



Zephyr's “smart” FPU support for ARM64 & RISC-V

Nicholas Pitre

[3] Floating-Point Numbers

What is a floating-point number?

A floating-point number is made of 3 components

- A radix or base (e.g. “10”)
- A signed integer significand (e.g. “-2345612”)
- A signed integer exponent (e.g. “3”)

Resulting value = significand \times base^{exponent}



[4] Floating-Point Numbers

But where is the point?

By convention, the point is usually next to the first significant digit of the significand:

- significand = -2345612
- value = -2_._345612



[5] Floating-Point Numbers

Final result

- base = 10
- significand = -2345612
- exponent = 3

Resulting value:

$$-2.345612 \times 10^3 = -2345.612$$



[6] Floating-Point Numbers

Binary Representation

- base = 2
- significand = -100100101001
- exponent = 1001

Multiplying by a power of 2 becomes a bit shift.

Resulting value:

$$-1.00100101001 \ll 1001 (9) = -1001001010.01$$



[7] Floating-Point Numbers

*Binary Representation

The first digit of a binary significand is always a 1

Once normalized, this always appear as "1.xxxxxx" so we may preserve only the fractional part ("xxxxxx") called the mantissa.

Decimal	Binary
1	1
2	10
3	11
4	100
5	101
1.0	1.0
2.5	10.1
3.125	11.001



[8] Floating-Point Numbers

The IEEE754 Standard

IEEE754 defines representation and interchange format.

The most common formats are:

Single Precision (32 bits)

- base is always 2 (implicit)
- one sign bit
- mantissa is 23 bits
- exponent is 8 bits (including sign bias)

Double Precision (64 bits)

- base is always 2 (implicit)
- one sign bit
- mantissa is 52 bits
- exponent is 11 bits (including sign bias)



[9] Floating-Point Numbers

IEEE754 floating-point representations

Table 1. Single Precision (32 bits)

sign	exponent	mantissa
bit 31	bits 30 .. 23	bits 22 .. 0

Table 2. Double Precision (64 bits)

sign	exponent	mantissa
bit 63	bits 62 .. 52	bits 51 .. 0



[10] Floating-Point Numbers

Multiplication of two numbers

Example pseudocode of a floating-point multiplication

```
multiply(a, b)
{
    /* determine resulting sign */
    result.sign = a.sign ^ b.sign;
    /* determine resulting exponent */
    result_e = a.exponent + b.exponent;
    result_e -= EXPONENT_SIGN_BIAS;
    /* recreate significands */
    s_a = a.mantissa | (1 << MANTISSA_BITS);
    s_b = b.mantissa | (1 << MANTISSA_BITS);
    /* perform fixed-point multiplication */
    result_s = (m_a * m_b) >> MANTISSA_BITS;
    /* account for multiply carry bit */
    if (result_s & (1 << (MANTISSA_BITS+1))) {
        result_s >>= 1;
        result_e += 1;
    }
    /* mask out leading 1 bit */
    result_s = ~(1 << MANTISSA_BITS);
    result.mantissa = result_s;
    result.exponent = result_e;
    return result;
}
```

NOTE | *Handling of rounding, denormal, overflow, infinity and NaN are missing from this example.*



[11] Floating-Point Numbers

Addition of two numbers (more on next slide)

This is even more complex than multiplication:

1. For each argument:
 - a. Extract significand and exponent values
 - b. If sign bit is set: negate the significand
2. Determine the operand with the smallest exponent
3. Shift its exponent rightward and increment its exponent until both exponents are equal
4. Add both significands together
5. If result is zero: we return 0 here

NOTE *Handling of rounding, denormal, overflow, underflow, infinity and NaN are missing from this list.*



[11] Floating-Point Numbers

Addition of two numbers (continued)

6. Extract resulting sign bit from the resulting significand
7. If sign bit is set: negate the resulting significand
8. Normalize the resulting significand to the desired number of mantissa bits by shifting it left or right, and adjusting the exponent
9. accordingly
10. Mask out the leading 1 from the significand to keep only the mantissa
11. Repack the sign, exponent and mantissa together
12. Return the result

NOTE *Handling of rounding, denormal, overflow, underflow, infinity and NaN are missing from this list.*



[12] Floating-Point Numbers

Floating-Point in software is SLOW

- Simple arithmetic operations require several CPU instructions.
- Floating-point multiplications require integer multiplications with 2x the number of significand bits:
 - 48 bits for single precision
 - 104 bits for double precision
- Divisions, especially if no native div instruction exists, may be very slow due to needed code loops.



[13] Floating-Point Numbers

Software Floating-Point Implementation Example

From the GCC news and announcements page (August 27, 2003):

Nicolas Pitre has contributed his hand-coded floating-point support code for ARM. It is both significantly smaller and faster than the existing C-based implementation.



Table is in the next slide.

[13] Floating-Point Numbers

The pure assembly version is 5 to 25 times faster than the generic C implementation... but still requires several CPU cycles:

Table 3. ARM32 assembly instruction count for some operations

operation	instructions
mulsf3 (common case)	28
mulsf3 (x or y power of 2)	20
muldf3 (common case)	42
muldf3 (x or y power of 2)	25
addsf3 (say 8.0 + 9.0)	43
adddf3 (say 8.0 + 9.0)	56
cmpsf2 (say 8.0 and 9.0)	14
cmpdf2 (say 8.0 and 9.0)	20
fixsfsi (common case)	12
fixdfsi (common case)	14



[14] Floating-Point Unit

What is an FPU?

A floating-point unit (FPU) is dedicated hardware to operate on floating-point values.

- Much faster than software
- Deals with IEEE-754 representation directly
- Dedicated machine instructions for:
 - addition
 - subtraction
 - multiplication
 - division
 - square root
- May perform transcendental functions natively
- May support single instruction / multiple data (SIMD)
- Much faster execution than software



[15] Floating-Point Unit

FPU's typically have their own register file

ARM's Vector Floating Point (VFP)

- 32 registers
- registers are 128 bits wide
- 512 bytes total

RISC-V's floating point extensions

- 32 registers
- "F" extension = Single-Precision
 - 32 bits wide
 - 128 bytes total
- "D" extension = Double-Precision
 - 64 bits wide
 - 256 bytes total
- "Q" extension = Quad-Precision Floating-Point
 - 128 bits wide
 - 512 bytes total



[16] Zephyr Project

Zephyr Project - www.zephyrproject.org

- Small RTOS
- Highly configurable
- Optimized for memory constrained devices



[17] Zephyr Floating-Point Options

Zephyr's Floating-Point Options

No FPU

- The lack of an FPU means floating-point usage is entirely software based. This is commonly called "soft-float".

Globally exclusive FPU usage

- An FPU is available but unmanaged. By default, threads may use the floating point registers only in an exclusive manner, and this usually means that only one thread may perform floating point operations.

FPU register sharing

- This option enables preservation of the hardware floating point registers across context switches to allow multiple threads to perform concurrent floating point operations.



[18] Zephyr FPU Sharing

FPU register sharing

Pros

- No special FPU reservation needed
- Threads don't need to be aware of other threads' FPU usage
- No risk of FPU register corruptions

Cons

- FPU register content must be preserved and restored across thread context switches, meaning...
- Thread context switching runtime overhead
- Increased memory usage



[19] Zephyr FPU Sharing

FPU context switching — Simple approach

Switch from thread A to thread B:

1. Push normal (ALU) register content onto the stack
2. Push FPU register content onto the stack
3. Store stack pointer into thread A's structure
4. Retrieve thread b's stack pointer from its structure
5. Pop FPU register content from the stack
6. Pop normal (ALU) register content from the stack
7. Resume execution on thread B

Steps 2 and 5 are the most costly.



[20] Zephyr FPU Sharing

Observations:

- Saving and restoring FPU register content on each context switch adds non-trivial overhead.
- Most threads don't perform floating-point operations at all.
- Even when a thread does floating-point operations, it is typical for the other threads not to do so.

Let's optimize this.



[21] Zephyr FPU Sharing

FPU context switching — Smarter approach

This is called "lazy" FPU context switching.

Switch from thread A to thread B:

1. Push normal (ALU) register content onto the stack
2. Disable FPU register access
3. Store stack pointer into thread A's structure
4. Retrieve thread b's stack pointer from its structure
5. Pop normal (ALU) register content from the stack
6. Resume execution on thread B

The most costly steps are gone.



[22] Zephyr FPU Sharing

FPU context switching — Smarter approach

If thread B starts using the FPU:

1. FPU access is disabled: an FPU exception is raised
2. Enable FPU register access
3. Push FPU register content into thread A's FPU save area
4. Retrieve FPU register content from thread B's FPU save area
5. Resume execution on thread B

NOTE | We can't use thread stacks to store FPU context anymore.



[23] Zephyr FPU Sharing

FPU context switching — Smarter approach

The lazy FPU context switching needs:

- A global variable pointing to the FPU-owning thread structure
- A per-thread FPU save area

Same memory usage overall.



[24] Zephyr FPU Sharing

Lazy FPU context switching

Let's optimize it further.

Switch from thread A to thread B:

1. Push normal (ALU) register content onto the stack
2. Disable FPU register access
3. Store stack pointer into thread A's structure
4. Retrieve thread b's stack pointer from its structure
5. If thread B "owns" the FPU: enable FPU register access
6. Pop normal (ALU) register content from the stack

Resume execution on thread B

Steps 2 and 5 are very light and quick.



[25] Zephyr FPU Sharing

Lazy FPU context switching

Table 4. Example of thread and FPU context switching

Event	FPU state	FPU owner
Start of A	Initially disabled	none
FPU usage by A	Gets enabled	A
Switch to B	Gets disabled	A
Execution of B	Remains disabled	A
Switch to C	Remains disabled	A
Switch to A	Is enabled for owner	A
Switch to B	Gets disabled	A
FPU usage by B	A context saved, enabled for B	B
Switch to A	Gets disabled	B
FPU usage by A	B context saved, enabled for A	A



[26] Zephyr FPU in Exceptions

FPU Context Switching upon Exceptions

Zephyr Exception Contexts:

- System Calls
- Interrupt ReQuest (IRQ)
- CPU Faults

Constraint:

- The FPU context may not be stored on the stack...
- therefore FPU usage may not happen in exception context



[27] Zephyr FPU in Exceptions

FPU usage may not happen in exception context ...

... Really?

Typical exception sequence:

1. Normal execution
2. Exception raised
 - a. ALU registers preserved on the stack
 - b. Execution of exception code
3. ALU registers restored from the stack
4. Normal execution resumed



[28] Zephyr FPU in Exceptions

FPU usage may not happen in exception context ...

... So let's prevent it.

Typical exception sequence:

1. Normal execution
2. Exception raised
 - a. ALU registers preserved on the stack
 - b. FPU access disabled
 - c. Execution of exception code
 - d. If current thread "owns" the FPU: re-enable FPU register access
 - e. ALU registers restored from the stack
3. Normal execution resumed



[29] Zephyr FPU in Exceptions

FPU usage may not happen in exception context ...

... Unless we let it.

Variation on the thread context switching:

1. Normal thread execution
2. Exception raised
 - a. ALU registers preserved on the stack
 - b. FPU access disabled
 - c. Execution of exception code
 - d. FPU access attempted: exception raised
 - i. Enable FPU register access
 - ii. Push FPU register content to owning thread's FPU save area
 - iii. Set FPU owner to none
 - e. FPU access resumed
 - f. Current thread no longer owns the FPU: disable FPU register access
 - g. ALU registers restored from the stack
3. Normal thread execution resumed



[30] Zephyr FPU in Exceptions

FPU usage may happen in exception context ...

... but not in a stacked exception.

Given:

- We cannot save FPU context on the stack
- Exception context is forgotten once the exception is done

We simply need to:

- Drop the exception's FPU context as well when the exception is done
- Prevent the exception context from being interrupted



[31] Zephyr FPU in Exceptions

FPU usage may happen in exception context ...

... but not recursively.

FPU usage in exception context:

1. Normal thread execution
2. Level 1 exception raised
 - a. FPU access disabled
 - b. Execution of exception code
 - c. FPU access attempted: level 2 exception raised
 - i. Enable FPU register access
 - ii. Push FPU register content to owning thread's FPU save area
 - iii. Set FPU owner to none
 - iv. Disable IRQs to level 1 exception's context
 - d. FPU access resumed
 - e. Current thread no longer owns the FPU: disable FPU register access
3. Normal thread execution resumed



[32] Zephyr FPU in Exceptions

FPU usage may happen in exception context ...

... if we're careful.

This assumes:

- IRQs (and therefore thread preemption) gets disabled for the remainder of the exception
- CPU fault exceptions that can't be masked don't use FPU at all.

Therefore FPU usage in exception (mainly syscalls, maybe IRQ handlers) should be rare and quick.



[33] Zephyr FPU Sharing Surprises

The AArch64's **va_arg**

*C code using **va_list***

```
#include <stdarg.h>
extern void bar(char *fmt, va_list ap);
void foo(char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    bar(fmt, ap);
    va_end(ap);
}
```



[34] Zephyr FPU Sharing Surprises

The AArch64's **va_arg**

Assembly output for **va_start()** usage

foo:

```
sub sp, sp, #272
[...]  
str q0, [sp, 80]  
str q1, [sp, 96]  
str q2, [sp, 112]  
str q3, [sp, 128]  
str q4, [sp, 144]  
str q5, [sp, 160]  
str q6, [sp, 176]  
str q7, [sp, 192]  
stp x1, x2, [sp, 216]  
stp x3, x4, [sp, 232]  
stp x5, x6, [sp, 248]  
str x7, [sp, 264]  
add x1, sp, 16  
bl bar  
[...]
```



[35] Zephyr FPU Sharing Surprises

The AArch64's **va_arg**

An FPU access exception is triggered on:

- printf()
- printk()
- log statement
- any other variadic function

Even if NO floating point arguments are used.

Implications:

- IRQs are excessively disabled in exception context
- Many CI tests fail due to lack of preemption



[36] Zephyr FPU Sharing Surprises

The AArch64's **va_arg**

Workaround: instruction simulation

```
for (;;) {
    uint32_t insn = *pc;
    /*
     * We're looking for STR (immediate, SIMD&FP)
     * of the form:
     *
     * STR Q<n>, [SP, #<pimm>]
     *
     * where 0 <= <n> <= 7 and <pimm> is a
     * 12-bits multiple of 16.
     */
    if ((insn & 0xffc003f8) != 0x3d8003e0) {
        break;
    }
    /* Zero the location as the above STR would have done */
    uint32_t pimm = (insn >> 10) & 0xffff;
    *(__int128 *) (sp + pimm * 16) = 0;
    /* move to the next instruction */
    pc++;
}
```

... then leave FPU access disabled and IRQs enabled.



[37] Zephyr FPU Sharing Surprises

RISC-V: No FPU Access Exception!

RISC-V has NO dedicated FPU access exception signal

Only an Illegal Instruction signal

Implication:

- Need to manually decode exception-triggering instructions to distinguish FPU access fault from actual illegal instructions.



[38] Zephyr FPU Sharing Surprises

RISC-V: No FPU Access Exception!

Instructions we're looking for:

- The OP-FP space
 - FADD.S, FSUB.S, FMUL.S, FDIV.S, FSQRT.S, FMIN.S, FMAX.S, FCVT.W.S, FCVT.L.S, FCVT.S.W, FCVT.S.L, FSGNJ.S, FSGNJN.S, FSGNJX.S, FLT.S, FLE.S, FCLASS.S, and the .D variants, etc.
- The LOAD-FP and STORE-FP spaces
 - FLW, FLD, FSW, FSD
- The fused multiply-add spaces
 - FMADD.S, FMSUB.S, FNMADD.S, FNMSUB.S, and the .D variants
- FP instructions in the RVC space
 - C.FLD, C.FLDSP, C.FLW, C.FLWSPP, C.FSD, C.FSDSP, C.FSW, C.FSWSP
- CSR instructions accessing FPU state
 - FRCSR, FSCSR, FRRM, FSRM, FSRMI, FRFLAGS, FSFLAGS, FSFLAGSI



Source code is in the next slide.

RISC-V illegal instruction handler (RV32 version)

```
/* determine if this is an Illegal Instruction exception */
csrr t2, mcause
li t1, 2 /* 2 = illegal instruction */
bne t1, t2, no_fp
/* determine if we trapped on an FP instruction. */
csrr t2, mtval /* get faulting instruction */
andi t0, t2, 0x7f /* keep only the opcode bits */
xori t1, t0, 0b1010011 /* OP-FP */
beqz t1, is_fp
ori t1, t0, 0b0100000
xori t1, t1, 0b0100111 /* LOAD-FP / STORE-FP */
beqz t1, is_fp
ori t1, t0, 0b0001100
xori t1, t1, 0b1001111 /* MADD / MSUB / NMSUB / NMADD */
beqz t1, is_fp
xori t1, t0, 0b1110011 /* SYSTEM opcode */
bnez t1, 1f /* not a CSR insn */
srli t0, t2, 12
andi t0, t0, 0x3
beqz t0, 1f /* not a CSR insn */
srli t0, t2, 20 /* isolate the csr register number */
beqz t0, 1f /* 0=ustatus */
andi t0, t0, ~0x3 /* 1=fflags, 2=frm, 3=fcsr */
beqz t0, is_fp
1: /* remaining non RVC (0b11) and RVC with 0b01 are not FP instructions */
andi t1, t2, 1
bnez t1, no_fp
srli t0, t2, 8
andi t1, t0, 0b00100000
beqz t1, no_fp
is_fp:
/* Process the FP trap and quickly return from exception */
la ra, fp_trap_exit
mv a0, sp
tail z_riscv_fpu_trap
no_fp:
[...]
```



[40] Zephyr FPU Sharing Optimization

RISC-V's FPU Clean/Dirty Status

RISC-V offers an FPU Dirty status bit.

This allows for further optimization:

- A "clean" FPU does not need saving back to memory.
- A "dirty" FPU is a good sign of active usage.
- The "active" state means we can proactively restore FPU context at thread context switch time.

Useful to avoid FPU exception processing overhead when the FPU is contended.



[41] Zephyr FPU Sharing Optimization

Smarter Lazy FPU context switching

Table 5. Example of thread and FPU context switching on RISC-V

Event	FPU state	FPU owner
Start of A	Initially disabled	none
FPU usage by A	Access trap: FPU gets enabled	A
Switch to B	FPU gets disabled	A
FPU usage by B	access trap: <ul style="list-style-type: none">• FPU is dirty:<ul style="list-style-type: none">◦ FPU saved into A• A marked as recent FPU user• FPU enabled for B	B

Table continues in the next slide.



[41] Zephyr FPU Sharing Optimization

Table 5. Example of thread and FPU context switching on RISC-V (continued)

Event	FPU state	FPU owner
switch to A	A is a recent FPU user: <ul style="list-style-type: none">• if FPU is dirty:<ul style="list-style-type: none">◦ FPU saved into B◦ B marked as recent FPU user• FPU restored for A	A
A does not use FPU	FPU remains clean	A
Switch to B	FPU is clean: <ul style="list-style-type: none">• FPU is not saved• A no longer recent FPU user• if B is recent FPU user:<ul style="list-style-type: none">◦ FPU restored for B• if B is not recent FPU user:<ul style="list-style-type: none">◦ FPU remains owned by A◦ FPU access disabled	Depends



[42] FPU Sharing and SMP

What about SMP ?

Table 6. Example of thread and FPU context switching on SMP

CPU 1			CPU 2		
Event	FPU access	FPU owner	Event	FPU access	FPU owner
FPU used by A	Enabled	A	FPU used by B	Enabled	B
Switch to C	Disabled	A	Switch to D	Disabled	B
Switch to B	Disabled	A	Switch to A	Disabled	B
FPU used by B	Trapper	A	FPU used by A	Trapped	B



[43] FPU Sharing and SMP

SMP Considerations

An Inter-Processor Interrupt (IPI) is needed to ask another CPU to flush its FPU context to memory.

If thread X starts using the FPU:

1. FPU access exception is raised
2. Push FPU register content into current FPU owner's save area
3. Clear FPU ownership
4. For each remote CPU:
 - a. If FPU owner != X: continue
 - b. Send that CPU an IPI to have FPU content saved to memory
 - c. Loop until that CPU's FPU ownership is cleared
5. Retrieve FPU register content from thread X's FPU save area
6. Resume execution on thread X



[44] FPU Sharing and SMP

More Surprises - Deadlock scenario:

<i>Table 7. SMP deadlock sequence</i>					
CPU 1			CPU 2		
Event	FPU access	FPU owner	Event	FPU access	FPU owner
FPU used by A	Enabled	A	...	Disabled	None
Switch to B	Disabled	A	Switch to A	Disabled	None
...	Disabled	A	A acquires spinlock L	Disabled	None
B tries to acquire spinlock L	Disabled	A	FPU used by A	Trapped	None

Table continues in the next slide.



[44] FPU Sharing and SMP

More Surprises - Deadlock scenario:

<i>Table 7. SMP deadlock sequence (continued)</i>					
CPU 1			CPU 2		
Event	FPU access	FPU owner	Event	FPU access	FPU owner
B spins waiting for L (with IRQs disabled)	Disabled	A	IPI sent to CPU 1	Pending	None
IPI pending but IRQs still disabled	Disabled	A	Wait for FPU owner on CPU 1 to no longer be A	Pending	None
Wait for CPU 2 to release L			Wait for CPU 1 to release A's FPU context		



[45] FPU Sharing and SMP

Deadlock Avoidance

Possibilities:

1. Forbid FPU usage when a spinlock is held
2. Momentarily re-enable IRQs in the contended spinlock case
3. Check for pending FPU IPI in the contended spinlock case
 - a. 1 is hard (think `va_arg` on ARM)
 - b. 2 would open races with locking in IRQ handlers
 - c. 3 was the chosen approach



[46] References

Source code references

- GCC's IEEE754 soft-float assembly implementation:
 - <https://github.com/gcc-mirror/gcc/blob/master/libgcc/config/arm/ieee754-sf.S>
 - <https://github.com/gcc-mirror/gcc/blob/master/libgcc/config/arm/ieee754-df.S>
- Zephyr's FPU handling code for ARM64:
 - <https://github.com/zephyrproject-rtos/zephyr/blob/main/arch/arm64/core/fpu.c>
 - <https://github.com/zephyrproject-rtos/zephyr/blob/main/arch/arm64/core/fpu.S>
- Zephyr's FPU handling code for RISC-V:
 - <https://github.com/zephyrproject-rtos/zephyr/blob/main/arch/riscv/core/fpu.c>
 - <https://github.com/zephyrproject-rtos/zephyr/blob/main/arch/riscv/core/fpu.S>
 - <https://github.com/zephyrproject-rtos/zephyr/blob/main/arch/riscv/core/isr.S>
- Zephyr's FPU test case for RISC-V:
 - https://github.com/zephyrproject-rtos/zephyr/blob/main/tests/arch/riscv/fpu_sharing/src/main.c



[47] The End

Questions?

