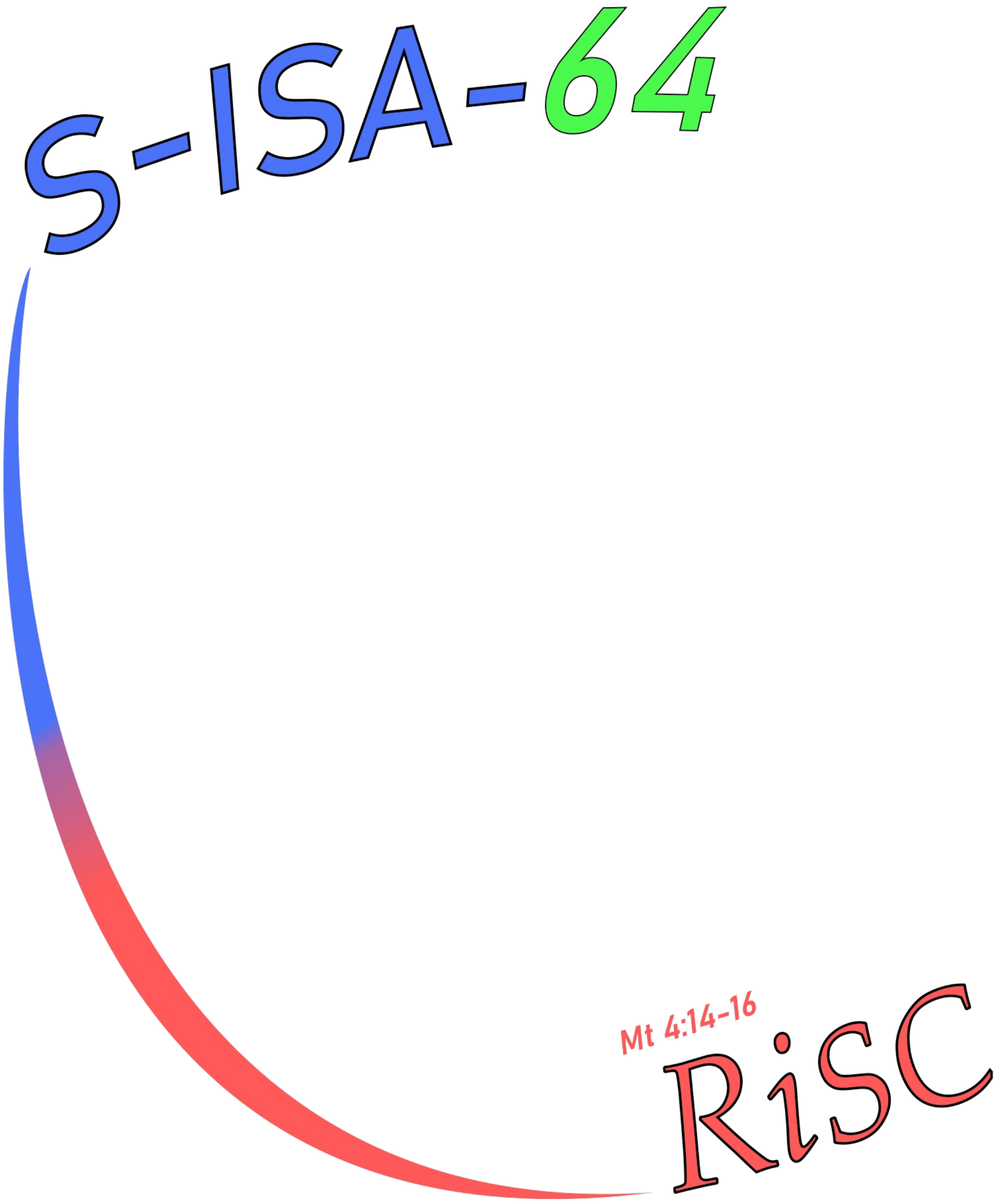
S-ISA-64 RISC ISA
Programmer’s Reference



Contents

Architecture Synopsis

S-ISA-64 (Often shortened to SISA64) is a reduced-instruction-set computer architecture, with an 8 bit opcode, and an opcode-specific number of individual byte arguments.

There are 256 general purpose registers, a program counter, and a stack pointer, all of which are 64 bits wide.

Main Memory is byte-addressable. I.e. each byte has a 64 bit address.

Values larger than a single byte are stored Big-Endian in memory.

There is an additional address space called the “device bus” which has a 64-bit addressable, 64-bit address space (As in, each address is 64 bits wide, and, reading or writing to a particular address will read or write 64 bits at once) The device bus offers a simple memory mapped I/O. It is used for interacting with physical components on the system.

The processor offers 2 main modes of execution- privileged and unprivileged.

Privileged mode is what the processor boots into and allows execution of all instructions and access to all of memory. Privileged code is trusted enough to be able to crash the implementation, and can set up invalid processor states. The provided emulator may crash if those states occur.

Unprivileged mode is effectively sandboxed, has a limited view of memory, and while it can execute a majority of the ISA, there are many things it cannot do.

Privileged mode code is what you would write a kernel in, and unprivileged mode code is what you would write a userland (Sandboxed, running on an operating system) application in.

Comparison with other ISAs

SISA64 has a single-byte-opcode, multi-byte-operand system very similar to old MOS 6502 processors. The memory model is a flat 64 bit address space. There is no alignment, segmentation, paging, or complicated address translation lookups in SISA64.

SISA64 is unusual in that every single instruction mnemonic directly corresponds with one and only one opcode. And the opcode determines the addressing mode used.

The opcode decoding logic is extremely simple, only needing to match an opcode of fixed size.

The assembly language is, in principle, equally simple. The assembler knows how to write bytes to a file- not anything else. Lines from the file are repeatedly processed by text replacement to achieve every complex or intricate stage of the assembly, and at the end all the assembler has is a list of “bytes”, “shorts”, “longs”, and “qwords” commands.

The ISA was designed to be fast and easy to implement while also being easy to write code for and fast to emulate in a portable manner.

Design Decisions

Most critical design decisions in the architecture were made for emulation speed or ease of programming.

SISA64 has an offset+max system for its usermode MMU because it is extremely fast to emulate and easy to implement.

SISA64 has a stack that grows upward because it is conceptually easier to remember than a stack that grows downward.

SISA64 has a big endian architecture because it is easier to tell what the bytes in memory are for numbers when reading assembly code than little endian… it also made writing the assembler slightly easier.

Instruction Set

Here is a compact data sheet for every instruction:

(hlt|nop)	;
become_user	*rid_off*, *rid_max*, *rid_rfaddr*, *rid_insns, saved_regs*
dread	*rid_addr*
dwrite	*rid_addr*, *rid_val*
syscall	;
imXX	*rid*, **literal**
(ld|st)XX	*rid*, **addr**
(ild|ist)XX	*rid*, *rid*
mov	*rid_dest*, *rid_src*
(lsh|rsh|and|or|xor)	*rid_dest*, *rid_src*
(compl|neg|bool|not)	*rid_dest*
(i|f|d)(add|sub|mul)	*rid_dest*, *rid_src*
(u|i|f|d)div	*rid_dest*, *rid_src*
(u|i)mod	*rid_dest*, *rid_src*
seYY	*rid_dest*
(u|i|f|d)cmp	*rid_dest*, *rid_compareme*, *rid_compareme*
(jmp|call)	**addr**
jnz	*rid_testme*, **addr**
ret	;
(get|set)stp	*rid*
(push|pop)XX	*rid*
ito(f|d)	*rid*
(f|d)toi	*rid*
(inc|dec)	*rid*
(ld|st)spXX	*rid*, **off32**
bswapZZ	*rid*
mnz	*rid_testme*, *rid_dest*, *rid_src*

Where XX = one of (64,32,16,8) determining how large a load/store/push/pop is happening.

Where YY = one of (32,16,8) determining the size of the integer in the register to sign extend.

Where ZZ = one of (64,32,16) determining the size of a value to perform a reverse byte reordering.

Where **addr** = an 8 byte big-endian unsigned integer memory address,

Where **literal** = an XX bit value, being XX/8 bytes long, which is to be loaded into a register.

Where *rid* = an 8-bit unsigned integer value representing the ID of a general purpose register, from 0-255. Anything after “rid” (such as *rid_testme*) indicates how the register is used in the instruction. If it is a “dest” then the result of the operation is always written to that register.

Where if there is a semicolon after the instruction name, it indicates that it has no arguments and the opcode plus arguments length is thus a single byte.

Where off32 = a 4 byte unsigned integer stack-relative offset.

Instruction Set Details

* All math operations operate on 1,2, or 3 registers, and the first one specified will always be the destination.
* Almost all instructions use a (destination, source) argument pattern. The only instruction mnemonics which break the (destination , source) pattern are stXX (st64, st32, st16, st8), which take a register id to store before the address to store them to.
* d stands for double (64 bit floating point) and f stands for float (32 bit floating point)
* u stands for unsigned, i stands for integer (signed)
* compare instructions write the result of the comparison into the dest register, and use the two source registers in the comparison.
* Compare instructions return -1 (complement of zero) for less than, 1 for greater than, or zero for equals. Floating point comparisons with infinity may result in erroneous equals comparisons.
* mod and div instructions may result in an error code being generated due to a math error. Dividing by zero is never valid.
* become_user, syscall, dread, and dwrite are too complicated to explain in this portion. They have their own chapters. **See “Device” for dread and dwrite. See “User Mode” for become_user and syscall.**
* ldspXX and stspXX read and write using a 32 bit unsigned integer offset from the stack pointer, rather than a 64 bit offset.

Execution Flow

When the implementation first starts, it copies a program image from a ROM into memory at address zero.
The device bus is initialized (di() in the emulator) and all registers are cleared.
Execution begins with all registers, including the program counter, set to zero.
The processor starts in **Privileged Mode**, which has full access to the system.
After some amount of execution, it will normally invoke “become_user” and enters **User Mode**. At this time, the privileged process’s program counter and stack pointer are saved. User mode does not have access to the privileged instructions and only has a small, finite number of cycles before it is preempted and control flow returns to the caller. User mode’s starting registers are determined by the register file passed by pointer to become_user. This includes the program counter, the stack pointer, etcetera.

User Mode has a limited view of memory restricted by the settings passed into become_user. Its address space goes from [off, off+max] in memory, as passed into become_user, which is re-mapped to [0,max].

Once user mode reaches a state which would cause an error (0 for halt, 1 for an attempted use of privilege change instructions, 2 for being preempted, 3 for erroneous attempts to use dread or dwrite, 4 for syscall, 16 for an integer math error, and 32 for a floating point math error) the processor will store the state of the general purpose registers, program counter, and stack pointer at the time of the error into the register file passed by pointer to become_user, return control to the calling privileged code, restore the privileged view of memory, and restore the privileged stack pointer and program counter. The privileged general purpose registers before become_user are lost and most of them are filled with values that the user had in there at the time of the error code being returned.

The error code from the usermode execution is placed in register zero. The error code is then handled, and the privileged process can choose whether to become_user again (perhaps continuing a pre-empted process, or after serving a syscall) or do something else.
After this continues for some time, the processor will eventually want to shut down. It does this by either having an exception in privileged mode or the “hlt” instruction. The device is then shut down (dcl() in the emulator) and the system closes.

Device

The device is an implementation of the instructions dread and dwrite which are to provide access to the I/O and other functions of the system.

dread and dwrite are privileged instructions and may not be executed in user mode, returning ERR_DEVICE (3)

The emulator provides an example of what a typical implementation might look like.

In hardware, dread() and dwrite() would provide blocking bios-interrupt access to a secondary bus on the system.

Address 0 is the serial terminal (getchar, putchar)

Address 1 can be read to discover the system’s memory size.

Address 2 provides access to a measurement of time in milliseconds if read with dread.

User Mode

As discussed previously, SISA64 has two privilege levels, “**Privileged**” and “**User**”.

Privileged mode has access to all of the instructions and the entirety of memory.
User mode has access to most of the instructions, but not all, and has only a small window into memory defined by an offset and maximum address.
To explain how user mode works and how to use it in an operating system implementation, you should first learn how to enter user mode.

```
become_user rid_off, rid_max, rid_rfaddr, rid_insns, saved_regs
```

---

rid_off: ID of the register containing the offset address of the user’s address space. The address in this register is a physical address that defines the user mode virtual address of zero.
If you wanted to have user mode start with a window into memory that started at 0x10000 (64k into memory) then you would put 0x10000 into a register and then pass the id of that register as the first argument to this instruction.

rid_max: ID of the register containing the maximum address that usermode will be able to access during its execution.
If usermode tries to access any memory that goes beyond this maximum address (either by trying to read beyond it or straddling the boundary with a multi-byte access) then it will actually read address zero instead.
If you wanted to have usermode start with a view into memory at [0x10000, 0x1FFFF] then you would put 0x10000 into a register and pass its id as the first argument, and put 0x1FFFF into another register, and pass its id as the second argument.

rid_rfaddr: ID of the register containing the address of the register file used for keeping track of the user-mode registers.
The register file is a fixed-size array of q-words (8 byte values) holding the big-endian values of the 256 general purpose registers, the program counter, and the stack pointer in that order. Upon invoking become_user, the processor loads *from* the register file *into* the registers, and upon returning to privileged execution, it stores *to* the registerfile *from* the registers, in the previously mentioned order.
The address of the register file is saved by the processor in a register during usermode execution for this purpose.

rid_insns: ID of the register containing the number of instructions to execute in usermode before preemption. **Note that a value of zero still results in a single instruction being executed.**

saved_regs: Maximum ID of register to pass/save, switching between user mode and privileged mode. To pass and save all registers, pass “255” in. To pass and save only the first register, pass “0” in.
Note that **when become_user is invoked, any registers not set by the register file contain the values that were set by the privileged code.** It is important, then, for a secure operating system to zero-out any unsaved registers before calling become_user.
Note that this changes the size of the register file. Saving all registers results in a 2064 byte register file, but saving only 8 (0-7) results in an 80 byte register file.

Returning to Privileged Mode

After invoking become_user the user-mode program will execute code in its restricted sandbox. Eventually, though, **Something** will happen, which causes control to return to the operating system. This **Something** can be any of the following:

* An hlt instruction is reached.
* the user mode code tries to illegally “become_user”
* The user mode code uses up its instruction count budget.
* The user mode code tries to execute dread or dwrite.
* The user mode code makes a system call.
* The user mode code has an integer math error (div or mod by zero)
* The user mode code has a floating point math error

Each one of these **Something**s returns a unique error code identifier upon control returning to the operating system.
Halt: 0
Privilege Error: 1
Preemption: 2
Device Error: 3
System Call: 4
Integer Math Error: 16
Float Math Error: 32

This error code is stored in register zero (rid = 0) upon return of control to the operating system.

The contents of user-mode registers are stored in the register file whose address was passed as the third argument to become_user as soon as this event occurs.
It is thus entirely possible for the privileged process to immediately set up the registers again and “become_user” again.

Before it does that, though, it may want to serve a system call or determine if it should switch to another process (if you are writing an operating system with multitasking).

Multithreading

The emulator does not, at the time of writing, yet support emulation of a multicore processor. However, this is how it should be implemented:

In the device address space…
(Recall that address 0 was used for the blocking serial I/O, 1 for querying the available memory size, and 2 for time in milliseconds)

* Address 3 should yield the number of physical cores/processors on the machine. If the system is single-processor, then it may return either zero or one.

* Address 4 should provide the starting address of a series of device addresses which can be written to, to start another processor. These addresses are the “Processor Kickstart Addresses”. Writing to these addresses, irrespective of the value, begins execution on that processor. If the processor is already running, the behavior is *implementation defined.* Reading a kickstart address is *undefined behavior*. Kickstarting a core that is already running is *undefined behavior*. There are exactly as many kickstart addresses as there are cores.

* Address 5 should provide the number of available hardware mutual exclusion addresses for locking and unlocking to guard portions of memory. If there are none available, zero is returned. *Mutual exclusion addresses are in the device address space, not the normal one.*

* Address 6 should provide the starting address of the mutual exclusion addresses.

Writing a “1” to a mutual exclusion address attempts to lock it, and writing “0” unlocks it. Trying to lock a mutual exclusion address (or “mutex”) if it is already locked (either by the current core or another core) causes the implementation to hang until the mutex is available to be locked. If no mutual exclusion addresses are available, zero is returned when reading Address 6. Reading a mutual exclusion address is *implementation-defined behavior*.

* Address 7 should provide the starting address of processor status variables. Every core, including the first core, should have one.

If there are no processor status variables available, Address 7 should return 0.

Processor status variables can be **read** to determine if a processor is running: 1 for running and 0 for not running. If a processor reaches an “hlt” instruction, or an exception in privileged mode, it stops execution and the implementation should return a 0 upon reading a processor status variable. That processor should then be immediately available for starting again via the appropriate processor kickstart address. These variables are ***atomic*** and if a processor is in the process of shutting down, it should be 1. **Writing** to a processor status variable is *implementation-defined behavior*.

* Address 8 should provide the starting address of processor error codes. *The error codes are only available to be read if that processor is currently turned off, and reading it in any other case is undefined behavior.* The main use case of the error code addresses should be to check the reason why a core halted execution.

Errata:
* Every core/processor should be able to start every other core on a multi-core system, if that core is turned off. If core 1 has halted execution, core 2 should be able to start it.
* Cores do not share registers.
* Cores begin execution with all of their registers set to zero. This means that when starting another core, a processor must write the appropriate boot code to address zero. This could be as simple as writing a jump there.
* Undefined behavior may crash the implementation while implementation defined behavior must provide a non-crashing functionality (which includes shutting down or hard-resetting the system). Crashing being as possibly deleterious as “Halt and catch fire” or “detonate built-in nuclear weapon”. Non-crashing behavior could simply be returning an unexpected value, immediately shutting down the system, or performing some implementation-specific useful behavior.
* Processors are not required to run at the same speed but must share the same memory.
* All processors have full access to the device bus.
* All bus accesses, memory or device, should be appropriately guarded with mutual exclusion if relevant. If two processors share a memory address 0xABCDEF, then there should be a mutex which each processor locks and unlocks to access that memory address. Inappropriately accessing a memory address that has been modified by another processor without acquiring a mutex lock is *implementation defined behavior* and may read inappropriate or partially-written values. It should not crash the implementation.
* Hardware mutex addresses need only support unlocking by the same processor that locked it. Trying to unlock a mutex from a different processor than the one that locked it is *undefined behavior.*

Individual Instruction Information

(While reading this, please refer to *Instruction Set)* Most instructions are fairly obvious.
`nop` does **no op**eration and executes the dispatch,
`hlt` **h**a**lt**s execution, returning an error code of zero to the privileged process or shuts down the system,
`become_user` is explained in *User Mode,*
`syscall` is also explained in *User Mode* but functions similar to *hlt*
`dread` and `dwrite` perform 64 bit **read**s and **write**s on the device bus and are privileged instructions, explained in *Device*.
`ImXX` loads an **im**mediate value (the second argument) into a register (the first argument)
`ldXX` **l**oa**d**s a value from a memory address (the second argument) into a register (the first argument)
`stXX` **st**ores a value from a register (the first argument) into a memory location (the second argument). it is the only instruction in the ISA that violates the dest,src syntax of most instructions.
`ildXX` **i**ndirectly **l**oa**d**s a value from a memory address *stored in a register* (the second argument) into a register (the first argument).
`istXX` **i**ndirectly **st**ores a value from a register (the second argument) into a memory address *stored in a register* (the first argument).
`mov` copies (or **mov**es) the contents of a register (the second argument) into another register (the first argument).
`lsh` and `rsh` perform arithmetic left and right shift on the value in a register (first argument) by the number of bits specified in another register (the second argument).
`and`, `or`, and `xor` perform bitwise logic on the contents of two registers (the first and second arguments).
`compl` inverts all bits in a register (the first argument)
`neg` treats the value in a register as a signed 64 bit integer and multiplies it by negative one.
`bool` transforms the contents of a register to represent its boolean evaluation if treated as an unsigned integer. If the contents are zero, it is set to zero. If the contents are non-zero it is set to one.
`not` does the opposite transformation of bool, setting the register to 1 if it has zero contents, or 0 if it has non-zero contents.
`iadd`, `isub`, `imul`, `udiv`, and `umod` perform **u**nsigned integer arithmetic on the contents of two registers, corresponding to **add**ition, **sub**traction, **mul**tiplication, **div**ision, and **mod**ulo. Due to the nature of two’s complement, `iadd`, `isub`, and `imul` all work on s**i**gned 64 bit integers, too.

`idiv` and `imod` perform s**i**gned integer arithmetic corresponding to **div**ision and **mod**ulo, respectively. They are not valid on unsigned values.

`seYY` **s**ign **e**xtends the value in a register from some smaller number of bits (one of: 8,16,32) to 64 bit.

`ucmp` and `icmp` compare two integers, either **u**nsigned or s**i**gned, respectively. The first argument is the destination register for the result of the comparison (-1 if the second argument is less than the third, 0 for equals, 1 for greater than)

`fcmp` and `dcmp` compare two floating point numbers having the same -1,0,1 system and same three-argument syntax as `ucmp` and `icmp`.

`jmp` **j**u**mp**s the program counter to a future location using an immediate value.

`call` jumps the program counter to a future location, but first it pushes the program counter onto the stack. It is supposed to be used for implementing function **call**s.

`jnz` **j**umps the program counter to a future location using an immediate value (the second argument, *if* a register (indicated by the first argument) is **n**ot **z**ero.

`ret` pops the program counter off the stack. It is meant to be used for **ret**urning from a function to the caller.

`getstp` retrieves the **st**ack **p**ointer and stores it into a register.

`setstp` sets the **st**ack **p**ointer to the value stored in a register.

`pushXX` **push**es the contents of a register onto the stack

`popXX` **pop**s a value off of the stack and puts it into a register.

`ito(f|d)` converts a s**i**gned integer into a **f**loat (32 bit floating point) or **d**ouble (64 bit floating point). `itof` converts to 32 bit float, `itod` converts to 64 bit float.

`(f|d)toi` converts a **f**loat (32 bit floating point) or **d**ouble (64 bit floating point) to a s**i**gned integer. `ftoi` converts from a 32 bit float, and `dtoi` converts from a 64 bit float.

`inc` and `dec` **increment** and **decrement** the value in a register, interpreting it as a signed or unsigned integer.

`ldspXX` **l**oa**d**s a value, from a location relative to the **s**tack **p**ointer, into a register, using a fixed immediate 32 bit offset. the offset is subtracted from the stack pointer.

`stspXX` **st**ores a value, from a register, into a location relative to the **s**tack **p**ointer, using a fixed immediate 32 bit offset. the offset is subtracted from the stack pointer.

`bswapZZ` **swap**s the **b**ytes in a register, switching endian-ness.

`mnz` conditionally moves one register to another (third argument to second) if the first register is not zero.

Calling Convention

Function Calls:

Function calls potentially overwrite all registers, so the caller must save any and all registers that they are using before calling a function.

System Calls:

System calls pass their arguments through registers 1-255. Register zero is ignored. This is because when returning from become_user, the processor overwrites register zero with the error code (which, for `syscall`, is 4) and in order to use its value, the privileged code would have to read the value from the register file. This would slow down the already-costly context switch.

The *type* or *id* of the particular system call functionality being used should be in register 1.

User Mode:

Compilers and/or program authors should know how many of their registers are saved in a system call or through pre-emption. This information should be provided through a system call. Registers greater than this number may be unsafe or insecure to use as preemption and task switching may result in those registers being periodically cleared, filled with garbage, or both. Generally, no operating system should ever save fewer than registers 0 through 7.