Introduction to Implied Instruction Set Computing (IISC) and Multiple Recuring Instruction Set Computing (MRISC)

By *Suminda Sirinath Salpitikorala Dharmasena*
6G, 1st Lane, Pagoda Road, Nugegoda 10250, Sri Lanka.
Telephone: + 94-(0)11-5 364614 E-Mail: {sirinath1978m}@gmail.com

[1] Abstract

The current paper introduces the concepts behind Implied Instruction Set Computing (IISC) and Multiple Recurring Instruction Set Computing (MRISC). In this architecture, the instructions are arranged in recurring fashion thus not needing any decoding. Moreover, this has active and passive instructions which enable significant improvement in instruction level parallelism.

[2] Introduction and Background

There are numerous computer architectures today. They can be categorized according to 1) stored program machine architectures / sequential machines / finite state machines / control flow machines[1] 2) Dataflow machines architectures 3) modern parallel computing models.

The initial stored programme computers had instructions which are executed in sequence, which resembled specification of algorithms as in mathematical proofs and solutions. The instructions are organised sequentially. The execution is also sequential with the exception of jump statements, with modify change this sequential execution. In addition, each instruction is identified by an op code. An instruction may also take operands. The execution cycle of these machines follow a fetch, decode, and execute cycle. In the fetch instruction are fetched from memory, then they are decode or identified and finally they are sent to the relevant FUs for processing. These processors generally have registers which are used for the programme counter and other general or special purpose use.

[1] Machines with Instruction Set Architectures (ISA) which can be modeled using Control Flow Graphs (CFG).

The stored programme concept above is generally attributed to John von Neumann. A break from this stored programme concept are a type of computers called dataflow machines which tries to model the computer operation in terms of data flowing through various operations. Computation occurs on availability of data (data driven) and the results are passed to perform the next computations. Many of these schemes had various overhead (memory wastage in some cases) including token matching overheads, thus making reducing the absolute performance that can be derived out of them. Also in pure Dataflow architectures computed results cannot be re used, i.e., results of a computation flows to the next computation and such results are not stored for further computational use. No Instruction Set Computer (NISC) has does not have many of the drawback of earlier Dataflow machines, but it would requite a high memory bandwidth than Systolic Architectures (discussed below) or machines using registers (discussed previously). In cases the inefficiency due to high memory bandwidth requirement may be superseded by the parallelism they provide.

Also there has been stored program Instruction Set Architectures with an element of parallelism in terms or different instruction classes executed. Very Long Instruction Word (VLIW) is an example of this.

There have been more modern developments like computing with multiple nodes. These schemes are very expensive and cumbersome. Also there are FPGA based solutions which are less expensive, but does give a great degree of parallelism, and are also register rich. FPGA needs to be re configured each change of programme and cannot load a programme with the ease a normal computer can do. Currently, there does not seem to be implementations where FPGAs are used to run a fully fledged Operating System. Currently, programming

FPGAs are not as straight forward as the coding and testing done in "normal" machines. Also the choice of programming languages are restricted, also if main streams languages are used there might be other constrain on the use of the language other than that imposed by the syntax of the language. Also there has been Systolic Architectures where a single Processing Element (PE) is replaced by many PEs and computation is achieved by orchestrating the dataflow between PE. These schemes increase throughput without increasing memory bandwidth. Use of multiple PE, in this scheme, will be very cumbersome, expensive and difficult to implement.

IISC and MRISC is loosely a hybrid of these approaches where the benefits of all can be derived with lesser disadvantages. It also eliminated the decode phase of the execution cycle.

[3] IISC and MRISC

[3.1] Architecture

In this architecture, there are FUs which are wired directly to registers. The connected registers are of two types 1) input registers, 2) output registers. When data is placed in the input registers an output is computed and placed in the output registers.

The instructions aim at actively moving data through registers, while the useful computation occurs passively as a side effect of this data movements.

Since the active instructions aim at only moving data therefore an extensive Instruction Set is not needed. The instructions will be simple move instructions or possibly select and target pair of instructions if an intermediate buffer of registers is used. In the latter case selected instructions are placed in some intermediate buffer of registers, and then these registers are placed into the input registers. Having instructions like move or a pair of select and target as discussed, makes it appear in a multiple recurring pattern. Therefore the instructions can be deduced by the position or occurrence therefore an op code is not needed to identify them. Therefore decoding is not needed.

There are other instructions patterns that can be used other than a simple move or a pair of select as target. E.g. move and set evaluation timer.

Certain IISC and MRISC architectures can be created where the operands of instruction are shared, i.e., an operand is common to many architectures.

[3.2] Compiler and Other Related Technicalities

One of the other main goals of IISC and MRISC design was to give as much control to the compiler over the processor. The compiler can organise what instructions run in parallel and which instructions are sequentially executed (opposed to the processor deciding what instruction to execute out of order.) In addition, it can also explicitly manage the cache.

In the implementation discussed in my patent document, a PE can execute instruction only from the cache. Instructions move data and instructions from shared memory to the cache.

[4] Discussion

[4.1] Functional Programming Based Argument

This scheme would show large performance improvements. This can be demonstrated using graph theory or functional programming. The latter is less tedious and also may be better received since much of the work using parallel algorithms use functional programming. The lazy evaluations of functions theoretically[2]

[2] In practice there may be efficiency problems in PE separately processing each line of execution and synchronizing using Message Passing Interfaces (MPI) than a single PE which can

would provide the possibility of tremendous parallelism. (A graph theory based argument is presented in SL Patent # 13819 document; in addition any comprehensive reference on Dataflow would introduce similar material). This can be shown as follows:

Let $x_1, x_2, \ldots x_n$ be inputs and $y_1, y_2, \ldots y_n$ be outputs.

Let $f_{11}, f_{12}, \ldots f_{nm}$ be functions which are defined interns of the inputs hold the results of functions evaluating functions $f'_{11}, f'_{12} \ldots f'_{nm}$. f_{ij} is such that can be evaluated together, i.e., for a given level i the function are note dependent on other function of the same level. n is the number of levels of functional evaluation and m is the number of functional evaluations per level (may vary from level to level) and $f'_{11}, f'_{12} \ldots f'_{nm}$ be a functions defined in terms of previously defined functions.

The specification for a computational task in functional terms can be in the form:

$f_{11}(x_1, \ldots x_n) = f'_{11}(x_1, x_2, \ldots x_n)$
…
$f_{1m}(x_1, \ldots x_n) = f'_{1m}(x_1, x_2, \ldots x_n)$
…
$f_{21}(x_1, \ldots x_n) = f'_{21}(f_{11}, \ldots f_{1m})$
…
$f_{2m}(x_1, \ldots x_n) = f'_{2m}(f_{11}, \ldots f_{1m})$
…
$f_{n1}(x_1, \ldots x_n) = f'_{1m}(f_{(n-1)1}, \ldots f_{(n-1)m})$
…
$f_{nm}(x_1, \ldots x_n) = f'_{nm}(f_{(n-1)1}, \ldots f_{(n-1)m})$

then,

$y_1 = f_{n1}(x_1, \ldots x_n)$
…
…
$y_m = f_{nm}(x_1, \ldots x_n)$

For each level the arguments of $(f_{11}, \ldots f_{1m}) \ldots (f_{n1}, \ldots f_{nm})$ can be evaluated separately.

inherently handle it, within a certain manageable size.

Any programme that can be written using a functional language can be written using the above scheme. Here $f_{11}, f_{12}, \ldots f_{nm}$ are equivalent to evaluation of function in terms of the inputs it received in terms of the results of previously evaluated functions and $f'_{11} \ldots f'_{nm}$ are equivalent to function definitions.

Each level takes the results evaluated from the previous level as an argument. This is loosely achieved by the active instructions which move data; and the evaluation of $f_{11}, f_{12}, \ldots f_{nm}$ in each level is loosely analogous to the passive evaluation by the functional units.

[4.2] Advantages

[4.2.1] Advantages Over Dataflow Machines

Furthermore, since the functional evaluation above is loosely analogous, when compared to Dataflow architectures, IISC and MRISC can re use previously generated values. This also would be advantageous over Dataflow machines. These values can be re used since the inputs and output are stored in registers and the values in these registers can be moved into other registers. This facilitates the use of previously generated values more than once if needed. (If the analogy was strict, then the result would be again a pure Dataflow machine since intermediate values again cannot be re used.) As mentioned above, IISC and MRISC do not have token passing and matching overheads as in some Dataflow machines.

[4.2.2] Advantages Over RISC/CISC Machines

Current, RISC and CISC have very complex hazard management and out of order execution schemes. In recent times there have been attempts to execute multiple instructions of different classes or possibly from different threads. Implementing these schemes are complex and requires space in the processor. IISC and RMISC overlooks these schemes in favour of passing the

burden of parallelising to the compiler which is more close to the source programme that the processor.

Out-of-order-execution looks at a limited window of instructions thus it does not provide the highest level of optimality in terms instruction scheduling. Out-of-order-execution is also a source of complexities in processors and does have an overhead in terms of wafer space and also contribute towards additional heat, therefore this architecture aims at providing a mechanism to easily parallelise computation at compile time opposed to handle parallelisation by the processor while executing.

The wafer space saved can be utilised for providing additional FUs.

[4.2.3] Over Other Parallel Machines

FPGA based implementations give great degree of parallelism but they cannot be used for general purpose computing. The same is true for Systolic Architectures. An IISC and MRISC gives the advantage of Systolic Architectures using a single processor. In a Systolic Architectures data is moved through and array of PE reducing the memory interaction, in IISC and MRISC what is done is that a similar process is done where data is moved through FUs within a PE reducing the memory bandwidth requirement. This would be less costly and in some cases would be even very much faster that traditional Systolic Architectures. If each PE is given its own cache (local memory space) space as in my patent document, then there would be similarities with Message Passing Architectures.

[5] Conclusions

In functional programming, functional arguments can be evaluated in parallel. This is the source of parallelism in functional programming. Using this new processor architecture would provide a higher level of parallelism can be subjected only to the following two constraints: 1) the availability of FUs, 2) the maximum implemented parallel active instructions. Programmes will run at the highest possible parallelism subjected to the above two limiting factors. If the maximum possible active instructions are sufficiently large then, constrain 2 above will not pose as a limiting factor.

Due to the fact that this new architecture has many of the advantages features across currently existing architectures and has lesser drawback, it can be argued that this processor architecture would generally have an edge over any given architectures. Connection Machines developed by Dr. Daniel W. Hillis gained about 2,000 times performance gain in certain class of applications. (This is a multiple node machine or a machine with many PE.) If a multi node machine is build using IISC and MRISC based PEs then a larger performance gain may be achieved. Certain Dataflow architectures, like the NISC, have gained up to about 16 times performance gains in certain tests. Bases on these figures it is my guess that the performance gain in IISC and MRISC would be slightly more per PE than the NISC for the same test cases since NISC will require a larger memory bandwidth.

At the current state of work the exact performance gains has not been quantified.

[5.1.1] Advantages in use of High Level Languages (HLL)

Certain parallelisation schemes, like the use of FPGA require that certain languages are used and in many cases does have more constrains that that of the language syntax. In other schemes like in the initial CM, the choice of language was limited. The use of this architecture does not impose such constraints.

The only possible complication may arise on use of pointer arithmetic with displaces a pointer's address by a large value (far

pointer arithmetic) or use of variable length parameter lists in C/C++.

[6] Direction for Future Work

Further work needs to be done in quantifying the absolute performance gains for currently used benchmarks. In addition, since the speedup is more particularly tied to algorithms which can be parallelised, therefore performance should be evaluated for most available algorithms. This requires a tremendous effort and research; so this has been set as a future direction.

This processor needs to be implemented. This is also set as a direction for future work.

In addition, after implementing the processor a compiler should be written, and subsequently applications should be ported onto this architecture. Initially the existing Open Source programmes would be the likely candidates for porting.

It is also the wish of the author that the academia would be inspired to undertake such research, since this is beyond the scope of an individual effort.

[7] References

The main reference for this paper is the web. Please search the web using the following key words:

- HCP
- HPCC
- CISC
- RISC
- Petri Nets
- Dataflow Machines
- Manchester Dataflow Machines
- Scheduled Dataflow Processor
- Fine Grain Parallelism
- NISC
- CCA Project
- Message Passing Architectures
- Systolic Architectures
- FPGA
- PRISM
- COMA (Cache-Only Memory Architecture)
- CM (Connection Machines)
- Thinking Machines
- High-Level Language Hardware Abstraction
- High-Level Language to Hardware Translation
- Reconfigurable Computing
- Out of Order Execution
- Data Hazards
- Control Hazards
- Pipeline
- Dr. Daniel W. Hillis
- Guy L. Steele
- Arthur H. Veen
- Jurij Silc

[1] Ralph Duncan, A Survey of J Parallel Computer Architectures, Control Data Corporation, February 1990 IEEE

[2] Gregory W. Donohoe, K.Joseph Hass, Stephen Bruder, Pen-Shu Yeh, Reconfigurable Data Path Processor for Space Applications, http://klabs.org/.

[3] Reiner Hartenstein, The Digital Divide of Computing, TU Kaiserslautern, http://hartenstein.de

[4] Jurij Silc, Borut Robic , And Theo Ungerer, Asynchrony in Parallel Computing: From Dataflow to Multithreading, September 1997

[5] Reiner Hartenstein, Data-Stream-Based Computing: Models and Architectural Resources, Kaiserslautern University of 'Technology, Germany. http://hartenstein.de

[6] M. Reshadi, B. Gorjiara, D. Gajski, "Utilizing Horizontal and Vertical Parallelism Using a No-Instruction-Set Compiler and Custom Datapaths", International Conference on Computer Design (ICCD), October 2005

[7] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, Susan Eggers, The WaveScalar Architecture, Transactions on Computer Systems (TOCS)

[8] The TRIPS Team, Scaling to the End of Silicon with EDGE Architectures, IEEE Computer Society, 2004
[9] Guy L. Steele Jr., W. Daniel Hillis, Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing, Thinking Machines Corporation, ACM 1986
[10] Guy L. Steele Jr., W. Daniel Hillis, Data Parallel Algorithms, Communications of the ACM 1986
[11] Arthur H. Veen, Dataflow Machine Architecture, ACM Computing Surveys 1986