Eager Recirculating Memory to Alleviate the Von Neumann Bottleneck

J Edwards[1] and S O'Keefe[2]

[1] Thoughtful Technology, Newcastle, UK jonny@thoughtfultech.co.uk
[2] YCCSA, University of York, York, UK simon.okeefe@york.ac.uk

Abstract. This paper presents an examination of channel based time delays and their application as units which perform storage and computation. We describe the implementation of compound arithmetic operations, and show that by recirculating the impulses along a channel, both memory and computation can be achieved on the same general channel unit. In addition, this approach has the further advantage of performing arithmetic simplification eagerly, so that the resultant use of memory is optimised by the intermediate processing during memory circulation phases.

1 Introduction

Central Processor Unit (CPU) design is fixated on binary data storage. This is not surprising since the development of powerful computation devices has become an integral part of the technological advancement of our modern society. A paper by Edwards *et al* [1] takes a step back from this approach, exploring an alternative method of performing computation without the storage and manipulation of binary data. The method proposed uses time delays across a channel as a representation of numbers and as a computation medium, and in [1] this system is shown to have the capacity to perform all forms of arithmetic computation.

This paper examines this approach in more detail, offering a recirculation method to store arithmetic operation on the channel. This allows processing to occur *on the same medium as storage*, effectively removing the "von Neumann Bottleneck" (explained in more detail in Section 2). Furthermore, since the storage medium is actively recirculating, operations can be processed and simplified in an *eager* fashion, so that the recirculated result is optimised for when it is externally accessed.

The paper is structured as follows: section 2 reviews the von Neumann Bottleneck to assess it's impact on decreasing CPU operation throughput. To provide the necessary background in time delay computation, section 3 summarizes the work presented in [1]. This section then expands on this previous work, to demonstrate that non-trivial computation can be performed, and discusses the arrangement of compound arithmetic operations such that they can can be processed by a single stack-less pass across a channel. The fourth section presents a discussion of how these signals on a channel might be recirculated using the

addition of a *compute flag* to communicate with other processing units and clocks. The paper concludes with a discussion of the continued development of the algorithmic capabilities of the single time based processing units.

2 The von Neumann Bottleneck

The **von Neumann Bottleneck** is a term coined by John Backus in his 1978 Turing Award Lecture[2]. It describes the imbalance between the speed of computation and the speed of memory access in CPUs designed using the von Neumann architecture. The imbalance arises because the speed with which data is acquired from Random Access Memory (RAM) is significantly slower than the speed at which a CPU can perform computation. This results in `wait` cycles while data is acquired, and ultimately slows overall computation time. Non-parallel solutions are mainly defined in terms of pipelining, enhanced caching and branch prediction methods [3]. Alternative architectures such as the **Harvard Architecture**[4] also manipulate the nature of memory by providing programme and data store with separate width data buses. To a large extent the current directions in chip development have masked this problem, by using ever larger caches and reducing processor sizes, in line with *Moore's Law*[5], to increase computation throughput. So far as the authors are aware, there are no current computational approaches, even in parallel systems, that allow memory to also perform computation *on the same processing unit.*

3 Time Delay Processing

Channel computation is defined as the arrangement of data, where data can represent both operation and operand, into a signal. This signal when communicated between a encoder and decoder, via a channel (which is defined as the communicative medium between the encoder and decoder, in the sense of Shannon [6]) facilitates an efficient decoding, resulting in the evaluation of the computation encoded in the data.

A discrete stream of these data (which can be loosely thought of as a program) can exist on one channel, and the system $encoder, channe, decoder$ (see Figure 3 can be though of as a unit of processing, with the encoder and decoder described by suitable finite state machines (implicitly with some requisite digital textitstate).

Fig. 1. The channel computation arrangement

Edwards *et al* [1] propose an encoding approach in this domain which utilises the temporal aspect of the communication to represent the data as a time delay between individual impulses (see Figure 2).

This bears some similarity to various schemes, including Pulse Width Modulation (PWM) [7] and Action Potentials [8].

It is shown then that addition can be performed by the concatenation of two values (see Figure 3) and this is easily extended to subtraction and comparison (See [1], section 3.2). The simplicity of this approach implies small encoding and decoding FSMs which in turn suggests beneficial real-world instantiation (which we discuss further in section 5).

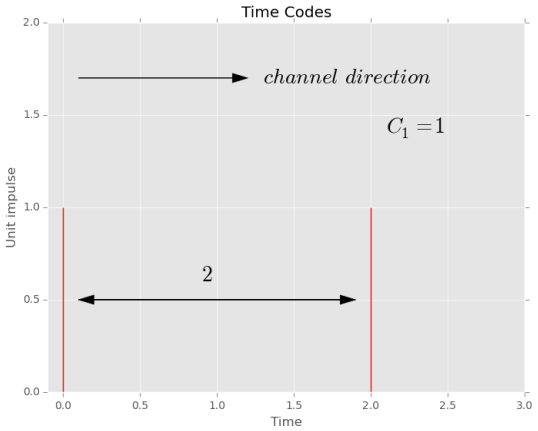

Fig. 2. A Time Delay Unit representing the value 2 as a delay between two impulses.

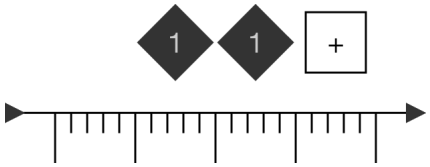

Fig. 3. Simple Addition can be performed by concatenation. The diagram can be read right to left, and shows the addition operator processing two 1 magnitude signals. This results in a single magnitude output of 2.

Furthermore, by the inclusion of two clocks measuring the relative speed of the impulses traversing the channel extends the arithmetic operations to multiplication and division. The term clock in this context is a unit of time measuring device which discretely subdivides the channel into absolutely equivalent time-steps within the encoder and decoder. Also implicit is the notion that the clocks start synchronised and equivalent, and a mechanism exists to scale their time

steps, effectively changing the meaning of a unit time-step at each end the channel. This is illustrated in Figure 4:

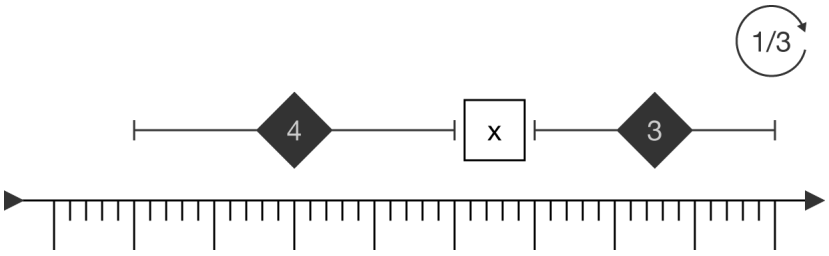

Fig. 4. Multiplication can be implemented using changes to the clock speed. The clock speed is shown at the top right of the diagram, so 1 cycle on this channel represents 3 to an external timer.

3.1 Representing Operands and Operators

A pressing design consideration of this system is to develop a form of representation for *operators*. The most straightforward method is to extend the representation so that it is a **tri-state channel** (states 0, 1, 2). Magnitude is represented by the time delay between impulses of value 1, and computation is represented by the time delay between values of 2. This is illustrated in Figure 5. A suitable (simple) FSM is necessary to map between the numerical magnitude representing the computation, and the actual operation.

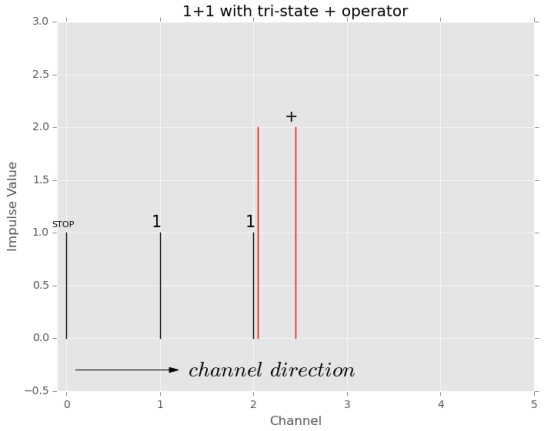

Fig. 5. Instructions can be coded as a separate state on the channel.

3.2 Performing Compound Operations

Addition and Subtraction A requirement for this computation system is that it should utilise the least theoretical "hardware" to perform calculations. There is some subtlety to the way compound operators are stored on the channel as there is no stack to store values from intermediate computations. Figure 6 demonstrates this for the concatenation of *addition* operators.

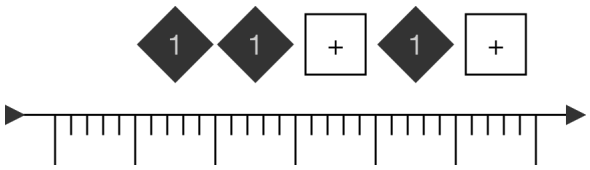

Fig. 6. Addition requires some form of internal memory when performed using an intermediate infix notation.

In evaluating an expression such as $2 + 3 + 3$, the processing unit requires *intermediate storage* to compute the compound expression. To resolve this difficulty, a simple modification of the position of the operator is required. For the system to perform addition no intermediate storage is required, so long as the addition/subtraction operators are arranged in a **prefix** manner. The processing algorithm for addition and subtraction then counts the number of operators and concatenates accordingly. A simple working example (in Figure 7) demonstrates compound addition of $2 + 2 + 3$.

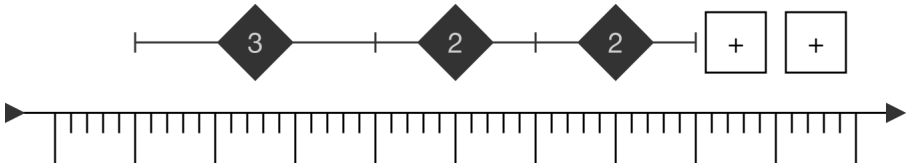

Fig. 7. Addition is a natural process, with a simple algorithm, when represented in prefix notation.

The algorithm starts timing at the first data impulse and simply ignores the number of impulses (encoding addition operators) past this point. It stops timing when the subsequent data impulse arrives. So for example, a prefix of `++` would result in skipping two impulses. The process is similar for subtraction, with the same labeling method as proposed in the original paper ([1] section 3.2) Figure 8 explains the compound arithmetic expression $4 - 2 + 1$ graphically.

Multiplication and Division Multiplication and Division are also possible with only clock speed modification. The most suitable position of the operator

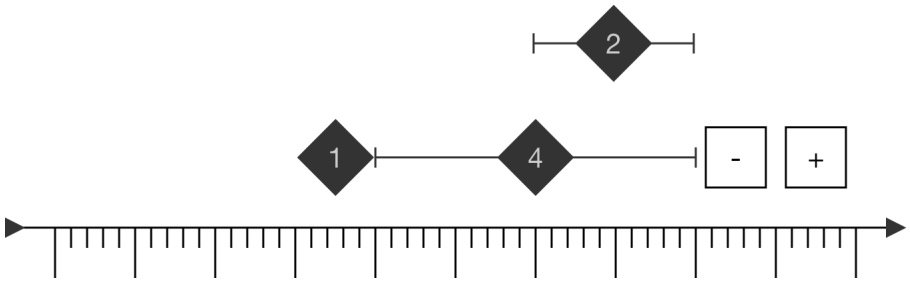

Fig. 8. Subtraction is also easily performed when specified as a prefix operation.

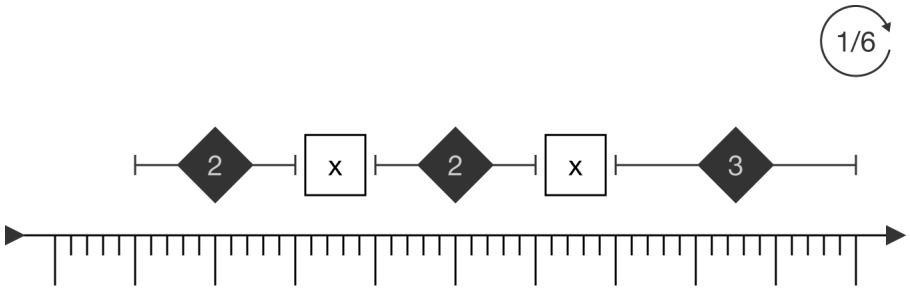

Fig. 9. Multiplication is naturally performed using infix. The first value is moved to the speed register to affect the multiplication. This illustrates evaluation of the expression $3 \times 2 \times 2$.

is **infix** as again this removes the necessity of the intermediate storage. The one caveat is that values must pass to the clock register before exiting the channel. A simple worked example evaluating the expression $3 \times 2 \times 2$ is shown in Figure 9. This is performed without the need for intermediate storage of operands and operators. Likewise division is similarly scaled. A simple worked example evaluating the expression 3/1/3 is shown in Figure 10.

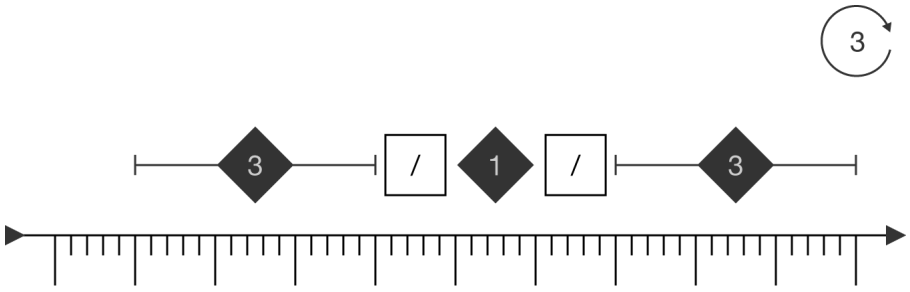

Fig. 10. Division is similar in process to multiplication. This illustrates evaluation of the expression 3/1/3.

As a final summary, Figure11 collects all the ideas above and demonstrates an arithmetic operation with arbitrary complexity, in this case $3+4\times(6+5/(3-2))$.

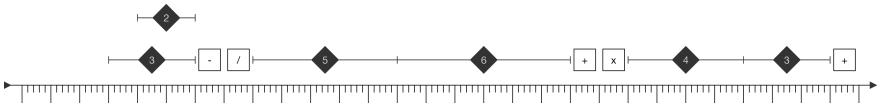

Fig. 11. Complex operations can be performed on a single channel.

4 Eager Recirculating Memory

The section above expands the description of the computational qualities of the arithmetic processes described in the original paper. This computational process can be further modified to become a memory system. This is achieved by the addition of recirculation of the impulses, so the decoder renecoder the signal by connecting two channels in a circular fashion to form a continuous loop. The computation from this loop can be revealed by the further addition of a **computation switch** which transfers the data *out of the loop*. Figure 12 represents this recirculation graphically.

When the switch is toggled and the circuit becomes open, computation occurs and the data emerges from the channel. Significantly, computation and storage

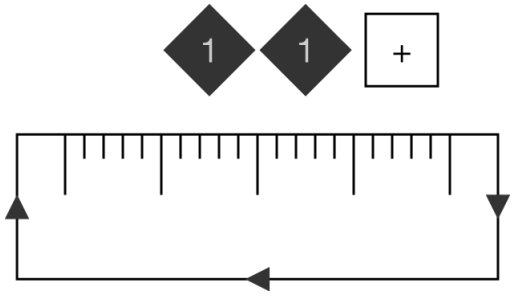

Fig. 12. Memory is implemented using recirculation of the impulses.

are held *together on the same channel*. One interesting aspect of the model is that during recirculation, computation can be silently performed, progressively simplifying the data stored in memory. This performs optimisation of the stored expression for free, and simplifies the memory constraints by removing any processable impulses prior to being made externally available through the setting of the computation switch. The simplest example application of this *eager memory* is the addition of $1 + 1$. This is the shown in Figure13, and is trivially simplified to 2 by one circulation of the impulses around the channel loop.

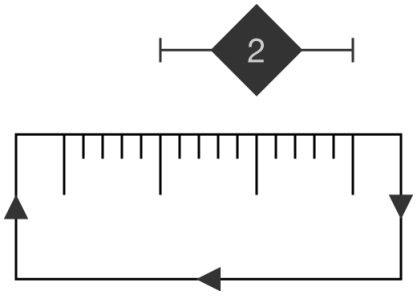

Fig. 13. Instructions can be simplified whilst circulating in memory.

Addition and subtraction operations only manipulate the direct signal on the channel, hence are amenable to simplification. In contrast, multiplication is more problematic since it produces multiplicatively larger operands which, as they are temporally encoded, increase the amount of time necessary for computation. Instead, it may (and we are actively researching) be more efficient to leave the operands in their unoptimised pre-calculation format as multiplicative tuples. An example of this is shown in Figure 14.

So, the output value is only evaluated when re-entering when the computation is directly decoded into it's resultant digital form. The enticing by-product of this is that large numbers can be encoded as multiplications, so the system has an

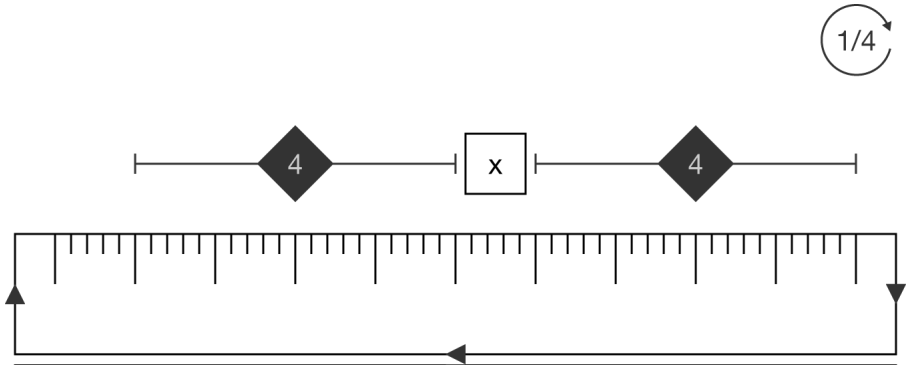

Fig. 14. The intermediate relative clock speeds used in multiplication may prohibit the optimisation of an expression held on the processing unit.

implicit computational compression scheme, although care must be taken with prime numbers (which we are currently investigating as a part of our further research).

The idea of *laziness* is not a new concept in the field of language design and computation. Several modern languages, particularly functional ones, operate an implicit laziness; holding on to computation without extending them out into memory. Even popular multi-paradigm languages like Python have a `yield` operator which stalls computation until a `do` step.

5 Conclusion and Future Developments

Time based computation offers an alternative theoretical strategy to traditional digital computation. In this paper we expand the model to incorporate more compound operations and discuss how they might be held in memory. Furthermore, we have demonstrated that individual time based units can perform *both* memory and computation and with a minimal modification to the originally presented framework. Using the discussed scheme, we have no delineation between memory and computation and this lack of bottleneck has the added advantage of allowing intermediate expression optimisation during storage. WHilst

The next step is to build a more general instruction set that incorporates Minimum Instruction Set Computing (MISC) type instructions. in the hope of building a fully turing complete universal processing unit This will probably involve the linking of individual units into more powerful arrays such that sequence, selection and iteration can occur as a part of inter-channel unit communication. It is also hoped that future research will assess the nature of optimising memory and demonstrate processes that may benefit from its application. It may be possible to perform a system warm-up whilst loading the data into memory, so that effectively much of an algorithmic process has already been worked through

prior to program inception. This is effectively a compilation phase for a more extensive time-based processing system.

6 Acknowledgments

The authors would like to thank Mark Hill, for help in preparation of the diagrams.

References

1. Edwards, J., O'Keefe, S., Henderson, W.D.: Unconventional arithmetic: A system for computation using action potentials. In: Proc. of Unconventional Computation and Natural Computation. (2014) 155–163
2. Backus, J.: Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. Commun. ACM **21**(8) (August 1978) 613–641
3. Shen, J.P., Lipasti, M.H.: Modern processor design : fundamentals of superscalar processors. McGraw-Hill Higher Education, Boston (2005) Index.
4. Cohen, B.: Howard Aiken, Portrait of a computer pioneer. The MIT Press (2000)
5. Moore, G.E.: Readings in computer architecture. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2000) 56–59
6. Shannon, C.E.: A Mathematical Theory of Communication. The Bell System Technical Journal **27**(3) (1948) 379–423
7. Maass, W., Bishop, C.M., eds.: Pulsed Neural Networks. MIT Press, Cambridge, MA, USA (1999)
8. Hopfield, J.J.: Pattern recognition computation using action potential timing for stimulus representation. Nature **376**(6535) (1995) 33–36