April 16, 2024

SEIS:
The Simple Extensible Instruction Set

Francisco Santana

TABLE OF CONTENTS

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 “zero”-page is designed to allow the program to access frequently used values in a single instruction. For the purposes of the simulator, the zero-page will be set 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.

REGISTERS

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

INSTRUCTIONS

All instructions will begin with an OPTY, which may change the context of the subsequent 29 bits.

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
OPTY			Determined by OPTY																												

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

OPTY	**Description**
000	Control instructions
001	32-bit integer arithmetic
010	32-bit floating-point arithmetic, including converting to/from a float
011	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:

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
0	0	0	OPCODE				0	<UNUSED>																Address				<UNUSED>			
							1	24-Bit Offset (Immediate)																							
							Unused in HALT, NOP, and RET																								

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

Integer Arithmetic Instructions

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.

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
0	0	1	ADD, SUB, MUL, DVU, DVS, MOD, AND, IOR/OR, XOR/EOR, BSL/ASL, BSR, ASR, ROL, ROR					0	<UNUSED>											Option				Source				Destination			
								1	15-Bit Option (Immediate)																						
			NOT					<UNUSED																							
			SXT					<UNUSED>																		Len		Target			
			CMP or TST					0	<UNUSED>											Right Value				Left Value				<UNUSED>			
								1	15-Bit Right Value (Immediate)																						

All instructions will set the ZF and OF registers accordingly.

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

Operation	**Description**	**OPCODE**
ROR	Rotate right a register by an immediate value or another register, *modulo* 32	01111
CMP	Compares a register to another register or immediate value (the destination is ignored)	10000
TST	Masks a register with a second register (ignores the destination) This sets the ZF if the two values are equal.	10001

Floating-Point Arithmetic Instructions

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
0	1	0	FADD, FSUB, FMUL, FDIV, FMOD					<UNUSED>												Option				Source				Destination			
			FNEG, ITOF, FTOI					<UNUSED>																							
			FCMP					<UNUSED>												Right Value				Left Value				<UNUSED>			
			FCHK					<UNUSED>																Target				<UNUSED>			

All instructions will set the ZF (for zero), OF (for negative values), EPS (if within an epsilon), NAN, and INF registers accordingly.

Operation	**Description**
FADD	Adds two registers
FSUB	Subtracts a register from another register
FMUL	Multiplies two registers
FDIV	Divides two registers
FMOD	Computes the modulo of a register to another register
FCMP	Compares two floating-point values, setting the floating-point registers appropriately (see FCHK)
FNEG	Negates a floating-point value
FREC	Gets the reciprocal of a floating-point value
ITOF	Converts an int to a float
FTOI	Converts a float to an int
FCHK	Checks a float, setting ZF if zero, OF if negative, EPS if it is zero within an error, INF if infinity, and NAN if not a number

I may add instructions to set the processor status flags appropriately depending on what values the floating-point status registers took on, such as setting the zero-flag if the epsilon-equality flag was set.

Register-Operation Instructions

The offsets are not sign-extended.

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
0	1	1	LBR, LSR, LLR, SBR, SSR, SLR				V	0	0	<UNUSED>														Address				Destination			
							V	0	1	12-Bit Offset																					
							V	1	0	<UNUSED>										Index											
							0	1	1	12-Bit Offset																					
							1	1	1	////		16-Bit Zero-Page Address																			
			TFR				<UNUSED>															Source					Destination				
			PUSH, POP				0	<UNUSED>					LP	BP	SP	VF	VE	VD	VC	VB	VA	V9	V8	V7	V6	V5	V4	V3	V2	V1	V0
							1	<UNUSED>								(16-Bit) Extension or Retraction in Bytes															
			LDR				/	0	Z	SHIFT		16-Bit Immediate Value																Destination			
								1	<UNUSED>			16-Bit Zero-Page Address																			

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 "⟶»" or "⟹»", in the place of a typical assignment syntax.

Operation	**Description**	**OPCODE**
LBR	Load byte to register	0000
LSR	Load short to register	0001
LLR	Load long to register	0010
SBR	Store byte from register	0011
SSR	Store short from register	0100
SLR	Store long from register	0101
TFR	Transfer register (supports transferring to/from any register)	0110
PUSH	Push a register value to the stack (any register)	0111
POP	Pop a register value from the stack (any register)	1000
LDR	Load a 16-bit shifted immediate value to a register, optionally zeroing out the register (determined by the "Z" bit), or a 16-bit zero-page address (the higher-order bits for the zero page are prepended)	1001
LOAD	Not a real instruction – splits into one or two instructions to load the immediate, which, for simplicity, may involve a no-op when loading a label address	////

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.

GAME PLAN

The simulator will be a standalone program that takes inputs from stdin and writes outputs to stdout. This is to have the graphical interface and simulator run as separate processes and, thus, allow me to write them in different programming languages, so that the GUI may be written in a language with good GUI support such as C#.net or Java and the simulator may be written in a faster compiled language like Rust. The processor's state may be queried using a set of commands, allowing the interface to query address and register values and pipeline state. The two processes will communicate using pipes.

The reason for my use of Rust for the bulk of the simulator code is to minimize memory bugs in the simulator and to use libraries that enable me to take full advantage of stdin and stdout by using powerful libraries like CLAP and SERDE to easily exchange messages between the simulator and the GUI and to use a library like Pest to parse the input code. Additionally, it has incredible unit testing features built into Cargo, its package manager and toolchain. This toolchain is consistent across platforms, so code that works on Windows *should* also work on Linux and MacOS.

I plan to use C# for the GUI code, since I have access to Visual Studio on my computer and can easily design the frontend for the simulator. C# does not have as strong of a testing feature set, but that is not of major concern. C# also enables dependency management through NuGet.

The simulator will allow stepping and running the program to completion, and, hopefully, pausing the simulator upon user request. The simulator, using the piped I/O to the GUI, can send register values, address values, pipeline state, and teletypewriter output, which the GUI can display to the user.

The simulator will include TTY support by using simulated hardware registers that the program can read from or write to. Volatile reads/writes are supported to circumvent caching mechanisms.

Versioning will be handled using Git.

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 (→ or ⇒) 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 *do not need to be null terminated by you*; the assembler will append the terminal null. If a null is added in the middle of a string, that is 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 2 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
`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 ==>> syntax will use the volatile version.
`LSR <ADDR>, D`	Loads the short at ADDR into D. Using the ==>> syntax will use the volatile version.
`LLR <ADDR>, D`	Loads the word at ADDR into D. Using the ==>> syntax will use the volatile version.
`SBR S, <ADDR>`	Stores the lower 8 bits of S into ADDR. Using the ==>> syntax will use the volatile version.
`SSR S, <ADDR>`	Stores the lower 16 bits of S into ADDR. Using the ==>> syntax will use the volatile version.
`SLR S, <ADDR>`	Stores the value of S into ADDR. Using the ==>> 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 <XLOAD>, D`	Expands to several LDRs.

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 `<XLOAD>` 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>]]
```

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 zero-page, code will be emitted to jump elsewhere and the code will be written past those pages.

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.

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.