You Can't Always Get What You Want

If you try sometimes you might find you get what you need – *but maybe not*. That's the problem with **caches**. As modern processors have sprouted increasingly complex memory hierarchies, they still have not solved the fundamental problem of ensuring that instructions and data are always ready when they are needed.

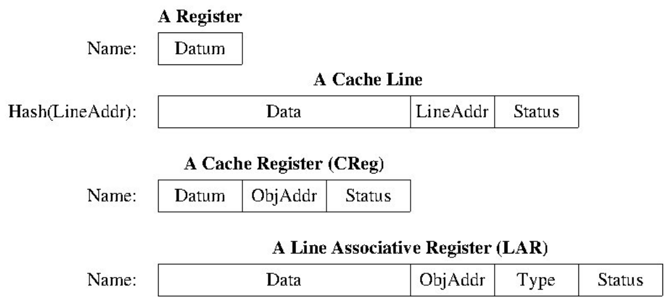
A value kept in a **register** is there when you need it. However, there are lots of memory objects conventional machines do not keep in registers. Instructions are not kept in registers, although there really isn't a good reason why not. There is a widely held belief that registers don't help with "spatial locality" – but that is true only if we assume that a register is just one object wide.

Of course, ambiguously aliased data cannot be kept in registers. For example, if a code refers to `a[i]` and `a[j]` a number of times, there are three possible circumstances:

Alias Analysis	Register Assignment
compiler knows i==j	share one register
compiler knows i!=j	two separate registers
compiler doesn't know	which of the above?

In fact, ambiguously aliased object references are the only reason a processor needs cache: without cache, the store and reload would become slow references to main memory. Even so, it is possible that the cache would be ineffective, because references to other objects might have had addresses hashing to the same cache line slot, thus evicting the desired object from cache. However, everything needed to resolve the ambiguity was in the processor – why not simply modify the register file to automatically update aliased objects in other registers?

CRegs & LARs. Our SC'88 paper, *CRegs: A New Kind of Memory for Referencing Arrays and Pointers*, introduced CACHE-REGISTERS to solve the ambiguous alias flushing problem by *associatively updating* registers whose address fields match. However, data CREGs don't make use of spatial locality – our work on LARs (LINE ASSOCIATIVE REGISTERS) over the past few years combines CREGs with SWAR (SIMD WITHIN A REGISTER) to take full advantage of spatial locality. The basic hardware cell structures are:



An example using Data LARs. In the following C code, suppose `i, j and k` are ambiguously aliased. Because each LAR records the current object position within its data field, the LAR code reduces memory loads, fewer if any of the lines have the same base addresses.. The architecture also provides implicit lazy stores. Each LAR is type tagged, and type information need not be encoded within arithmetic instructions. The LAR code as written is scalar; however, if the data are properly aligned, replacing `ADDS` and `ANDS` with `ADDV` and `ANDV` would perform the SWAR parallel operations on a "line" of data at a time.

```
C Code          Conventional RISC    LARs
nasty(int*i,    LW $t1,j(0)          LOADSW d1,d0,d0,j
int*j, int*k)   LW $t2,0($t1)        LOADSW d2,d0,d1,0
{               LW $t3,k(0)          LOADSW d3,d0,d0,k
*i=*j+*k;       LW $t4,0($t3)        LOADSW d4,d0,d3,0
*k=*i&*k;       LW $t5,i(0)          LOADSW d5,d0,d0,i
}               ADD $t6,$t2,$t4      LOADd d6,d0,d5,0
                SW $t6,0($t5)        ADDS d6,d2,d4
                LW $t7,0($t5)        ANDS d4,d6,d4
                LW $t8,0($t3)
                AND $t9,$t7,$t8
                SW $t9,0($t3)
```

Instruction LARs. Instruction LARs remove the instruction fetch process from the execution of each instruction, replacing it with separate, explicit, instructions that load of compressed blocks of instructions. If a load requests a block that is already in another instruction LAR, the decoded instruction block is logically copied without any memory activity. Control flow targets are specified using an instruction offset within an instruction LAR, rather than by comparatively lengthy memory addresses. The overall result is a smaller memory footprint, improved utilization of memory bandwidth, and complete freedom from misses during instruction processing.

Status. Krishna Melarkode's 2004 M.S. Thesis, *Line Associative Registers*, first introduced the concepts of LAR as an extension of CREGs. Since SC2011, significant progress has been made in development of a complete compiler for a C dialect and a Verilog hardware target. The current design uses 2,048-bit lines in both the instruction and data pipelines. It supports a wide variety of scalar and vector operations, with implicit type conversions handled for 8, 16, 32, and 64-bit object sizes. Watch **`Aggregate.Org/LAR`** for more information.

This document should be cited as:

```
@techreport{sc12lar,
author={Paul Eberhart, Matt Sparks and Henry Dietz},
title={You Can't Always Get What You Want},
institution={University of Kentucky},
address={http://aggregate.org/WHITE/sc12lar.pdf},
month={Nov}, year={2012}}
```

Aggregate.Org
UNBRIDLED COMPUTING

SC12
Salt Lake City, Utah