Procedia Computer Science

Volume 51, 2015, Pages 1828–1837

ICCS 2015 International Conference On Computational Science

Transmathematical Basis of Infinitely Scalable Pipeline Machines

James A.D.W. Anderson[1]

University of Reading, UK
j.anderson@reading.ac.uk

Abstract
We describe infinitely scalable pipeline machines with perfect parallelism, in the sense that every instruction of an inline program is executed, on successive data, on every clock tick. Programs with shared data effectively execute in less than a clock tick. We show that pipeline machines are faster than single or multi-core, von Neumann machines for sufficiently many program runs of a sufficiently time consuming program. Our pipeline machines exploit the totality of transreal arithmetic and the known waiting time of statically compiled programs to deliver the interesting property that they need no hardware or software exception handling.

Keywords: transreal arihtmetic, von Neumann Machine, pipeline

1 Introduction

A recent report sets out some mathematical challenges to developing machines and algorithms that operate at Exascale [12]. Delivering general purpose computation, on a machine with multiple von Neumann cores, is an ambitious goal. Here we examine the more limited question of how to construct a pipeline machine of arbitrary size. Such a machine has some long latency but then produces a result every clock tick. It executes an entire, in line, program, on successive data, every clock tick, giving it perfect parallelism. To achieve this we require unbreakable pipelines where, apart from physical errors, a core is guaranteed to pass on its data every clock tick. This requires a total system of computation that has no logical exceptions: we use transmathematics – a collection of techniques that rely on an extension of real arithmetic, called transreal arithmetic [9]. We report three architectures. Our first architecture uses a single instruction, implemented in a fixed-point, transinteger arithmetic. This architecture was tested in FPGA and in software simulation in a custom Java program. The architecture was then modified, with the introduction of a second instruction, to operate on trans-floating-point numbers, making it more suitable for scientific computation. Finally the architecture was completely pipelined. These latter two architectures were simulated only in software. We then obtained very approximate estimates of the die area and power consumption of the final architecture that lead us to believe that exascale computing is achievable now.

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2015

doi:10.1016/j.procs.2015.05.408

2 Transmathematics

Transreal arithmetic is a superset of real arithmetic. It defines division in terms of the lexical reciprocal so that $\frac{a}{b} \div \frac{c}{d} = \frac{a}{b} \times \frac{d}{c}$ for all real numbers $a, b, c, d \in \mathbb{R}$. This definition contains the usual definition of division as multiplication by the multiplicative inverse but additionally allows division by zero. All real numbers divided by zero produce non-finite numbers. All positive numbers, $0 < k \in \mathbb{R}$, divided by zero are equal to the definite number infinity: $k/0 = \infty$. Similarly $-k/0 = -\infty$. Zero divided by zero is the definite number nullity: $0/0 = \Phi$. Nullity was introduced in [3]. It is an unordered number that lies outside the range from negative infinity to positive infinity. Arithmetic on non-finite numbers reduces to arithmetic on ratios of real numbers. Transreal arithmetic is axiomatised and proved consistent by a machine proof in [9]. A human, constructive proof of the consistency of both transreal and transcomplex arithmetic is give in [14]. A tutorial on transreal arithmetic appears in [6]. Classical and many-valued logics are unified by arithmetising them in transreal arithmetic in [10]. Importantly all real limits, derivatives and integrals are extended to transreal form in [8][15]. Hence many scientific applications may be extended from real to transreal form, making them immune to division by zero errors – because dividing by zero is not an error in the new paradigm of transmathematics.

A transreal extension of two's complement arithmetic was implemented in both hardware (FPGA) and software simulation [5]. This arithmetic uses the binary code for the most negative integer to encode nullity. This removes bias by removing this one, excess, integer. The excess integer is known as the *weird number* because it has the paradoxical property $-n = n < 0$. The adjacent binary codes are used to encode positive and negative infinity so that the arithmetic is saturated. This prevents wraparound where the sum of two positive numbers can be negative or zero and the sum of two negative numbers can be positive or zero. Thus arithmetic on transreal two's complement numbers faithfully represents the topology [8] of transreal numbers.

A transreal extension of IEEE floating-point arithmetic [1][2] was implemented in software. Trans-floating-point arithmetic [7] uses the bit pattern which encodes IEEE's negative zero to encode nullity. In [8] it is argued that IEEE's negative zero is a category error: it is a limit not a number. This difference is illustrated in the transreal evaluation of the tangent function where IEEE's negative zero alternately computes infinity with the correct and wrong signs at intervals of $\pi/2$ radians. Transreal arithmetic is total so it has no Not-a-Number (NaN) objects. Instead IEEE's NaNs are mapped to real numbers. This adds one binade which may be used either to double the range of real numbers or else to halve the threshold at which numbers underflow to denormal form. The benefits of these alternative strategies have not yet been explored. The transreal infinities are stored in the most extreme bit pattern with all exponent and mantissa bits set to one. This leaves the sign bit to encode the sign of infinity, as usual. Thus trans-floating-point arithmetic faithfully represents the topology of transreal numbers.

In principle it would be possible to define all arithmetical exceptions, on a case by case basis, as violations of transreal topology. This would produce a single set of exceptions for both trans-two's complement and trans-floating-point arithmetic so that the two arithmetics could be substituted for each other depending only on the precision that is required. Clearly this topological approach to defining machine errors will succeed with any computer arithmetic that faithfully represents the transreal topology [8] or the transcomplex topology [14].

Trans-floating-point arithmetic is simpler that IEEE 754 arithmetic because it has no negative zero and no NaNs. It also has a total set of relational operators – *less than, equal to, greater than* – which can be combined with *not* to produce an irredundant set of operators, unlike both IEEE floating-point arithmetic and real arithmetic [7]. In real arithmetic the logical operator *not* is redundant so that, for example, $x \not< y$ is identical to $x \geq y$.

In summary transreal arithmetic is a total system of arithmetic which is in the process of being extended to many areas of mathematics [20][18][13] to create an entire transmathematics which allows division by zero. This new paradigm promises to make the implementation of scientific applications more reliable by removing division by zero as a cause of error. It offers a topological basis for categorising machine errors. It offers the advantages of more range or else more precision in floating-point arithmetic. This extension of the real values, encoded by floating-point bits, may be exploited in ordinary computation. Critically transreal arithmetic simplifies the implementation of arithmetical pipelines that always respect the topology of transreal numbers and do not break on division by zero.

3 The von Neumann Machine

The von Neumann machine was introduced by a memo, written by John von Neumann, in 1945 [17]. The von Neumann machine has been extraordinarily successful but it has features which force its computational efficiency to decline with increasing size. Each of these can be addressed by a suitable form of dataflow machine.

Firstly the von Neumann bottleneck [11] restricts the amount of data that can be Input to or Output from a computer so that the I/O bandwidth, needed to keep a von Neumann core busy, grows proportionately with the number of cores. When data are not ready, the von Neumann machine stalls until data are ready. Hence, for any particular bandwidth, the amount of time spent stalled, increases with the number of cores. In particular the von Neumann bottleneck manifests in hardware when data enter at the edge of a computer chip and multiple cores are present in the surface of the chip. The bottleneck arises because the amount of I/O grows linearly with the perimeter of the chip but the number of cores grows quadratically, in proportion with the area of the chip. In theory this problem could be solved by arranging I/O in the surface of a 2 Dimensional chip but no practical technology to do this has been demonstrated. Should 3D chips ever become practical, the bottleneck would arise even where 2D I/O is implemented. A 3D chip would require a volumetric form of I/O to be free of the bottleneck. However there is an alternative way of avoiding the bottleneck without resorting to exotic geometries: use a dataflow machine where data are input at an edge of a chip and flow over many cores before being output at an edge of the chip.

Secondly the von Neumann machine has a linear address space [17] which requires conditional instructions, such as *if-then-else*, to perform non-local jumps in memory. A non-local jump in hardware is limited by the physical constraint that signals cannot propagate faster than the speed of light so a sufficiently long jump cannot be completed in one clock tick and the von Neumann core is forced to stall, as above. Thus the efficiency of a von Neumann machine decreases with increasing physical size. This inefficiency can be avoided by employing a physical, 2D address space by jumping, say, down to satisfy the *then* part of a conditional and jumping right to satisfy the *else* part. In this way the jump is kept local, regardless of the size of the code in the *then* and *else* parts. With a square array of processors, the jump may be kept to the adjacent core so that it is always completed within one clock tick. The jumps implicit in subroutine call and return can be handled by calling programs inline and by forbidding recursive programs in favour of iterative ones. The jumps implicit in iterations may be handled by loop unrolling, where the waiting time of the algorithm is known, and can be ameliorated by arranging that the loop entry and exit points are physically close when the waiting time is unknown. These strategies require a very large number of cores, which increases the degree of parallelism in a pipeline machine but they place requirements on pipeline compilers. It is possible to copy code to effect recursion in a pipelined way but this strategy is expensive.

Thirdly the von Neumann machine has a fixed address space [17], typically encoded in a single machine word, so programs cannot grow beyond this size. This problem is obviated by making all addressing relative to the current core so that a fixed length of address encodes a fixed address horizon not a fixed address space. In principle a dataflow machine with address horizons can be scaled to any size.[1]

Taken together these methods keep the efficiency of a suitable dataflow machine constant, despite increasing size of the machine, but they imply constraints on the way that programs are laid out: data must flow, from edge to edge of a chip, through the cores in its surface; all jumps must be local, iteration is preferred to recursion and loops must be unrolled with a known waiting time or else with physically close loop entry and exit points; memory must be addressed in a horizon not in a fixed memory space.

4 Infinitely Scalable Pipeline Machines

Our criticism of the von Neumann machine, in the section immediately above, has given us some basic constraints for building infinitely scalable pipeline machines which we amplify here. First, though, we observe that any program for a digital computer can be converted into arithmetic by the process of Gödelisation [16]. Where the Gödelisation is carried through in transreal, specifically transinteger, arithmetic we are guaranteed that an arithmetical machine can execute a Gödelised program without any arithmetical exceptions, because transarithmetics are total. This somewhat trite existence proof has the very great psychological benefit that it guarantees that computer architectures can be constructed that have no logical exceptions. This encourages the designer to find such systems.

The constraint that data must flow from edge to edge of a chip, over the cores in its surface, is easy to achieve but combining such chips in a machine raises the issue of timing. In the absence of a compressible I/O medium, we are obliged to clock all of the cores from a common time domain. We do not require that the entire machine operates in a constant time, only that the cores within an address horizon do. This gives us a physical constraint on the size of the address horizon. It must be chosen so that clock skew, within the horizon, is less than a clock tick. Hence data sent by different routes to a single core can be guaranteed to arrive at the same time within a horizon. In other words data arrival at a core, within a horizon, is a synchronisation point. In practical terms it is easier to satisfy this constraint if a common time domain is imposed or if data paths are made linear, rather than allowing arbitrary 2D addressing, at some large scale. For example that boards have a linear address space but that chips within a board have full 2D addressing. At a larger scale, boards, within a cabinet, might have full 2D addressing but cabinets, within a network, might have linear addressing and so on, up to whatever scale can be synchronised with available technology.

Address horizons have a second, very important, advantage. They are small and can be much smaller than the address space of the machine. This means that tokens within a dataflow machine can use fewer bits to specify an address. This makes the token smaller so that more cores can be fitted on a chip. It also means that fewer address lines are needed, which reduces the risk of transmission desynchronisation within a token.

[1]We can conceive of building machines that are larger than the visible universe. Let us imagine that matter to build the machine and energy to run it, can be harvested from the vacuum energy of the universe. If independent civilisation construct large machines and join them together then some different parts of the machine may be beyond the visible horizon, as seen from every point in the machine. Alternatively if a civilisation builds a large machine and uses it for a long time then the expansion of the universe may carry parts of the machine beyond the visible horizon.

The constraint that all jumps must be local is easy to satisfy for conditional instructions. If the waiting time is known (the number of iterations of a loop or the number of calls of a subroutine is known) then loops may be unrolled and subroutines may be inlined. If the waiting time of a loop is not known then the problem can be finessed, in practical systems, by unrolling within some locality of the machine, such as a cabinet. The outermost loop can then be represented by a large data store whose role is to circulate data through the computational surface. Inlining of subroutines, with an unknown waiting time, can be finessed by storing both the subroutine's code and data in the store. As usual any sufficiently large problem can be handled only by serialising components so, for any fixed size of machine, there is a limit to the size of problem that can be solved in parallel.

With current technology, there will be insufficient I/O to drive each individual core in the edge of a chip. This means that there are programs where it is necessary to expend several clock cycles to input and/or output data for a single program run. This implies a cadence where computation occurs only on the beat, when all data are ready. This reduces the efficiency of a program. However, in some practical cases, such as in molecular dynamics, many copies of a program can be placed in the machine so that they operate on the same data as they flow past the programs. Thus many programs can be executed, on successive data, within one clock tick. In this case the cadence of the entire machine can be a fraction less than one. In many applications the machine will have a low cadence (many clock ticks between computations) during the set-up time when program and initial data are loaded onto the machine, followed by a period of high cadence computing, followed by a low cadence during the tear-down time in which finalised data are taken to backing store, are passed to a network of conventional computers, or are otherwise used.

In a large machine, I/O tap points will be wanted, at different scales, so that an appropriate size of virtual machine can be used. Perhaps the simplest, practical scheme is to treat cabinets as a single machine, with a local area network allowing arbitrary data input and output between cabinets.

5 Experimental Architectures

We have implemented three transreal architectures in software simulation and one, the first, in hardware. The bottom layer of the software simulation is implemented in Java. It uses an event-driven paradigm. In one second of real time, it can simulate all of the processing that goes on in one virtual clock tick of a machine composed of 30,000 transreal cores. The simulator provides instrumentation of the simulated machine and can generate system dumps, utilisation reports and both diagrams and movies of the simulated processing. A particularly useful feature of the simulator is that it records sufficient state to perform backward steps so that when a bug is detected in a simulated program, processing can be unrolled to trace where the bug came from. The simulator and associated tools have been developed throughout this ongoing project.

The first architecture is also simulated in hardware using a Digilent development board with a Virtex-II Pro, FPGA chip from Xilinix. This simulated architecture is a square array of 9 cores laid out in a 3×3 grid so that we can test the connectivity of hardware cores.

The first and all subsequent architectures are dataflow architectures that pass tokens of a fixed length. The tokens have a header and a datum. The header holds a horizon address and a few bits of control data. Thus a token carries both control and data. We have experimented with various sizes of header but a size of 16 bits seems to provide good performance in large, simulated machines. The typical use case is to flood tokens onto the compute surface to instantiate a program in the machine before a job is executed. In a typical case a job contains one or more repetitions of a set-up phase, in which initial data are added to a program, a compute phase, where the program is executed, and a tear-down phase, where final data are flushed from the machine. In compute intensive tasks the compute time necessarily dominates the processing time so that the overall processing time tends to the compute time.

All of the architectures use transreal cores that read inputs from, and write outputs to, pipeline registers laid out in the cardinal directions east, north, west, south. We use a fused logic so that within one clock tick: data is read from the input ports, is processed and is written to the output ports. The machine is clocked at I/O speeds, which are very much slower than typical processor speeds and have a correspondingly lower power consumption per unit time. Clocking at I/O speed is essential to maintaining a constant throughput. Hence these machines achieve "peak performance" all of the time. A second advantage of a pipeline bus is that it allows much more flexible routing, and therefore much more flexible layout, than can be achieved in a systolic array [19].

All of the architectures use a multiply-add instruction with results written conditionally to the cardinal ports. This is a 2D arrangement of the 4D Perspex Machine introduced in [4]. The condition is the sign of the result. Transreal numbers have four signs [9]: positive, zero, negative and nullity. This means that the sign is encoded, irredundantly, in two bits and each cardinal direction can handle a separate condition.

The first architecture implemented a fixed-point, transinteger arithmetic [5]. The second and third architectures implemented a trans-floating-point arithmetic [7] but with an additional, general purpose, bitwise operator and shift to allow the packing and unpacking of the sign, exponent and mantissa fields of a floating-point number. This was found to be more efficient than relying on just the arithmetical operation. The third architecture pipelined the core so that the entire machine can be run as a pipeline.

We implemented both scripting languages and compilers. The first architecture was tested by loading compiled code into the FPGA and the software simulator, comparing results, and cross-checking with implementations of the same algorithm implemented in two different programming languages and with hand computations performed by three people. The second and third architectures were tested, at a larger scale, only in software simulation, against canons implemented in two or three languages. The architecture can be thought of as an extreme Reduced Instruction Set Computer with extreme pipelining. We implemented some basic mathematical functions and found that 64-bit operations to obtain the integer part of a transfloat, obtain the fractional part of a transfloat, divide one number by another, find the square root of a single number, and a 64-bit exponential each occupy between 25 and 100 cores. All of these algorithms have a throughout of one result per clock tick.

We found that one line of raw Fortran90 typically turns into one core as competing influences play out. Division expands to less than 50 cores, which is a large number, but a multiply add and relational operator all fold into one core, similarly a logical operator and shift fold into one core. The overhead on counted loops and function call and return are all optimised to zero cores by loop unrolling and inlining. This very high code density may be due, in part, to the fact that we were developing the scripting language and compiler on the above tasks so the generated code is close to being hand tuned. We have some extremely early estimates of performance. Hand coding of the mathematics libraries achieves a core utilisation of about 60%. Programs generated by script or compiler have a core utilisation of about 25%. Our strategy is to lay out library functions and compiler templates as two dimensional patches. A patch has an outer boundary which defines its I/O, making it convenient to stitch patches together, and an inner boundary, on and inside which the computation is done. Patches sometimes have inner boundaries that can be unfolded or folded like a skein. This folding can be used to fill in gaps, so achieving higher utilisation. Sometimes computation and routing of different computational paths, within and between patches, can be overlaid in the same cores, thus achieving hyperdense layouts. We estimate 5% wastage at the edge of the chip to perform chip I/O, 20% wastage at the edge of the chip as a margin around unfoldable patches, 20% wastage as gutters between patches, 10% wastage from conditionals where only one path is used by a datum, 50% wastage due to routing within a patch. Combining these multiplicative factors gives a typical wastage of 73% or, equivalently, a utilisation of 27%.

We performed piecewise emulations of the core of a commercial molecular dynamics program and calculated its performance on a 2,000,000 core board. We were able to store 500 copies of the program in the board and execute it with a common data stream. Thus 500 program runs were executed per clock tick, giving an asymptotic cadence of 0.002.

As we have the machine only in simulation, obtaining real-world metrics presents some difficulties. We decided to measure performance in terms of unmodified Wassenaar FLOPs. The Wassenaar arrangement[2] is a legal treaty that controls the export of dual-use technologies that have a legitimate commercial use and may have a military use. The treaty sets out a legal definition of a FLOP which is modified by a weighting factor that is less than one to reflect the inherent inefficiencies in vector and array processors. As our machine is of neither kind we did not apply a weighting factor to it. The legal definition then specifies the FLOP rate of a machine as the product: clock rate in Hz $\times$ number of cores $\times$ number of instructions executed in a core per clock tick. A multiply-add is to be counted as two instructions. Measures of non-floating-point instructions are not specified by the treaty but all such architectures are controlled by a combination of use cases, technology used in fabrication, and data rates.

We specified a slow machine with a 250 MHz clock and 2 M cores, in a single, large board, which delivers 1P unmodified Wassenaar FLOPs, which we write as 1PWFLOPs. We gave this specification, together with a component list, to a commercial silicon designer and asked for an estimate of die area and power consumption. We were told that we might achieve 64k cores on the largest available, 28nm chip, without stitching, and that the high power consumption, due to high gate utilisation, may well be completely offset by the low clock speed. Indeed we were advised to consider increasing the clock speed. We then asked about wafer-scale fabrication. We were advised as follows. Firstly as the cores are so small, it should be possible to map out dead cores and route around them so that there is no upper limit to the size of chip that can be fabricated with stitching. Secondly clock and power distribution should be achievable on a 300mm wafer, which should deliver 1PWFLOP. Thirdly handling heat distribution on a wafer may well be achievable but would require a detailed design effort. If this expert opinion translates into practice then an exascale machine could be composed of 1,000 wafers. This number of components can be wired with high reliability but there remains a problem of physical failures in the wafer, board and network.

These figures lead us to believe that exaflop performance is achievable on a power budget of 20 MW. This is a very rough estimate and should be interpreted in the light of two, contradictory influences. Firstly FLOP rates, as usually measured on high-performance machines, are whole system measures[3] not WFLOPS. This overestimates the performance of our machine. Secondly program runs with fractional cadences (many runs per clock tick) are exceptionally useful in interactive and real-time tasks so their value is underestimated by FLOP rates.

[2]See `http://www.wassenaar.org`

[3]See `http://www.top500.org`

6 Discussion

Let us characterise the time to execute successive program runs, on given datasets, in a parallel machine with multiple von Neumann cores and in a pipeline machine. In the following two equations, the second subscript, m, denotes a multiple von Neumann machine and the second subscript, p, denotes a pipeline machine. T_j is the total time to execute a job composed of P_r program runs. T_p is the time for one program run – excluding T_d, which is the time to read one dataset. The reader is free to add a term to account for the time taken to write a dataset at the end of the computation. Note that here T_d is the cadence of an inline program in a pipeline machine. C is the number of cores. P_i is the number of instantiations of a program in a machine. Note that, in a multiple von Neumann machine, $P_{im} = C$ and we write C explicitly in Equation 1. D is the time to distribute data between cores and is, generally, a function of C. We also characterise the time order, for large P_r, at a level of detail that supports an illuminating comparison.

In the following formula, for a multiple von Neumann machine, the time to execute a job, T_{jm}, in a von Neumann machine with one core is given when $C_m = 1$, $D_m = 0$.

$$T_{jm} = P_r(T_{pm} + T_{dm})/C_m + D_m = O(P_r(T_{pm} + T_{dm})/C_m) \tag{1}$$

In a pipeline machine, the time to execute a job, T_{jp}, is given by the following equation. If a single dataset can be shared between instantiated programs then $P_{ip} > 1$ but if a dataset cannot be shared then $P_{ip} = 1$.

$$T_{jp} = (P_r - 1)T_{dp}/P_{ip} + (T_{pp} + T_{dp}) = O(P_r T_{dp}/P_{ip}) \tag{2}$$

Of course both machines compute the same results but the critical thing to note is that the time order of a multiple von Neumann machine is a function of program length but the time order of a pipeline machine is independent of program length. Therefore, for sufficiently many programs runs of a sufficiently time consuming program, a pipeline machine is faster than a multiple von Neumann machine. As this is the use case of supercomputing, we expect pipeline machines to outperform multiple von Neumann machines on those jobs that are suitable for pipeline processing.

The physical efficiency of pipeline machines is assured by the fact that they move data a very short distance, between cores, on each clock tick. This distance is minimised where the cores are small, as occurs in RISC machines. We have explored architectures with just one and two instructions, making them extreme RISC machines, with extremely small cores. The size of cores is further reduced by operating on short, relative addresses in a horizon, rather than on a long, absolute address in an address space. The size of cores is further minimised by using transreal arithmetic and total bitwise operations. Neither of the instructions we use have any exceptions so no exception handling circuitry is needed.

The class of pipeline machines we consider, above, operate on programs for which the waiting time to completion is known. In this case no software exception handling is needed. If a program enters a logical error state it may be terminated. If handling of this state is wanted then a separate program can wait until the waiting time has expired and then act on the fault, say, by reporting it. Physical faults may be handled in the same way by emitting fault reports at a scheduled time. The report, itself, indicates a failure and any failure of a report to arrive, after the waiting time, is a very serious fault indeed. This is a very significant issue for the discipline of Software Engineering – it introduces exception-free programming and exception-free hardware.

There are many ways to characterise the class of pipeline machines we have described. One way is to see them as active-memory machines where data in each memory location (core) is processed. With this insight we see that such a pipeline machine can emulate a multiple, von Neumann machine.[4] In practice we envisage pipeline machines being used as the compute surface in a machine that has a very large token store that circulates tokens through the pipeline machine. In this way, unrolling loops to the outermost loop, instantiated in the token store, allows general iterations where no useful bound on waiting time is known. We further envisage that this entire machine will be used in a conventional cluster of von Neumann machines that prepare input data for the pipeline machine and handle its output.

7 Conclusion

The efficiency of von Neumann machines declines with increasing size but we describe pipeline machines that retain constant efficiency. These machines have perfect parallelism in the sense that every instruction of an inline program is executed, on successive data, on every clock tick and programs with shared data effectively execute in less than a clock tick. Our pipeline machines do not need exception handling and are faster than single or multiple von Neumann machines for sufficiently many program runs of a sufficiently time consuming program.

7.1 Acknowledgments

Thanks are due to Dr Oswaldo Cadenas for developing the FPGA simulation and to Mr Steve Leach for developing the software simulators and tools, and for proposing cadence as a method of program analysis.

References

[1] Ieee standard for binary floating-point arithmetic. Archived at `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=30711` 1985.

[2] Ieee standard for floating-point arithmetic. Archived at `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4610935` 2008.

[3] J. A. D. W. Anderson. Representing geometrical knowledge. *Phil. Trans. Roy. Soc. Lond. Series B.*, 352(1358):1129–1139, 1997. Archived at `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1692011/pdf/9304680.pdf`

[4] James A. D. W. Anderson. Perspex machine. In Longin Jan Lateki, David M. Mount, and Angela Y. Wu, editors, *Vision Geometry XI*, volume 4794 of *Proceedings of SPIE*, pages 10–21, 2002. Last accessed in March 2015 at `http://www.bookofparagon.com/Mathematics/SPIE.2002.Perspex.pdf`

[5] James A. D. W. Anderson. Perspex machine xi: Topology of the transreal numbers. In S.I. Ao, Oscar Castillo, Craig Douglas, David Dagan Feng, and Jeong-A Lee, editors, *IMECS 2008*, pages 330–33, March 2008. Archived at `http://www.iaeng.org/publication/IMECS2008/IMECS2008_pp330-338.pdf`

[6] James A. D. W. Anderson. Evolutionary and revolutionary effects of transcomputation. In *2nd IMA Conference on Mathematics in Defence*. Institute of Mathematics and its Applications, Oct. 2011. Archived at `http://www.ima.org.uk/_db/_documents/Anderson.pdf`

[7] James A. D. W. Anderson. Trans-floating-point arithmetic removes nine quadrillion redundancies from 64-bit ieee 754 floating-point arithmetic. In *Lecture Notes in Engineering and Computer*

[4]A proof of the Turing completeness of our pipeline machine is known but has not been published.

Science: Proceedings of The World Congress on Engineering and Computer Science 2014, WCECS 2014, 22-24 October, 2014, San Francisco, USA., volume 1, pages 80–85, 2014. Archived at `http://www.iaeng.org/publication/WCECS2014/WCECS2014_pp80-85.pdf`

[8] James A. D. W. Anderson and Tiago S. dos Reis. Transreal limits expose category errors in ieee 754 floating-point arithmetic and in mathematics. In *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering and Computer Science 2014, WCECS 2014, 22-24 October, 2014, San Francisco, USA.*, volume 1, pages 86–91, 2014. Archived at `http://www.iaeng.org/publication/WCECS2014/WCECS2014_pp86-91.pdf`

[9] James A. D. W. Anderson, Norbert Völker, and Andrew A. Adams. Perspex machine viii: Axioms of transreal arithmetic. In Longin Jan Lateki, David M. Mount, and Angela Y. Wu, editors, *Vision Geometry XV*, volume 6499 of *Proceedings of SPIE*, pages 2.1–2.12, 2007. Last accessed in March 2015 at `http://www.bookofparagon.com/Mathematics/PerspexMachineVIII.pdf`

[10] James A.D.W. Anderson and Walter Gomide. Transreal arithmetic as a consistent basis for paraconsistent logics. In *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering and Computer Science 2014, WCECS 2014, 22-24 October, 2014, San Francisco, USA.*, volume 1, pages 103–108, 2014. Archived at `http://www.iaeng.org/publication/WCECS2014/WCECS2014_pp103-108.pdf`

[11] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, Aug. 1978. Archived at `http://dl.acm.org/citation.cfm?id=359579`

[12] Jack Dongarra, Jeffrey Hittinger, John Bell, Luis Chacon, Robert Falgout, Michael Heroux, Paul Hovland, Esmond Ng, Clayton Webster, and Stefan Wild. *Applied Mathematics Research for Exascale Computing.* U.S. Department of Energy, 2014. Archived at `http://science.energy.gov/~/media/ascr/pdf/research/am/docs/EMWGreport.pdf`

[13] Tiago S. dos Reis. Números transreais: matemática ou devaneio? In *14° Seminário Nacional de História da Ciência e da Tecnologia, 2014.*, 2014. Archived at `http://www.14snhct.sbhc.org.br/arquivo/download?ID_ARQUIVO=1899`

[14] Tiago S. dos Reis and James A. D. W. Anderson. Construction of the transcomplex numbers from the complex numbers. In *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering and Computer Science 2014, WCECS 2014, 22-24 October, 2014, San Francisco, USA.*, volume 1, pages 97–102, 2014. Archived at `http://www.iaeng.org/publication/WCECS2014/WCECS2014_pp97-102.pdf`

[15] Tiago S. dos Reis and James A. D. W. Anderson. Transdifferential and transintegral calculus. In *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering and Computer Science 2014, WCECS 2014, 22-24 October, 2014, San Francisco, USA.*, volume 1, pages 92–96, 2014. Archived at `http://www.iaeng.org/publication/WCECS2014/WCECS2014_pp92-96.pdf`

[16] Kurt F. Gödel. Über formal unentscheidbäre satze der principia mathematica und verwandter systeme. *Monatschefte für Mathematik und Physik*, 38:173–199, 1931. Last accessed in March 2015 at `http://www.w-k-essler.de/pdfs/goedel.pdf`

[17] M. D. Godfrey and D. F. Hendry. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(1):11–21, 1993. Archived at `http://dl.acm.org/citation.cfm?id=612553`

[18] W. Gomide and T. S. dos Reis. Números transreais: Sobre a noção de distância. *Synesis - Universidade Católica de Petrópolis*, 5(2):153–166, 2013. Archived at `http://seer.ucp.br/seer/index.php?journal=synesis&page=article&op=view&path%5B%5D=413&path%5B%5D=241`

[19] H. T. Kung and C. E. Leiserson. *Introduction to VLSI Systems*, chapter Algorithms for VLSI processor arrays. Addison-Wesley, 1979. Archived at `http://dl.acm.org/citation.cfm?id=578480`

[20] Alberto A. Martinez. *The Cult of Pythagoras: Math and Myths.* University of Pittsburgh Press, 2012.