

Simple Instruction Set Architecture 16

Programmer's Manual and Teaching Resource *1st Edition*

By David MHS Webster

Table of Contents

Contents

Introduction.....	3
Compiling.....	4
Workflow.....	7
Extending.....	9
Assembler Documentation Overview.....	11
Moving things around, Variables.....	15
Control flow.....	18
Math.....	25
Segment Control.....	26
Privilege.....	27
Krenel.....	29
Use Cases and FAQ.....	30
Addendum Dedicatory.....	35

Introduction

Sisa16 is an interpreted instruction set designed for replacing the CPU and BIOS in a computer. Instead of having to rewrite and recompile your kernel, your shell, your browser, your drivers, [...] for every new platform, you simply have to write a BIOS for sisa16 (getchar, putchar, interrupt, clock) and every program should run. **You basically only need a C compiler for the target architecture, and then you can get your full operating system on the target without recompiling it. You only need to write a small BIOS, and compile Sisa16 itself for the target. The BIOS implementation being much less complicated than rewriting an entire OS.**

Hardware which is specific to a particular platform can be interacted with through interrupts in a standardized fashion. Client/server protocols, Serial I/O, device peek/poke, etcetera can be implemented through the three functions (putchar, getchar, interrupt). Interaction with a hardware clock can be implemented through a clock() function. The example system uses a pure serial interface, for development purposes. Testing is done on Debian 11.

Sisa16 brings identical, reproducible, hard-realtime preemptive multitasking to any platform, even one without such abilities. While this means Sisa16 cannot take advantage of multithreaded processors very well, this is not its intended use case. The target is ideally a single-threaded 32 bit or 64 bit microcontroller or retro hardware.

“Let all that you do be done with love.”

Compiling

If you are using an ordinary Unix-like environment such as Linux, BSD, Msys2, or Cygwin, and your compiler is a relatively recent version of gcc, clang, or tinc, then you should be able to just call ‘make’ and everything will build, including the test programs. If this does not work or you wish to understand the compilation process in detail...

On most platforms, this will compile the assembler, which has a built-in emulator:

```
cc assembler.c -DUSE_UNSIGNED_INT -o sisa16_asm
```

Relevant Compiler Flags

-DUSE_UNSIGNED_INT

Define the macro “USE_UNSIGNED_INT” which indicates that “unsigned int” in C is a compatible 4-byte unsigned binary integer type. If “unsigned long” is such a type but not unsigned int, then you should omit this define.

-DUSE_TERMIOS

Recommended building flag. Enables usage of fnctl and termios in the textmode driver, not only to provide “raw” mode, but also to provide the ability to toggle non-blocking getchar by calling an interrupt.

-DNO_DEVICE_PRIVILEGE

Allow getchar, putchar, and interrupt to be called in user mode.

-DNO_PREEMPT

Disable Pre-emptive execution. This speeds up the emulator significantly, but disables the ability to restrict the execution cycles of user mode.

-DPREEMPT_TIMER=0x100000

Set the number of instructions between pre-emptions in User mode. Pre-emption is used to guarantee that userland code will always return to a privileged kernel.

-DPREEMPT=codeToCausePreemption()

If hardware interrupts can be used to directly provide preemption to sisa16, this is where you would do it. By default, an instruction counter is used.

-DFUZZTEST

Signal to the driver that arbitrary code may be executed, prevent harm to the host.

-DUSE_COMPUTED_GOTO

Enable usage of the GNUC extension for taking the address of a goto label, which speeds up the emulator tremendously and makes compilation much faster.

-DNO_FP

Disable the floating point unit. This is recommended if the host does not support the IEEE 754 32 bit binary floating point standard.

-DNO_SIGNED_DIV

Disable the signed integer division and modulo instructions. You should only do this if your system does not use a standard two's complement layout.

-DNO_SEGMENT

Disable the segment.

-DNO_SIGNAL

Disable hooking of SIGINT in the emulator and debugger, and SIGSEGV in the debugger.

-DNO_EMULATE

Disable emulate and emulate_seg instructions.

You can test if the assembler built successfully like this:

```
./sis16_asm -C
```

-C tells it to **C**heck the environment and exit. Read it carefully.

The exact same flags and semantics apply to building the emulator and debugger.

```
cc isa.c -DUSE_UNSIGNED_INT -o s16_emu
cc debugger.c -DUSE_UNSIGNED_INT -o s16_dbg
```

You can now test a simple program. Assuming that you are using the default serial text mode driver and you have a relatively ordinary terminal or terminal emulator, try the “wave” program...

```
./sis16_asm -i wave_runner.asm -o wave.bin
./sis16_emu wave.bin
```

You should now see a “wave” printed out to the screen, forever. You can disassemble the program by doing this:

```
./sis16_asm -dis wave.bin 0
```

this will show you the “machine code” in assembly again. It should look like this:

```
gek@Katherine:~/Programming/Simple_ISA$ s16_asm -dis wave.bin 0
//Beginning Disassembly

section 0x0;
la      0x03          ;//0x000000 :
lfarpc   ;//0x000002 : Region Jump to 0
halt     ;//0x000003 : End of Control Flow
halt     ;//0x000004 :
halt     ;//0x000005 :
halt     ;//0x000006 :

//Reached End.

gek@Katherine:~/Programming/Simple_ISA$
```

(if you know the language, you should be able to follow the control flow to 0x30000 and explore the rest of the binary. Use ‘-h’ to view help.)

If you wish to install the assembler and emulator, simply copy them into a suitable location in the system path, such as “/usr/bin” or “/usr/local/bin”. Copy the manual pages s16_asm.1, s16_dbg.1, and s16_emu.1 to a suitable place as well, such as “/usr/local/share/man/man1”. This can also be done by calling “make install” with administrative privileges.

Workflow

A program for SISA16 is written with one “main” assembly file, header files which are included with “ASM_header” and data files which are embedded with “ASM_data_include”.

The program, if its ‘main’ file is “main.asm” can then be built with

```
sis16_asm -i main.asm -o main.bin
```

Warnings may be emitted during assembly. Errors will cause the the assembler to abort and emit an error message. If there were no errors, then the assembler will report that it “Successfully assembled main.bin”. The file “main.bin” is a Sisa16 memory image ROM.

You can run this program with the emulator by calling:

```
sis16_emu main.bin
```

In most microcontroller workflows, a “ROM” (Read-Only Memory) image, is typically programmed onto an EEPROM (Electrically Erasable Programmable Read-Only Memory) or burned onto a non-reprogrammable ROM chip before being put on a board and powered on.

You can imagine the Sisa16 architecture as a very simple single-board computer in which the ROM is copied entirely to a random-access memory chip on power-on. The CPU then begins fetching, decoding, and executing instructions, occasionally reading from the RAM chip and interfacing with devices, until either a halt instruction is executed or the power is removed.

If you are suspicious of a program being incorrectly assembled, you can disassemble it either piece-by-piece or wholly:

```
sis16_asm -dis main.bin <location>
```

will disassemble main.bin starting at <location>. Standard C syntax for numbers is used, the same as in the assembly language itself. TIP: Control flow always starts at 0.

```
sis16_asm -fdis main.bin <location>
```

will perform a full disassembly from <location>. If zero, the entire binary is disassembled.

Normally, comments are emitted into disassembly about what the program is likely doing at that point in control flow, to assist in the reverse engineering process. These may provide optimization tips and useful insights into the language.

If you wish to not have comments, make sure to include the -nc argument before -dis or -fdis

```
sis16_asm -nc -fdis main.bin 0
```

will disassemble main.bin from zero without emitting line comments.

Now, if you've verified that your program was assembled correctly and you still find you are having bugs with your programs, you might want to try the debugger.

```
sis16_dbg main.bin
```

will run the debugger on main.bin. *If you are frequently re-assembling and debugging your program, it'd be awful smart to put it in a script somewhere!*

The debugger allows you to view a disassembly of your code, manipulate registers (including the program counter), arbitrarily start and stop execution of your program, restart your program from the beginning, alter memory and manipulate control flow. You can arbitrarily jump to any point in your program, test routines, change the values of variables in memory, and more.

Extending

Sisa16 is meant to be easily extended to suit your needs. To have Sisa16 programs interact with C libraries, you should write a “d.h” file which implements these functions:

```
“Device Init”
static void di();
“Device Close”
static void dcl();
“Getchar” – corresponding to the getchar instruction.
static unsigned short gch();
“Putchar” – corresponding to the putchar instruction.
static void pch(unsigned short a);
“Interrupt” – corresponding to the interrupt instruction
static unsigned short interrupt(unsigned short a,
    unsigned short b,
    unsigned short c,
    unsigned short stack_pointer,
    unsigned short program_counter,
    unsigned char program_counter_region,
    UU RX0,
    UU RX1,
    UU RX2,
    UU RX3
);
“Clock”
clock_ins()
```

These functions (and one macro) need to be implemented into d.h (or provided through includes) for sisa16 to run properly.

[This may look daunting, but compared to the effort of rewriting a kernel to support an entirely new architecture, it is nothing]

Also note that you can choose to add new instructions to the architecture. If you choose to do that, you must:

In isa.h:

- 1) add the necessary goto label with a dispatch
- 2) add the entries to the computed goto table and switch case

in instructions.h:

- 3) add the instruction to the assembler, including the number of arguments and the conversion to a “bytes” command, and increase the “n_insns” constant’s size. There are three arrays and one variable you must change, in total. You may also want to add a comment that will be emitted in disassemblies.

In sisa16_asm.1:

- 4) write documentation in the sisa16_asm.1 manual page about your addition

Assembler Documentation Overview

The developer documentation for the sisa16 assembler is included in the manual page sisa16_asm.1, and should be provided to all programmers interested in writing programs for the platform.

The documentation for the emulator is included in the manual page sisa16_emu.1.

The assembler manpage provides a quick overview of the commandline args, every instruction in the instruction set, and the (dis)assembly-time features of Sisa16. It should serve as a fast reference when one is needed, whereas this manual serves primarily to teach high level concepts.

A nicely-formatted, overly friendly re-listing of every single instruction and assembler feature with a minimal complete example and a lollipop is specifically omitted from this manual for the following reasons:

- 1) Doing so would be too dissimilar from real micro controller documentation. Read official documentation on almost any real popular micro controller. They're poorly formatted tables with minimal comments, and often contain broken english and horrible spelling errors. Wrong information is common. *Consider any such errors in this manual a "feature".*
- 2) The document is already too large
- 3) I don't think it's necessary.
- 4) Exercise for the reader.

Important notes about the assembly language which may confuse readers:

- * the semicolon is not used to denote a comment, it is used as a separator. This is also a function of the vertical bar or pipe character.
 - * Multiple instructions may be written on the same line.
 - * Python-style `"#"` comments as well as C++ style `"//"` comments are supported.
 - * All instructions that take more than one byte of arguments will typically use SPLIT syntax.
 - `%0xEEFF%` is semantically equivalent to `0xEE`, `0xFF` for instruction arguments.
 - `%/0x1298CADF%` is semantically equivalent to `0x12`, `0x98`, `0xCA`, `0xDF` for instruction arguments.
 - `??13.7%` will parse the floating point number 13.7 and generate a comma separated list of four bytes similar to previous.
 - `%-10%` will take the decimal number ten and put it in a 32 bit variable, bitwise negate it, add one, and output the comma separated big endian list of bytes comprising it, effectively outputting negative ten twos complement.
 - `%&0xABCDEF%` will generate a three-byte split, useful for ``farld`` and ``farst`` instructions.
- This is an artifact of how the assembler works.

Assembly Process

Author's note: If you are not interested in writing a compiler or assembler yourself, I would recommend skipping this section. These details are largely unimportant for writing programs for Sisa16.

The Sisa16 assembler is a very simple text-replacement based assembler. Macros and assembly directives are evaluated, instructions are parsed, file construction code is generated, and the binary is written in that order. The best way to explain how the assembler works is bottom upwards.

Lines are read and parsed from the .asm file one at a time, with backslashes escaping the newline to simulate an extended line (Some exceptions apply- comments and string literals cannot escape the newline). This is at the very top of the main loop. The file is processed twice, so that forward referencing can be done (needed for forward jumps) (Search for “/*Second pass to allow goto labels*/”). This is the loop which contains the main loop which iterates over individual lines.

At the very bottom, the assembler has a few basic things it can do (Search in assembler.c for “/*VERY_BASIC_THINGS_IT_CAN_DO*/”)

- * bytes can be written to the output counter. (“bytes”)
- * shorts can be written to the output counter. (“shorts”)
- * the output counter can be moved. (“section”)
- * the output counter can be moved to a region boundary (“region”)
- * A region of memory can be filled with a single byte value. (“fill”)
- * the output counter can be moved specifically on the second pass (“asm_fix_outputcounter”)
- * the status of the assembler can be printed to standard out (“asm_print”)
- * A region restriction can be started, overriding a page restriction. (“asm_begin_region_restriction”)
- * A page restriction can be started, overriding a region restriction. (“asm_begin_page_restriction”)
- * The active restriction can be ended. (“asm_end_restriction”)
- * All macro variables can be dumped. (“asm_vars”)
- * Assembly can be halted. (“asm_quit”)

These very basic operations are parsed in a semicolon-separated fashion until the end of the line or a vertical bar. After this is finished, if a vertical bar is present in the line, everything before the vertical bar is removed and the line is re-evaluated through all steps of the process.

It should be noted there are some other basic operations that the assembler can perform, but they are recognized at the very top of the main loop rather than the bottom:

- * The assembler can include arbitrary data with ASM_data_include
- * The assembler can push a file onto the parsing stack with ASM_header, which starts parsing that file until it ends, and then returns to the current one.
- * The assembler can include a string literal at any point in the file. These are recognized before preprocessing.

An instruction is a sequence of bytes comprised of an opcode and arguments. All opcodes and arguments are converted directly to “bytes” commands by text replacement before the very bottom.

This is why multi-byte arguments must be specified as comma separated integers. Instruction expansion is the stage where opcodes and their comma-separated byte arguments are turned into bytes commands. (Search for “/*INSN_EXPANSION_STAGE*/” in assembler.c)

- * First, an instruction is recognized before any vertical bars on the line. It is checked to see if there is a longer instruction could be made from the same spot, to prevent, for instance, “llb” from being impossible to parse due to “lb” being inside of it.
- * Then, the number of arguments is counted and verified. If an incorrect number of bytes are given for arguments, then an assembly-time error is emitted.
- * Finally, a “bytes” command replaces the name of the instruction.
- * These steps are repeated until an instruction is not recognized on the current line.

Before this, though, Macro definitions must be recognized. VAR# lines. (Search assembler.c for “/*MACRO_DEFINITION_STAGE*/”).

A VAR line has two portions:

- * A portion specifying the name of the macro, between the pound symbols
- * A portion specifying the contents of the macro, after the second pound symbol.

Macro names are validated to ensure that they won’t generate conflicts with Sisa16 assembler’s reserved namespaces, keywords, or instructions. Bad macro names generate errors. Macros that are **redefining** or have the potential to **conflict** emit a warning and set a flag. Non-redefining macros are checked on the second pass to ensure that they have the same value as they did the first time, as this is almost certainly a programming error. Macro definition lines do not proceed to the instruction expansion or code generation phases of assembly, nor do they take vertical bars into account (The vertical bar is *part* of the macro). It is important to note that Sisa16’s assembler does not allow for more than about 65528 macros to be defined. This should be enough for any well-written program, but it may be an issue if you are writing a compiler for the platform.

Before this, macro expansion must take place. (Search for “/*MACRO_EXPANSION*/”). On VAR# lines, only some of the macros are allowed to be expanded, such as the @ sign. Otherwise, all macros are expanded. Macros are expanded only if they occur before a vertical bar. The last defined macro that matches, which does not overlap with a longer matching macro will be expanded first. Macros are only expanded before a vertical bar. This is true even for a VAR# line, so you can prevent built-in macros such as the @ sign from being evaluated by prefixing the macro definition with a vertical bar.

Macro expansion (or, preprocessing) repeatedly identifies macros and replaces them until either the macro replacement limit is reached, or there are no macros left to be expanded. Macro expansion may be interrupted if a pre-preprocessing directive is discovered at any point during macro expansion before a vertical bar on a “normal” line.

Pre-preprocessing is the step which takes place before macro expansion. Syntactic sugars such as “:label:” and “.mymacro:” are handled.

You should now have a reasonably strong grasp of how the assembly language is parsed and how the final file is created.

Design Decisions and Critique

If you are a programming language fanatic, you may recognize that this is a very unusual, or at least uncommon way of writing a programming language parser and lexer. The code certainly has its flaws, but I still think it's better than most assemblers I've seen. It is also very small.

You might also recognize that unlike most assembly languages, you may have multiple instructions on a single line. This is mostly for organizational purposes- When writing a subroutine in Sisa16, you probably already have a rough grasp of what the C statements it equates to would look like. I like to organize instructions that represent a single **idea** on a single line. If I just want to increment a variable, I don't want to put loading the variable, incrementing the register, and saving it back to be on separate lines. It would be "var++", or "var = var + 1" if you were writing it in a high level language, so I think it should be conceptually organized the same way in the code. (It also made writing macros easier...)

My design decisions with the assembler strayed from others largely out of disgust with the stuffy, visually ugly display of many assembly languages.

Moving things around, Variables

Sisa16, like many other architectures, is a load/store architecture. There is no “mov”, but there is an “lda”. You can load an immediate value into a register, load from a memory location, store to a memory location, indirectly load from memory through a register, indirectly store to a memory location through a register, and copy values between registers. Loads and stores are typically within the same region as the program counter, unless they are “far” memory loads and stores. The stack is always in the zero region.

The following bit of code increments the single byte value at address 500 (dec) in the current region:

```
;lda %500%;lb 1;add;sta %500%;
```

(note that lb 1;add; can be replaced with aincr; and achieve the same result)

This, in order:

- * Loads the value from M[500] into register A.
- * Puts the immediate value 1 into register B.
- * Adds the registers A and B and puts the result in A.
- * Stores the value in the A register at M[500].

if there is an array of 8 bytes at location 0xE0AA and you want to load the byte indicated by register ‘a’ this is how you’d do it:

```
;lb 8;mod;llb %0xE0AA%; add; ca; ilda;
```

This, in order:

- * Loads the single-byte immediate value “8” into register B.
- * Performs an unsigned integer modulo of register A by register B and stores the result in A.
- * Loads the 16 byte value 0xE0AA into register B.
- * Adds registers A and B.
- * transfers the contents of the 16 bit “A” register to the “C” register.
- * Uses the contents of the C register to perform an indirect single-byte load into a.

Both of these examples operate within the current region. To access all of memory, you need to use “far” memory commands.

If there is short variable at 0xAFEE23, you can load it into register A like this;

```
;farllda %&0xAFEE23%;
```

Or store it like this:

```
;farstla %&0xAFEE23%;
```

Of course, you can access more than just single variables. You can also do indirect memory accesses. Assuming that the aforementioned memory location is actually the start of an array of two-byte values, of length 13, and you have an index into it in register A...

```
;sc %0xAF%;lb 13; mod;lb 2;mul;llb %0xEE23%; add; ba; farillda;
```

In order, this:

- * Sets the C register to 0x00AF, the region in which the array resides.
- * loads 13 (decimal) into register B
- * Performs a modulo operation, A%B and stores the result in A. This guarantees that no out-of-bounds memory access occurs if A is larger than length of the array.
- * Loads 2 into register B. (An element in the array is two bytes in size)
- * Multiplies A and B and stores the result in A. This corrects for the size of elements in the array.
- * Loads 0xEE23 into register B.
- * Adds registers B and A, and stores the result in A.
- * moves the value from register A to register B.
- * performs a “far memory load” into A

the effect is thus like this piece of C code:

```
register short A;

/*
   A had an index placed in it at some point...
*/

A = ((unsigned short*)0xAFEE23)[A%13];
```

If you’ve looked at a disassembly of compiled C code in almost any other language, this should seem very straightforward to you. Note that unlike a more complex assembly language like x86, there is no “lea” instruction which performs a fused integer multiply-add, nor a MOV instruction which can compute array offsets automatically. Arithmetic operations always have their results place in the accumulator (register A for 16 bit operations, register RX0 for 32 bit operations) rather than being specified in the arguments of an instruction.

You could also have performed the operation like this:

```
;lb 13; mod;lb 2;mul;lrX0 %/0xAFEE23%;rx1a;rxadd;cbrX0;farillda;
```

This form is more general.

Large Copies

Let's say you have a lot of memory you want to copy, all at once. You could write a routine which copies individual bytes one at a time. Indeed, this would be the most flexible. However if you are copying large amounts of data this could be very slow. There are two instructions that can help you: `farpage1` and `farpage2`.

```
;lla %10%; sc %20%; farpage1;
```

This copies 256 bytes from the page in memory indexed by "20" decimal, to the page indexed by "10" decimal. In the memory map, there are 65,536 pages of size 256 bytes.

`farpage2` swaps the order of the arguments. The page indexed by C is the destination, while A is the source.

Control flow

Control flow in SISA16 uses fixed (rather than relative) addressing.

```
;sc %0x500%; jmp;
```

This unconditionally jumps to the address 0x500 **Within the program counter's region**. This is also sometimes referred to as a “branch”. Sisa16 also has conditional branching. The following code jumps to 500 if the A and B registers are equal:

```
;cmp;sc %0x500%;jmpifeq;
```

Or conversely, if they are **not** equal:

```
;cmp;sc %0x500%;jmpifneq;
```

Notice the usage of “cmp”. There are four comparison instructions, “cmp”, “rxcmp”, “rxicmp”, and “fltcmp”.

cmp will set A to zero if A is less than B, one if A equals B, and two if A is greater than B.

rxcmp puts the exact same values in A, but performs tests on RX0 and RX1 instead of A and B, respectively.

rxicmp functions identically to rxcmp, but treats RX0 and RX1 as signed, so -1 will be less than 0.

fltcmp functions identically to rxicmp but does 32 bit **floating point** comparisons. The semantics are more important with fltcmp:

- * If RX0 holds the floating point value for negative zero and RX1 holds positive zero, or vice versa, they are considered equal.
- * Positive infinity is always considered greater than everything except itself.
- * Negative infinity is always considered less than everything except itself.
- * NAN is considered equal to everything.

Alright, but how do you use this to create an “if” statement? You can use **labels**.

In Sisa16, labels are simply assembly-time variables which contain nothing but a memory address. You can create them like this, on their own line:

```
VAR#myLabel#@
```

Alternatively, you can use one of these syntactic sugars:

```
:myLabel:
myLabel:
```

(Note: you cannot have a label on the same line as code)

The following complete program is a truth machine. Any key pressed except the “1” key will result in immediate termination, while “1” will result in printing “1” forever.

```
..main:
getchar;
putchar;
lb '1'; cmp;
sc %Lbl_false%; jmpifneq;
sc %Lbl_true%; jmp;

:Lbl_false:
    halt;
:Lbl_true:
    la '1';
    cpc;
    putchar;
    jmp;
```

You may have noticed that Sisa16 uses 16 bit numbers for its control flow. The program counter, for instance, is 16 bit. How, then, does one execute code outside the first 64k?

The CPU has an 8-bit register called the “Program counter region” which specifies the 64k portion of memory that the program counter is currently loading opcodes and operands from. It also specifies the region that memory loads and stores will happen in, by default. Lda, stla, ldc, ilda and many others use this register for determining the region to get their data from. You can get the value of this register with the `cpcr` instruction.

It is changed by these instructions:

- * lfarpc
- * farcall
- * farret
- * farjmprr0

it is very typical (for reasons that will be explained later) to want the code of your program to be outside the first region of memory. So it is very typical to see this:

```
section 0x10000;
    //code...

section 0;
    la 1;
    lfarpc;
```

If you haven't read the manual pages, note the use of "section" to set the code generation location. The empty area in memory between the "lfarpc" and the code at 0x10000 is filled with zeroes. Also note that the order of the two sections in the assembly file is irrelevant. Code generation may occur in any order the programmer desires.

Switch Case

In SISA16 it is possible to create a jump table, an equivalent of switch-case in C.

Here is an example program which reads a single byte from the keyboard and prints its hexadecimal value to standard out:

```
section 0;
    sc %10%; jmp;
section 10;
    getchar;
    apush;
    lb 4;rsh;
    lb0xf;and;
    lb 7;mul;
    llb %printbytehex_jmptable_1%; add;ca;jmp;
VAR#printbytehex_jmptable_1#@
    la0x30;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x31;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x32;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x33;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x34;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x35;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x36;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x37;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x38;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x39;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x41;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x42;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x43;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x44;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x45;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x46;putchar;sc%printbytehex_jmptable_1_end%;jmp
VAR#printbytehex_jmptable_1_end#@
    apop;lb 15;and;lb7;mul;
    llb %printbytehex_jmptable_2%;add;ca;jmp;
VAR#printbytehex_jmptable_2#@
    la0x30;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x31;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x32;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x33;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x34;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x35;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x36;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x37;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x38;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x39;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x41;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x42;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x43;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x44;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x45;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x46;putchar;sc%printbytehex_jmptable_2_end%;jmp
VAR#printbytehex_jmptable_2_end#@
    halt;
```

This program written in C would look like this:

```
int main(){
    unsigned char a = getchar();
    switch((a & 0xf0)>>4){
        case 0: putchar('0'); break;
        case 1: putchar('1'); break;
        case 2: putchar('2'); break;
        case 3: putchar('3'); break;
        case 4: putchar('4'); break;
        case 5: putchar('5'); break;
        case 6: putchar('6'); break;
        case 7: putchar('7'); break;
        case 8: putchar('8'); break;
        case 9: putchar('9'); break;
        case 0xA: putchar('A'); break;
        case 0xB: putchar('B'); break;
        case 0xC: putchar('C'); break;
        case 0xD: putchar('D'); break;
        case 0xE: putchar('E'); break;
        case 0xF: putchar('F'); break;
    }
    switch(a & 0x0f){
        case 0: putchar('0'); break;
        case 1: putchar('1'); break;
        case 2: putchar('2'); break;
        case 3: putchar('3'); break;
        case 4: putchar('4'); break;
        case 5: putchar('5'); break;
        case 6: putchar('6'); break;
        case 7: putchar('7'); break;
        case 8: putchar('8'); break;
        case 9: putchar('9'); break;
        case 0xA: putchar('A'); break;
        case 0xB: putchar('B'); break;
        case 0xC: putchar('C'); break;
        case 0xD: putchar('D'); break;
        case 0xE: putchar('E'); break;
        case 0xF: putchar('F'); break;
    }
    return 0;
}
```

Just from the visual shape and indentation of the code, you can tell there are similarities, and perhaps you understand now why C is often called a “portable assembly language”. Here’s what’s happening:

- * First, we jump to 10 (decimal) in the binary. This is to leave some space for the stack. We technically don’t need to do this in this instance (The getchar instruction would simply be overwritten with apush), but it makes the code easier to think about.

- * A character is read from the standard input.

- * The high hex digit is checked and a computed jump is done to one of sixteen different possible values it could have- 0 through F in hex. This hex digit is printed.

- * The same is done for the low hex digit.

Note the “lb 7; mul;” before the jump in the sisa16 assembly code. This is the width of each jump table entry. Just like accessing an array of multi-byte elements, the stride must be taken into account when performing a computed jump in a jump table. This was calculated by hand.

Astute readers may notice that this code could be made substantially smaller by simply looping the switch-case, and moving the “putchar” to the end of the each switch case. *Consider this an exercise.*

Subroutines

You can create subroutines which use the stack in sisa16. The stack pointer is 16 bit, so it always points somewhere within the first 64k of memory. *This is why you might not want to put your code in the zero region.* There are four main instructions to use when writing subroutines:

```
* call
* ret
* farcall
* farret
```

If you know that you are only ever going to call the subroutine from the same region it exists in, such as if your entire program were to fit inside of a single 64k region, then you should use “call” and “ret”. They are faster and use less memory. Farcall and farret should be used for procedures which may be invoked from outside of the same 64k region they are in.

Here is a simple program which uses 16 bit call and ret to create a simple program which multiplies two decimal digits entered by the keyboard.

```
section 0;sc %L_main%; jmp;
section 1000;
VAR#proc_printbytehex#sc %1000%;call;
    //retrieve our argument.
    pop %3%; apop; push %4%;
    //push it
    apush;
    lb 4;rsh;
    lb0xf;and;
    lb 7;mul;
    llb %printbytehex_jmptable_1%; add;ca;jmp;
VAR#printbytehex_jmptable_1#@
    la0x30;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x31;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x32;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x33;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x34;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x35;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x36;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x37;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x38;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x39;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x41;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x42;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x43;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x44;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x45;putchar;sc%printbytehex_jmptable_1_end%;jmp
    la0x46;putchar;sc%printbytehex_jmptable_1_end%;jmp
VAR#printbytehex_jmptable_1_end#@
    apop;lb 15;and;lb7;mul;
    llb%printbytehex_jmptable_2%;add;ca;jmp;
VAR#printbytehex_jmptable_2#@
    la0x30;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x31;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x32;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x33;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x34;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x35;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x36;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x37;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x38;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x39;putchar;sc%printbytehex_jmptable_2_end%;jmp
    la0x41;putchar;sc%printbytehex_jmptable_2_end%;jmp
```

```

        la0x42;putchar;sc%printbytehex_jmptable_2_end%;jmp
        la0x43;putchar;sc%printbytehex_jmptable_2_end%;jmp
        la0x44;putchar;sc%printbytehex_jmptable_2_end%;jmp
        la0x45;putchar;sc%printbytehex_jmptable_2_end%;jmp
        la0x46;putchar;sc%printbytehex_jmptable_2_end%;jmp
VAR#%printbytehex_jmptable_2_end#@
ret;
section 500;
VAR#L_main#@
    getchar;putchar;
    lb 0x30;sub;
    apush;
    la 0xa; putchar; la 0xd; putchar;
    getchar;putchar;
    lb 0x30;sub;
    bpop;
    mul;
    apush;
    la 0xa; putchar; la 0xd; putchar;
    proc_printbytehex;
    la 0xa; putchar; la 0xd; putchar;
halt;

```

This program receives two decimal digits as input and multiplies them to create a result, which is printed in hex. Entering other characters has erroneous effects, but displays a result.

Math

There are seven classes of arithmetic instruction in Sisa16.

- * 16 bit arithmetic. Div and Mod are unsigned.
- * 32 bit arithmetic. Div and Mod are unsigned, except for...
- * 32 bit signed division instructions, rxddiv and rximod.
- * 32 bit floating-point arithmetic.
- * Comparison instructions.
- * Floating-point and Integer conversion ops.
- * Integer increment/decrement instructions.

Idea for implementers: It might be worthwhile to add vector operations to the ISA.

Operations are for twos complement integers, default unsigned, unless otherwise specified.

Here is a table of the mathematical operations, for reference:

Instruction	Operation	Destination register
cmp	A <=> B	A
rxcmp	RX0 <=> RX1	A
rxicmp	RX0 <=> RX1 (signed)	A
fltcmp	RX0 <=> RX1 (float)	A
add	A + B	A
sub	A - B	A
mul	A * B	A
div	A / B	A
mod	A % B	A
rxadd	RX0 + RX1	RX0
rxsub	RX0 - RX1	RX0
rxmul	RX0 * RX1	RX0
rxdiv	RX0 / RX1	RX0
rxmod	RX0 % RX1	RX0
rxidiv	RX0 / RX1 (signed)	RX0
rximod	RX0 % RX1 (signed)	RX0
fltadd	RX0 + RX1 (float)	RX0
fltsub	RX0 - RX1 (float)	RX0
fltmul	RX0 * RX1 (float)	RX0
fltdiv	RX0 / RX1 (float)	RX0
and	A & B	A
or	A B	A
xor	A ^ B	A
compl	~A	A
rxcompl	~RX0	RX0
lsh	A << B	A
rsh	A >> B	A
rxand	RX0 & RX1	RX0
rxor	RX0 RX1	RX0
rxxor	RX0 ^ RX1	RX0
rxlsh	RX0 << RX1	RX0
rxrsh	RX0 >> RX1	RX0
nota	(A==0)	A
boolify	(A!=0)	A
logor	A B	A
logand	A && B	A
aincr	A+1	A
adecr	A-1	A
rxincr	RX0+1	RX0
rxdecr	RX0-1	RX0
rxitof	RX0 (signed) to float	RX0
rxftoi	RX0 (float) to signed	RX0

Segment Control

The “Segment” is the way that Sisa16 accesses more than 16 megabytes of memory. It can allow access to a 40 bit address space.

These are the instructions that interact with the segment:

`seg_ld`: load segment RX1 into page RX0 of Main Memory. (1 byte) (AB)

`seg_st`: store segment RX0 from main memory into page RX1 in the segment. (1 byte) (AC)

`seg_pages`: get the number of pages in the segment in RX0. (1 byte) (B3)

Note that there is no way to “ild” or “ist” directly with the segment. 256 byte “pages” are moved from main memory into the segment, or moved back. This means that large copies are actually relatively fast between main memory and the segment, but that loading a small amount of data is expensive.

Using `seg_ld` and `seg_st` in conjunction with `farpagel` and `farpagest` can result in incredibly fast copies to and from the segment.

The segment is a fixed size and access to it is controlled by the active segment mask and offset, which are configured in privileged mode. `seg_config` configures the offset and mask, and `seg_getconfig` retrieves the active offset and mask. An offset equal to the number of pages means the segment is disabled for the user process.

Privilege

Sisa16 is relatively unique in how it implements privilege rings into the architecture- No Memory Mapping Unit is used. It uses Memory segmentation to prevent user-mode code from interfering with other user code. Instructions executing in user mode simply cannot address any memory they cannot access. While this means that each mode can only use 16 megabytes of memory, it makes the emulator extremely fast, and it makes using the privilege scheme extremely easy.

When sisa16 boots, it starts in privileged mode. To enter user mode, you must either call `priv_drop`, or `emulate`. Memory is cleared to zero before booting, so `priv_drop` ordinarily won't do anything. The user code will simply reach a halt instruction, opcode 0. You have two options: Use the "emulate" instruction to write a program into user memory and begin execution at location 0, or use ``user_farista`` to copy an entire program, byte-by-byte, into the target.

`Emulate` will copy the contents of privileged mode main memory into user memory and begin execution at location 0. To execute arbitrary code, then, you must write a "bootloader" which will jump into the code you want to run in user mode, and it must be written at location zero **before** `emulate` is called. Note that sisa16 itself and by extension privileged mode boots directly to location zero as well. This means that you will need to, at runtime, overwrite location zero with a bootloader.

Location zero is also part of the stack (region 0) so you should be careful to preserve what is there and copy it back when `emulate` returns.

"But how does `emulate` return?"

`Emulate` returns when any one of the following occurs:

- A privileged-only instruction is executed. (15)
- A driver instruction is executed (16: `getchar`, 17: `putchar`, 18: `interrupt`)
- A `syscall` is made (19)
- A preemptive context switch occurs. (255)
- A 'halt' instruction is reached. (0)
- An illegal opcode is executed (0)
- An illegal segment access occurs (5)
- An error code is emitted. (Any other value of register A.)

Whenever any one of these occurs, the state of the user mode is saved (registers and all) so that you can return to it later with `priv_drop`.

To write a very simple single-process kernel, you can write a loop that looks roughly like this:

```
emulate();
while(1){
    if(a == 0xff){task_ric();priv_drop(); continue;}
    else {puts("Error");break;}
}
```

See krenel.hasm for a complete monolithic kernel.

Preemption

Sisa16 implements pre-emption not based on time, but rather based on the number of instructions executed. Every time an instruction is executed, an instruction counter is incremented (by some amount, varying based on the “cost” of that instruction).

When this instruction counter surpasses a certain value, control will return to privileged code- meaning that either emulate or priv_drop will return. If preemption was the reason for returning, the A register will be 255. The preemption mechanism can be changed to be best suited for the hardware.

Syscall

A special instruction exists in sisa16 specifically for the purpose of trapping to create system calls- syscall. The A register will be 19 after returning from priv_drop or emulate, your privileged code simply needs to respond to it.

The BIOS functions

The BIOS functions- getchar, putchar, and interrupt- are privileged. Whenever they are called in user mode, control immediately returns to privileged code. Privileged code can respond to them just like syscalls.

The Segment

The segment, the majority of memory available to sisa16, is restricted access by a “mask” and “offset.

This mask and offset can be configured in privileged mode. It remains active even through context switches and task_sets, so it must be changed by the kernel each time.

An example Kernel.

An example kernel is written in krenel.hasm. It is called “Krenel” (spelled exactly like that). It features a simple round robin scheduler, with several basic system calls implemented for process management and Disk I/O. It can easily be extended to add new system calls for any driver you may implement.

Krenel

Krenel is the extremely simple kernel I have written for Sisa16. Here are its features at the time of writing:

- Round robin scheduler supporting all 8 of Sisa16's hardware tasks.
- Hard Realtime- the worst case execution time is very easy to compute and relatively small.
- Various interrupts and passthroughs for controlling the disk and I/O
- process control and Interprocess communication

It can be re-worked to include additional functionality as needed for an expanded device, or used as the basis for an entirely new kernel.

Use Cases and FAQ

Sisa16's main use case is to replace the CPU in a machine with one whose bytecode is known- similar to the original vision of Java.

If you have a C compiler for a viable platform, you can get SISA16 running on it, with all of its features. Any platform can have preemptive multitasking and segmented memory.

Q: If I have a C compiler for a target architecture, why wouldn't I just compile <such and such kernel>

A: You're going to have to re-write that kernel and all of its drivers to support the hardware...

What if the architecture doesn't have compatible preemption?

And after you have a kernel, you'll need a shell, a browser, a package manager... all of which will have to be compiled.

If you wanted to port Linux, which is written in C, to a new architecture, not only would you need a C compiler, but you'd need to port every program which runs on Linux to the new architecture. Many programs make assumptions about the architecture they're running on, and they might not be true anymore. By creating a standardized virtual processor, you can circumvent this problem entirely. SISA16 only requires writing an emulated BIOS. As soon as Sisa16 is ported, which is much simpler than a kernel, every program ever written for Sisa16 (using compatible interrupts) will run.

Q: Why would I want to emulate a computer that doesn't exist? Why not use an ISA that is popular?

A: SISA16 is designed for emulation speed and portability. Its instructions are very close to "real" hardware instructions on most platforms, enough so that it should be relatively easy to write a JIT. The ISA is just wide enough to allow reasonable utilization of the hardware, without cutting out extremely low end microcontrollers, maybe even retro systems like M68K computers. Finally, virtually all existing instruction sets which are actually popular have intellectual property issues. Fast emulators for them cannot be placed in the public domain, because the ISAs are not public domain. Risc-V would seem to be the best solution, but its ISA is far too broad and emulating it would be much slower.

To give you an idea of how **fast** SISA16 can run, my Ryzen laptop can execute more than a single Sisa16 instruction per cycle- over 4 Gigahertz. My i7-6700 gets slightly over a billion. Admittedly, this is in tight loops, but that's where computers spend the most time anyway.

Q: How is this better than any other trusted interpreter system?

A: This one doesn't run on top of a kernel. You run an operating system inside of it. It's not exactly the same as a trusted interpreter OS, but rather like a Motherboard abstraction layer. A full CPU/BIOS replacement. Operating systems written in Sisa16 rely on Sisa16 to correctly execute programs in the same way that OSes written on bare metal rely on the physical CPU.

Q: Is rewriting a BIOS really easier than patching a kernel?

A: Yes, it really is easier. A kernel which aims to be fast must make a lot of hardware-specific assumptions, and mix them around everywhere. But if the “hardware” is an abstraction layer, providing a small set of standardized low-level functions to access the hardware, then this effect can be negated. This series of small low level functions can and should be platform-specific, outside of a portable kernel.

Part of Sisa16’s goal is to separate kernel/OS development from driver development.

Q: “small set of low level functions?” I’m suspicious.

A: Most pieces of specialized hardware on a computer have high level APIs which are what end-user software interacts with (OpenGL, OpenAL, Vulkan...)

At the kernel level, this distinction still exists. Between for instance, the gpu driver and the X11/Wayland renderer. X11 and Wayland expect a certain set of functions from the GPU driver, in the form of OpenGL calls, memory map access... etcetera.

To make it clear- this project does not remove the difficulty of writing a driver. It simply shifts the difficulty away from operating system development. **If you expect to provide a `drawTriangle()` function as part of a standardized instruction set to your kernel, driver implementers will still need to write `drawTriangle()` for their respective pieces of hardware.** However, since this is now *Outside the realm of kernel development* these functions can be used even by other operating systems. These functions are – correctly – tied to the system-specific configuration, rather than a kernel and OS, which provides scheduling, ABI, permission control, and so on. ***The maintainers of Graphics Card Q do not have to write separate drivers for every single OS under the sun- they write a single driver, once, for the Sisa16 BIOS implementation, and every single OS ever written for Sisa16 can take advantage of it equally.***

Imagine if this is how things were done normally. Then, Linux would never have had driver issues, because it could have just used ***dows drivers. Free operating systems would have replaced proprietary ones decades ago.

Q: Isn’t the Sisa16 driver then **part** of the kernel?

A: Is a Virtual Machine program part of an operating system you run inside of it? Is an Emulator part of a game you play? Furthermore, isn’t this a philosophical question, rather than pragmatic?

Q: Isn’t part of the point of an operating system to abstract away the hardware?

A: There are much bigger problems in OS development than hardware abstraction- such as how schedulers run, how daemons are started, and so on and so forth. Monolithic versus Modular isn’t forced on an OS by Sisa16 either- a system daemon running in user mode can be designated to control a device by passing through interrupts and such.

Q: Isn't part of the point of OS development to get full utilization out of hardware, and doesn't running an emulator sort of ruin that?

A: Partially, yes. It is true, you could always do better by compiling to true native code. However, given how CPUs continue to increase in power exponentially over time (although not as extreme as it used to be) it can be considered to be worth it due to the enormous boosts in portability.

The same could be said about Javascript, yet thousands of developers write in it every day. Game engines are written in Javascript.

Q: Isn't 16 megabytes per process awfully limiting?

A: No, not really. First of all, that's only the memory mapped, executable portion of memory, per process. The Segment can be as large as you want. A single process could potentially have hundreds of gigabytes of memory in the Segment.

Second, if you look at most processes running on a typical system, the vast majority of them use less than 16 megabytes, and if you actually look at the amount of memory a process actually uses rather than how large their virtual address space is (which is what sysmon shows) then you will find it is rather small.

Third, Sisa16 is designed for microcontrollers and low-end systems, and due to how memory segmentation works in Sisa16, making per-process main memory much larger would cause the number of processes available at a time to be very small.

Fourth, much of the memory usage you see in software today comes from bloat. Huge dynamically linked libraries. Javascript interpreters doing almost nothing. Anything which uses electron for instance is going to have an enormous base memory usage. Most software is just extremely poorly designed.

Q: You complain about interpreters being unnecessary overhead, but isn't Sisa16 just *another one*?

A: See for yourself.



clock.asm	gek	12	34152	23.7 MiB
-----------	-----	----	-------	----------

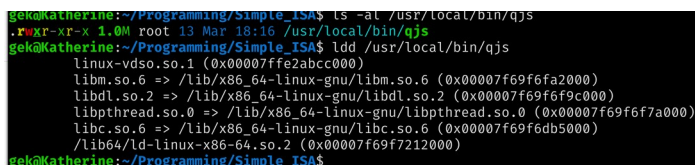
That's Krenel running a program. Now, the host is doing a trick here to reduce memory usage on the system, that's why only 24 megs are in use. Linux is stealing cycles to futz with the MMU and whatnot. The emulator is tiny itself, the x86_64 program can fit inside of a C64's memory map when dynamically linked, at around 49k. For full transparency, Here are both the dynamically linked and static variants.



```
gek@Katherine:~/Programming/Simple_ISA$ ls -al /usr/bin/sisa*
-rwxr-xr-x 102k root 14 Mar 1:57 /usr/bin/sisa16_asm
-rwxr-xr-x 143k root 14 Mar 1:57 /usr/bin/sisa16_dbg
-rwxr-xr-x 49k root 14 Mar 1:57 /usr/bin/sisa16_emu
gek@Katherine:~/Programming/Simple_ISA$

gek@Katherine:~/Programming/Simple_ISA$ ls -al /usr/bin/sisa*
-rwxr-xr-x 844k root 14 Mar 1:51 /usr/bin/sisa16_asm
-rwxr-xr-x 860k root 14 Mar 1:51 /usr/bin/sisa16_dbg
-rwxr-xr-x 746k root 14 Mar 1:51 /usr/bin/sisa16_emu
gek@Katherine:~/Programming/Simple_ISA$ ldd /usr/bin/sisa16_asm
not a dynamic executable
gek@Katherine:~/Programming/Simple_ISA$
```

Compare the assembler, which is practically a full IDE, to QuickJS, a small Javascript implementation in C:



```
gek@Katherine:~/Programming/Simple_ISA$ ls -al /usr/local/bin/qjs
-rwxr-xr-x 1.0M root 13 Mar 18:16 /usr/local/bin/qjs
gek@Katherine:~/Programming/Simple_ISA$ ldd /usr/local/bin/qjs
linux-vdso.so.1 (0x00007ffe2abcc000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f69f6fa2000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f69f6f9c000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f69f6f7a000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f69f6db5000)
/lib64/ld-linux-x86-64.so.2 (0x00007f69f7212000)
gek@Katherine:~/Programming/Simple_ISA$
```


Node seems deceptively small, however NodeJS hides most of its size in dynamically linked libraries, The actual executable is 26k, but it loads a behemoth 36M SO, alongside a dozen others...

```
gek@Katherine:~/Programming/Simple_ISA$ ls -al /usr/bin/node
-rwxr-xr-x 26k root 11 Aug 2021 /usr/bin/node
gek@Katherine:~/Programming/Simple_ISA$
gek@Katherine:~/Programming/Simple_ISA$ ls -al /lib/x86_64-linux-gnu/libnode.so.72
-rw-r--r-- 36M root 11 Aug 2021 /lib/x86_64-linux-gnu/libnode.so.72
gek@Katherine:~/Programming/Simple_ISA$
```

Not even sim65, a MOS 6502 emulator, can beat Sisa16. The same CPU used by the NES and C64:

```
.rwxr-xr-x 89k root 19 Aug 2021 sim65
.rwxr-xr-x 78k root 19 Aug 2021 sp65

gek@Katherine:~/Programming/Simple_ISA$ ldd /usr/local/bin/sim65
linux-vdso.so.1 (0x00007ffc59be9000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fc4626fb000)
/lib64/ld-linux-x86-64.so.2 (0x00007fc462916000)
gek@Katherine:~/Programming/Simple_ISA$
```

z80emu is a little less than half the size of Sisa16's emulator, but again, that's an 8 bit micro and it's ONLY the CPU. No devices. I don't think it even uses string.h

```
B .r--r--r-- 20k root 6 Mar 15:57 z80emu.a
B gek@Katherine:~/Programming/Simple_ISA$
```

The overhead for a sisa16 implementation is **miniscule** compared to other interpreters.

You might argue that Sisa16, in its intended use-case, is likely larger than its size here. Afterall- the BIOS would have to implement a lot more. *Firstly, I disagree, HOWEVER*, to make a fair comparison, you'd have to look at how large the Linux kernel is.

```
gek@Katherine:~/Programming/Simple_ISA$ ls -al /boot/vmlinuz-5.10.0-12-amd64
-rw-r--r-- 6.8M root 7 Mar 16:06 /boot/vmlinuz-5.10.0-12-amd64
gek@Katherine:~/Programming/Simple_ISA$
```

This is just the kernel image itself, which is compressed, without blobs. The init Ramdisk is quite hefty...

```
gek@Katherine:~/Programming/Simple_ISA$ ls -al /boot/initrd.img-5.10.0-12-amd64
-rw-r--r-- 50M root 9 Mar 19:08 /boot/initrd.img-5.10.0-12-amd64
gek@Katherine:~/Programming/Simple_ISA$
```

And firmware is another 7.8 Megs

```
gek@Katherine:/usr/lib/firmware$ du -h
4.7M ./intel-ucode
56K ./RTL8192E
8.0K ./dsp56k
716K ./rtl_bt
324K ./rtw89
660K ./rtw88
1.2M ./rtlwifi
8.0K ./isci
12K ./keyspan_pda
68K ./cis
136K ./rtl_nic
8.0K ./av7110
7.8M .
gek@Katherine:/usr/lib/firmware$
```

Even if the overhead for writing a BIOS implementation for Sisa16 increased its size twentyfold, it would still be much, much smaller than Linux plus any other interpreter. I actually believe that a freestanding Sisa16 implementation would be much smaller than a hosted implementation, due to the lack of necessity for a full Libc, complexity of MMU management and threading code, and of course, a native implementation wouldn't be running the assembler, it would run the emulator.

Q: Alright, so maybe the interpreter and BIOS would be pretty small. But, wouldn't the kernel written on top of Sisa16 end up being very large?

A: I wrote a monolithic kernel with its own tiny shell in Sisa16's standard library, and it compiled down to "libc_pre.bin". Let's check out how big it is:

```
.rw-r--r-- 19k root 14 Mar 2:26 libc_pre.bin
.rw-r--r-- 2.0k root 14 Mar 2:26 libc_pre.hasm
```

Granted, KRENEL is pretty lackluster in featureset. Imagine it being 1024 times larger- 19 Megabytes.

A thousand times larger and it would still be way smaller than Linux!

What's more, most of the space taken up by the .bin is actually **reserved space** for the malloc implementation's allocation bitmap- just zeroes. The kernel fits in this space:

```
VAR#LIBC_KRENEL_BEGIN#2549
VAR#LIBC_KRENEL_END#3871
VAR#libc_COMMAND_COM#4119
VAR#libc_alloc_page_map#4432
gok@Katherine:~/Programming/Simple_ISA$
```

1322 bytes. Less than 1.5k for the kernel itself. The entire Libc without the reserved space for the allocator is 4,432 bytes, or 4.33k.

Admittedly, Krenel hardly does as much as Linux does. Only one scheduler, no permissions, IPC can go haywire, and serial-only. But it could fit onto an NES cartridge.

The shell takes up less than 400 bytes. It does even less, *but that could really change in just a couple K!*

With space efficiency like that, Imagine how much more could be written in a 6 megabyte KRENEL!

Addendum Dedicatory

Thanks and Praise go to my Creator, Lord, and Savior, Yeshua, the Son of God, YHVH, Jesus of Nazareth, the One true Living God.

Believe on him and you shall have eternal life.