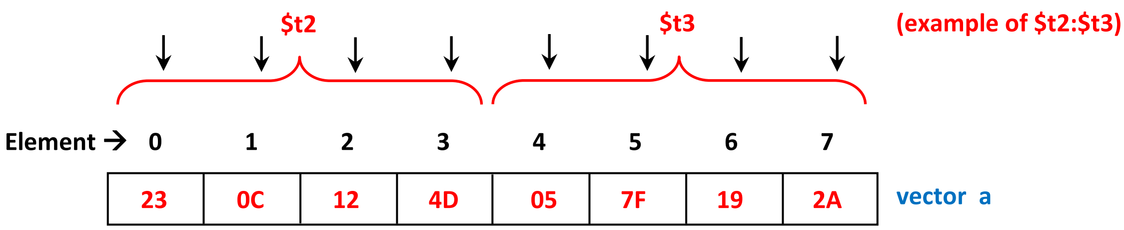
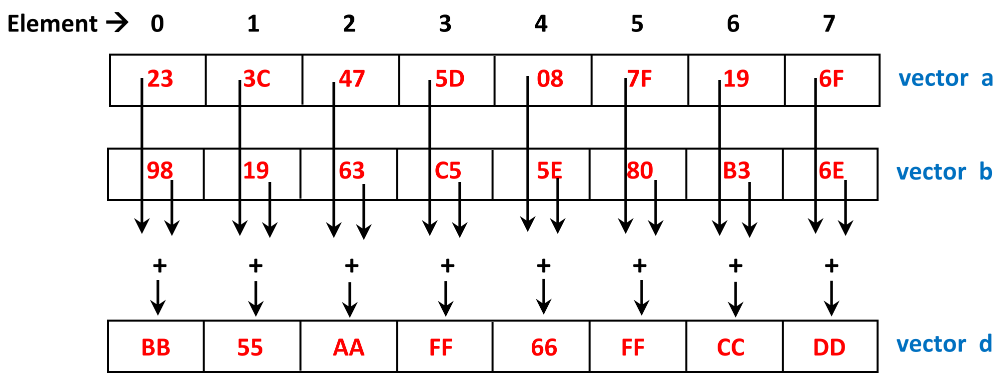
Fall 2018 Semester Project – SIMD Enhanced MIPS Instructions



Example 1: *Vector Add Saturated (unsigned)*

To add 8 bytes to another 8 bytes in one instruction we can use
vec_addsu d, a, b

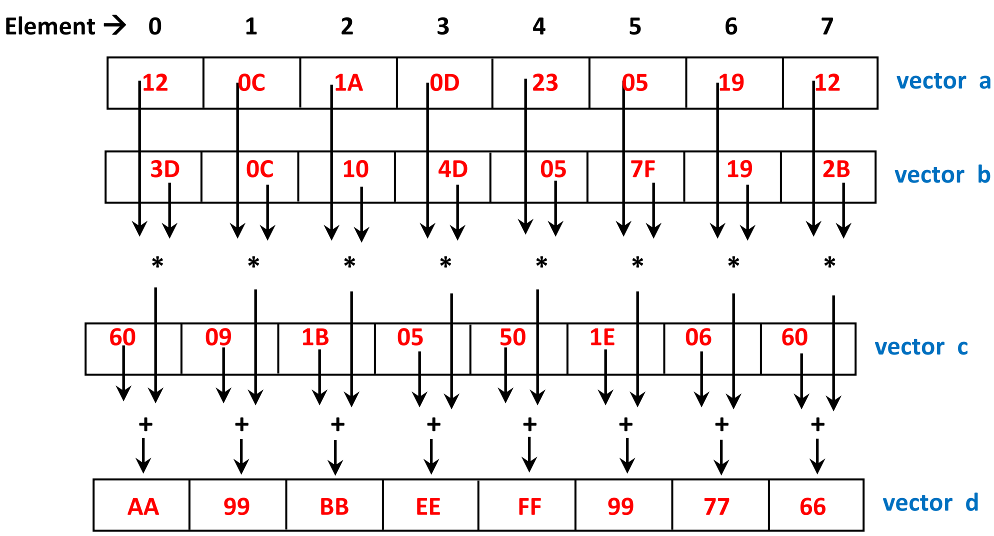
where each element of **a** is added to the corresponding element of **b**. The unsigned-integer (no-wrap) is placed into the corresponding element of **d**.



vec_addsu is the AltiVec analog of the add unsigned bytes (no wrap) available in the PowerPC scalar instruction set.

Example 2: *Vector Multiply and Add*

To multiply the vector elements in **a** by the vector elements in **b** and then add the intermediate result to the vector elements in **c**, storing each resulting element in vector **d**, in one instruction and in one rounding, we can use **vec_madd d, a, b, c.** Note that the sum of the intermediate product with elements in vector **c** are "truncated" for a half-length results placed into vector **d**.

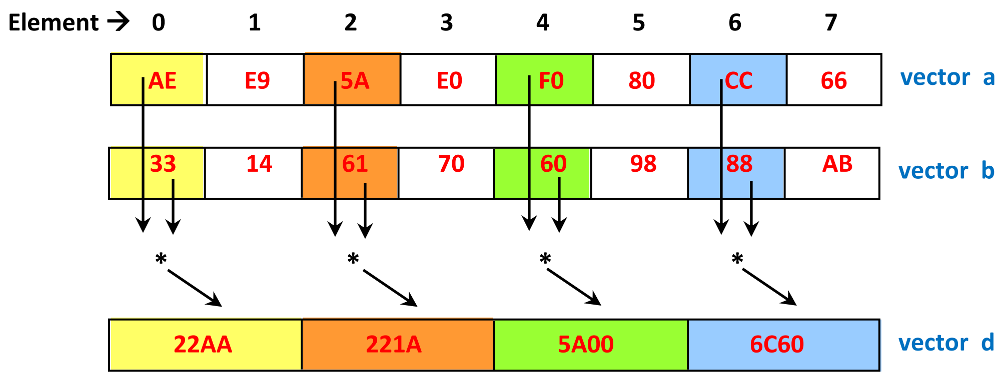


vec_madd is the AltiVec analog of the multiply-add fused available in the PowerPC scalar instruction set. The AltiVec unit on the PowerPC goes beyond the common general purpose microprocessor instructions.

Example #3: *Vector Multiply Even Integer*

vec_mule d, a, b

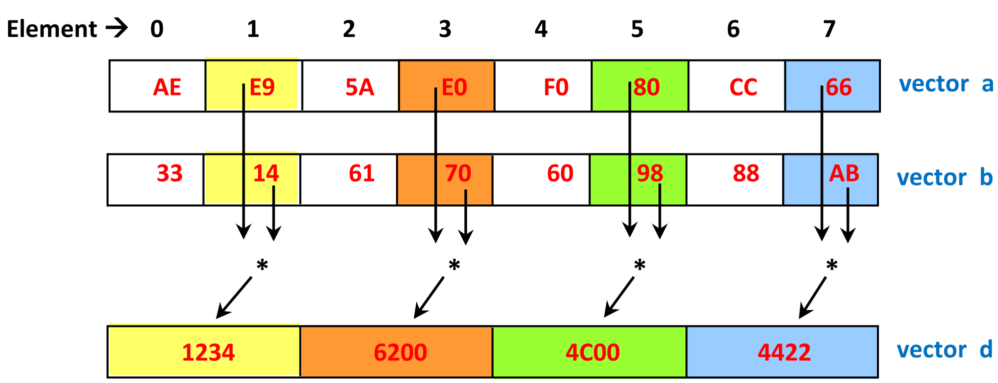
Each element of vector **d** is the full-length (16-bit) product of the corresponding high (i.e. even) half-width elements of vector **a** and vector **b**.



Example #4: *Vector Multiply Odd Integer*

vec_mulo d, a, b

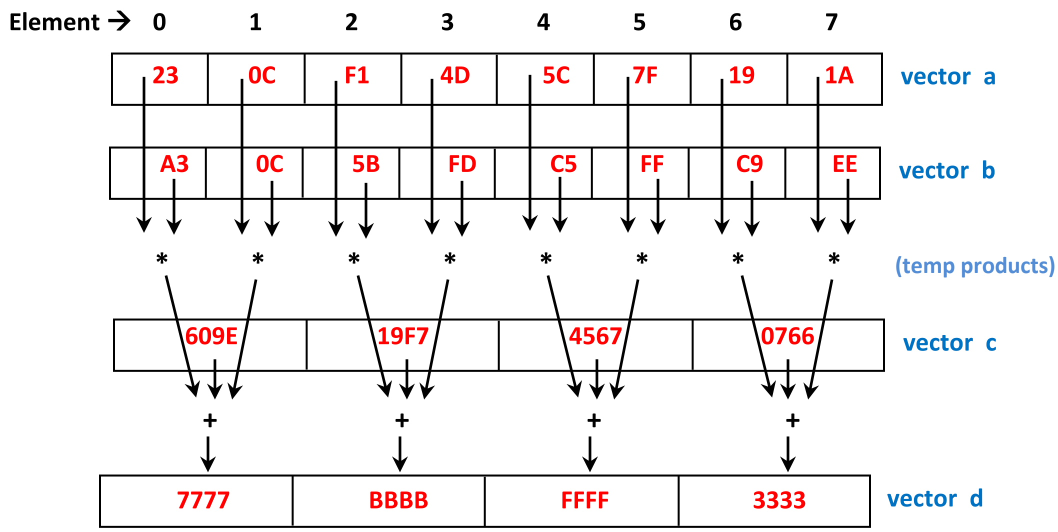
Each element of vector **d** is the full-length (16-bit product of the corresponding low (i.e. odd) half-width elements of vector **a** and vector **b**.



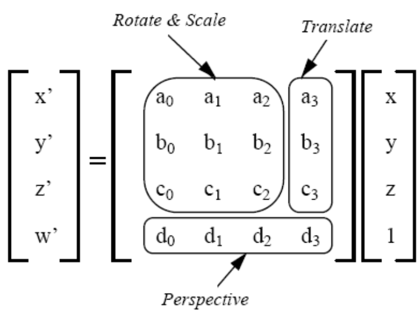
Example #5: *Vector Multiply Sum Saturated*

vec_msums d, a, b, c

Each element of vector **d** is the 16-bit sum of the corresponding elements of vector **c** and the 16-bit "temp" products of the 8-bit elements of vector **a** and vector **b** which overlap the positions of that element in **c**. The sum is performed with 16-bit saturating addition (no-wrap).



Exciting new 3D games are coming to market every day. Typically, computations that manipulate 3D objects are based on 4-by-4 matrices that are multiplied with four element vectors many times. The vector has the X,Y, Z and perspective corrective information for each pixel. The 4-by-4 matrix is used to rotate, scale, translate and update the perspective corrective information for each pixel. This 4-by-4 matrix is applied to many vectors.

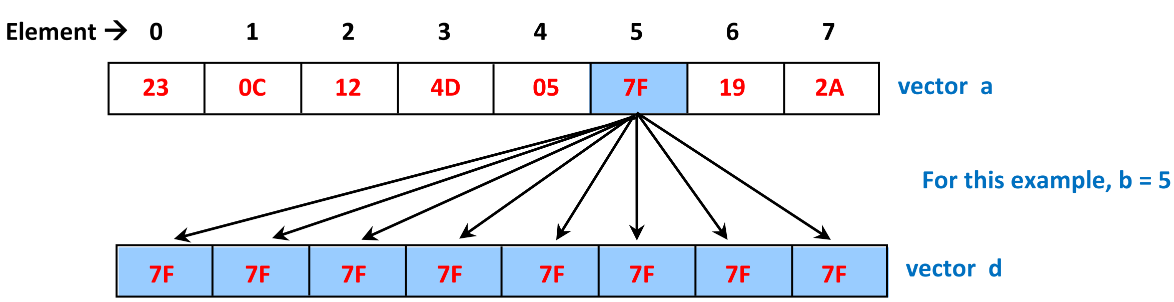


$$x' = a_0x + a_1y + a_2z + a_3$$

Example 6: *Vector Splat*

vec_splat d, a, b

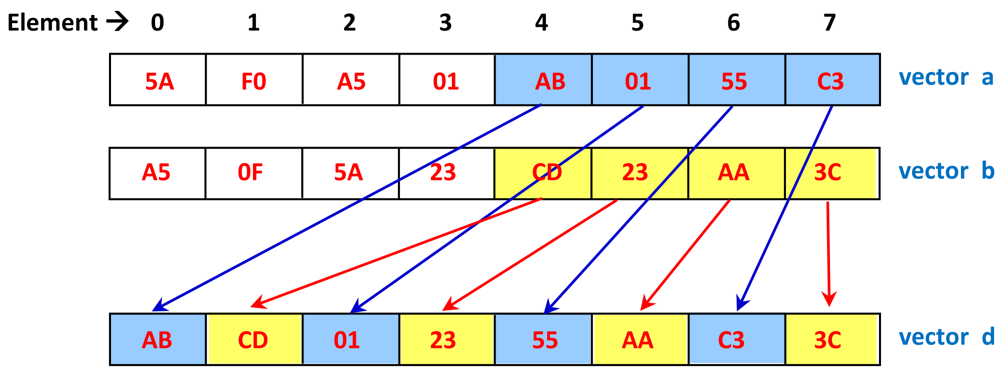
The "splat" instruction is used to copy any element from one vector into all of the elements of another vector as shown in the diagram below. Each element of the result vector **d** is component **b** of vector **a**.



Example #7: *Vector Merge Low*

vec_mergel d, a, b

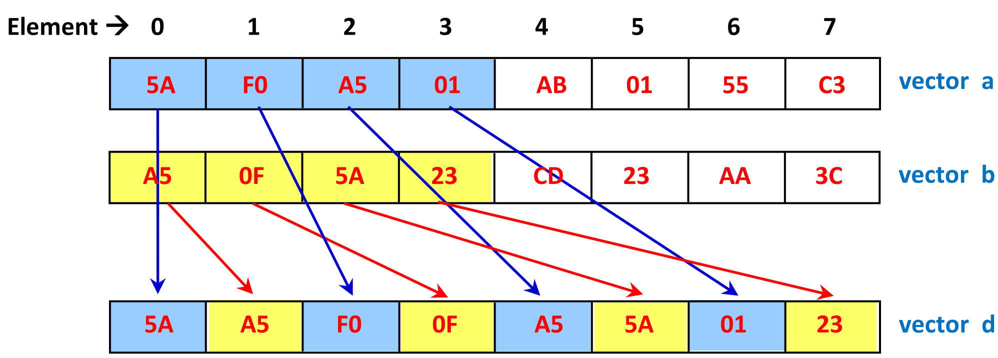
The even elements of the result vector **d** are obtained left-to-right from the low elements of vector **a**. The odd elements of the result are obtained left-to-right from the low elements of vector **b**.



Example #8: *Vector Merge High*

vec_mergeh d, a, b

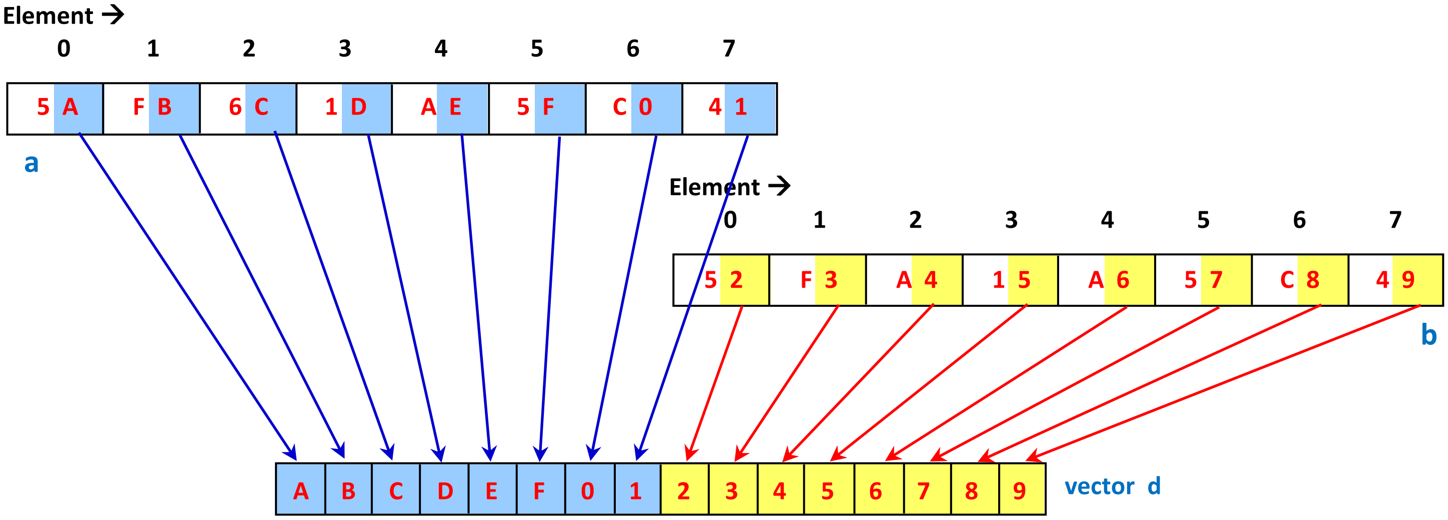
The even elements of the result vector **d** are obtained left-to-right from the high elements of vector **a**. The odd elements of the result are obtained left-to-right from the high elements of vector **b**.



Example #9: *Vector Pack*

vec_pack d, a, b

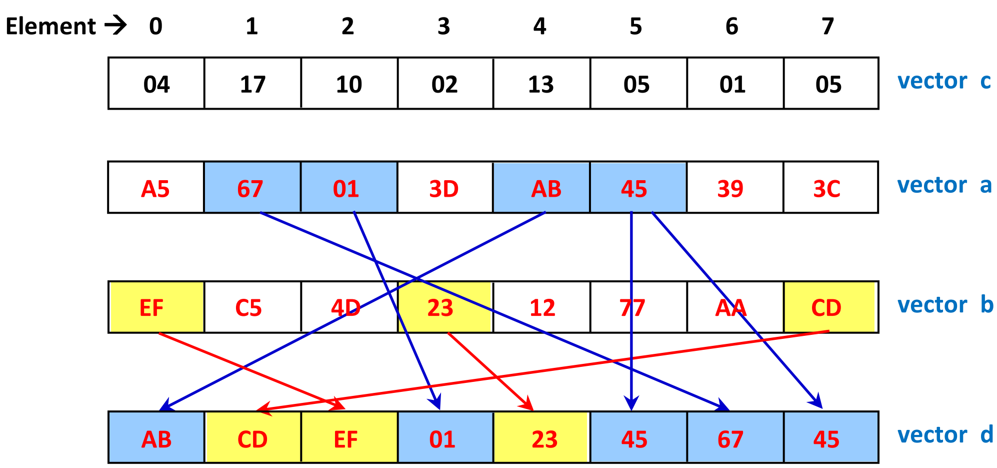
Each high element of the result vector **d** is the truncation of the corresponding wider element of vector **a**. Each low element of the result is the truncation of the corresponding wider element of vector **b**.



Example 10: *Vector Permute*

vec_perm d, a, b, c

The "permute" instruction fills the result vector **d** with elements from either vector **a** or vector **b**, depending upon the “element specifier” in vector **c**. The vector elements can be specified in any order.

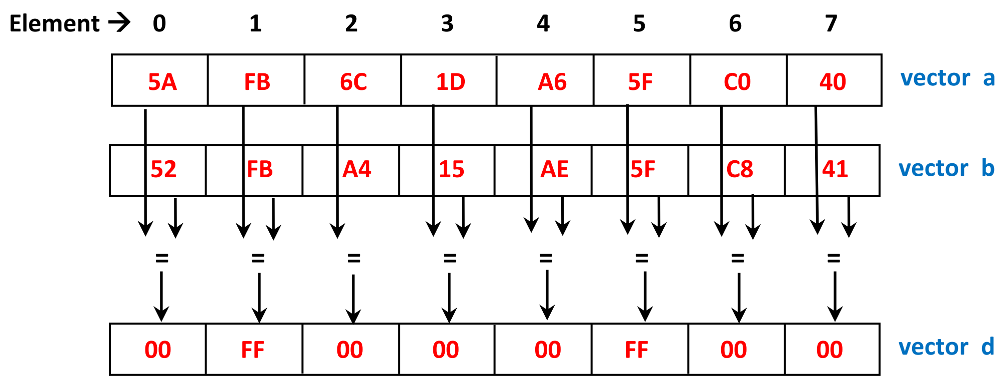


vec_perm uses vector c as a sophisticated mask and assigns corresponding values of the operands **a** and **b** to the **d** vector. For example, $element_4$ of **a** is mapped to $element_0$ of the **d** and $element_7$ of **b** is mapped into $element_1$ of **d**. Thus, each “element specifier” in vector **c** has two components: the most-significant-half specifies an element from vectors **a** or **b** (0=a, 1=b); the least-significant-half specifies which element within the selected vectors (0..7).

Example #11: *Vector Compare Equal-To*

vec_cmpeq d, a, b

Each element of the result vector **d** is TRUE (all bits = 1) if the corresponding element of vector **a** is equal to the corresponding element of vector **b**. Otherwise the element of result is FALSE (all bits = 0).



Example #12: *Vector Compare Less-Than (unsigned)*

vec_cmpltu d, a, b

Each element of the result vector **d** is TRUE (all bits = 1) if the corresponding element of vector **a** is less than the corresponding element of vector **b**. Otherwise the element of result is FALSE (all bits = 0).