

August 1, 2024

SEIS: The Simple Extensible Instruction Set

Francisco Santana

OVERVIEW

This instruction set takes pages from the MOS 6502 instruction set and the ARM instruction set. Namely, it will be a RISC processor with a one word, one instruction paradigm, a 64K “zero”-page (which can be accessed using a 16-bit addressing immediate), and a 64K stack. It will have sixteen variable registers and a handful of processor state registers. It will also enable register-to-register transfers, which is the only way to access the processor-state registers.

The “short”-page is designed to allow the program to access frequently used values in a single instruction. For the purposes of the simulator, the short-page will be located in the second page – right after the stack. The first page will be reserved for the application code, but the application code does not have to live on the first page. In addition, pages will be 65,536 bytes in size.

This instruction set architecture is a clean-sheet general-purpose architecture. Some of the key features of this instruction set architecture are:

- Support for loading values of 8 (bytes), 16 (shorts), and 32 (long) bits in width.
- Explicit sign-extension instructions.
- Typical 5-stage RISC pipeline.
- 16 32-bit registers, which are general-purpose.
- Separate control registers for the program counter, stack base, stack pointer, and return pointer.
- Transfer between any two registers.
- ARM-like register storage.
- Words are 32 bits in width.
- Memory-mapped I/O (which is currently not implemented in the simulator).

REGISTERS

Canonical Name	Number	Description
<i>V0-V7</i>	0-7	Variable registers, conventionally for subroutine parameters and return values, but may be used for any purpose
<i>V8-VF</i>	8-15	Variable registers, for any purpose
<i>PC</i>	16	30-bit program counter (least significant bits are appended, 00)
<i>SP</i>	17	14-bit stack pointer (most significant 16 bits = 0x0001, least significant 2 bits = 00)
<i>BP</i>	18	14-bit base pointer (most significant 16 bits = 0x0001, least significant 2 bits = 00)
<i>LP</i>	19	30-bit link pointer (least significant bits appended = 00)
<i>ZF</i>	20	Zero flag
<i>OF</i>	21	Overflow flag
<i>EPS</i>	22	Epsilon equality flag
<i>NAN</i>	23	Not-a-number flag
<i>INF</i>	24	Infinity flag

In assembly, *all registers are referred to by their canonical names*. This is to prevent any confusion that may arise from using the incorrect register. Accessing the PC will always yield the address of the instruction accessing the PC. Addressing invalid registers will yield zero in the simulator. ZF, OF, EPS, NAN, and INF, if loaded, will yield zero in the higher-order bits and the stored value in the lower-order bit.

In a similar fashion to how the MOS Technology 6502 processor works, the SP and BP will be 16 bits wide for the simulator, meaning that the stack can support up to 64 KB of size, but the upper 16 bits are always set to 0x0001, meaning that the stack starts at about 64K from the start of the address space. The stack grows up, and the registers will point to (not above) the top of the stack, and pushing values to the stack will subtract from the SP. The stack is used to store register values, and so there are operations explicitly meant to use the stack. Values may be pulled from the stack directly. The endianness is big-endian.

The assembly syntax primarily uses trigrams like ADD or SUB, but some instructions will use tetragrams like PUSH or FMUL. Assembly syntax puts the destination register on the rightmost operand slot, as it is consistent with the physical layout of the bits of an instruction word.

INSTRUCTIONS

All instructions will begin with an OP_{TY}, which may change the context of the subsequent 29 bits.

[illegible]

Here are the descriptions of what kinds of instructions each OPTYPE supports, which is not finalized:

<i>OPTY</i>	Description
<i>000</i>	Control instructions
<i>001</i>	32-bit integer arithmetic
<i>010</i>	32-bit floating-point arithmetic, including converting to/from a float
<i>011</i>	Register transfers, loads, and stores

Leaving bits available for new instructions allows me to add extensions like *vector extensions*, *wide type support*, or privileged *supervisor instructions* later into the development process, should I (or an adopter) decide to do so.

I will simply list instructions that will be implemented by the simulator. Assume that any opcode *not mapped to an instruction* is a no-op.

Control Instructions

The immediate offset is sign-extended to a 32-bit integer. It is also always shifted two bits to the left. The layout of a control instruction is as follows:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F			
0	0	0	OPCODE	0	<UNUSED>																		Address		<UNUSED>									
				1	24-Bit Offset (Immediate)																													
				Unused in HALT, NOP, and RET																														

Operation	Description	OPCODE
<i>NOP</i>	Reserved no-op instruction	0000
<i>HALT</i>	Stops the processor	0001
<i>JMP</i>	Jumps to an address relative to the PC or given by a register	1001
<i>JSR</i>	Jumps to a subroutine, setting the LR with the address of the following instruction	1000
<i>RET</i>	Returns from the subroutine by setting the PC to the value of the LR	1010
<i>JEQ/JZE</i>	Performs a jump if the zero flag is set	0010
<i>JNE/JNZ</i>	Performs a jump if the zero flag is <i>not</i> set	0011
<i>JGT</i>	Performs a jump if neither the zero flag nor the overflow flag are set	0100
<i>JLT</i>	Performs a jump if the zero flag is unset but the overflow flag is set	0101
<i>JGE</i>	Performs a jump if the overflow flag is not set or the zero flag is set	0110
<i>JLE</i>	Performs a jump if the overflow flag is set or the zero flag is set	0111

Immediate values are not sign-extended in the Option (Immediate) form. They are, however, sign-extended in the Right Value (Immediate) form. Len is the width, in bytes, of the value minus one.

[illegible]

All instructions will set the ZF and OF registers accordingly.

Operation	Description	OPCODE
<i>ADD</i>	Adds a register to another register or immediate value	00000
<i>SUB</i>	Subtracts a register from another register or immediate value	00001
<i>MUL</i>	Multiplies a register to another register or immediate value	00010
<i>DVU</i>	Divides a register by another register or immediate value, for unsigned integers	00011
<i>DVS</i>	Divides a register by another register or immediate value, for signed integers	00100
<i>MOD</i>	Computes the modulo of a register and another register or an immediate value	00101
<i>AND</i>	Computes the bitwise AND of a register and another register or an immediate value	00110
<i>IOR/OR</i>	Computes the inclusive OR of a register and another register or an immediate value	00111
<i>EOR/XOR</i>	Computes the exclusive OR of a register and another register or an immediate value	01000
<i>NOT</i>	Computes the bitwise NOT of a register	01001
<i>SXT</i>	Sign-extends a register (uses the third form)	01010
<i>BSL/ASL</i>	Logical shift left a register by an immediate value or another register, <i>modulo 32</i>	01011
<i>BSR</i>	Logical shift right a register by an immediate value or another register, <i>modulo 32</i>	01100
<i>ASR</i>	Arithmetic shift right a register by an immediate value or another register, <i>modulo 32</i>	01101
<i>ROL</i>	Rotate left a register by an immediate value or another register, <i>modulo 32</i>	01110

The offsets are not sign-extended.

The only instruction that can utilize any processor register is the TFR instruction. The “V” bit for the load and store instructions represents a *volatile* variant, which bypasses the cache entirely. In assembly, it is represented by using a “super arrow”, which is either “ \longrightarrow ” or “ \Longrightarrow ”, in the place of a typical assignment syntax.

No loads/stores will automatically shift register values, but they will shift immediate values. If a load or store accesses a misaligned address, the instruction will take additional cycles to complete. If a load reads a zero, it will set the zero flag.

USER MANUAL

This project has three major executables: the assembler (`seis-asm`), the simulator (`seis-sim`), and the graphical user interface (`gui.exe`).

Assembly Syntax

The syntax is like what you'd expect from the ARM assembly syntax, except the destination register is the third register, which follows either an arrow (\rightarrow or \Rightarrow) or a comma.

The assembler allows for the use of constants and for the locations of instructions to be specified by using directives (syntactically like attribute macros in Rust).

The assembler can take multiple files. Only the first file is made available on the first page, but subsequent files are placed in free pages (after the stack and zero page). The location directive allows you to specify where you want to start placing data or instructions. All labels and constants are visible to all files passed to the assembler. Duplicate constants and labels are erroneous, even across files.

The assembler will place instructions at the beginning of a page. Wherever possible, the assembler will try to inline constants. If that is not possible, the assembler will either insert loads to store the constant into the destination register (and use the destination register as the option operand) or throw an error if that is not possible because the destination was an operand.

Strings are automatically null terminated by the assembler. Nulls in strings are considered an error.

The assembler has been made using Pest, which provides rustc-style error messages telling you where an error is and why the error is an error.

Control Instructions

INSTRUCTION	EFFECT
NOP	Stalls the processor
HALT	Stops the processor
JMP <JUMP>	Jumps unconditionally
JSR <JUMP>	Jumps and sets the LR to 1 + PC
RET	Jumps to the address provided by the LR
JEQ <JUMP>	Jumps only if ZF is set
JNE <JUMP>	Jumps only if ZF is not set
JGT <JUMP>	Jumps if ZF is not set and OF is not set
JLT <JUMP>	Jumps if ZF is not set and OF is set
JGE <JUMP>	Jumps if ZF is set or OF is not set
JLE <JUMP>	Jumps if ZF is set or OF is set

All <JUMP> parameters take on the following syntax:

SYNTAX	RESULT
label	The assembler will try to resolve the location of the label relative to the current instruction. If the label cannot fit, it throws an error.
[+/-]imm	Jumps imm instructions before or after the PC at the instruction. (imm is multiplied by 4)
Vx	Jumps to the address specified by the register (lower-order two bits are ignored)

Integer Arithmetic Instructions

The final comma may be replaced by an arrow. S stands for “Source”, O stands for “Option”, and D stands for “Destination”. These will set the ZF if the result yields a zero and the OF if the result overflows or underflows.

INSTRUCTION	EFFECT
ADD S, O, D	Adds S and O and writes the result to D
SUB S, O, D	Subtracts O from S and writes the result to D
MUL S, O, D	Multiplies S by O and writes the result to D
DVU S, O, D	Divides S by O and writes the result to D
DVS S, O, D	Divides S by O and writes the result to D (this is signed)
MOD S, O, D	Computes S modulo O and writes the result to D
AND S, O, D	Bit-and S and O and writes the result to D
IOR S, O, D	Bit-or S and O and writes the result to D
XOR S, O, D	Bit-xor S and O and writes the result to D
NOT S, D	Writes the bit-not of S to D
SXT T D	Where T is byte, short, or word (which results in a nop), sign-extends D to the full width of the register
BSL S, O, D	Binary shift left S by O bits and writes the result to D
BSR S, O, D	Binary shift right S by O bits and writes the result to D
ASR S, O, D	Arithmetic shift right S by O bits and writes the result to D
ROL S, O, D	Rotate left S by O bits and writes the result to D
ROR S, O, D	Rotate right S by O bits and writes the result to D
CMP L, R	Subtracts L from R without writing to a register
CMP S L, R	Performs a signed comparison. ZF is set as normal, but OF is set if $L < R$.
TST L, R	Bit-and L and R without writing to a register

Floating-Point Arithmetic

All operations will set ZF if the result is zero, OF if the result negative, EPS if the result is zero within an error, INF if the result is infinity, and NAN if the result is not a number.

INSTRUCTION	EFFECT
FADD S, O, D	Adds S and O and writes the result to D
FSUB S, O, D	Subtracts O from S and writes the result to D
FMUL S, O, D	Multiplies S by O and writes the result to D
FDIV S, O, D	Divides S by O and writes the result to D
FMOD S, O, D	Computes S modulo O and writes the result to D
FCMP L, R	Subtracts L from R without writing to a register
FNEG S, D	Negates S and writes the result to D
FREC S, D	Computes the reciprocal of S and writes the result to D
ITOF S, D	Converts S to a float and writes the result to D
FTOI S, D	Converts S to an int and writes the result to D
FCHK R	Sets the ZF, OF, EPS, INF, and NAN registers

Register Operations

INSTRUCTION	EFFECT
LBR <ADDR>, D	Loads the byte at ADDR into D. Using the \Rightarrow syntax will use the volatile version.
LSR <ADDR>, D	Loads the short at ADDR into D. Using the \Rightarrow syntax will use the volatile version.
LLR <ADDR>, D	Loads the word at ADDR into D. Using the \Rightarrow syntax will use the volatile version.
SBR S, <ADDR>	Stores the lower 8 bits of S into ADDR. Using the \Rightarrow syntax will use the volatile version.
SSR S, <ADDR>	Stores the lower 16 bits of S into ADDR. Using the \Rightarrow syntax will use the volatile version.
SLR S, <ADDR>	Stores the value of S into ADDR. Using the \Rightarrow syntax will use the volatile version.
TFR S, D	Copies the value in S into D.
PUSH <STACK>	Pushes registers into the stack OR adds a value to the stack register.
POP <STACK>	Pops registers from the stack OR subtracts a value from the stack register.
LDR imm, R[.p]	Will load imm to R. If p is specified, will not zero out the rest of the register, but will shift the value by $16 * p$.
LDR &zpa, R	Will add the value of zpa to the base address of the zero page and store the result in R
LOAD <X>, D	Expands to several LDRs to load X to D.

All <ADDR> parameters can take on the following syntaxes:

SYNTAX	RESULT
@addr	Adds addr to the base address of the zero page and loads the value at the resulting address
Vx	Uses the value stored in the register as the effective address
Vx + off	Adds off to the value stored in the address and uses the result as the effective address
Vx[Vy]	Adds the value in Vy to the value in Vx and uses the result as the effective address
%off	Subtracts off from the stack base pointer and uses that as the effective address

The <X> parameter can take on the following syntaxes:

SYNTAX	RESULT
INT FLOAT	Evaluates into the binary representation of the value and expands into one or two loads. If a zero would be loaded into the upper bits, only one load gets generated.
LABEL	Evaluates into two loads to load the address of the label. If the upper bits are 0, a NOP is emitted in the second instruction.
.CONST	Evaluates the binary representation of the value of the constant and expands into one or two loads. If a zero would be loaded into the upper bits, only one load gets generated.

Directives

A directive has the following syntax:

```
#[DIRECTIVE = VALUE]  
#[DIRECTIVE]
```

For now, there is only one directive: it is used to specify *where* to put data or code.

Location

```
#[Location = <LOCATION>]
```

This directive will set the location of any instructions and labels following it.

Attempting to write code to the stack page or zero page using this directive will yield an error. If writing code goes into the stack or short-page, code will be emitted to jump elsewhere and the code will be written past those pages. The assembler will warn you if you may potentially be overwriting other code or data.

Labels

label_ident:

Given the label_ident, binds an address to a label. Labels do not consume locations.

Data Blocks

```
#<float|byte|short|word|string>: {  
    <list of data>  
}
```

This puts data inline at the location in question. The data type specified is what your data block's data is cast into. If there is a mismatch, an error is thrown. If the type is "string", you can put strings there, which are null-terminated. The suggested use is to use a label at the beginning of the data block to point at the data block.

If you want to make a randomized data block, use the following syntax:

```
#<float|byte|short|word>?[min, max; count]  
#<float|byte|short|word>?[min, max; count %seed]
```

The first syntax will create a totally randomized array of count items between the minimum and maximum values. The optional seed, seen in the second syntax, will deterministically generate those values.

This was added during the development of the benchmarks to eliminate the need for manually writing data and hoping it is sufficiently random.

Constants

```
.const_ident = value
```

Binds the value specified to the const_ident specified. This is not stored in memory – for that, use a data block.

Contribution

I wrote all the code for the project.

However, my project also relies on third-party tools. Below is the list of libraries I directly used in my project.

Rust Libraries

[rand](https://crates.io/crates/rand) by Alex Crichton and Diggory Hardy, *et al* (<https://crates.io/crates/rand>)

[rand_chacha](https://crates.io/crates/rand_chacha) by Diggory Hardy, *et al* (https://crates.io/crates/rand_chacha)

[clap](https://crates.io/crates/clap) by Kevin K., *et al* (<https://crates.io/crates/clap>)

[inquire](https://crates.io/crates/inquire) by Mikael Mello, *et al* (<https://crates.io/crates/inquire>)

[serde](https://crates.io/crates/serde) by David Tolnay, *et al* (<https://crates.io/crates/serde>)

[serde_json](https://crates.io/crates/serde_json) by David Tolnay, *et al* (https://crates.io/crates/serde_json)

[toml](https://crates.io/crates/toml) by Alex Crichton, Eric Huss, and Ed Page, *et al* (<https://crates.io/crates/toml>)

[pest](https://crates.io/crates/pest) by Dragoş Tiselice, *et al* (<https://crates.io/crates/pest>)

[pest_derive](https://crates.io/crates/pest_derive) Dragoş Tiselice, *et al* (https://crates.io/crates/pest_derive)

[ratatui](https://crates.io/crates/ratatui) by Orhun Parmaksiz, Arijit Basu, and Leon Sautour, *et al* (<https://crates.io/crates/ratatui>)

[crossterm](https://crates.io/crates/crossterm) by Timon, *et al* (<https://crates.io/crates/crossterm>)

[rayon](https://crates.io/crates/rayon) by Josh Stone and Niko Matsakis, *et al* (<https://crates.io/crates/rayon>)

[wasm-bindgen](https://github.com/rustwasm/wasm-bindgen) and [wasm-pack](https://github.com/rustwasm/wasm-pack) by the rustwasm team (<https://github.com/rustwasm>)

C#.NET Libraries

[Newtonsoft.Json](https://www.newtonsoft.com/json) by James Newton-King, *et al* (<https://www.newtonsoft.com/json>)

[Tomlyn](https://github.com/xoofx/Tomlyn) by Alexandre Mutel, *et al* (<https://github.com/xoofx/Tomlyn>)

[WPF](https://learn.microsoft.com/en-us/dotnet/desktop/wpf/?view=netdesktop-8.0) by Microsoft (<https://learn.microsoft.com/en-us/dotnet/desktop/wpf/?view=netdesktop-8.0>)