A Compiler Target Model for Line Associative Registers

Paul S. Eberhart, Henry G. Dietz

University of Kentucky
Department of Electrical and Computer Engineering
Lexington, Kentucky, 40508

Abstract. Line Associative Registers (LARs) are the basis for a new class of processor architectures in which memory accesses are minimized by explicitly managing wide lines of instructions and data in processor registers. The design of LARs has significant commonality with a number of existing technologies which have been more or less widely adopted, however, we firmly believe that LARs-based design, which will employ a highly unusual execution model discussed in the remainder of this paper, has ever increasing potential for performance gains over conventional designs utilizing hierarchies of caches and registers. In order to effectively test and utilize this new design, suitable development tools must be written. This paper attempts to describe the implications of a LARs-based architecture for compiler writers, and demonstrate that the benefits of such a design can be harnessed with the use of conventional programming languages. At this time, a HDL verification model implementing a simple LARs-based architecture has been completed, and progress has begun on developing a set of software development tools based on the LLVM compiler infrastructure is underway.

Introduction

Line Associative Registers (LARs) are the basis for a new class of processor architectures in which memory accesses are minimized by explicitly managing wide lines of instructions and data in processor registers. This paper provides an overview of the compiler target model and code generation issues associated with this new class of architecture.

Line Associative Registers fill the role of both registers and caches in a traditional memory hierarchy, bringing many of the advantages of each while avoiding their more egregious faults. This work descends from a previous project, CRegs, which implemented a subset of the design features, and the rise of SWAR (SIMD Within A Register) designs. The LARs design has evolved chiefly to address the ever increasing rift between the speed of CPUs and the main memory that feeds them.

Each individual LAR consists of a many word line for data, as well as a source address, an offset, a type tag, and a dirty bit. The source address and dirty bit function as one would expect in a typical cache. The type tag is set on load,

and indicates the arithmetic type of the data. Because altered data is marked dirty, there is no need for an explicit store mechanism, and in fact all stores are conducted lazily as memory cycles become available. The data in the lines can be addressed as a single scalar value, pointed to by the offset, or as a vector for SWAR operations. In either case, the type fields are used to handle type conversions as an implicit part of arithmetic instructions, casting the output value to the type of the destination LAR.

This architectural design gives rise to a number of significant advantages over a traditional cache and register hierarchy. Perhaps the most important feature of the deign is that, thanks to the tag field, LARs are able to automatically resolve ambiguous aliases, allowing values which could not be kept in classical registers to remain at the top of the memory hierarchy. It is also extremely important that LARs are entirely statically managed; unlike a cache, a compiler can make decisions about allocation, which allows for both more optimal utilization and more consistent timing than could be obtained from a similarly sized set of registers and cache memory. Enabling the compiler to statically manage the memory hierarchy is not a novel concept [1], however, statically managed LARs enable greater benefits that simple prefetching and cache hinting.

Because instructions are executed from LARs, the architecture presents an extremely unusual execution model. Chiefly, the model is novel in that instruction memory does not appear as a static linear space, but rather a sequence of interlinked blocks. In exchange for this violation of convention, which does invalidate many of the assumptions made when targeting classical Von Neumann or Harvard architectures, instructions can be stored and fetched in compressed bocks, further reducing memory traffic. This behavior also makes code replication extremely inexpensive, as repeated executions of the same code incur no additional fetches.

The current effort to implement a toolchain for this design is based on retargeting the LLVM compiler infrastructure to support a LAR-based design. The majority of the features are to be implemented within the infrastructure, however, explicit handling of instruction fetches is beyond the flexibility of LLVM, and will therefore be handled by a custom optimizing assembler.

Rationale

Currently, most architectures attempt to hide this increasing disparity with complex multilevel caches using complex dynamic replacement policies. While in the past such a cache hierarchy has been reasonably successful in hiding memory latency, such designs incur a number of ever more serious problems as the disparity between CPU speed and memory latency grows. In particular, dynamic mechanisms used to manages caches create extremely variable memory access latency, which impedes efficient code scheduling, while making inefficient use of the large portion of modern chips devoted to on-board cache memory, and the replacement mechanisms themselves. The design of LARs has been chiefly motivated by this observation that the classical Von Neumann execution model is ill-suited

to a world where random memory access is so exorbitantly expensive. While a LARs-based design breaks many of the assumptions found in conventional Von Neumann or Harvard style load-store architectures, it closely resembles the designs of early Von Neumann machines [2], where memory accesses had to be scheduled around delay lines, mechanically rotating drums, or other devices with non-constant access time, perhaps more so than any other current architecture.

History

Research into LARs began with the master's thesis of Krishna Melarkode [3], in 2004, and has since developed in scope and complexity, to include a hardware verification model, software development toolchain, and a considerable amount of theoretical work into the unexpectedly complex implications of such a design. It is worthwhile to take a moment to directly consider the two technologies which inspired the LARs design; CRegs, which implemented a subset of the design features, and the SWAR (SIMD Within A Register) designs. It is also noteworthy that there are a number of developmental and technological parallels between LARs and EPIC/VLIW [4] designs. However, we we firmly believe the LARs solution is more general and elegant than it's VLIW cousins, in that it completely removes many of the memory hierarchy problems that VLIW designs merely help to hide. Also like VLIW, LARs-based designs require considerable intervention by the software toolchain to package code into suitable parallel blocks.

SWAR In the 1990s, a variety of architectural features for operating on multiple data objects within a single machine word, collectively known as SWAR (SIMD Within A Register) technologies, were added to a number of popular architectures. These designs included Intel's MMX [10, 11] and AMD's 3DNow! [12] for x86, and a number of earlier extensions for PA-RISC [7–9]. These changes have been met with widespread adoption, and can be found in almost all modern architectures. Many compilers now default to using SWAR operations where possible, for example, IA32 GCC defaults to SSE for floating point arithmetic. This success has come despite the fact that they significantly alter the programming model, because of the substantial performance gains afforded by utilizing SWAR technology. While the name does not immediately suggest it, these gains come chiefly from relief from small random accesses to memory, by grouping collections of similar small operations into single sequential reads [5]. In order to ease adoption of SWAR, many of the implementations provide both parallel and scalar instructions. The oxymoronic scalar SWAR instructions simply ignore all but a single field – the one in the lowest bitfield in the word.

Later developments in SWAR technology have focused on extending the size and variety of operations which can be performed in a single operation. Early implementations supported only small integers, enabled by simply selectively cutting the carry chain in the existing integer ALU. Later designs have expanded the variety of supported data types, such as packing two 32-bit floats into a single

64-bit operation in 3DNow![12], and extending the path widths to 128 bits in AltiVec and SSE. This expansion continues even today. For example Intel's now defunct Larrabee was slated to extend data path width to 512 bits [13].

CRegs Given that ever wider registers and datapaths, and SWAR-like processing of data within them, are clearly the way of the immediate future, the question becomes what most limits performance of very wide SWAR? We suggest the answer is the same one that inspired the invention of CRegs (Cache Registers) [15, 16] in the late 1980s.

Given enough registers, the compiler can ensure that nearly every object can be accessed from a register when it is needed. However, this is not true of ambiguously aliased objects. For example, if a[i] is loaded into one register and a[j] into another, should a change of the value in the register holding a[i] also change the value in the register holding a[j]? If a[i] and a[j] are ambiguously aliased, the compiler can't tell. Thus, the compiler is forced to generate code that flushes the new value of a[i] to memory and then re-loads the value of a[j] from memory. This problem is common in code working on arrays and/or pointers. One could argue that ambiguous aliases are not very common in most programs, but the flush/reload cost is high enough to make them a significant performance issue even when they are relatively rare. More importantly, using SWAR-like methods to operate on wide lines can be expected to dramatically increase the number of ambiguously aliased references. In the above example, even if a[i] and a[j] are not aliased, they may be in the same SWAR line in memory, and thus would need to be treated as ambiguously aliased references. In effect, this false sharing [17] forces many apparently unambiguous object references to become ambiguously aliased line references. CRegs solve the ambiguous alias flush/reloadIn effect, this "false sharing" [17] forces many apparently unambiguous object references to become ambiguously aliased line references. CRegs solve the ambiguous alias flush/reload problem by using hardware to associatively update aliased objects in registers. The mechanism proposed in the current paper, LARs, can be seen as fundamentally extending CRegs to work with wide lines rather than single objects in registers problem by using hardware to associatively update aliased objects in registers. The mechanism proposed in the current paper, LARs, can be seen as fundamentally extending CRegs to work with wide lines rather than single objects in registers.

Unfortunately, CRegs have a major impact on the programming model. Fundamentally, a CReg is a register that holds both a datum and the address from which it was fetched. If a CReg's datum value is changed, then the datum values in any other CRegs whose address fields match also have their datum changed to match. This changes the programmer model partly because there is now an address field that can be operated upon, but more significantly because the memory access model is entirely different. For example, by associating a dirty bit with each CReg, store instructions become entirely unnecessary. Despite the idea being twenty years old, the strangeness of requiring a new instruction set design has prevented CRegs from being widely applied. The only commercial

implementation to date is the IA64 Advanced Load mechanism [18], which does not achieve the full benefit because it uses its CReg-like mechanism only as a filter for memory references rather than as a replacement for conventional registers and cache.

The associativity concepts in CRegs were also discussed for instruction memory access in an attempt to eliminate the instruction fetch cycle. A short block of instructions loaded into an Instruction CReg could be associatively copied into the instruction register when needed. The concept is somewhat similar to the trace cache and loop recognition mechanisms introduced by Intel in the P4 and i7 [19, 20], but allows more direct compiler control and potentially higher efficiency.

Architectural Overview

Without introducing a particular instruction set or implementation, any LARs based architecture would have a number of distinctive properties, which stem directly from the nature of LARs. It is these properties which make a LARs-based design attractive for environments where random memory accesses exhibit high latency, and these same properties which substantially affect the programming model. The following sections discuss those structures required for a general-purpose architecture based on LARs for both the instruction and data path, which differ from a conventional processor design.

The Line Associative Register

In general, a LAR is a very wide register, which contains a number of machine words worth of data, and several fields worth of metadata. At minimum, this metadata will contain an address field, which contains specific information to locate the source from which the LAR was loaded in memory, and to reference to individual fields of the LAR. A processor can be designed to utilize LARs for both instructions and data. However, instructions and data exhibit different access patterns, and therefore benefit from, or even require, slightly different features to be handled efficiently. Therefore, just as most modern architectures employ separate instruction and data caches, two slightly different structures are used for the two paths; Instruction LARS (ILARs) for instructions and Data LARs (DLARs) for data. The following sections provide details for both Data LARs and Instruction LARS.

DLARs Data LARs are tagged wide registers that replace the functionality of both registers and cache in the data path. Each LAR holds enough data for very wide SWAR operation; scalar operations on any field also are supported. The tags include address and status bits (e.g., "dirty") similar to what would be found in a traditional cache entry, but additionally specify the size and type of data (signed, unsigned, float, etc.) in the LAR and a specific "current field" address within the LAR, adequate to address each field with the object size set

to the minimum supported value for a given implementation. As in a CReg, the use of a dirty bit and address field to detect changed and aliased values allow for the elimination of a large portion of unnecessary memory accesses. Historically, a number of earlier architectures have employed type tagging in memory. Intel's iAPX432 [21] was perhaps the most extreme in this regard; the entire memory of an iAPX system was type tagged with an object system, and accessed through a privilege controlled segment-offset mechanism. Unfortunately, employing type tagging in memory incurs expensive penalties, in the form of bandwidth overhead to move tags to and from main memory, and awkward mechanisms to manipulate the tags. In contrast, tagging data copied into a LAR with quite a lot of information is cheap, in that it fetches no additional data from main memory, and easy. By tagging each LAR with type information set at load time, it becomes possible to simplify the instruction set and increase architectural regularity, while simultaneously making type-conversion instructions unnecessary. It also is possible to take advantage of the fact that address field of the LAR is essentially a typed pointer, so address operations can be scaled by object size much as the C programming language does for pointer arithmetic. Because of this capability, scalar operations on objects within a LAR have the full flexibility to access any field, not just the one in the lowest bits. A generic diagram of the layout for a DLAR bank is provided below.

Table 1. Data LAR Structure

LAR NR	Data	Address		WDSZ	TYP	D
		TAG	OFFSET			
	2^m blocks	$2^n - 2^m$ bits	m Bits			1 bit
D0						
D1						
D2						
...	...	...	...	...	...	...
Dxx						

ILARs ILARs support only a fixed field size (one instruction), and have only address and data fields, as the others would be superfluous. However, ILARs' complication comes from their effect on the execution model. Because instructions are always fetched in blocks determined at compile time, and are addressed by block and offset within an ILAR, it is trivial to apply a compression algorithm on ILAR-sized blocks of instructions, or multiples thereof. Instruction blocks, therefore, must be explicitly fetched and decompressed into instruction LARs before they become accessible, as individual instructions may literally have no address in memory. This fetch cycle is only coupled to execution by the explicit issue of fetch instructions, making an entirely LARs-based architecture not technically a Von Neumann design.

Instruction Set

The typed, SWAR capable design allows for an extremely rich set of functions from a small instruction set; the same instruction encoding can be used for analogous operations on all supported data types, and executed on the proper functional unit based on the type field of the data LAR being acted upon. Likewise, scalar and vector operations can simply differ by a switch. Scalar operations do require extra fields, as the offsets must be specified . The only instruction which must be varied for each individual data type is the load for moving data into DLARs, as this is the mechanism by which the type field is set at load time.

Calling/Return Behavior

One of the more substantial challenges presented by an architecture which has no permanent linear address space is the problem of resolving return addresses. Return addresses, in the form of a block and offset, must be resolved at the time the call is made, so that the block containing the return point can be loaded into an ILAR and decompressed before the return completes. To this end function calls and returns are managed by a hardware stack mechanism, specifically designed to be implementable with a multitude of underlying mechanisms, which offer different trade offs between performance and hardware complexity. As examples, return stack can be a FIFO as the name suggests, managed by a more complicated replacement policy, or, to trade performance for hardware complexity, simply nonexistent. To maintain the stack, call instructions look very much like a normal return instruction; it prefetches the destination block, into a stack-like arrangement of ILARS which are not expressly addressable. On return it is then possible to associatively update the ILAR line being returned to. The pre-fetch area (stack) will be invisible to the user, and can, at the discretion of the implementer, buffer the entire ILAR, the uncompressed headers and compressed instruction block, only the header, or, again simply do nothing. If the ILAR being returned to is still cached in the stack, one of the addressable ILARs can be associatively updated from the "invisible" one in the stack, providing a clean, fast return. If the ILAR data has spilled from the stack, or was never actually stored, the information which has been saved can be used to pre-fetch the ILAR into which the return points. If no information is available, there will be a considerable stall while the suitable block is loaded from main memory, which is comparable to the worst case on more conventional designs.

Current Implementation

Despite being a work in progress since 2004, research into LARs is still very much an ongoing project. A number of steps have been completed toward better understanding of the demands and ramifications of LARs-based designs. On one front, several students have implemented a simple LARs-based design in the Verilog hardware description language, and in doing so have demonstrated that the required circuit size and complexity is feasible for a modern processor design.

In parallel, there has been theoretical work into the ramifications of LARs on the programming model, which is now reaching a practical stage, in the form of a rudimentary compiler currently under development.

Hardware Verification Model The current reduced-size synthesizable test core, which uses 8 64bit data LARs and 6 1024Bit instruction LARs, synthesizes to an FPGA in approximately 50,000 4-input LUTs, with a maximum clock frequency of 60MHz, prior to optimization. This large size is to be expected, since the LARs effectively encompass the portion of a modern chip devoted to both the register file and to on-die caches.

Software Currently, a rudimentary compiler is being built within the LLVM infrastructure [22] to target an internal "straw-man" LARS-based architecture. While this compiler is still incomplete, it appears that compiling for LARs will only require changes comparable to those required for any porting effort. However, a relativity complicated assembler is necessary to handle packaging blocks of code and inter-block scheduling in the output language. The LLVM infrastructure was chosen as a starting point for several reasons. First, using an existing compiler infrastructure avoids the cumbersome overhead involved in writing common components not affected by the architecture; such a language fronted, and tools for manipulating the intermediate language. Secondly, the LLVM code base is considerably more accessible than many of it's well-established predecessors. The most important impetus behind the decision to use an existing infrastructure is however that doing so demonstrates that conventional tools and code can be adapted to LARs-based designs, despite the wildly different . The use of a standard fronted in particular will demonstrate that standard code, can be built for a LARs-based target without any programmer-visible changes.

Summary

Line Associative Registers (LARs) are the basis for a new class of processor architectures in which memory accesses are minimized by explicitly managing wide lines of instructions and data in processor registers. LARs fill the role of both registers and caches in a traditional memory hierarchy, bringing many of the advantages of each while avoiding their more egregious faults. An architecture based on LARs will present an extremely unusual execution model, which is better suited to the severe latency between processor and main memory present in modern systems than more conventional designs. While this execution model will break many assumptions from a conventional Von Neumann architecture, it should be possible to build unmodified code in existing, popular languages with no more intervention than is required to port between more traditional architectures.

References

1. Brent, G. A. "Using program structure to achieve prefetching for cache memories", University of Illinois, Department of Computer Science, Urbana, IL, USA, 1987
2. M. R. Williams, *A History Of Computing Technology*, Second Edition, IEEE Computer Society Press, Los Alamitos, CA, 1997, pp.296-380
3. Krishna Melarkode. Line Associative Registers. Master's Thesis, University of Kentucky, October 2004
4. J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, "Parallel processing: a smart compiler and a dumb machine," in SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction, vol. 19, no. 6. New York, NY, USA: ACM Press, June 1984, pp. 37-47. [Online]. Available: http://dx.doi.org/10.1145/502874.502878
5. Randall Fisher. General- purpose SIMD Within A Register: Parallel processing on consumer engineering. Purdue University, Ph.D. Thesis Proposal, November 1997.
6. Randall J. Fisher and Henry G. Dietz. The Scc Compiler: SWARing at MMX and 3DNOW! In Larry Carter and Jeanne Ferrante, editors, Proceedings of the 12 th International Workshop on Languages and Compilers for Parallel Computing, La Jolla, California, August 1999. Springer- Verlag.
7. Ruby B. Lee. Subword Parallelism with MAX-2. IEEE Micro, 16(4):51-59, August 1996.
8. Ruby Lee and Jerry Huck. 64-bit and multimedia extensions for the PA-RISC 2.0 architecture. In Proceedings of Compcon '96, Technologies for the Information Superhighway. Digest of Papers, 152- 160, Los Alamitos, California, 1996. IEEE Computer Society Press.
9. Ruby B. Lee, Multimedia Extensions for General-Purpose Processors, Proc. IEEE Workshop Signal Processing Systems, pp. 9-23, Nov. 1997.
10. Intel Corporation. MMX technology overview. Technical report, Intel Corporation, February 1997. Formally at http://developer.intel.com/drg/mmx/.
11. Intel Corporation. Intel Architecture MMX technology: Programmer's reference manual. Technical report, Intel Corporation, March 1996.
12. Advanced Micro Devices, Inc. AMD Extensions to the 3DNow! and MMX Instruction Sets Manual
13. Larry Seiler, Doug Carmean, Eric Springle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan Larrabee: A many-core x86 Architecture for visual computing.
14. Deepu Talla, Lizy Kurian John, Doug Burger, Bottlenecks in Multimedia Processing with SIMD Style Extensions and Architectural Enhancements
15. H. Dietz, C. H. Chi, CRegs: a new kind of memory for referencing arrays and pointers, Supercomputing 88, pp.360-367, Jan 1988.
16. Peter Dahl, Matthew O'Keefe, Reducing memory traffic with CRegs, Proceedings of the 27th annual international symposium on Micro-architecture, pp 100-104, Nov 1994.
17. J. Torrellas, H.S. Lam, J.L. Hennessy, False Sharing and Spatial Locality in Multiprocessor Caches, June 1994 (vol.43 no. 6) pp 651-663
18. C.McNairy, D.Soltis, "Itanium 2 Processor Microarchitecture", IEEE Micro Vol. 23 Issue 2, pp.44- 55, March 2003.
19. Eric Rotenberg, Steve Bennett, Jim Smith, Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching, In Proceedings of the 29th International Symposium on Microarchitecture, 1996, 2434

20. G. Hinton, D. Sagar, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the Pentium 4 processor." Intel Technology J. Q1 2001.
21. "Intel iAPX432 General Data Processor Architecture Reference Manual", 171860-001
22. Chris Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization", Masters Thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.