Simple Instruction Set Architecture 16

Programmer's Manual and Teaching Resource
1st Edition

By David MHS Webster

Table of Contents

Contents

Introduction

Sisa16 is a virtual machine based instruction set architecture which mirrors many features and quirks of real-life ISAs while being useful and unique in its own right. Its portability is novel while still being powerful enough to create extremely dynamic and interesting programs. It has priviled execution and memory protection… even though it has no Memory Mapping Unit.

Learning how to write code for SISA16 will give you an appreciation of low level programming, the fundamentals of systems engineering, as well as provide you with many new tools to use in your own projects. As an interpreted language, SISA16 is extremely competitive in speed, size, and overall footprint. The emulator can be made to interface with any C library by creating a “driver”, and the emulator can be embedded trivially into any C program as an extension language. This “driver” behaves like a BIOS for the virtual machine.

Whereas most assembly language courses are limited to blinking LEDs and printing characters to a tiny display, Sisa16 can be made to render color graphics, play sounds, use a network connection, work with files, and interact with the host in a manner which is accessible to, and even extensible by first year Computer Engineering students. A hypothetical driver could attempt to mirror real-life hardware interfaces, or just be something that works. This provides enormous flexibility to educators and developers alike, and makes Sisa16 useful in its own right. *See “**Extending**”*.

The assembler is designed for rapid prototyping and debugging. A full debugger with the ability to manipulate registers and view a disassembly is available. A unix shebang can be added to the top of an assembly file and it can be executed as a script. Helpful and specific error messages are displayed. Assembly-time syntax checking and validation is done to avoid programming errors. Special helper features are built into the assembler to provide the programmer with assistance in writing a working program.

Sisa16, including all resources, code, examples and documentation, lies fully in the public domain. It is my gift to you in the hopes that you find it useful, or at least entertaining. An example driver is provided which can load files into the segment and interact with the terminal. A complete set of tools for working with the language is provided. An SDL2 device driver is included for your amusement.

“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 tinyc, 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_SDL2 -lSDL2 -lSDL2_mixer
Include access to an SDL2 device, which allows for rendering to a screen.

-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. Only active if not using SDL2.

-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 standar 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:

./sisa16_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 sisa16_emu
cc debugger.c -DUSE_UNSIGNED_INT -o sisa16_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...

./sisa16_asm -i wave.asm -o wave.bin
./sisa16_emu wave.bin

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

./sisa16_asm -dis wave.bin 0

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

```
gek:~/Programming/Simple_ISA$ sisa16_asm -dis wave.bin 0
//Beginning Disassembly

section 0x0;
//<Region Boundary 0x000000 >
la          0x0a                ;//0x000000  :
putchar                         ;//0x000002  : <DEVICE> write A to device.
la          0x0d                ;//0x000003  :
putchar                         ;//0x000005  : <DEVICE> write A to device.
la          0x01                ;//0x000006  :
lfarpc                          ;//0x000008  : Region Jump
halt                            ;//0x000009  : End of Control Flow
halt                            ;//0x00000a  :
halt                            ;//0x00000b  :
halt                            ;//0x00000c  :

//Reached Halt/Invalid Opcode Limit. Disassembly finished.
```

(if you know the language, you should be able to follow the control flow to 0x10000 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 sisa16_asm.1, sisa16_dbg.1, and sisa16_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

sisa16_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:

sisa16_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:

sisa16_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.

sisa16_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

sisa16_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.

sisa16_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
);

It should be noted, that if your library requires initialization and finalization, and you enable the “emulate” feature of the instruction set, that you should check the “EMULATE_DEPTH” variable as every emulation level will invoke di upon initialization and dcl upon finalization.

An emulation depth of zero means that the virtual machine is truly initializing or finalizing, rather than simply encapsulating.

Relevant emulator variables accessible from the driver-

```
unsigned char R                     :error flag.
u M_SAVER[2][0x1000000]             :Memory. Kernel[0] and User[1]
unsigned char *SEGMENT              :Segment pointer.
UU SEGMENT_PAGES                    :Number of 256 byte “pages” in the segment.
sisa_regfile REG_SAVER[2]           :Saved registers for Kernel[0] and User[1]
```

You should use these variables in your driver to achieve the best interoperability with Sisa16.

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"*. Hand-holding to this extent would hinder the educational value of these tools.
2) It is important that all Engineering students learn how to read unix manual pages. Love it or hate it, every utility they will use in their career will probably be documented by them. Better they learn how to read them now than later.
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 is 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 boundar ("region")
* A region of memory can be filled with a single byte value. ("fill")
* the ouput 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 pre-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. Vasm is probably the nicest looking I've ever seen, but I still prefer Sisa16. Even the inappropriate syntax highlighting of my favorite editor still leaves Sisa16 a far prettier language.

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.

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

```
;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 (hex!) 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 only in the first 64k “region” of memory, the “Zero region”. To access the rest of memory, you must use the far memory system.

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;lb2; 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 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, but also more dangerous. It is possible using this pattern to access arrays that cross region boundaries, since incrementing beyond the region boundary will not loop to the beginning of the 64k region as with the previous examples. However, care must be taken that the array is properly aligned. A single element in the array cannot straddle the region boundary. That is to say, you cannot have a two-byte “short” value starting at 0xAFffFF, as the indirect memory access would

not properly load the byte at 0xB00000 as the low byte, but rather the one at 0xAF0000. This is an intentional design choice of the architecture, as it mirrors the quirks of a real-life microprocessor which uses 16 bit arithmetic.

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: farpagel and farpagest.

```
;lla %10%; sc %20%; farpagel;
```

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.

farpagest 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. 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. This is in opposition to the IEEE 754 standard, but is intentional: a NAN should always be considered erroneous for a floating point value. It should never be used as a magic value.

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:
```

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 a secret 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 is changed by these instructions:
* lfarpc
* farcall
* farret
* farjmprx0

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 hexidecimal 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 your entire program fits 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.
	astp;lb3;sub;ca;ilda;
	//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 eight classes of arithmetic instruction in Sisa16.
* 16 bit arithmetic. Div and Mod are unsigned.
* 16 bit fixed point multiplies. Works regardless of sign. Available for every possible location of the “binimal” point.
* 32 bit arithmetic. Div and Mod are unsigned, except for…
* 32 bit signed division instructions, rxidiv and rximod.
* 32 bit floating-point arithmetic.
* Comparison instructions. Available for 16 bit unsigned, 32 bit unsigned, 32 bit signed, and floating point.
* Floating-point and Integer conversion ops. The instructions are rxitof and rxftoi, which convert a signed integer to float and float to signed integer, respectively.
* Integer increment/decrement instructions.

Note to Implementers: This is a serious avenue for expansion. Vectorized instructions are all the rage now. I considered adding vectorized ops and fused multiply-adds but ultimately decided against it. “Too CISCy”.

All of them operate in fundamentally the same way: previously in the program, operands will have been loaded into registers. The operation is performed, and the result is stored into the “primary” of those two registers. 32 bit arithmetic operations always store in RX0. 16 bit arithmetic operations and all comparisons store the result in register A. All 16 bit arithmetic operations use registers A and B. All 32 bit arithmetic operations occur between RX0 and RX1.

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, or about a terabyte.

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_realloc: resize the segment to size specified by RX0. if RX0 is 0, the Segment is freed. (1 byte) (AD)

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 can be resized with seg_realloc. Failure to resize the segment will generate error 7.

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 (16,17,18)
- A syscall is made (19)
- A preemptive context switch occurs. (255)
- A 'halt' instruction is reached. (0)
- An illegal opcode is executed (0)
- 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;}
}
```

(C to assembly translation not provided, but you can see a similar and more fleshed out example in the included standard library, emulation.hasm)

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.

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 “Driver”

The driver 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 for the active task.

An example Kernel.

An example kernel is written in emulation.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.
- Fixed memory usage
- Pass-through for getchar and putchar
- Interrupts ten and 13 passed through as well, along with matching syscalls.
- After it shuts down, it notifies the user by printing out a message
- Cleans up after dead tasks.
- Shutdown syscall (A=0xDEAD) <no arguments>
- Kill PID (A=0xDE00) B=pid to kill
- Disk Write (A=0xDE01) B=user page to write, RX0 = destination on disk, nonaligned. Return value is 1 if it succeeded.
- Sleep (A=0xDE02) <no arguments>
- Disk Read (0xDE03) B=user page to read into RX0 = source on disk, nonaligned. Return value is 1 if it succeeded.
- Exec (0xDE04) B=user region to create new process from. It will be made into region 1 (default) of the new process, and a bootloader for it will automatically be writetn.
- IPC_write (0xDE05) RX0 =Address in target, B= the value (1 byte) you want to write, C = the PID of the process you want to write it in.
- Fork (0xDE06) <no arguments> returns 1 if it succeeded.
- GetPID (0xDE07) <no arguments> returns the PID of the current process. Valid PIDs are 0-8 with the default configuration of KRENEL.
- OwnSegment (0xDE08) <no arguments> The current PID will be given ownership of the segment if it is not locked. It will be locked if this can be granted. The return value is 1 for success, 0 for failure.
- Release Segment (0xDE09) <no arguments> If the current PID owns the segment, the lock on the segment will be released.

Challenges

I've taught you enough about sisa16 assembly language to do almost everything in it that you would do in C.

If you feel comfortable enough with the language, I would recommend trying some of these challenges:

Very Easy:
* Write translations of C control flow structures into Sisa16. For loops, while loops, If/else/elseif, switch-case, etcetera.
* Implement a simple clock that counts hours, minutes, and seconds.
* Print the first 10 Fibonacci numbers given two starting numbers. You can use the Hexidecimal printing routine from this manual.

Easy:
* Write a program that takes in hex numbers entered via the keyboard and prints them in binary. Halt the program if a non hex digit is entered.
* Allow the user to type in a string on standard input, handle a sentinel character such as newline (0xA ascii) and spit it back out to them. Also try providing the length of the string, strlen.
* Write a calculator program that takes in multi-digit integer numbers in a base of your choice and prints the result of various math operations on them (add, subtract, multiply, divide, modulo). A four-function calculator.

Medium:
* Create a bootloader program that can take in a sisa16 ROM image from standard input, and then begin executing it. For full compatibility, Make sure that when the bootloaded program begins executing, all registers are set to zero. Bonus points if you can utilize the entire address space. The program should be able to boot itself. TIP: remember that when the program counter reaches the end of a 64k region of memory, it loops back around to the beginning.
* Write a routine that can display floating point numbers in decimal.
* Execute code in User mode.
* Add new functions to the standard library.

Hard:
* Create a brainfrick interpreter which has at least 64k 8bit cells and 64k of program space. Bonus points if you can make it work with more.
* Write an assembler for Sisa16. It need not be compatible with the existing syntax- in fact there may be some merit to inventing your own. You can receive source code on standard in, or read from a file. Implement some way which allows people to create forward jumps and labels. Bonus points if it's actually written **in** sisa16 assembler.
* Interpret arithmetic with order of operations. 5 * 3 + 2 * 7 should correctly parse as 15 * 14 which is 210.

* Write an emulator for a simpler CPU. It could be a real one, or it could be one you make up. If you're looking for real ones to emulate, the 6502 is an obvious choice. Provide some means of interacting with it, whether that's memory mapped I/O or a couple added instructions.

Very Hard:
* create an interpreter for an oldschool language like Basic.
* Compile a high level language like C to sisa16. You could make your own. Advanced users might want to investigate writing an LLVM backend.
* Write an emulator for sisa16… in sisa16! Make sure to allow stepping through instructions. If you're feeling lucky, make it able to execute itself. TIP: Use the Segment.
* Write a simple kernel for Sisa16, similar to Krenel. Try implementing a more complicated device driver in d.h, and implementing syscall support for it in your kernel.