, as there are a lot of pins to configure. First, we will configure pin 5 of the buzzer, which, unlike the buttons, will use an OUTPUT mode (since it emits a sound to the outside).

```
setup:
  # buzzer
  la a0, buzzerPin
  lw a0, 0(a0)
  li a1,  0x03 #OUTPUT
  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, pinMode
  lw ra, 0(sp)
  addi sp, sp, 4
```

Then, to configure the rest of the buttons, we created a loop that goes through all the selected buttons.

First, in setup, after initializing the buzzer, we initialize the variables in our loop:

```
  la   s0, button_C4
# Button list
  li s1, 0

# Position in the list
  la   t3, button_count
  lw   s2, 0(t3)

# List length
```

Next, we move on to the `loop_buttons` function, where we will:

- Check if we have already gone through the entire list

```
loop_buttons: bge s1, s2, end_loop_buttons # loop until all the
    buttons are positioned
```

- If this is not the case, we scroll through the data to find the pin number we want.

```
# Take position of the button
slli s3, s1, 2
add t1, s0, s3 # shift
lw s4, 0(t1) # Button value mv a0,s4
```

- Once the position has been obtained, we configure the button with `INPUT_PULLUP` mode, as they are buttons.

```
li a1, 0x05 # INPUT_PULLUP
addi sp, sp, -4
sw ra, 0(sp)
jal ra, pinMode
lw ra, 0(sp)
addi sp, sp, 4
```

- As we indicated in the statement, we are going to use interrupts, so we have to assign them now knowing which button we are on (that is, we take advantage of the fact that the order of `button_handlers` is the same as the order followed by the button array). For more information on how to assign interrupts, see example 2. We will indicate what the ISRs are like in the next point:

```
# Attach Interrupts
# Transform digitalPin into interrupt
mv a0, s4
addi sp, sp, -4
```

```
sw ra,0(sp)
jal ra, digitalPinToInterrupt #
    digitalPinToInterrupt(interruptpin);

lw ra,0(sp)
addi sp, sp, 4 # Search the correct pointer la t0, button_handlers
add t1, t0, s3
lw a1, 0(t1) # Let the interrupt jump when button is pressed
    (on_low)
li a2, 0x03
addi sp, sp, -4
sw ra,0(sp)
jal ra, attachInterrupt #
    attachInterrupt(digitalPinToInterrupt(buttonPin[i]),
    handler[i], ON_LOW);
lw ra,0(sp)
addi sp, sp, 4
```

- We add up a position and re-enter the loop

```
addi s1, s1, 1 # next button
j loop_buttons
```

- If we have finished going through the list, we return to main by executing a ret (or a `jr ra`).

```
end_loop_buttons: ret
```

Step 3: Loop This step can be done in two ways: with polling (i.e., constantly checking the status of all buttons) or with interrupts (pressing the button triggers an interrupt). The easiest way to do this exercise is to assign an interrupt to each button and, when pressed, have each one have its own ISR. Therefore, before starting the loop, we will create each of the necessary ISRs and assign them to the buttons. Refer to case 2 using interrupts for a better understanding.

In this case, we use a "flag" change since the `tone()` function underneath launches "tasks" or "processes" underneath, which is not safe to do in an ISR.

```
# ISR handleButton0:
    la t0, anyPressed
    lw t1, 0(t0)
    li t2, 0 # Assign button
    sw t2, 0(t0)
    jr ra

handleButton1:
```

```
    la t0, anyPressed
    lw t1, 0(t0)
    li t2, 1 # Assign button
    sw t2, 0(t0)
    jr ra

handleButton2:
    la t0, anyPressed
    lw t1, 0(t0)
    li t2, 2 # Assign button
    sw t2, 0(t0)
    jr ra

handleButton3:
    la t0, anyPressed
    lw t1, 0(t0)
    li t2, 3 # Assign button
    sw t2, 0(t0)
    jr ra
```

Then, inside the loop, we check the status of the `anyPressed` flag and add a small `delay` so that the watchdog does not trip.

This loop checks that, if the flag is not at -1, the tone corresponding to that position sounds.

```
loop: la t0, anyPressed
      lw s0,0(t0)
      li t1, -1
      bne s0, t1, startTune
      la t0, time_delay
      lw a0, 0(t0)
      addi sp, sp, -4
      sw ra,0(sp)
      jal ra, delay # delay(200);
      lw ra,0(sp)
      addi sp, sp, 4
      j loop
```

Step 4: Start Tune Start playing the tone.

In this case, once we have the position of the button that has been pressed, we search the memory array for the corresponding note value, similar to when we searched for the button.

To prevent the tone from having an indefinite duration, we have set a timeout of 200. Once the

tone is played, it returns to the loop.

```
startTune:
  # Clean value
  la t0, anyPressed
  li t1, -1
  sw t1, 0(t0)

  # Search value
  la t0, note_C4
  slli s1, s0, 2
  add t1,s1,t0
  lw s2, 0(t1) # Note value

  # Play Tone
  la t0, buzzerPin
  lw a0, 0(t0)
  mv a1,s2
  li a2, 200

  # tone(buzzerPin, notes[i], duration);
  addi sp, sp, -4
  sw ra,0(sp)
  jal ra, tone
  lw ra,0(sp)
  addi sp, sp, 4

  j loop
```

USER MANUAL AND USE CASES FOR THE SAIL-BASED MODEL

In this chapter, we will present the user manual for the new Sail-based simulator engine, using a simple example program, and show different use cases for the tool.

User manual

This user manual describes the different functionalities integrated into the new RISC-V simulator based on the Sail specification language.

Init page

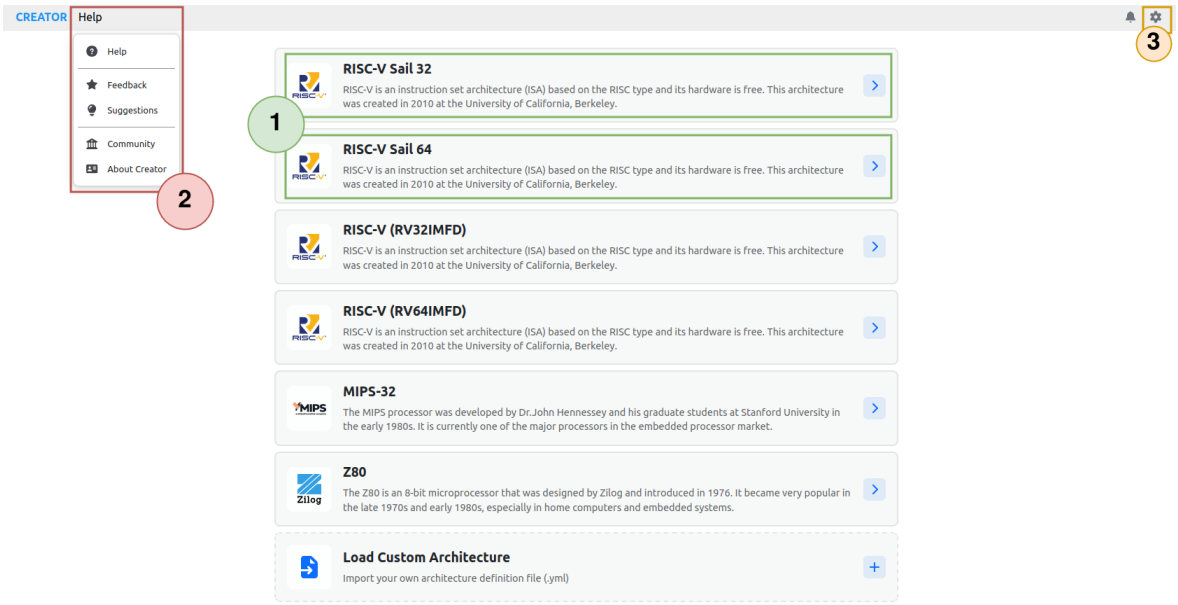

Figure 36: Simulator home page

The simulator's home page lists the available architectures for simulation, as shown in Figure 36.

First, in the central area, two variants of the RISC-V architecture are implemented with Sail. Remember that for each module, the complete set of instructions for each architecture is available. You can access the simulator directly by selecting the architecture variant to be used, as shown in the highlighted area **1**. On the other hand, we have the help menu in case any error or unwanted behavior is identified while using the tool, as shown in the highlighted area **2**. Finally, there is a configuration menu where you can edit the simulator's characteristics, as shown in the highlighted area **3**.

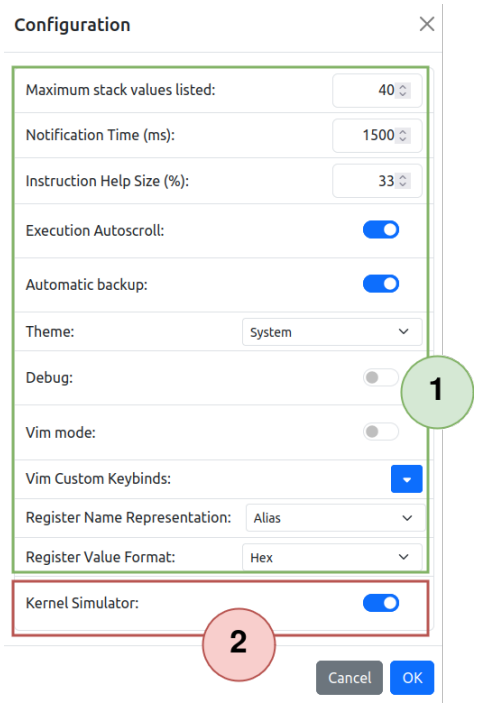

Figure 37: Módulo de configuración del simulador web

The Figure 37 corresponds to the simulator configuration menu. This menu allows the user to configure different aspects of the simulator to adapt its execution to their needs. The highlighted area **1** shows additional visual aspects of the simulator configuration, such as displaying messages during debugging, the code editor, and how records are represented, among others. The highlighted area **2** allows the user to select whether the user wants to use the kernel implementation integrated into the simulator itself or a kernel developed by the user for program execution.

Once the simulator is configured through the modules presented, it proceeds to select the architecture, as shown in the highlighted area **1** of Figure 36, and accesses the simulation module.

Simulation module

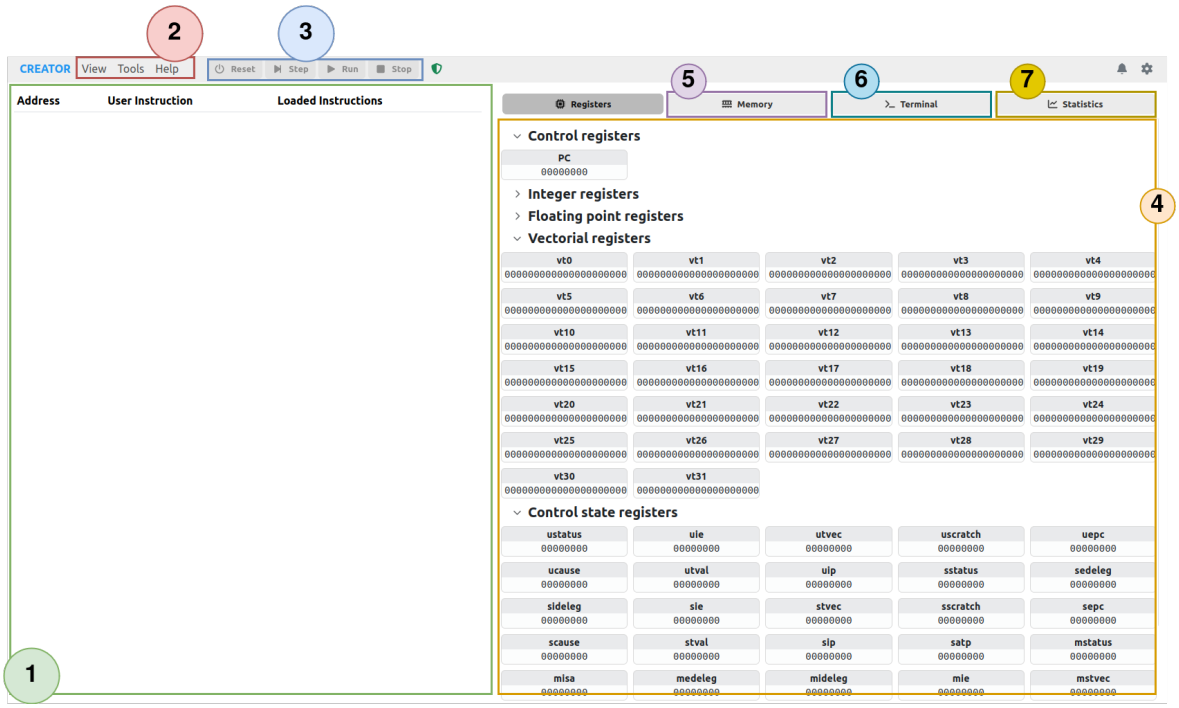

Figure 38: Módulo de simulación del simulador Web

As shown in Figure 38, different sections are identified within the simulator's execution module. Moving from left to right, the module's elements are identified.

On the left side, different areas are identified:

1. Instruction execution and debugging area: this section shows the coding and address of the compiled assembly instructions, allowing you to track the current status of the architecture. Also, the user can set a breakpoint on a specific instruction to stop execution and debug its behavior if the program was run without interruptions, not step by step.

2. Simulator navigation bar: allows you to navigate the simulator's views and elements. In the *View* section, you can navigate to the code editing module or the architecture information module; the *Tools* section allows you to load examples of the architecture, use the calculator, or use the remote service to flash on a microcontroller; and the *Help* section, as shown in Figure 36, allows you to access the simulator's help and contact module.

3. Simulator execution bar: This allows the user to execute their programs step by step, run them without interruptions, stop execution, and reset execution.

On the right side, the following areas are identified:

4. Architecture status: Displays the current state of the architecture's registers. On each register, you can see the value stored in the register and view it in different representations (hexadecimal, decimal, IEEE 754, etc.).

5. Architecture Memory View: Displays the contents of the Memory and is updated during program execution.
6. Terminal View: shows the program's input and output during execution.
7. Statistics View: shows the program's execution statistics, displaying which instruction types are executed.

In the Tools area, there is a button that opens a menu of programs to run in the simulator. Its function is to learn about programming in the simulator and understand the architecture's functionalities, as shown in Figure 39. This menu can be accessed via the Tools button in Figure 38.

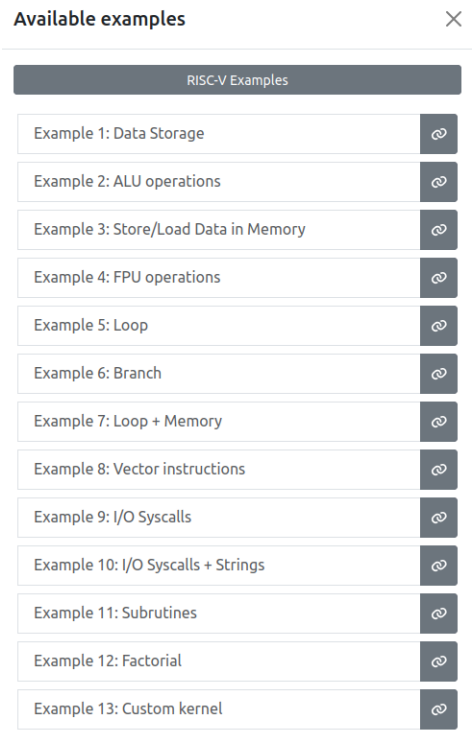

Figure 39: RISC-V program examples menu.

In the same tools area, there is a button that redirects to an IEEE 754 calculator/converter between decimal, hexadecimal, and single and double precision floating point values, in 32 and 64 bits, as shown in Figure 40. It can be accessed through the tools menu, as shown in Figure 38.

Figure 40: Calculator/converter IEEE 754 of decimal, hexadecimal, simple, and double precision values .

The architecture area shows the user the configuration set for that variant in the simulator, as shown in Figure 41. You can navigate through it to see the instructions available to the user within the simulator. As shown in the highlighted area **1**, the user can view the instructions, pseudo-instructions, directives, and registers available in the architecture. Finally, in the highlighted area **2**, the user is shown a summary of the configuration established in the simulator, including the word size, the name of the main function, memory alignment, parameter-passing convention, etc.

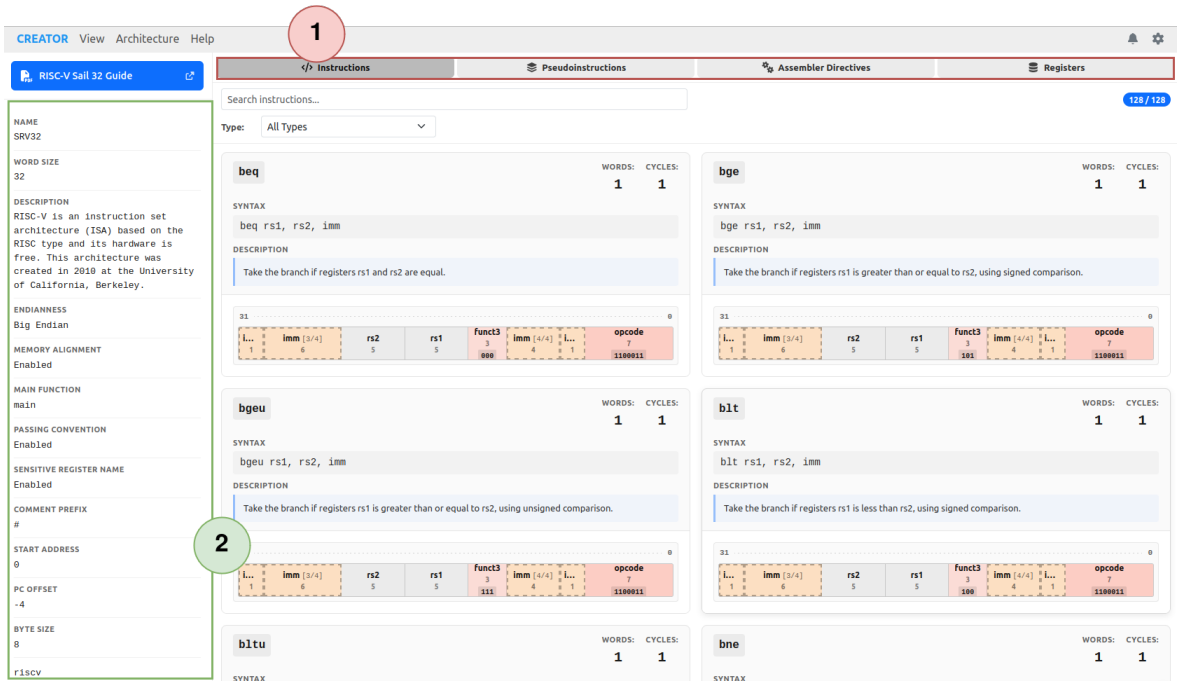

Figure 41: 32-bit RISC-V architecture configuration.

Code editor module

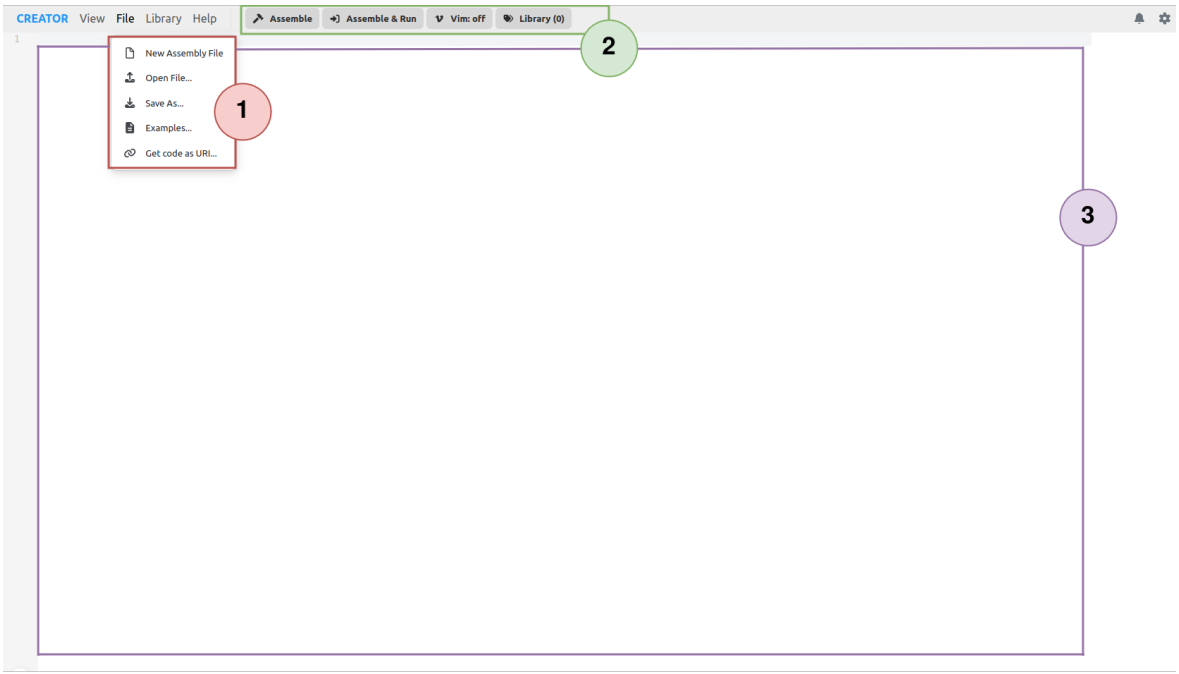

Figure 42: Program coding module.

As shown in Figure 42, different parts or modules are identified in the editor: program coding and compilation (highlighted area **2**), file creation, loading, and downloading (highlighted area **1**); and library creation, loading, and downloading (*Library* menu button).

Files can be created using the "New File" button in the highlighted area **1**. In addition, this menu can be used to load files that have been previously coded for the project, load examples such as those in Figure 39, and even download files coded in the editor (highlighted area **3**). After creating or loading files in the editor, a message will appear in the code editing view to indicate whether the operation was successful.

For libraries, there is a menu for creation and loading (*Library* menu options). Once the program with which you want to generate the library has been coded (**it is important that the program does not contain the (*main*) functionality, otherwise the library cannot be generated**), it is compiled and, once the user is notified that it has been compiled correctly, the tool is asked to generate a new library. This option downloads the created library to the user's device. On the other hand, to load libraries, use the "Load" option in the library menu and select the libraries to be included in the editor. Once the library has been loaded correctly, a message will be displayed on the screen to notify the user.

For coding, the contents of the file will be displayed. Once the files required to generate the program or library have been encoded, proceed to compile and generate the binary to be executed by pressing the compilation button in the highlighted area **2**.

Once the program has been generated, you can download it as a library (if it does not contain the

program's main functionality) or access the simulation module to execute it.

Program execution in the web simulator

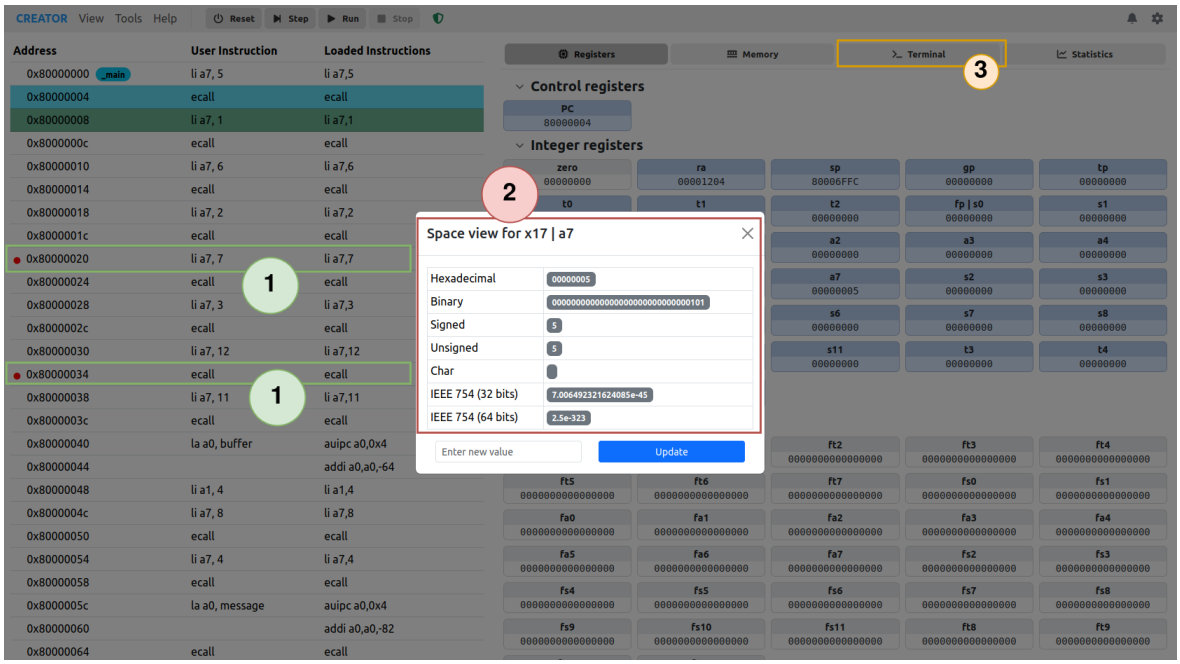

Figure 43: Execution of a program that makes system calls for the user to introduce data.

Figure 43 shows the execution of a program that contains system calls. The system calls executed in this program prompt the user to enter, via the terminal input module, integer values, single- and double-precision floating-point values, a character, and a *string*. The user inserts them as execution progresses, and the system call interrupts execution. As the user inserts them through the terminal input module (within the highlighted area **3**), the values entered are printed by the terminal output.

Highlight the "breakpoints" in the execution, marked by the user as shown in the highlighted area **1**. These breakpoints are used to stop execution at the marked instruction if, for example, you want to run the program without interruptions, but you want the simulator to stop execution when it is about to execute the instruction. This behavior is known as a debugging module or execution debugging.

Finally, it should be noted that the contents of the affected registers during execution, as shown in the highlighted area **2**, reflect the changes that have occurred in both the registers and memory regions affected by the execution at the moment when execution was stopped. In this case, it is correctly shown that the value "5" has been loaded into register **a7**, which specifies the type of operation to be executed when making a system call (*ecall*). This call prompts the user for an integer value via the terminal.

For a better understanding of system calls and other instructions used in the program, it is recommended to consult (while the coded programs are running) the information on system calls and other coded instructions. This information can be found in the simulator documentation provided to

support the user.

Use cases with the new functionalities of the simulation tool.

This section will address different use cases of the simulation tool that show the functionalities of the new Sail-based simulator and the compiler tool.

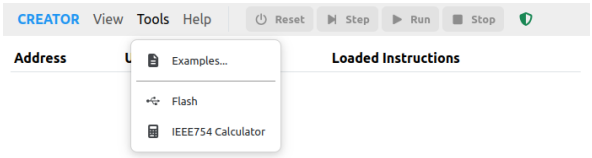

Figure 44: Examples menu.

Notice that the code of use cases was in the examples module in the tools menu, as shown in Figure 44. The last use case was with a custom file that it will be shown in their section.

Execution of a program with a custom kernel coded in assembly language.

For this use case, the "*Example13.s*" of the examples menu. This code it is an implementation of a custom kernel that shows the user how to implement a kernel. This example can be modified in the Editor's view to explore the simulator tool's full capacity.

It is important to note that the user must disable the simulator's kernel to apply their own kernel implementation. And in case the user does not want to use the example kernel, their main function of the kernel should be called "kernel". The reason is to simplify the tool if it must start at the "main" function or at "kernel" function.

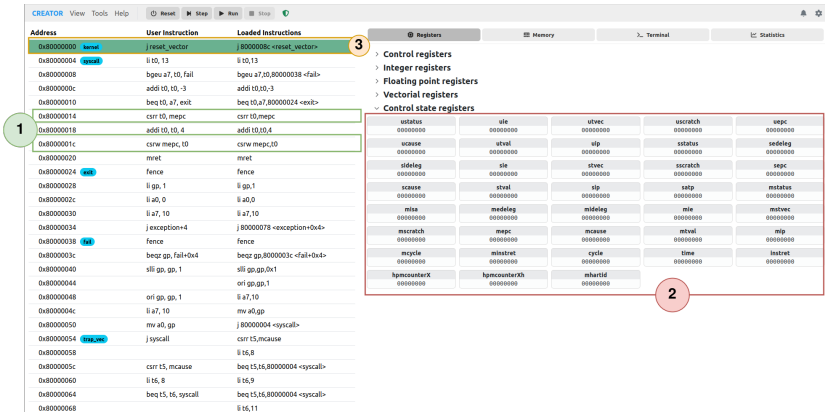

Figure 45: Execution of a custom kernel.

As shown in Figure 45, in the highlighted section **1**, a program has been generated that modifies the control and status file register, allowing us to establish the desired architecture configuration before entering the main function. These modifications will be reflected in the file register in the highlighted area **2**, such as the address to which it jumps when an interrupt or exception occurs; the address to which it will jump when returning from one execution mode to another (from machine mode to user

mode, for example); or from which execution mode it comes from at the moment the interrupt or exception occurs. Finally, the highlighted area **3** shows the start address of execution, which in this case is the kernel, not the main function.

Execution of a program with vector instructions.

For this use case, use the "*Example8.s*" from the examples menu. This code it is an implementation of how to load, operate, and store vectors. This example can be modified in the Editor's view to test with other vector data types, for example, floats, integers, bytes, or halfs.

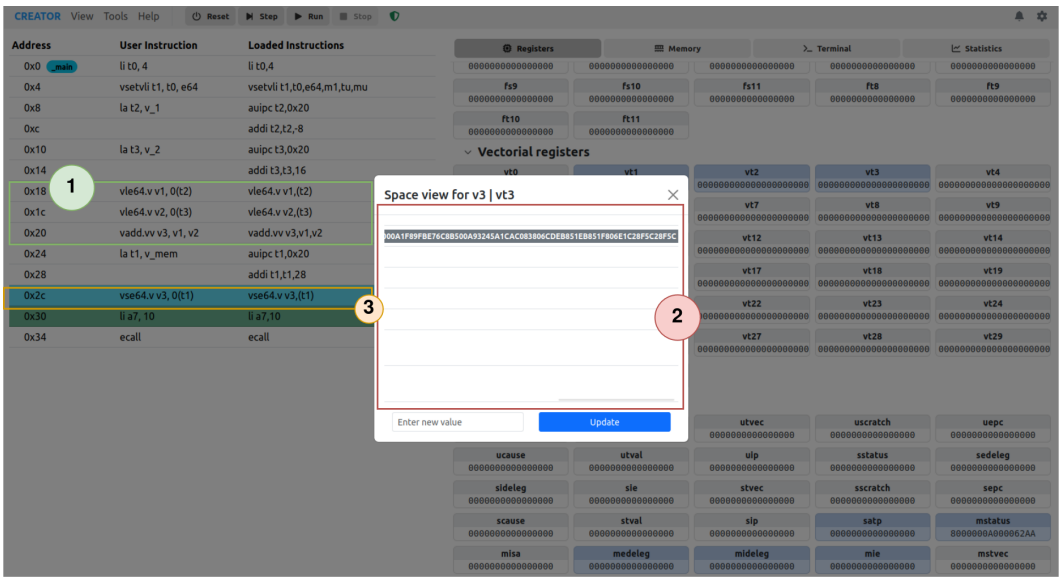

Figure 46: Program execution with vectors.

As shown in Figure 46, in the highlighted section **1**, the program execution is computed as the sum of two vectors y, the result of which is stored in the highlighted area **2**. The hexadecimal value shown is the result of the sum. However, the result will be stored at the memory address specified by the storage instruction in the highlighted area **3**. Once stored in memory or the execution has finished, the result value can be checked in the memory view.

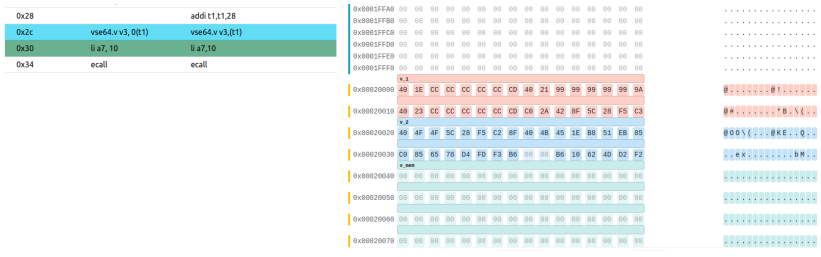

Figure 47: Result vector values stored in memory.

As shown in Figure 47, the result is successfully stored in memory in "v_mem" address assigned to store the result vector.

Execution of a program compiled with a library which was previously generated.

For this use case, two files have been used for the development.

```
# auxiliar progam code
.section .data
.align 4
pi:
    .float 3.141516

.section .text.init
.globl circumference_area

circumference_area:
    la t0, pi
    flw ft0, 0(t0)
    fcvt.s.w ft1, a0
    fmul.s ft2, ft1, ft1
    fmul.s fa0, ft2, ft0
    li a7, 2
    ecall
    jr ra
```

Figure 48: Auxiliary assembly program which calculate the area of a circumference.

```
# main program code
.section .data
.align 2
radio:
  .word 8

.section .bss
.align 8
tohost: .dword 0

.section .text.init
.globl _main

# Complete your main function here
_main:
  la t0, radio
  lw  a0, 0(t0)
    call circumference_area
    li a7, 10
    ecall
```

Figure 49: Main assembly program which call the function of the library.

The listing file in Figure 48 is used to generate a library that is downloaded after compilation and then loaded into the simulator. The listing file in Figure 49 is used to call the program's main function, and within the main function, the function from the library is called.

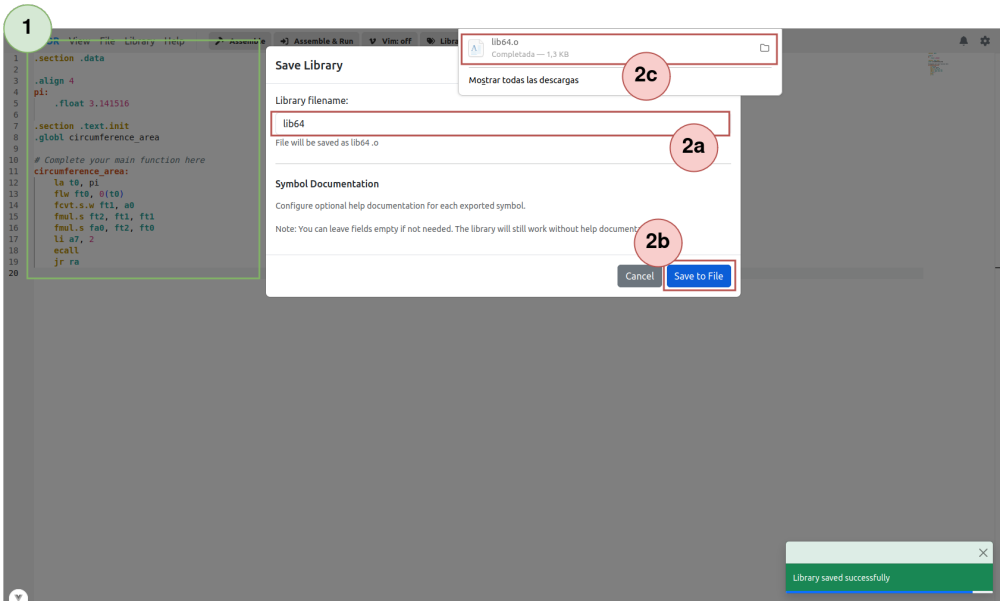

Figure 50: Compile and save assembly code of the library.

In Figure 50, the library (highlighted area **1**) is coded, which does not contain the main function of the program, since the library cannot contain the main function of the program. Once compiled, the library is created and downloaded (*Library* menu button) as shown in the highlighted areas **2**.

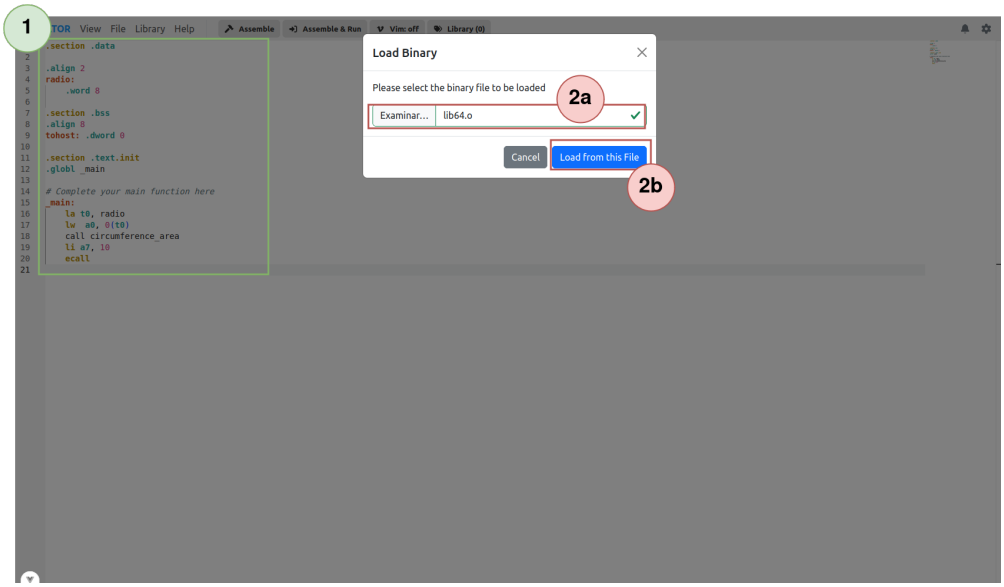

Figure 51: Import library to compile with main function.

Once the library has been generated, we proceed to import the library and encode the program, as shown in Figure 51. To do this, we select the "Load library" option from the *Library* menu and load the previously downloaded library (highlighted areas **2**). Once loaded, replace the program code with that of the main function that calls the library function (highlighted area **1**), compile the program, and open the simulation view to run it.

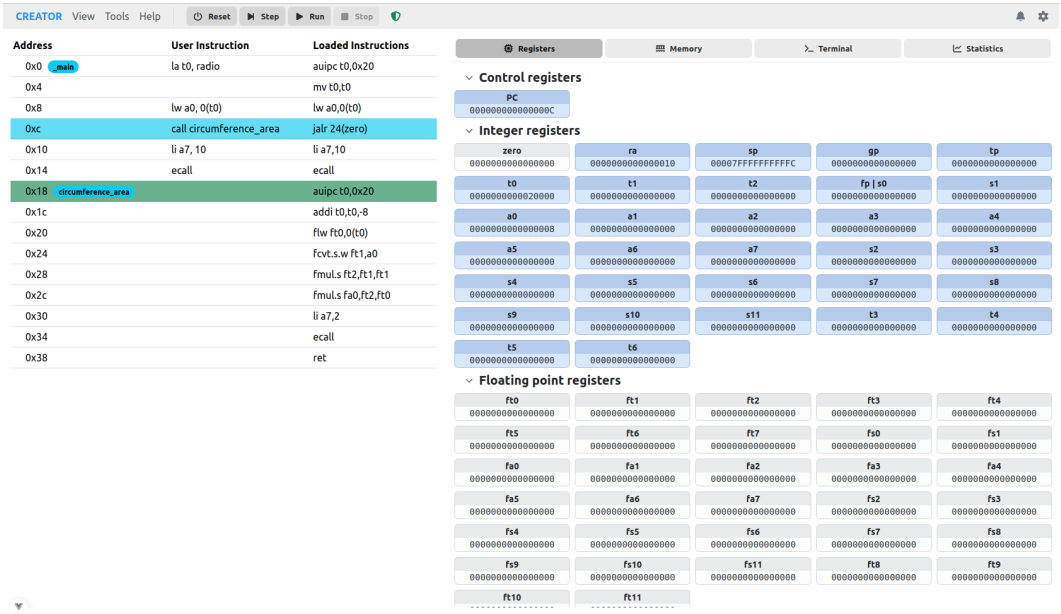

Figure 52: Execution of a binary generated with a library.

In the simulation view, the program is executed and, as shown in Figure 52, the simulator correctly executes the call from “main” to the function that calculates the area of a circle.