McMaster University

Advanced Optimization Laboratory

Title:
Unified Tables for Exponential and Logarithm Families

Authors:
Christopher Kumar Anand and Anuroop Sharma

AdvOL-Report No. 2009/02

May 2009, Hamilton, Ontario, Canada
Unified Tables for Exponential and Logarithm Families

Christopher Kumar Anand
McMaster University
and
Anuroop Sharma
Optimal Computational Algorithms, Inc.

Accurate table methods allow for very accurate and efficient evaluation of elementary functions. We present new single-table approaches to logarithm and exponential evaluation, by which we mean that a single table of values works for both \( \log(x) \) and \( \log(1 + x) \), and a single table for \( e^x \) and \( e^x - 1 \). This approach eliminates special cases normally required to evaluate \( \log(1 + x) \) and \( e^x - 1 \) accurately near zero, which will significantly improve performance on architectures which use SIMD parallelism, or on which data-dependent branching is expensive.

We have implemented it on the Cell BE SPU (SIMD compute engine) and found the resulting functions to be up to twice as fast as the conventional implementations distributed in the IBM Mathematical Acceleration Subsystem (MASS). We include the literate code used to generate all the variants of exponential and log functions in the paper, and discuss relevant language and hardware features.

Current interest in implementations of elementary functions is driven by an interest in greater accuracy and by the continuing need to adapt to new hardware environments, e.g., VLIW (very long instruction word) [Story et al. 1999] and SIMD (single instruction multiple data) [Anand and Kahl 2009]. There are many implementation approaches, see [Muller 2006], but the most successful on floating point hardware are piecewise table-based methods. The most important and generically applicable contribution to the improvement in table-based methods is the Accurate Tables approach of Gal [Gal 1986], [Gal and Bachelis 1991], in which special (argument,result) pairs are found such that both argument and result are very close to representable numbers.

The present paper offers an improvement to this approach which allows \( e^x - 1 \) and \( \log(1 + x) \) to be computed using the same tables as \( e^x \) and \( \log x \), and eliminates the need for special cases, thereby improving performance, especially on SIMD architectures, and architectures with expensive branches.

Furthermore, the reduction in the number of tables will reduce the memory footprint of the math library, which is likely to improve overall system performance.
in a cached or private-memory architecture. This work was motivated by the fact that all of these features are becoming more common, and playing a larger role in determining overall system performance.

This approach depends on the use of fused multiply-add instructions, whose use in accurate function evaluation is well known, prompting proposals to include them in language specifications, e.g., [Gustavson et al. 1999].

Tables have been used for a long time in logarithm, [Tang 1990], exponential, [Tang 1989], and offset exponential, [Tang 1992] evaluations, but the use of a single table for both variants is novel. On the Cell BE SPU, we see a doubling of performance for \( \log(1 + x) \) and \( e^x - 1 \). This performance difference would not have been so great in previous-generation scalar architectures, but we feel it is representative of expected future differentials, since register-level SIMD has now migrated to all significant architectures, and branches are unlikely to become less expensive.

This paper is divided into sections describing the algorithms for exponentiation and logarithm. In the first section, we show in detail the methods of computing \( 2^x \) and \( 2^x - 1 \) with a shared table, and the reason the computation is accurate. In the next section we discuss the SPU implementation on a general level, and support for this work in the language. For full details, we have included literate code for both log and exp families as appendices. In the fifth and sixth sections, we present accuracy and performance information. And we conclude with remarks on possible approaches to obtaining more correctly rounded results and a remark about implementation in future hardware.

We hope that these algorithm descriptions will be useful to other implementers, and to make implementation easier, we have included in appendices the source code used to generate both families of functions. The source code is amply commented to make it understandable to someone not familiar with our development language.

1. EXPONENTIAL

1.1 \( 2^x \)

To simplify the exposition, we present our algorithm for base-2 exponentiation.

Our algorithm to calculate \( 2^x \) follows a standard pattern: Extract the integer part, \( \lfloor x \rfloor \), of the argument, and construct the exponential bits in the floating point representation of \( 2^{\lfloor x \rfloor} \). We have

\[
x = \lfloor x \rfloor + f
\]

and

\[
2^x = 2^{\lfloor x \rfloor} 2^f.
\]

Use some significant bits in the residual, \( f \), to look up an offset \( \mu \), and a multiplier, approximately equal to \( 2^\mu \). Evaluate a polynomial approximation on the offset residual, \( f = \mu + f' \),

\[
p(f') \approx 2^{f'}
\]

and finally put the results together:

\[
2^x = 2^{\lfloor x \rfloor} 2^\mu 2^{f'} = 2^{\lfloor x \rfloor} 2^\mu p(f').
\]
Since multiplication by a power of 2 is exact in floating point arithmetic if and only if it doesn’t cause an overflow or underflow, the order of operations is not important in this case. An efficient implementation, which properly handles exceptional inputs and subnormal inputs is a bit more work, but this is the principle.

Our contribution is the modification of this approach which works for $2^x - 1$ with the same set of table values $(\mu_{\text{accurate}}, 2^\mu_{\text{exact}})$. To do so, we need to modify this procedure to use the rounded value $[x + 1/2]$, which is either the desired exponent, or the next higher exponent. As a result the multipliers are in the range $(1/\sqrt{2}, \sqrt{2})$.

### 1.2 $2^x - 1$

The function $e^x$ exhibits extreme loss of precision at 0, in fact all subnormal numbers map to 1. For this reason, the function $e^x - 1$ is usually provided in math libraries, from which any function of $e^x$ in the neighbourhood of zero can be computed accurately. To simplify the exposition, we will refer to the equivalent function $2^x - 1$ in describing our algorithm.

There are two key ideas: after applying range reduction, approximate $2^{f'} - 1 \approx p(f')$ rather than approximating $2^{f'}$, as is commonly done; round to the integer exponent rather than truncating, so the first reduced interval surrounds zero rather than abutting it, and rearrange the computation so that intermediate values are either exact or have exponents smaller than the final result. We’ll look at the last point first. Let $x = I + \mu + f'$ be a decomposition, with $I = [x + 1/2]$ the rounded value. This is not the exponent in the final result, so it results in extra computation, but it is necessary to prevent catastrophic rounding errors elsewhere in the computation:

$$2^x - 1 = 2^I \cdot 2^\mu \cdot 2^{f'} - 1$$

(1)

$$= 2^I \cdot 2^\mu \cdot (2^{f'} - 1) + (2^I \cdot 2^\mu - 1)$$

(2)

$$= (2^I \cdot 2^\mu) \cdot p(f') + (2^I \cdot 2^\mu - 1),$$

(3)

where the parentheses in the final expression show the correct order of computation: a multiplication $2^I 2^\mu$ with exact result; a multiply-add $2^I \cdot 2^\mu - 1$, whose result retains full precision; and a final multiply-add which retains full precision.

We need two properties of the values $\mu$ which will be stored in the table. Let $N$ be the number of intervals, and $\delta \ll 1$ the maximum distance of accurate (but inexact) table values from the centre of each interval. For the intervals $[-1/N, 0)$ and $[0, 1/N)$, we use the exact value $\mu = 0$. For the other intervals $[i/N, (i + 1)/N)$, $\mu$ is within $\delta/N$ of the centre of the interval. The choice of $N = 2^N$ will depend on the memory available for table values. The larger the table, the lower the order of the approximating polynomial and the tighter our bounds on expected maximum error. We have found $N = 128$ and order 5 to be a good compromise. The size of $\delta$ is more difficult to pick. There is a trade off between having more accurate table values, which increase the number of correctly rounded results in the general case, and having values closer to the centre of the interval, which reduces the number of values for which we have a weaker bound on the maximum error. Since these are only weak bounds, and expectations of well-roundedness, it would be worthwhile to search the set of possible accurate table values to find ones for which the floating-point operations are more faithful than guaranteed by the bounds, we have not
performed this search.

Using these properties, we consider rounding errors case by case, in an informal way, with the goal of explaining why the method works, and not proving a bound on the maximum rounding error.

**Case I.** If $I = 0$ and $\mu = 0$, then $\mu$ and $2^\mu$ are exact, so $f'$ is exact. $2^I 2^\mu - 1 = 0$, so $2^x - 1 = p(x)$ which is chosen to be accurate within an ulp.

**Case II.** If $I = 0$ and $\mu \neq 0$, then $|f'| < 1/(2N) + \delta/N$, and

$$|\mu - (2i + 1)/(2N)| < \delta/N,$$

for some integer $i$. This case is accurate because

$$\frac{|(2^x - 1) - (2^\mu - 1)|}{|2^x - 1|} < \frac{1}{2^i} + O\left(\frac{1}{N}\right) + O(\delta).$$

So, the correction $2^\mu p(f')$ almost always has a lower exponent than $2^\mu - 1$, and the final multiply-add may not round correctly, but it cannot introduce larger ulp errors. When $|i| > 1$, the chance of incorrect final rounding decreases, so more answers will be correctly rounded if more intervals are used. An exact bound for (5) for particular $N$ can be obtained by applying the intermediate value theorem to the numerator and denominator, and noting that $2^x$ is convex, so the slope is increasing, so only the slopes at the endpoints need to be evaluated (numerically). The asymptotic expression (5) follows by substituting the approximation

$$2^x - 1 = x \log 2 + O(x^2),$$

into the LHS after dividing both sides by $2^x$

$$\frac{|2^x - 2^\mu|}{|2^x - 1|} = \frac{|1 - 2^f'|}{|1 - 2^{-x}|} = \frac{|f'| + O(|f'|^2)}{|x| + O(|x|^2)}$$

and the fact that $|x| > |i|/N$ while $|f'| < 1/(2N)$.

**Case III.** If $I > 0$, then a similar analysis shows that the relative contribution of $2^I \mu p(f')$ grows progressively smaller, so the number of correctly rounded cases will not decrease as $I$ increases. Because $2^x$ is convex, the estimates become gross overestimates of the relative contribution, but that is acceptable.

**Case IV.** If $I < 0$, then the convexity does not help, but a simpler analysis shows that as $I$ becomes increasingly negative, the contribution of the polynomial diminishes. This is the region in which $2^x - 1$ is increasingly insensitive, so the expected large errors in $p(f')$ are reduced by being multiplied by $2^I$, while $2^I + \mu - 1$ approaches $-1$ and can have at most a $1/2$ ulp rounding error.

Putting the cases together, and we expect no more than an ulp error, plus the maximum error in the polynomial approximation.

2. **LOGARITHM**

Our method is successful in unifying all of the functions in this family, including both $\log(x + 1)$ and $\log x$ for bases 2, $e$ and 10, including the use of a single look-up table, and the reduction to a single significant execution path (plus minor cases to clean up exceptions). The single DSL function which generates the whole family is included as an appendix, but in this section, we simplify the exposition.
by restricting our attention to the function \( \log_2(x + 1) \), except where accuracy arguments are different for the \( \log_2 x \) cases.

All efficient floating-point log algorithms depend on extracting the exponent bits from the binary floating point representation, to get the leading term in a decomposition of the logarithm. We evaluate the the floating point number

\[
x + 1 = 2^e f'
\]

as

\[
\log_2(x + 1) = e + \log_2(f').
\] (6)

The difficulty with this strategy is that although \( x \) is an exactly representable floating point number, \( x + 1 \) need not be. For very small \( x \), the error relative to \( |x| \) may be very large. This is the reason that the function \( x \rightarrow \log(1 + x) \) is included in standard libraries. The approach we have previously used (similar to the approach of [Gal and Bachelis 1991]) is to use a special approximation for \( x < 1 \), and use an accurate \( \log(x) \) computation when \( x \geq 1 \). This multiple-case approach is very expensive on SIMD architectures in which all cases are executed in parallel and the appropriate result is selected, and even on scalar architectures, such data-dependent branches are increasingly expensive. For this reason, we developed a new approach requiring no special cases.

Like other table approaches, after taking care of the floating point exponent, we use piecewise approximations for arguments in the range \([1, 2)\). Unlike conventional table methods, we use multiplicative range reduction,

\[
x \rightarrow \mu_x x - 1
\] (7)

where \( \mu_x \) is a table value, constant on each of \( N \) intervals, chosen so that

\[
|\mu_x x - 1| < 1/N.
\]

In the \( \log(x) \) case, this fused multiply-add has an accurate result, because the exponent of \( x \) is always larger than the exponent of the result, except for the first interval, in which case we can use \( \mu = 1 \), so that the answer is exact. The result, \( \tilde{x} \), is in a subinterval of \((-1/N, 1/N)\); and in the final step, we can use a polynomial approximation to calculate \( \log(1 + \tilde{x}) \).

For the \( \log(1 + x) \) function, substituting \( x + 1 \) for \( x \) in the procedure above would result in catastrophic loss of precision, but a careful recombination of operations can avoid this. For reasons of efficiency, we reduce the input to the interval \([2, 4)\)

\[
2f' = 2^{1-e}(1 + x).
\]

The reduction operation can be rewritten in terms of the argument \( x \), and rearranged

\[
(2f') \rightarrow \mu(2f') - 2
= \mu 2^{1-e}(x + 1) - 2
= (\mu 2^{1-e})x + (\mu 2^{1-e} - 2).
\] (8)

to preserve accuracy through the use of two multiply-adds (and a multiply). The floating-point translation into three instructions may lose up to \( e \) bits of accuracy,
but the magnitude error in the reduced value is \(< 2^{-52}\). As long as the integer \(e\) is nonzero, the resulting loss of accuracy in the final result

\[
\log_2(x + 1) = e - \log_2 \mu + p \left( (\mu 2^{1-e})x + (\mu 2^{1-e} - 2) \right),
\]

where \(p\) is a polynomial approximation of \(x \mapsto \log_2(1 + x)\), is restricted the rounding error in the polynomial, which will also be on the order of an ulp, because the input is smaller than \(1/N\) and the coefficients are decreasing.

In this method we choose exact values \(\log_2 \mu\) such that \(e - \log \mu\) is exact. Given the relatively narrow range of possible exponents, \(|e| < 1024\), it is sufficient to choose floating point values whose mantissas end in ten zeros. Among such values, we seek accurate values with the property that \(1/\mu\) is very close to an exact floating point number. We put the pairs \(\log_2 \mu\) and round\((1/\mu)\) into the table.

3. IMPLEMENTATION

The functions were implemented in a Domain Specific Language (DSL) embedded in the functional programming language Haskell. Code was generated using Coconut (COde CONstructing User Tool) development environment, see [Anand and Kahl 2009].

The main advantages of the Haskell embedding are: the ease of adding language features using type classes and higher order functions, and the strong static typing in Haskell which catches many patterns abuses at run time. Some features could be easily implemented using C preprocessor, or C++ template features, but others, notably table look-ups, would be cumbersome to implement and very difficult to maintain. See [Anand and Kahl 2009], for a full account of the language, and the results of implementing a single-precision special function library for the Cell BE SPU.

The double precision library, SPU DP MASS, was implemented using this tool, and is distributed starting with the Cell BE SDK 3.1. It was found to be four times faster than the best alternative, SIMD Math, implemented in C using processor intrinsics. Most of the improvements stem from efficient patterns for table look-ups and leveraging higher levels of parallelism through partial unrolling. The performance improvements reported here are relative to the faster MASS implementations developed using Coconut, so the differences are the result of the improved algorithm and not more efficient implementation.

In future work, we will show that this approach works well for other SIMD architectures. However, some of the patterns which are efficient on one implementation of a single instruction set architecture will not necessarily be the same for another implementation, and patterns for different architectures will be even more different. Our approach is not to automatically discover these patterns, but to use different extensions to the language for different architectures. For this reason it is useful to review the features of the Cell BE SPU of interest to high-performance numerical software developers, and how they may be exploited.

3.1 Cell BE SPU

The Cell BE (Broadband Engine) architecture contains a POWER architecture microprocessor together with multiple compute engines (SPUs) each with 256K of
private local memory, which in our case, put a limit on table size when multiple special functions are used together to process blocks of data.

All SPU computation uses Single Instruction Multiple Data (SIMD) instructions operating on more than one data element packed into a register in parallel. The SPU has a single register file with 128 bit registers. Most instructions operate on components, i.e., four 32-bit integers or 2 64-bit double precision floating point numbers, adding, multiplying or shifting each element in parallel. We call these pure SIMD instructions, to distinguish them from operations operating on the register contents as an array of bytes, or a set of 128 bits.

Since SIMD applies a single instruction to multiple data, it follows that multiple data elements are treated in the same way. The additional cost of branches in an architecture without deep reordering and branch prediction puts a premium on implementations without any exceptional cases. So it is usually faster to make two versions of a computation and then select the right one based on a third predicate computation than it is to branch and execute one of the two computations. For this reason, all our special functions on the SPU are branch-free. Such branch-free implementations are also useful in real-time applications requiring deterministic execution time.

For special function evaluation it means that there are no fast paths. All computation takes the slowest route; so if one branch requires a polynomial of order six, the other branches might as well use the same order. This can get complicated if, for example, some branches require odd polynomials and others generic polynomials. We provide different mechanisms for simplifying the implementation of such sharing. We support looking up multiple coefficients, we support switches, and we support hijacking of internal polynomial evaluation by external callers. This first and last are seen in the code included via Appendices.

The SPU has two dispatch pipelines, an even pipeline corresponding roughly to computation, and an odd pipeline including load-store and bit/byte permutations acting on the whole register. By their nature, special functions are computationally bound, so several patterns make special efforts to use odd instructions wherever possible, including in look-ups and switches.

3.2 Partial Unrolling

In almost all cases, we have found performance increases when computing exceptions, extracting exponents, etc., in words rather than double-words. To take advantage of this, we provide 4-way parallel implementations in most cases. SIMD double precision instructions dictate 2-way parallelism, so we are unrolling by an extra factor of two.

Unrolling by hand is cumbersome and error-prone, so we provide consistent packing and unpacking instructions (which use the odd pipeline) and functions which map functions of registers to functions of pairs of registers and single registers. Zero lines of code are duplicated in our library as part of unrolling.

4. PERFORMANCE

To test the performance of this new approach, we implemented \( \log_e(x + 1) \), \( e^2 - 1 \), and inverse hyperbolic cosine and tangent using the methods described in double precision in Coconut targeting the Cell BE SPU. We compared these to the DP SPU
Fig. 1. Five functions initially implemented as part of the Cell BE SDK 3.1, and reimplemented using the methods of this paper. The SPU has two pipelines, so the expected cycle time after inlining/unrolling is the number of instructions in the more crowded pipeline. Shown are the number of instructions per double precision input for both odd and even pipelines, and both the released versions, and the new versions implemented for this paper.

MASS (Mathematical Acceleration Subsystem) versions, as distributed in the Cell BE SDK 3.1. The results, shown in Figure 1, show an average 70% expected performance improvement. To eliminate the effect of in-lining and instruction scheduling, these results are the instruction counts for the pure functions alone (no loop overhead or array referencing). To reach this level of performance, careful thought must go into instruction selection, and it is impossible to say that the implementations are equally efficient, but in both cases, significant efforts were made to tune performance. The complete code used to generate the current versions are available in the appendices, and the other versions are available in C-language form in the SDK.

Although the forward hyperbolic functions use exponentials, in the same way that the inverse hyperbolic functions use logarithms, we did not generate new versions of those functions because the SPU has weak double precision reciprocal performance, and reciprocals are required to take advantage of the new $2^z - 1$ approach.

5. ACCURACY

We tested each of the functions by simulating execution using Coconut for at least 10000 inputs over the full range, and compared the results to computations carried out in Maple with precision of 500 significant digits.

The accuracy is acceptable for a high-performance library, see Table I, but it is likely that accuracy would be improved by searching for polynomials with better rounding behaviour using the procedure outlined in [Brisebarre et al. 2006] or even by searching through combinations of table values and candidate polynomials, although it is not clear whether the larger search space could be restricted to a reasonable size in some way. This is a question for further research.
Faster Exponential and Logarithm

<table>
<thead>
<tr>
<th>function</th>
<th>max error (ulps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>1.55</td>
</tr>
<tr>
<td>exp2</td>
<td>1.66</td>
</tr>
<tr>
<td>expm1</td>
<td>1.80</td>
</tr>
<tr>
<td>exp2m1</td>
<td>1.29</td>
</tr>
<tr>
<td>log</td>
<td>1.78</td>
</tr>
<tr>
<td>log1p</td>
<td>1.79</td>
</tr>
<tr>
<td>log1p</td>
<td>1.11</td>
</tr>
<tr>
<td>log2</td>
<td>1.00</td>
</tr>
<tr>
<td>acosh</td>
<td>2.01</td>
</tr>
<tr>
<td>asinh</td>
<td>2.20</td>
</tr>
<tr>
<td>atanh</td>
<td>1.46</td>
</tr>
</tbody>
</table>

Table I. Accuracy, as represented by maximum error in ulps.

6. CONCLUSION

We have demonstrated considerable performance improvements by using a novel accurate table approach to calculating \( \log(1 + x) \) and \( e^x - 1 \), and functions evaluated using these functions. The most significant improvement is a more than doubling of the performance of \( \log(1 + x) \). Significant improvements can be expected for all SIMD architectures, and probably on VLIW (very large instruction word) architectures as well.

By unifying the use of tables in this way, we hope that we are laying the foundation for the inclusion of accurate tables in multi-core hardware, where the cost of including such tables would be reduced both by the reduced number of such tables and by the possibility of sharing the tables among multiple cores. This would significantly increase the opportunities for in-lining of such efficient implementations by compilers, and is a topic which should be pursued by manufacturers.

REFERENCES


7. APPENDIX A: DSL SUMMARY

To make the paper self-contained, we include in Table II a description of all the machine instructions we use in the paper. Machine instructions are set in **bold**. They operate on 128-bit vectors, and all of the instructions in the examples can be interpreted to be operating on four 32-bit values in parallel. Functions which translate between Haskell values and DSL register values are set in *italics*, and words from the DSL vocabulary are *underlined*. These display changes are applied automatically by *lhs2TeX* to all source code. The examples (other than cube root) in this paper have been edited for length, but otherwise appear as they do to Coconut developers.

Haskell is sparse of syntax. Blocks are delimited by indenting, lists by [ , ], and tuples by ( , ), both in defining types and constructing data. Data and type constructors begin with a capital letter. With this information, and with existing code to refer to, it is possible for undergrads to start programming on day one.

8. APPENDIX B: LITERATE LOG FAMILY

To avoid cloning code, Coconut math functions have expert interfaces, which allow the user to get special variants with varying levels of exception handling and non-standard input ranges, etc. The expert interfaces also return a list of labelled values (of variables and expressions) are returned to facilitate development and analysis. In the simplest usage, the labelled values are a convenient alternative to debug printing, but exposing variable values to the caller in Haskell, rather than strings, makes data mining in Haskell much easier than it would be using a scripting language like perl.

```
log2Family :: (SPUType n, HasJoin n, MemRegion c n cm)
    ⇒ MathOptions → (n, n) → Bool → ((n, n), [(String, (n, n))])

log2Family (MathOptions base exceptions) v isP1Case = (result,
    []
)

List of labelled values omitted here.
```

```
]]

where
```

For the \( \log(x) \) function, if the input is subnormal,

```
inputSubnorm = pMapPo dfcgt (undoubles2 $ 2 **(-1022)) v
```

use floating point multiplication, *dfm*, to boost it into the normal range for processing, and use a *selb* to select between the two cases. For the \( \log(x + 1) \) case, add 1 to the input.

```
adjustedSubnorm = pMapPo dfm v $ undoubles2 $ 2 **(256)
normalized =
```


<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>andc</td>
<td>bitwise and with complement</td>
</tr>
<tr>
<td>selb</td>
<td>bits in third argument select corresponding bits in first or second argument</td>
</tr>
<tr>
<td>shufb</td>
<td>bytes in third argument index bytes to collect from first two arguments</td>
</tr>
<tr>
<td>fma, fs</td>
<td>32-bit floating point fused multiply-add, subtract</td>
</tr>
<tr>
<td>fcmgt</td>
<td>32-bit floating point compare $&gt;$, producing a 32-bit mask in each word, e.g., for use in selb</td>
</tr>
<tr>
<td>mpya, mpy</td>
<td>16-bit integer multiply fused with 32-bit add, multiply</td>
</tr>
<tr>
<td>mpyui, mpyi</td>
<td>(unsigned) integer multiply with immediate arguments</td>
</tr>
<tr>
<td>rotqbiib</td>
<td>rotate whole quadword up to 7 bits left, number given by immediate</td>
</tr>
<tr>
<td>roti</td>
<td>rotate each 32-bit word by an immediate constant number of bits</td>
</tr>
<tr>
<td>shli</td>
<td>shift each 32-bit word left by an immediate constant number of bits</td>
</tr>
<tr>
<td>ceqi</td>
<td>for each output word, set each bit to 1 if the corresponding input word matches immediate constant, and to 0 otherwise</td>
</tr>
<tr>
<td>ceq</td>
<td>for each output word, set each bit to 1 if the corresponding input words match, and to 0 otherwise</td>
</tr>
<tr>
<td>dfcgt</td>
<td>64-bit double floating point compare $&gt;$, producing a 64-bit mask in each doubleword</td>
</tr>
<tr>
<td>dfm</td>
<td>64-bit double floating point multiplication</td>
</tr>
<tr>
<td>dfa</td>
<td>64-bit double floating point addition</td>
</tr>
<tr>
<td>dfs</td>
<td>64-bit double floating point subtraction</td>
</tr>
<tr>
<td>dfma, dfms</td>
<td>64-bit double floating point fused multiply-add, multiply-subtract</td>
</tr>
<tr>
<td>clgt</td>
<td>32-bit logical compare $&gt;$</td>
</tr>
<tr>
<td>and</td>
<td>bitwise and</td>
</tr>
<tr>
<td>a</td>
<td>32-bit integer add</td>
</tr>
<tr>
<td>dfceq</td>
<td>64-bit double floating point compare $==$</td>
</tr>
<tr>
<td>or</td>
<td>bitwise or</td>
</tr>
<tr>
<td>xor</td>
<td>bitwise xor</td>
</tr>
<tr>
<td>dfnms</td>
<td>64-bit double floating point fused multiply-subtract and then negate</td>
</tr>
<tr>
<td>rotqbyi</td>
<td>rotate quadword by byte immediate</td>
</tr>
<tr>
<td>cgt</td>
<td>32-bit signed compare greater than $&gt;$</td>
</tr>
<tr>
<td>orc</td>
<td>bitwise or with complement</td>
</tr>
<tr>
<td>rotmi</td>
<td>rotate and mask out immediate, provides logical right shift</td>
</tr>
</tbody>
</table>

Table II. Summary of SPU instructions used in this paper.

```haskell
if isP1Case then
  pMapPo dfa v $(undoubles2 $ 1)
else
  pMapPPP selb v adjustedSubnorm inputSubnorm
```

The use of Haskell control flow if-then-else is used to control the generation as preprocessor directives can be used in C. The main advantage to using Haskell control flow is that decisions about code generation can involve arbitrary computations, and arguments passed as typed parameters, with all the advantages of using a programming language has over using a nest of untyped string operations without scoped variables.

In most of our double precision SPU implementations of elementary functions,
we found that additional parallelism can be extracted by operating on 4 values at a time for part of the computation, \( e.g. \) for identifying exceptional cases. This could be done by manually unrolling and replacing select code sequences, or by having the compiler unroll and look for patterns. Since the correctness of any such substitution depends on a case-by-case analysis of floating point binary representations, automatic identification would be challenging. On the other hand, unrolling by hand is tedious and is fertile ground for typos, so we provide a set of higher-order functions to control unrolling, packing and unpacking without duplicating code by hand. The functions \( p2\text{wrd} \) and \( \text{wrd}2P \) pack and unpack a pair of register values (using the high-order words) into a single value, and the functions \( \text{pMap} \) with different suffixes map their first argument over both pairs and single register arguments, with \( P \) identifying pairs, and \( o \) identifying one argument. For example, we see above that \( \text{dfa} \), add doubles, takes two arguments the second of which is a constant, so \( \text{pMapPo} \) is used to unroll this operation, mapping over the two input vectors, but using the same constant for both instances, whereas bit-wise selection, \( \text{selb} \), takes three arguments, all of which are unrolled variables.

Put high words from four doubles into one vector for testing exceptional values, and exponent extraction.

\[
\text{mergeDbls} = p2\text{wrd normalized}
\]

Use bit manipulation in the odd pipeline to form the high words of the doubles \( (2^{20} + 0xf000 + \text{expBits}) \), as follows: isolate the two bytes containing the exponent bits, and prepend \( 0x00ff \),

\[
\text{keep2bytes} = \text{shufB mergeDbls mergeDbls} \\
[\text{shufb0x00, shufb0xff, 0, 1, shufb0x00, shufb0xff, 4, 5} \\
, \text{shufb0x00, shufb0xff, 8, 9, shufb0x00, shufb0xff, 12, 13}]
\]

rotate the bits to align the exponent bits to the byte boundary (rotating zero bits out of one word and into its neighbour),

\[
\text{alignRight} = \text{rotqbii keep2bytes 4}
\]

now extract the two middle bytes from each word (containing \( 0xf000 + \text{expBits} \)) and merge them with the top two bytes of \( 2^{20} \) which contain the sign, exponent and four zero mantissa bits

\[
\text{expAsDbls} = \text{shufB alignRight (undoubles2} \ § 2 \ \uparrow \ 20) \\
[16, 17, 1, 2, 16, 17, 5, 6, 16, 17, 9, 10, 16, 17, 13, 14]
\]

Note: negative inputs produce garbage results, but this case is caught by exception handling. This version reduces the number of even instructions by extracting the exponent and mantissa with \( \text{shufb} \), and using a \( \text{rotqbii} \) to align the fraction.

Subtract the \( 2^{20} + 0xf \) nibble from the result, and subtract exponent bias, \( 0 \times 3ff \). Note that this must be done using 64-bit doubles, because there are no “packed double precision” instructions.

\[
\text{expOffset'} = \text{pMapPo dfa (wrd2P expAsDbls) (undoubles2} \ § 2 \ \uparrow \ 20 + 0 \times 3ff)
\]

In the \( \log(x) \) case, if the input is subnormal we will subtract 256 from the answer to adjust for multiplication by \( 2^{256} \).
expOffset =
    if isP1Case then
        expOffset'
    else
        pMapPP dfs expOffset' subtractSubnorm
subtractSubnorm = wrd2P $ selb (unwrds4 $ head $ wrds $ idSim $ undoubles2 $ 0)
    (unwrds4 $ head $ wrds $ idSim $ undoubles2 256)
    (p2wrd inputSubnorm)

The constant coefficient includes the accurate table value, the offset for the exponent
and the offset for subnormals.

c0PexpPart = pMapPP dfs expOffset c0

The fractional part is decomposed using \( \log(x/\alpha) = \log x - \log \alpha \), where \( c0 = \log \alpha \)
and \( \text{mult} \approx \alpha \). The \( c0 \) values are exact, and chosen so that the \( \text{mult} \) values are
accurate. This is a reverse accurate table method, in the sense that the image of
the function is exactly representable, but the argument is not. The direction

\[ [c0, \text{mult}] = \text{lookupWrdsPairVals} (19, 8) \log2Table BN.nat4096 mergeDbls \]

The \text{lookupWrdsPairVals} returns two pairs of 128-bit registers, each containing two
doubles. The argument \((19, 8)\) tells the Haskell which bits (little-endian order) are
used to form a key. This version takes a single (packed) register value as input,
and extracts the key bits in this form to save instructions. The BN.nat4096 is a
variable whose value is never used, but whose type encodes the size of the array.
This allows the Haskell type checker to match array sizes. The size of the array of
values, and the width of the key field are checked when the code graph is generated,
and in the SPU ISA simulator.

The accuracy in the accurate table method plays a role in two places. This computing
the fractional offset, it is important to use a fused multiply-subtract or two
fused multiply-adds in the \( \log(1 + x) \) and \( \log(x) \) cases respectively. Increased accuracy
is obtained by by using an accurate table value for the multiplication. Because
the reduction rule for logarithm is multiplicative, we reduce with an accurate scale
factor, and constant offset, so that we can use a single polynomial approximation
for all intervals.

fracMOffset =
    if isP1Case then
        expPart'
    else
        pMapPP dfs fracMOffset' subtractSubnorm
fractionSubnorm = wrd2P $ selb (unwrds4 0 $ head $ wrds $ idSim $ undoubles2 $ 0)
    (unwrds4 0 $ head $ wrds $ idSim $ undoubles2 256)
    (p2wrd inputSubnorm)

In the \( \log(x + 1) \) case, we must do the reduction in two steps we reduce to the
fraction by multiplying by \( 2^{-e+1} \text{two1MExponent} \),

\[
\text{let two1MExponent} = \text{wrd2P$}
\text{selb (unwrds4 0 x7ff000000) (unwrds4 0 x00000000) mergeDbls}
\]
calculated with one \text{selb} instruction as the complement of the exponent bits, ex-
tracted from the floating point number by selecting 1 bits for 0 exponent bits, and
0 bits otherwise. This single instruction does four extractions, negations and sub-
tractions of one. This is then multiplied by the input resulting in a double value
with the same mantissa, but exponent \(-1\), \( i.e. \frac{1}{2} \leq 2^{-e-1}x < 1 \).
frac1p = pMapPP dfm v two1MExponent
in pMapPPP dfma mult frac1p $pMapPPP dfma mult two1MExponent (undoubles2 2)

In the \( \log(x) \) case, because all the inputs are known to be positive, normalized numbers, we can insert the exponent bits we want using a single \texttt{selb}:

\[
\text{else let }
\quad \text{fracLeft} = \text{selb mergeDbls (unwrds4 0 x3FF000000) (unwrds4 0 x7FF000000)}
\quad \text{frac} =
\quad \text{let } (v1, v2) = \text{normalized}
\quad \text{in (shufB v1 fracLeft [16, 17, 18, 19, 4, 5, 6, 7, 24, 25, 26, 27, 12, 13, 14, 15], shufB v2 fracLeft [20, 21, 22, 23, 4, 5, 6, 7, 28, 29, 30, 31, 12, 13, 14, 15])}
\]

and the reduction is simpler, requiring a single fused operation,

\in pMapPPP dfms mult frac (undoubles2 1)

Note that we could have inserted the exponent directly into the normalized inputs, but inserting them into the packed \texttt{mergeDbls} reduces the number of even instructions.

We use a polynomials to calculate \( \log x \) or \( \log(x+1) \) of the reduced value, obtained using Maple.

\[
\text{evalPoly} = \text{hornerPDBl (c0PexpPart : fixedCoeffs) fracMOffset}
\text{fixedCoeffs} = \text{map ((\lambda v \rightarrow (v, v)) \circ unquads2) isP1Case}
\text{then } [0 \times 3FE71547652B82FE, 0 \times BF7C71547652B9043, 0 \times FAE709E080ED7, 0 \times FB71469DCAE28A8]
\text{else } [0 \times 3FF71547652B82FE, 0 \times BF71547652B9043, 0 \times FDE709E080ED7, 0 \times BFD71469DCAE28A8]
\]

In base 2, the polynomial evaluates to the result, otherwise we have to convert bases. We use fused multiply-add’s to improve final rounding.

\[
\text{resultNormal} = \text{case base of}
\quad \text{MO2 } \rightarrow \text{evalPoly}
\quad \text{MO10 } \rightarrow \text{pMapPoP dfma evalPoly log10ScaleHigh $pMapPoP dfma evalPoly log10ScaleLow}
\quad \text{MOe } \rightarrow \text{pMapPoP dfma evalPoly logeScaleHigh $pMapPoP dfma evalPoly logeScaleLow}
\quad _ \rightarrow \text{error $"LogDb1.log \ base " + \ show \ base + $" not supported"}$
\]

At some call sites, we may know that the inputs are always in the normal range, or it may be faster to deal with exceptional values in the context of the main function, so we optionally return this result, otherwise we use \texttt{select} a special case.

To save cycles and reduce latency, the exceptional cases and the exceptional flag are calculated in 32-bit words, and converted back using \texttt{wrd2P} and \texttt{wrdLogical2P}.

\[
\text{result} = \text{case exceptions of}
\quad [] \rightarrow \text{resultNormal } -- \text{no exceptions}
\quad _ \rightarrow \text{pMapPPP selb (wrd2P specialVal) resultNormal (wrdLogical2P notSpecial)}
\]
For $\log x$, positive or $\pm 0$ inputs result in nonNaN outputs, similarly inputs $\geq -1$ inputs for $\log(1 + x)$.

```plaintext
outputNotNaN =
  if isP1Case then
    p2wrd $ pMapPo dfcgt v $ undwrds2 0 xBFF0000000000001
  else
    p2wrd $ pMapPo dfcgt v $ undwrds2 0 x8000000000000001
```

Output is negative infinity if input to $\pm$.

```plaintext
isNegInf =
  if isP1Case then
    p2wrd $ pMapPo dfceq v $ undoubles2 (-1)
  else
    p2wrd $ pMapPo dfceq v $ undoubles2 0
```

The output should be $\infty$ if the input is $\infty$.

```plaintext
isPosInf = ceq (SPU.and mergeDbls outputNotNaN) $ unwrds4 0 x7ff00000
```

To minimize the number of even instructions, pack bytes for the three exceptions as $[isNegInf, nonNaN, isPosInf, isPosInf]$ to test together.

```plaintext
exceptionBytes = shufB
  (shufB outputNotNaN isNegInf [16, 0, 0, 0, 20, 4, 0, 0, 24, 8, 0, 0, 28, 12, 0, 0])
isPosInf
[0, 1, 16, 16, 4, 5, 20, 20, 8, 9, 24, 24, 12, 13, 28, 28]
notSpecial = ceq exceptionBytes $ unwrds4 0 x00ff0000
```

This ordering of bytes allows us to extract the sign bit and the top mantissa bit which should be set for a QNaN for four inputs with a single `xor`. We need to disambiguate the multiple `xors` visible in Haskell using the `SPU` namespace.

```plaintext
signQNaN = SPU.xor exceptionBytes $ unwrds4 0 x00080000
```

Merge the sign and first mantissa bits into the high-word of the appropriate exceptional value:

```plaintext
specialVal = selb signQNaN (unwrds4 0 x7ff00000) (unwrds4 0 x7ff7fff)
```

9. **APPENDIX C: LITERATE EXP FAMILY**

The type signature

```plaintext
expFamily :: forall val c cm ⊙ (HasJoin val, SPUType val, MemRegion c val cm) ⇒ Bool
        → MathOptions
        → ((val, val) → (val, val) → (val, val)
          , [(val, val)]
          , (val, val)
          )
        → (val, val)
```
→ (((val, val), (val, val)), (((String, (val, val)))))

expFamily isM1Case (MathOptions base exceptions) (subsSel, subsCoeffs, subsArg) \( \nu = ((\text{if } \text{null exceptions}
\text{then resultNorm}
\text{else if exceptions} \equiv [\text{MOSubnormal}]
\text{then resultWithSubnorm}
\text{else result, evalPoly})

)

List of labelled values omitted here.

)

where

Scale the input according the the base of the exponential to put the scaled integer value into the high-order 11 bits into the mantissa. These bits will become the exponent.

\[
v\text{ScaledOffset} = \text{case base of}
\begin{align*}
\text{MO2} & \rightarrow p\text{MapPo dfa v offsetBump} \\
\text{MOe} & \rightarrow p\text{MapPoo dflna v invLog2 offsetBump} \\
\text{MOeX2} & \rightarrow p\text{MapPoo dflna v invLog2X2 offsetBump} \\
\text{MO2m1} & \rightarrow p\text{MapPoo dflnms v (undoubles2 1) offsetM1Bump} \\
\text{MO2p1} & \rightarrow p\text{MapPo dfa v offsetM1Bump} \\
\text{MO10} & \rightarrow p\text{MapPoo dflna v log10InvLog2 offset}
\end{align*}
\]

Base MOeX2 represents the exponential function \( e^{2x} \), MO2m1 represents \( 2^{-x-1} \) and MO2p1 represents \( 2^{x-1} \). These functions are used in calculation of Hyperbolic functions and other functions not in the standard library.

To improve rounding during range reduction, we use high and low-order double values,

\[
[\text{invLog2, invLog2Low}] = \text{map undwrds2}
\begin{align*}
[0 \times 3FF71547652B82FE, 0 \times 3C777D0FFDA0D25] \\
[0 \times 40071547652B82FE, 0 \times 3C8777D0FFDA0D25] \\
\text{log10InvLog2} = \text{undwrds2 0 \times 400A934F0979A371}
\end{align*}
\]

and use both rounded and ceilinged values.

\[
\begin{align*}
\text{offsetBump} & = \text{undoubles2} (4096 + 2048 + 1023 + 0.5) \\
\text{offset} & = \text{undoubles2} $ 4096 + 2048 + 1023 \\
\text{offsetM1Bump} & = \text{undoubles2} (4096 + 2048 + 1023 + 0.5 - 1) \\
\text{offsetM1} & = \text{undoubles2} $ (4096 + 2048 + 1023 - 1)
\end{align*}
\]

Merge high words of vScaledOffset into a single quadword to increase parallelism to 4 in the construction of the output exponent from the integer part of the input an the handling of exceptional cases.

\[
v\text{OffsetWrd} = p2\text{wrd vScaledOffset}
\]

Rotate the bits to put the integer from the high mantissa bits into the exponent position.
expBitsWrd = equivJoin "only use bits 1 to 11, so both are the same"
[rotLeftOdd 12 vOffsetWrd, roti vOffsetWrd 12]

Since performance is bounded by the number of even instructions, we provide a
preferred computation using rotLeftOdd, a function which uses the minimum num-
ber of odd instructions to rotate the register left. An alternate version using even
instructions is provided to the compiler for cases when the function is in-lined into
an even-limited context.

Compare to smallest offset value which will result in a normalized $2^x$ output. If
the $2^x$ is subnormal (or zero)

twoXSubnorm = cgt (unwrds4 $0 \times 0b80100$) vOffsetWrd

and if the output does not saturate to infinity (i.e., if the first 4 bytes of the double
precision number $4096 + 2048 + 1023 + 1024 + 0.5$)

isNotInf = cgt (unwrds4 $0 \times 0bff000$) vOffsetWrd

In the $2^x$ case, if we are not processing excpetions, subnormal outputs are trunc-
cated to zero, otherwise in this case and for $\infty$ output, we set the exponent to the
smallest normal value, $2^{-1022}$. In the case of $2^x - 1$, $\infty$s do not convert into NaNs
during the computation but some negative inputs which produce $-1$, rather than
subnormal outputs, can still generate infinity and NaN bit patterns, so they also
need to be mased out.

expBitsExcept = if null exceptions $\lor$ isM1Case
    then andc expBitsWrd twoXSubnorm
    else selb expBitsWrd (unwrds4 $0 \times 00100000$) (orc twoXSubnorm isNotInf)

Mask out the non-exponent bits.

expBits = wrd2P $\&$ SPU.and expBitsExcept dblExpBitsInWord

Range reduction: Now construct the reduction to $(-0.5...0.5)$, in two steps: first,
truncate the pseudo fixed-point version of the input to the most significant fractional
bits, using the odd instruction shufb.

v3Bytes = pMapPo shufB1 vScaledOffset
[0, 1, 2, shufb0x00, shufb0x00, shufb0x00, shufb0x00, shufb0x00, shufb0x00, shufb0x00]

Next subtract a modified fixed offset, to get a truncated version of the input with
0.5 added. Now subtract this from the original input, to get the low-order part of
the fraction. This operation is exact, because the resulting exponent is not larger
than the exponent of the input.

frac = case base of
    MO2 $\rightarrow$ pMapPP dfs v (pMapPo dfs v3Bytes offset)
    MOe $\rightarrow$ pMapPP (\lambda vv v3 $\rightarrow$ dfina vv invLog2Low
        (dfms vv invLog2 (dfs v3 offset))) (dfs v3Bytes)
    MOeX2 $\rightarrow$ pMapPP (\lambda vv v3 $\rightarrow$ dfina vv invLog2LowX2
        (dfms vv invLog2X2 (dfs v3 offset)))
In the case of bases other than 2, a second fused operation is required for a correctly rounded result.

**Exceptional Inputs (Part I):** Detect ±∞ as inputs. Note that we are saving instructions by using the packed version of the input, and that this only works if input NaNs have a higher-order bit set (e.g. if they are QNaNs).

\[
\begin{align*}
posInf & = \text{ceq } v\text{OffsetWrd } 0 \times 7ff00000 \\
egInf & = \text{ceq } v\text{OffsetWrd } 0 \times fff00000
\end{align*}
\]

Detect the range which truncates to zero. Because we are using words and not double words, we are truncating some values which should have resulted in the smallest subnormals.

\[
isZero = \text{SPU.\text{and} } (\text{cgt } \text{unwrds4 } 0 \times 40b7cd7f v\text{OffsetWrd}) \quad \text{-- round to 0} \\
(\text{cgt } \text{unwrds4 } 0 \times fff00001 v\text{OffsetWrd}) \quad \text{-- but not NaN}
\]

To prevent large negatives from creating NaNs which propagate to the output, replace frac by 1 if vOffsetWrd is infinite or the output is zero.

\[
\fracOr1 = \text{pMapPp } \text{selb } \text{frac } (\text{undoubles2 } 1) \ (\text{wrd2P } \text{SPU.\or } \text{isZero } \text{posInf})
\]

**Table Lookup:** Now look up the table value, c₀, in the same interval as the fraction. The double precision value c₀AT is much closer to c₀ than an ulp. Also look up \( \text{mult} = 2^{c₀} \).

\[
[\text{mult}, c₀AT] = \text{lookupWrsPairVals } (7, 0) \ (\text{map } (\text{snd} \ \text{exp2Tables}) \ \text{nat256}) v\text{OffsetWrd}
\]

Calculate the offset from the table value.

\[
\fracMc0 = \text{pMapPp } \text{dfs } \fracOr1 \ c₀AT
\]

Now, \( \fracMc0 \) is very small number in range \([-2^{−8}..2^{−8}]\). Exponential functions \( 2^{\fracMc0} \) or \( 2^{\fracMc0} − 1 \) are approximated by minimax polynomials determined using Maple.

Because \( e^x \) is in-lined into functions which sometimes bypass its results, we provide a mechanism to the caller for evaluating an alternative polynomial. The passed function \( \text{subsSel} \) may insert code to do the substitution. This only makes sense for even-limited code, in which case \( \text{subSel} \) will evaluate to a partially evaluated \text{shufb} instruction.

For the compiler to automatically make this kind of substitution would require the annotation of code graphs with predicates on each node indicating which input value ranges will later be discarded. This is a long-term goal for Coconut, but in the meantime, the mechanism presented here is much safer than manual cloning and editing, because clones do not need to be maintained, and the “switch” instructions will necessarily act in concert.
evalPoly' = hornerPDb1 coefs $ subsSel subsArg fracMc0
coefs = zipWith subsSel subsCoeffs $ map (s2Pair o undwrds2) $
if isM1Case
   then [0 x3FE62E42FEFA39EF, 0 x3FCEBFBDFF82C3FB
          , 0 x3FAC6B08D7049F20, 0 x3F83B2ABD4AD2546, 0 x3F55D88577EA318]
   else [0 x3FF000000000000, 0 x3FE62E42FEFA3685
          , 0 x3FCEBFBDFF82C175, 0 x3FAC6B09B179A259, 0 x3F83B2ABFD0DDC31]

In the $2^x - 1$ case, the polynomial is odd with zero constant coefficient, so an
additional multiplication is needed.

\[
evalPoly = \begin{cases} 
    (pMapPP dfm evalPoly' fracMc0) & \text{if isM1Case} \\
    \text{evalPoly'} & \text{else}
\end{cases}
\]

**Put it together:** Having calculated the power of the reduced value, we use floating
point multiplication to add in the exponent bits in \text{expBits} and scale by the (exact)
table values \text{mult}.

In the $e^x - 1$ case, the only exceptional case is the $\infty$ result. We propogate
it by adding $\infty$ in forming \text{expPoly}. Large negatives and $-\infty$ are handled in the
calculation of \text{expBits}.

\[
\text{zerolnInf} = \text{wrd2P} \, \text{selb} \, \text{dblExpBitsInWord} \, (\text{unwrds} \, 0 \, \text{x0}) \, \text{isNotInf}
\]
\[
\text{expPoly} = \begin{cases} 
    (pMapPPP \text{dfma} \text{expBits} \text{evalPoly} \text{zerolnInf}) & \text{if isM1Case} \\
    \text{pMapPP dfm expBits evalPoly} & \text{else}
\end{cases}
\]

Now scale by \text{mult}, taking

\[
\text{resultNorm} = \begin{cases} 
    \text{pMapPPP dfm expPoly mult} & \text{if isM1Case} \\
    \text{pMapPP dfm expPoly mult} & \text{else}
\end{cases}
\]

**Exceptional Inputs (Part II):** In the $2^x$ case, we still need to take care of
exceptional values. We do this in parallel with the normal computation in a single
quadword by calculating one multiplicative adjustment which is 1 in the normal
case, 0 or $\infty$ in those exceptional cases, and, if the result will be a subnormal, we
construct a subnormal adjustment using integer add \text{a}, based on the (adjusted)
exponential bits. Starting with the 1 or subnormal cases:

\[
\text{denormExpAdjust} = \text{a} \, (\text{unwrds} \, 1022 \, \text{x} \, 2 \, \text{w} \, 20) \, \text{expBitsWrd}
\]
\[
\text{subnorm1} = \text{selb} \, (\text{unwrds} \, 0 \, \text{x} \, 3FF00000) \, \text{denormExpAdjust} \, \text{twoXSubnorm}
\]

we override for $\infty$ or 0:

\[
\text{subnorm1Inf} = \text{selb} \, \text{dblExpBitsInWord} \, \text{subnorm1 isNotInf}
\]
\[
\text{subnorm1Inf0} = \text{wrd2P} \, \text{SPU. and} \, (\text{andc} \, \text{subnorm1Inf isZero}) \, \text{dblExpBitsInWord}
\]
This works because the special values, 0 and $\infty$, are idempotent among non-NaNs. In the $2^x - 1$ case, we get the subnormal outputs directly from the polynomial evaluation, and the careful order in which the multiplier from the table and exponent bits are added.

\[
\text{result} = \text{if } \text{isM1Case} \\
\quad \text{then} \\
\quad \text{resultNorm} \\
\quad \text{else} \\
\quad p\text{MapPP dfm resultNorm subnorm1Inf0}
\]

For pow we want a separate version which handles subnormal outputs but not other exceptions.

\[
\text{resultWithSubnorm} = p\text{MapPP} \\
\text{dfm resultNorm (wrd2P $\&$ SPU.\text{and (andc subnorm1 isZero) dblExpBitsInWord})}
\]

Note on NaN: an input NaN will propagate to frac via floating point operations.