ΕΠΛ221:
Οργάνωση Υπολογιστών και Συμβολικός Προγραμματισμός

Εργαστήριο Αρ. 8

Εισαγωγή στην Αρχιτεκτονική
ARMv8

AArch64 Floating-point and NEON

Πέτρος Παναγή, PhD

Integer Registers used for Instructions

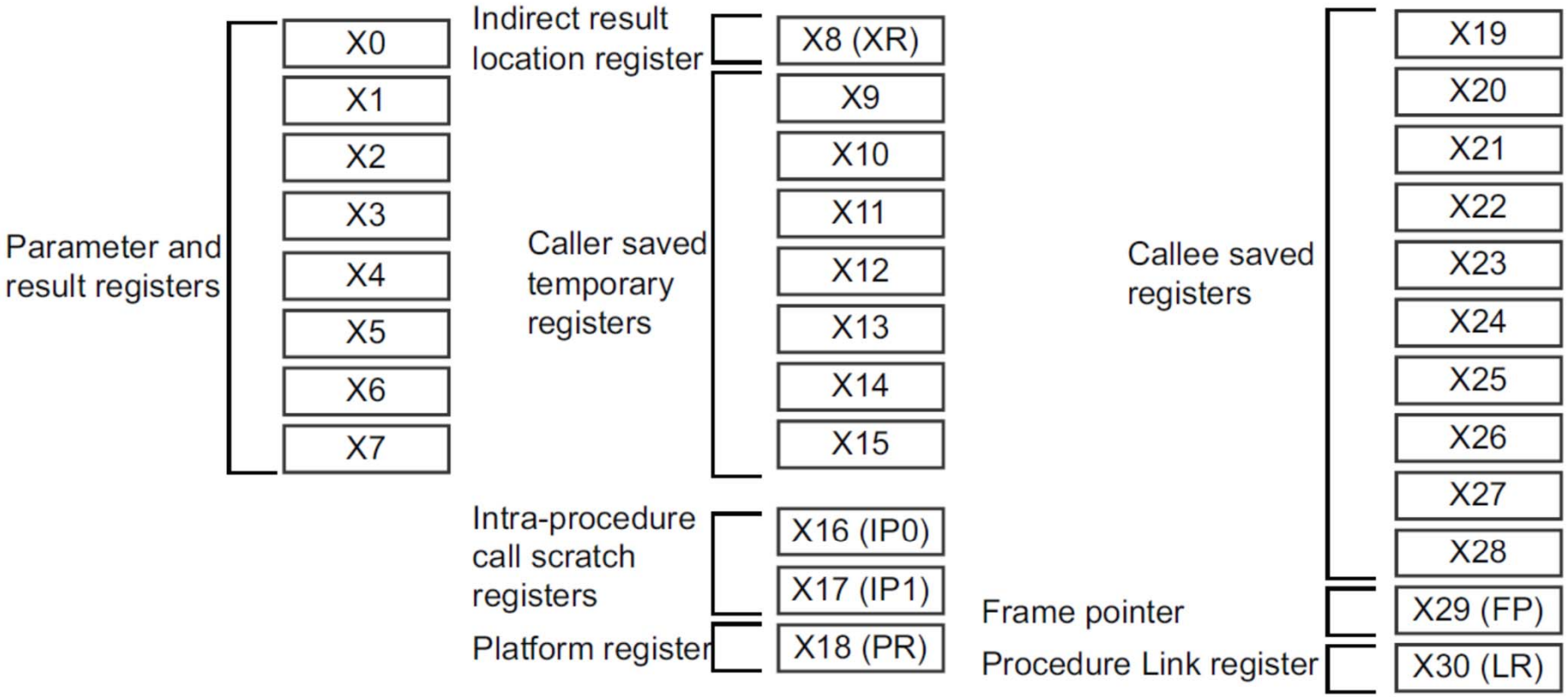
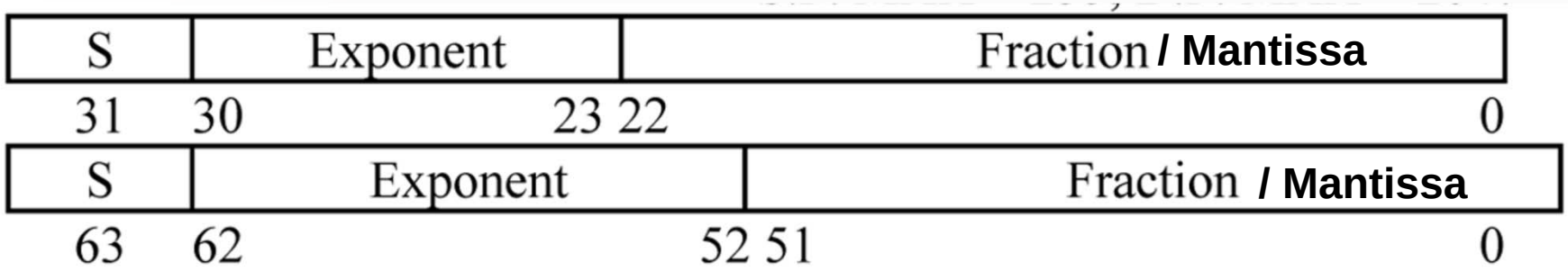


Figure 9-1 General-purpose register use in the ABI

IEEE 754 Single and Double Precision Floating point Numbers



http://ieeexplore.ieee.org/document/4610935/
https://www.h-schmidt.net/FloatConverter/IEEE754.html

Floating Point Sizes

Table 4-7 Operand name for differently sized floats

Precision	Size (bits)	Name
Half	16	Hn
Single	32	Sn
Double	64	Dn

Floating Point Registers

- 32 × 64-bit D registers D0-D31. The D registers are called double-precision registers and contain double-precision floating-point values.
- 32 × 32-bit S registers S0-S31. The S registers are called single-precision registers and contain single-precision floating-point values.
- 32 × 16-bit H registers H0-H31. The H registers are called half-precision registers and contain half-precision floating-point values.

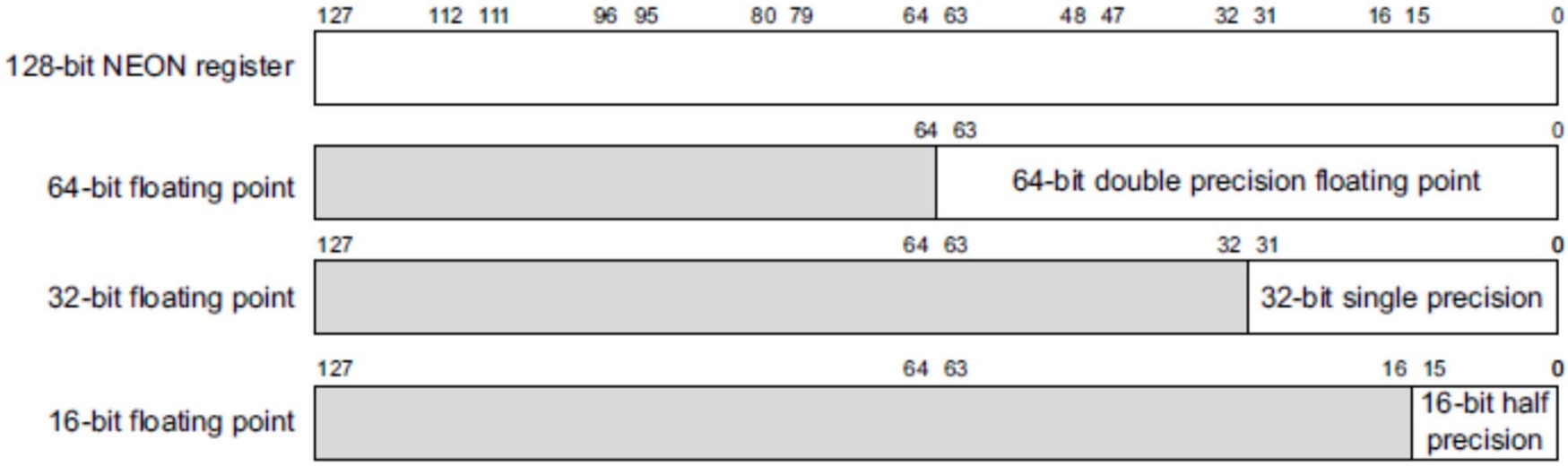


Figure 7-3 Floating-point register divisions

Floating-point register organization in AArch64

5.1.2 SIMD and Floating-Point Registers

The ARM 64-bit architecture also has a further thirty-two registers, v0-v31, which can be used by SIMD and Floating-Point operations. The precise name of the register will change indicating the size of the access.

Note Unlike in AArch32, in AArch64 the 128-bit and 64-bit views of a SIMD and Floating-Point register do not overlap multiple registers in a narrower view, so q1, d1 and s1 all refer to the same entry in the register bank.

The first eight registers, v0-v7, are used to pass argument values into a subroutine and to return result values from a function. They may also be used to hold intermediate values within a routine (but, in general, only *between* subroutine calls).

Registers v8-v15 must be preserved by a callee across subroutine calls the remaining registers (v0-v7, v16-v31) do not need to be preserved (or should be preserved by the caller). Additionally, only the bottom 64-bits of each value stored in v8-v15 need to be preserved[1]; it is the responsibility of the caller to preserve larger values.

The FPSR is a status register that holds the cumulative exception bits of the floating-point unit. It contains the fields IDC, IXC, UFC, OFC, DZC, IOC and QC. These fields are not preserved across a public interface and may have any value on entry to a subroutine.

The FPCR is used to control the behavior of the floating-point unit. It is a global register with the following properties.

- The exception-control bits (8-12), rounding mode bits (22-23) and flush-to-zero bits (24) may be modified by calls to specific support functions that affect the global state of the application.
- All other bits are reserved and must not be modified. It is not defined whether the bits read as zero or one, or whether they are preserved across a public interface.

When Used in Function Calls

AAPCS64: Role of floating-point registers

Register	Role
V0	Return value (for floating-point values)
V0 ... V7	Arguments in function calls (for floating-point values)
V8 ... V15	Callee-saved registers. The bottom 64 bits of these registers are preserved across calls. Upper 64 bits must be saved by the caller if needed.
V16 ... V31	Temporary registers (trashed across calls)

Floating-point register organization in AArch64

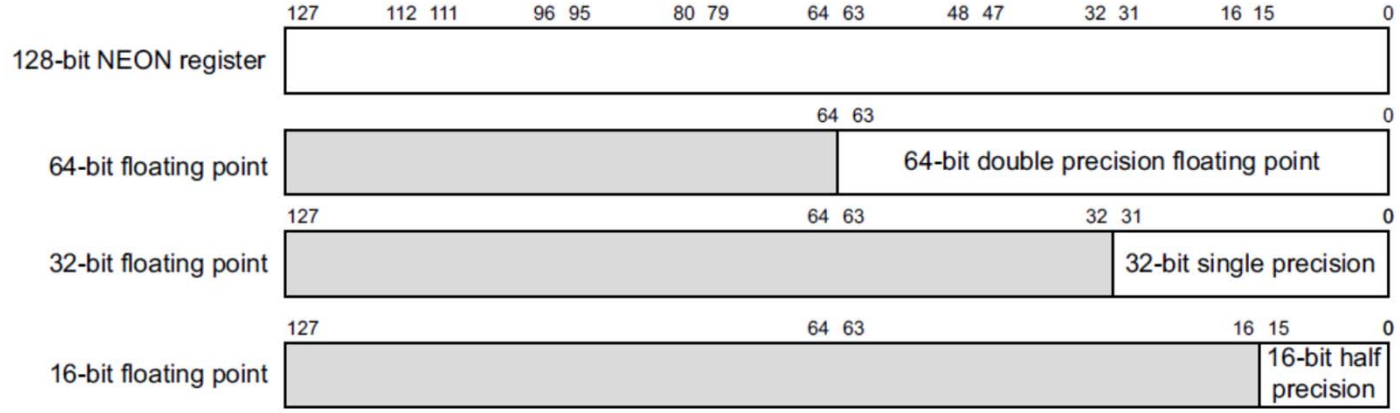


Figure 7-3 Floating-point register divisions

The F prefix and the float size is specified by the floating-point ADD instruction:

```
FADD Sd, Sn, Sm  // Single-precision
FADD Dd, Dn, Dm  // Double-precision
```

The half-precision floating-point instructions are for converting between different sizes:

```
FCVT Sd, Hn  // half-precision to single-precision
FCVT Dd, Hn  // half-precision to double-precision
FCVT Hd, Sn  // single-precision to half-precision
FCVT Hd, Dn  // double-precision to half-precision
```

Floating-point parameters

Floating-point values are passed to (and returned from) functions using the floating-point registers. Both integer (general-purpose) and floating-point registers can be used at the same time. This means that the floating-point parameters are passed in the floating-point H, S or D registers and other parameters are passed in integer X or W registers. The AArch64 *Procedure Call Standard* mandates hardware floating-point wherever floating-point arithmetic is required, so there is no software floating-point linkage in AArch64 state.

A detailed list of instructions is given in the *ARMv8-A Architecture Reference Manual*, but the main floating-point data processing operations are listed here to show the kind of things that can be done:

Table 7-1

`FABS Sd, Sn`	Calculates the absolute value.
`FNEG Sd, Sn`	Negates the value.
`FSQRT Sd, Sn`	Calculates the square root.
`FADD Sd, Sn, Sm`	Adds values.
`FSUB Sd, Sn, Sm`	Subtracts values.
`FDIV Sd, Sn, Sm`	Divides one value by another.

Floating-point parameters

`FMUL Sd, Sn, Sm`	Multiplies two values.
`FNMUL Sd, Sn, Sm`	Multiplies and negates.
`FMADD Sd, Sn, Sm, Sa`	Multiplies and adds (fused).
`FMSUB Sd, Sn, Sm, Sa`	Multiplies, negates and subtracts (fused).
`FNMADD Sd, Sn, Sm, Sa`	Multiplies, negates and adds (fused).
`FNMSUB Sd, Sn, Sm, Sa`	Multiplies, negates and subtracts (fused).
`FPINTy Sd, Sn`	Rounds to an integral in floating-point format (where `y` is one of a number of rounding mode options)
`FCMP Sn, Sm`	Performs a floating-point compare.
`FCCMP Sn, Sm, #uimm4, cond`	Performs a floating-point conditional compare.
`FCSEL Sd, Sn, Sm, cond`	Floating-point conditional select if (cond) Sd = Sn else Sd = Sm.
`FCVTSty Rn, Sm`	Converts a floating-point value to an integer value (ty specifies type of rounding).
`SCVTF Sm, Ro`	Converts an integer value to a floating-point value.

Stack Frame Layout

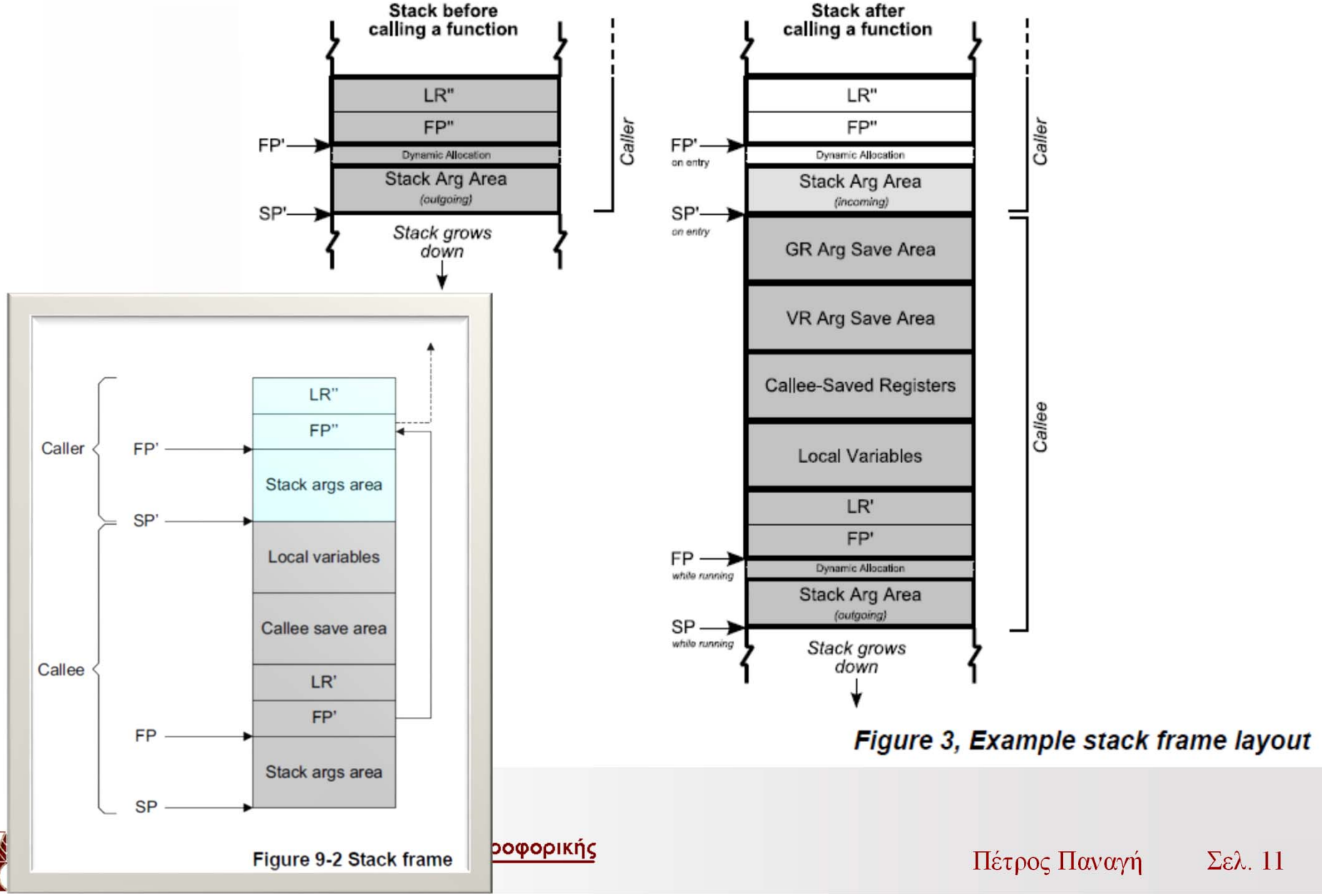


Figure 3, Example stack frame layout

int printf (const char * format, ...);

format

C string that contains the text to be written to stdout.
It can optionally contain embedded *format specifiers* that are replaced by the values specified in subsequent additional arguments and formatted as requested.

A *format specifier* follows this prototype: [see compatibility note below]
`%[flags][width][.precision][length]specifier`

Where the *specifier character* at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument:

specifier	Output	Example
d *or* i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	
%	A % followed by another % character will write a single % to the stream.	%

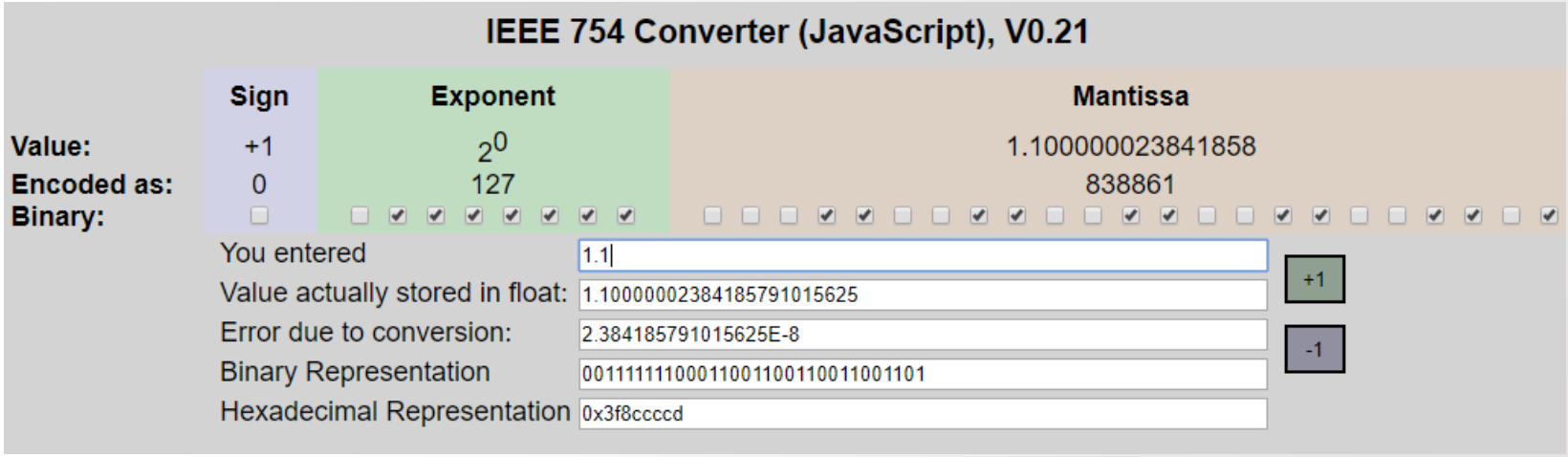
Examples

Lab8_example1.s - Read a Float and print it on the Screen

Lab8_example2.s - Read a Float addid to its self and print it on the Screen

Lab8_example3.s

```
Please give a Float: 1.1
You have given the value 0x3F8CCCCD
```



Examples

Lab8_example4.s

```
Please give a Double: 1.1
You have given the value 1.100000
You have given the value in HEX: 0x3ff199999999999a
```

Lab8_example5.s

```
Please give a Double: 1.1
You have given the value 1.100000
You have given the value in INT64: 1
Please give a Double: 123456789.123456789
You have given the value 123456789.123457
You have given the value in INT64: 123456789
Please give a Double: 123456789.987654321
You have given the value 123456789.987654
You have given the value in INT64: 123456789
```

Convert float to integer (FCVTxU, FCVTxS) instructions encode a directed rounding mode:

- Towards zero.
- Towards $+\infty$.
- Towards $-\infty$.
- Nearest with ties to even.
- Nearest with ties to away.