Finally, to detect the difference between an underflow value and a real zero, a wide OR is performed on the underflow zone of width W_Z, with

$$W_Z = E_{\min} - Q_{\text{lsb}} = E_{\max}$$

For draft standard posit sizes, the simplified formulas are the following:

$$W_O = \frac{N^2}{8} + \frac{3N}{4} - 2$$
$$W_Z = \frac{N^2}{8} - \frac{N}{4}$$
$$W_R = \frac{N^2}{4} - \frac{N}{2} + 1$$

Associated numerical values reported in Table 2.8.

Verification of functionnal correctness of the operators

In order to verify that the proposed architectures are correct, the following functional tests were first run in sofware:

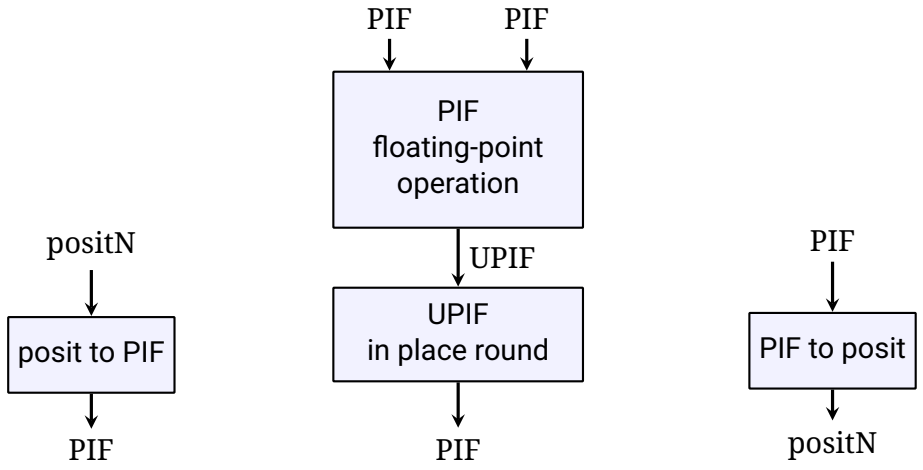
- Exhaustive test against softposit of posit8 and posit16 addition and multiplication.
- Some corner case tests of quire addition/subtraction and conversion back to posit for posit16.
- Exhaustive test for addition/product of IEEE16 against SoftFloat, for the five IEEE-754 rounding modes.

Then the VHDL file produced by the Vivado HLS compiler for a 16-bit posit adder was exhaustively tested using a VHDL simulator. Scripts and sources to reproduce this experiment are accessible from the MArTo repository.

Finally, the standard posit16 multiplier was synthesized, placed and routed for the Zynq FPGA of a Zybo board using the Vivado HLS toolchain, and the resulting FPGA circuit was exhaustively checked against SoftPosit executed on the ARM core of the Zynq. All these tests were successful.

2.3 Case study: comparing IEEE-754 and posit hardware implementation cost

Many works have compared the accuracies of posit and IEEE-754 floating-point formats [46]–[49]. However, these works didn't evaluate the performance implication of replacing IEEE-754 by posit. This is the aim of this case



(a) Memory read (b) Arithmetic operation (c) Memory write

Figure 2.11: Architecture of a PAU using posits as a memory-only encoding, with PIF registers and PIF-to-PIF operators.

study, which will evaluate two possible architecture for posit arithmetic units. The first one is the one described on figure 2.4, with posit used both as memory and internal register format. As shown on the figure however, this mostly adds a decoding step overhead to a classical floating-point operation (the PIF have no subnormal to support that simplify a bit these operators compared to IEEE-754, but the posit encoding is actually very similar to subnormal handling). As an alternative, posit can be used as memory format only, decoded on the memory path, and PIF as register format. The block-diagram of this architecture is presented on figure 2.11.

In order to avoid double-rouding and related issues, PIF in register should always represent the rounded result of the operation. This is the reason of the “UPIF inplace round” block of Figure 2.11b shows an UPIF in place round block. This block is roughly equivalent to the UPIF to posit encoder of figure 2.6, with the difference that a mask is shifted instead of the representation to. PIF to posit conversion that occurs when writing the value back to memory (as shown on figure 2.11b) is a simplification of the UPIF to posit encoder. Indeed, PIF values stores exactly representable posit values, so all the rounding logic can be removed.

2.3.1 Comparison of operator area and latency

Qualitative comparison

A first “qualitative” comparison between posit and IEEE-754 operators can be performed at the architecture level. There is no 1-to-1 match to the decoder

step in IEEE-754 arithmetic operator implementation, as the representation already has fixed-width fields. However, it can be compared with the detection of special cases. On the IEEE-754 side, one OR reduction on the exponent bits is needed to detect subnormals, another OR reduction on the significand bits is needed to detect zeros, and two similar AND reductions are needed to detect respectively NaN and infinities. The two OR reduction have a similar input width in total to the posit decoder OR reduction block. Normalization of subnormal values requires a LZC and a shifter, that operate on the fraction bits. On the posit side, the combined LZOC + shifter operates on a wider value. Besides, the fact that it is an LZOC and not an LZC inclurs internal AND reductions that matches the cost of the IEEE-754 ones. All in all, the posit decoding process cost is expected to be very similar to the handling of special cases and subnormal values in IEEE-754.

The PIF arithmetic operator implementations are very similar to IEEE-754 operators, so area and latency should be quite similar.

Handling rounding and special cases encoding in IEEE-754 requires one integer addition of the representation width, a few gates to determine if the rounding should occur according to the rounding mode, and a few bitwise masking operations to handle the generation of NaN/infinities if required. By contrast, posit rounding and encoding process which requires the same addition, but also a potentially large shift seems more costly.

As a whole, it is expected that posit operators require more resources and have a higher latency compared to IEEE-754 operators for identical widths formats. The expected overhead coming mostly from an expensive rounding and encoding process.

Comparison with state of the art

As we eventually observe that posits are larger and slower than IEEE floats, it is important to be convincing that we are not using a substandard posit implementation. For this purpose, Tables 2.9, 2.10, 2.11 and 2.12 gather the results of best-effort comparisons with the state of the art in posit hardware at time of writing. It shows that MArTo definitely improved this state of the art.

There is less pressure to show that the MArTo implementation of IEEE floats is efficient. A comparison with Xilinx implementation of IEEE floats is provided in Table 2.14. There, the line labeled Xilinx Float corresponds to IP used by Vivado HLS when using the `float` and `double` data types in the HLS C++ (hence the lack of 16-bit results). This hard IP is the industry standard when using Vivado, and can be considered a state-of-the-art implementation

Table 2.9: Comparison with [34] for standard posit addition and product

	Op	Format	LUT	DSP	Delay (ns)
[34]	+	<16, 1>	391	0	32.4
		<32, 2>	981	0	40.0
	×	<16,1>	218	1	24.0
		<32, 2>	572	4	33.0
MArTo	+	<16, 1>	**299**	0	**24.2**
		<32, 2>	**704**	0	**33.9**
	×	<16,1>	**213**	1	**19.4**
		<32, 2>	**483**	4	**28.9**

As no sources is provided, we report as-is the figures from [34], obtained for a Zynq-7000 (xc7z020clg484-1) with Vivado 2017.4. To limit the possible effect of tool improvement on the synthesis, MArTo synthesis results have been obtained for the same part with Vivado HLS/Vivado 2018.3, the oldest version available for download at time of experimentation.

Table 2.10: Comparison with [33] on standard posit addition and product

	Op	Format	ALM	DSP	Cycles	FMax (MHz)
[33]	+	<16, 1>	~500	0	~49	~550
		<32, 2>	~1000	0	~51	~520
	×	<16, 1>	~330	1	~35	~**600**
		<32, 2>	~600	**1**	~38	~**550**
MArTo	+	<16, 1>	**274**	0	**11**	**564**
		<32, 2>	**696**	0	**17**	**562**
	×	<16, 1>	**280**	1	**15**	**600**
		<32, 2>	**452**	2	**21**	445

Synthesis reported in [33] trarget Stratix V FPGA. Results are read from a graphic plot, hence the approximate values. As there is no version of the Intel HLS toolchain that supports both Stratix V and the C++ 11 standard used in MArTo, the C++ to HDL compilation is done using Vivado HLS. The obtained HDL is then synthesised and routed for Stratix V using Quartus. Despite being baroque, this toolchain seems to give good results, except for the <32, 2> product where it lacks the knowledge of the target's DSP possible configurations. Indeed, the product is computed using a 36x36 configuration of the DSP block, where a 27x27 configuration would be faster.

Table 2.11: Comparison with [35] on posit<32,6> addition and product

	Op	LUTs	DSPs	Cycles	Delay (ns)
[35]	+	946	0	**5**	4.1
	×	854	**1**	**6**	4.4
MArTo	+	**792**	0	**5**	**3.9**
	×	**435**	2	**6**	**4.1**

MArTo synthesis have been performed using Vivado HLS/Vivado 2020.1 using part xc7vx330t-ffg1157-3. Experimental settings of [35] use the same part, but tool version is not reported.

Table 2.12: Comparison with [50] for standard posit addition and product

	Op	Format	LUT	DSP	Delay (ns)
[50]	+	<16, 1>	383	0	27.25
		<32, 1>	939	0	35.8
	×	<16,1>	**201**	1	20.9
		<32, 1>	571	4	29.2
MArTo	+	<16, 1>	**300**	0	**25.5**
		<32, 1>	**672**	0	**34.5**
	×	<16,1>	205	1	**19.2**
		<32, 1>	**472**	4	**28.8**

MArTo synthesis have been performed using Vivado HLS/Vivado 2020.1 using part xc7z020clg484-1.

of floating-point for Xilinx FPGAs. It supports some of the IEEE features, such as infinity and NaN encoding. However, it is not IEEE-compliant: although the memory format is that of IEEE floats, subnormals are flushed to zero to save resources.

Considering that the Xilinx Float adders use DSP blocks to implement some of the shifters, the hardware costs of Xilinx adders and IEEE adders (obtained with MArTo) are really comparable. This illustrates that the overhead of subnormal handling in floating-point adders is small. Conversely, there is in Table 2.14 a very large difference in the resources used in multipliers. This demonstrates the cost of hardware subnormal handling in floating-point multipliers.

The Xilinx IP pipelining also seems to be fixed, and do not benefit from a relaxed clock constraint to reduce the latency, hence their large latency.

Since Xilinx floats lack subnormal support, the following bases on MArTo only the posit versus IEEE comparisons.

Comparison between posit and IEEE-754 operators

Table 2.13 compares combinatorial implementations of posits and floats of the same size on addition and multiplication. In this table, the "posit→posit" lines present results for the classical posit operators of Figure 2.4. The "PIF→PIF" lines presents results for the posit-compatible PIF operators that use the architecture of Figure 2.11b, including the inplace round component.

A first observation is that posit arithmetic is indisputably both larger and slower than IEEE-754 arithmetic. This contradicts the comparison in [34], which seems to use a very poor IEEE implementation.

As expected, the PIF-to-PIF operators are lighter and faster than the posit-to-posit ones. They still pay the price in area of a wider significand datapath (see Table 2.7) compared to IEEE operators: for the adders, PIF-to-PIF consume more LUTs than IEEE operators; for multipliers, they consume more DSP blocks (there is a step effect due to the discrete nature of DSP blocks). Again we observe in the IEEE multipliers the logic cost of subnormal support, but we also observe a comparable cost in the PIF multiplier, essentially due to the inplace round logic. Still, the PIF to PIF operators achieve delays that are closer to those of IEEE operators than to those of posit operators, which was their main motivation.

Note that the area cost of PIF/posit conversions (altogether about half the size of a complete IEEE adder) must still be paid in a posit arithmetic unit that uses the PIF-to-PIF approach. Only its delay (altogether about half the delay of a complete IEEE adder) is avoided. However, there is another advantage

Table 2.13: Synthesis results of combinatorial operators

(a) Combinatorial adder

	N	LUT	(ratio)	delay	(ratio)
posit→posit	16	312	1.33	11.1 ns	1.27
	32	647	1.49	15.8 ns	1.33
	64	1550	1.59	21.6 ns	1.35
PIF→PIF	16	237	1.01	9.7 ns	1.10
	32	562	1.29	12.9 ns	1.08
	64	1244	1.27	14.7 ns	0.92
IEEE→IEEE	16	234	1	8.8 ns	1
	32	434	1	11.9 ns	1
	64	976	1	16.0 ns	1

(b) Combinatorial multiplier

	N	LUT	(ratio)	DSP	delay	(ratio)
posit→posit	16	182	1.03	1	11.3 ns	1.39
	32	466	1.37	4	15.8 ns	1.62
	64	1213	1.58	16	21.1 ns	1.48
PIF→PIF	16	120	0.68	1	7.8 ns	0.96
	32	291	0.86	4	11.5 ns	1.17
	64	695	0.90	16	15.3 ns	1.08
IEEE→IEEE	16	176	1	1	8.1 ns	1
	32	340	1	2	9.8 ns	1
	64	768	1	9	14.3 ns	1

(c) Posit - PIF conversion operators

	N	LUT	delay
posit→PIF	16	61	2.59 ns
	32	106	4.74 ns
	64	278	5.52 ns
PIF→posit	16	41	2.12 ns
	32	98	2.50 ns
	64	301	2.83

Table 2.14: Synthesis results of pipelined operators

(a) Pipelined adder

	N	LUT	Reg.	DSP	cycles	delay
Posit	16	320	128	0	4	2.69 ns
	32	719	460	0	7	2.83 ns
	64	1635	1207	0	10	2.93 ns
IEEE	16	193	137	0	4	2.90 ns
	32	435	337	0	6	2.88 ns
	64	1001	880	0	10	2.99 ns
Xilinx Float	32	167	355	2	10	2.43 ns
	64	628	758	3	10	2.43 ns

(b) Pipelined multiplier

	N	LUT	Reg.	DSP	cycles	delay
Posit	16	213	80	1	4	2.85 ns
	32	443	198	4	6	2.93 ns
	64	1140	811	16	12	4.10 ns
IEEE	16	189	122	1	4	2.69 ns
	32	381	246	2	6	2.74 ns
	64	783	801	9	8	2.67 ns
Xilinx Float	32	82	151	3	5	2.72 ns
	64	115	494	11	10	2.75 ns

in a PIF-to-PIF PAU: the hardware for these conversions is naturally shared between different operations (such sharing is also possible in principle in a posit-to-posit PAU, but then it will restrict instruction-level parallelism).

Table 2.14 compares pipelined versions of the same operators, targeting a frequency of 333 MHz (3ns cycle time), and producing one output per clock cycle. As the latency estimated by the Vivado HLS tool is usually pessimistic, the reported latencies are obtained by an automated exploration that finds the smallest pipeline depth allowing the design to run with the target clock period. The script performing this exploration is open source, and is also accessible from the MArTo repository for reproducibility.

There is no PIF to PIF line in this table: for this setup, the PIF to PIF approach fails to provide any latency improvement (the arithmetic operators require the same number of cycles, and sometimes require one more cycle).

Table 2.15: Synthesis results for a sum of 1000 products (U: Unsegmented, S32 and S64: Segment sizes of 32 and 64 bits).

		LUT	Reg.	DSP	cycles	delay
quire 16	U	1200	1026	1	1019	2.70 ns
	S32	978	1062	1	1021	2.68 ns
	S64	1004	958	1	1019	2.36 ns
quire 32 (512 bits)	U	5884	6235	4	1031	*3.65* ns
	S32	3641	7237	4	1040	2.89 ns
	S64	3513	5189	4	1033	2.78 ns
Kulisch 32 (559 bits)	S32	3624	7632	2	1034	2.937
	S64	3612	5165	2	1026	2.801
IEEE Float 32		840	711	2	6012	2.92 ns
IEEE Float 64		1798	1723	9	8015	*3.33* ns
Xilinx Float 32		445	544	3	6008	2.72 ns
Xilinx Float 64		809	1386	11	8013	2.70 ns

We therefore choose not to report these results, which we consider synthesis artifacts as they are inconsistent with the expectations and with Table 2.13.

The cost of supporting all rounding modes in IEEE

Tables 2.13 and 2.14 report result for IEEE operators that only support round to nearest, ties to even. However, supporting the five IEEE-754 rounding modes increase only very slightly the hardware cost. For instance, adding this support to the 32-bit adder increases its area from 434 to 458 LUTs and actually decreases the delay from 11.9 to 11.7ns (another synthesis artifact). It remains well below the posit cost.

2.3.2 Quire versus standard operations

Synthesis results for the quire are given in Table 2.15, where we use MArTo to write a C++ loop that performs the sum of 1000 products and return the result as a posit. They are compared to a similar loop using floating-point Kulisch accumulator, and using regular floating-point hardware.

Quire is presented in unsegmented (U) version along with two segmented versions (S32 and S64 for segments of 32 or 64 bits). For 32 bits, the unsegmented version is not able to achieve 3ns cycle time, due to the long carry

propagation.

The Kulisch accumulator used here is also based on a 2's complement segmented accumulator [45, variant 3], with an IEEE-compliant final conversion to float (round to nearest, ties to even). The implementation has been validated against MFPR [51] simulations.

Unsurprisingly, the cost and performance of a posit32 quire and a Kulisch accumulator for 32 bits floats are almost identical.

An exact accumulator consumes vastly more resources than standard operators: a factor 10 for 32-bit floats (a smaller factor for posits, but only due to the higher cost of the standard operators). Such factors should not come as a surprise: the 512 bits of the posit32 quire are indeed 18 times the 27 bits of the posit32 significand. This is the price of the accuracy of an exact accumulator.

Another advantage of exact accumulation is that it offers a latency reduction proportional to the latency of the floating-point or posit adder (here a factor 6-7). This is thanks to the fact that the accumulation loop is an exact fixed-point addition, which offers opportunities to exploit more parallelism in its computations[39], [42].

Detailed synthesis results of the quire sub-components are given in Table 2.16. The *quire addition* line reports the cost for the architecture of Figure 2.9, including the large shifter and the fixed-point accumulation loop. This component accepts a new input every cycle. The two other lines describe the conversion of the quire result back to posit. The *carry propagation* is a loop component that adds zeroes, and is mostly merged with the *quire addition* component. However, there is an irreducible latency for the final carry propagation once the accumulation is over.

The latency overhead of the expensive conversion from quire to float or posits is easily amortized for large loops. However, it is also clear that a hardware quire will be very inefficient when used for small sequences of operations (e.g. fused multiply and add, complex arithmetic, small matrices or convolutions, etc).

2.3.3 Case study conclusion

This case study both allowed to demonstrate the relevance of having an HLS library for arithmetic components, and to compare the hardware cost of posit and IEEE-754 basic operation implementation.

Regarding the first point, it is simple with such a library to replace the arithmetic type in use for a given application by using a few typedefs. This allows easy experimentation to determine the format that gives the best accuracy/performance trade-off for a given application.

Table 2.16: Detailed synthesis results of hardware posit quire

			LUT	Reg.	Cycles	Delay (ns)
posit16	Quire addition	U	618	885	4	2.576
		S32	403	585	3	1.886
		S64	444	606	3	1.984
	Carry propagation	S32	6	390	3	1.539
		S64	2*	261	2	1.651
	Quire to posit		480	166	3	2.735
posit32	Quire addition	U	3609	4986	7	3.212
		S32	1305	2265	3	2.791
		S64	1389	2276	3	2.791
	Carry propagation	S32	281	2874	8	2.851
		S64	189	2391	7	2.183
	Quire to posit		1845	1457	17	2.878

On the second point, the take-away message of this study is that the indisputable complexity of the IEEE-754 standard, much attacked by posit proponents, does not necessarily translate into expensive hardware. Among the features that the posit system discards as useless, most (in particular overflow management, NaNs, and directed rounding mode) were designed to be implementable at very little cost. The only really expensive feature is subnormal support, due to rounding happening in a variable position of a bit vector. Posit arithmetic, despite the simplicity and elegance of the number system, involves such variable-position rounding, and therefore entail an overhead that is comparable in nature to subnormal support.

2.4 Limits and future work

A first issue with the current library design is the imposed default choice for IEEE-754 number rounding mode. This is however only a design issue and could be solved by adding a policy parameter to the `IEEENumber` template class determining the default rounding mode.

A more fundamental issue comes from the limits of C++ template metaprogramming. While the C++ template system has been shown to be Turing-complete, many obstacles hinders the development of a full library based only on this system.