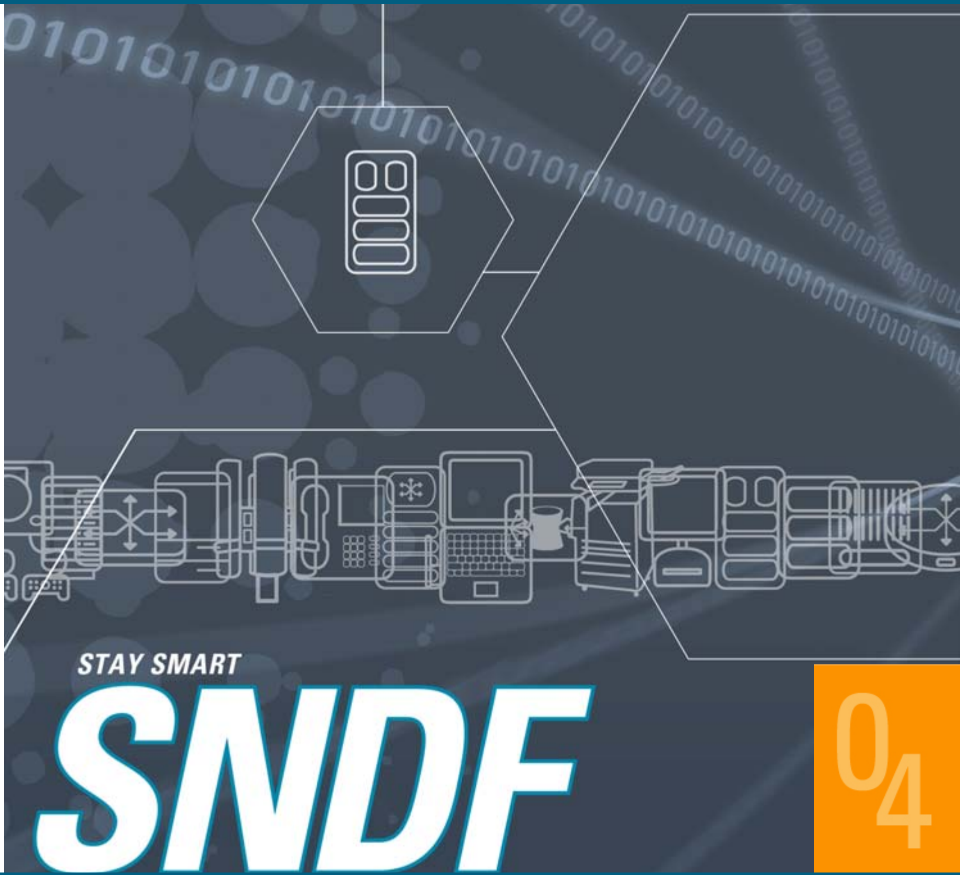
SMART NETWORKS
DEVELOPER FORUM
DALLAS 2004

April 26–29, 2004 • Hyatt Regency Dallas
www.motorola.com/sndf

H1119 - Introduction to AltiVec - Ten easy ways to Vectorize your code

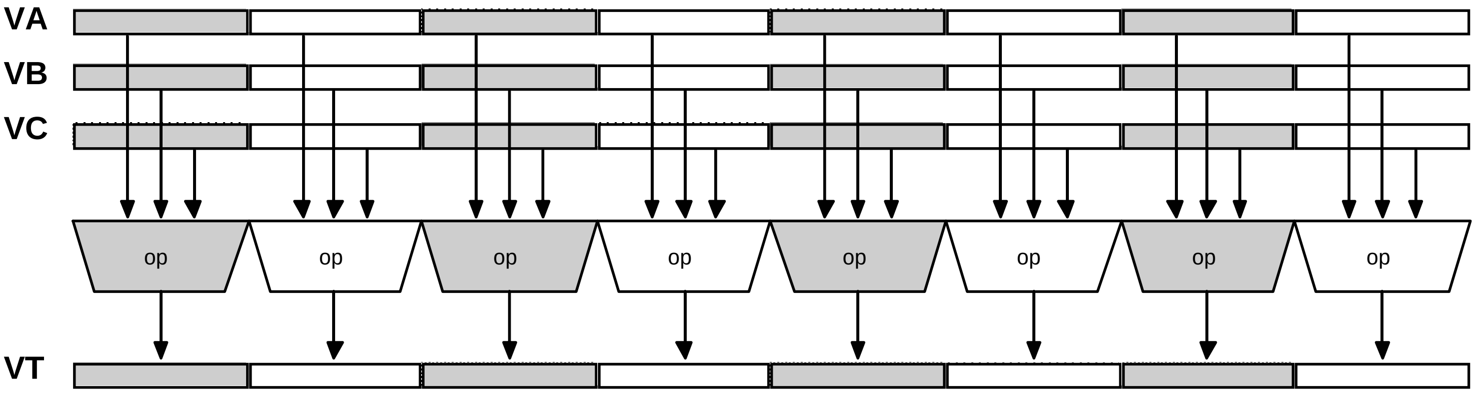


Sergei Larin
Sr. SW Engineer
CPD Applications Engineering

Launched by Motorola
freescale™
semiconductor

What is a Vector Architecture?

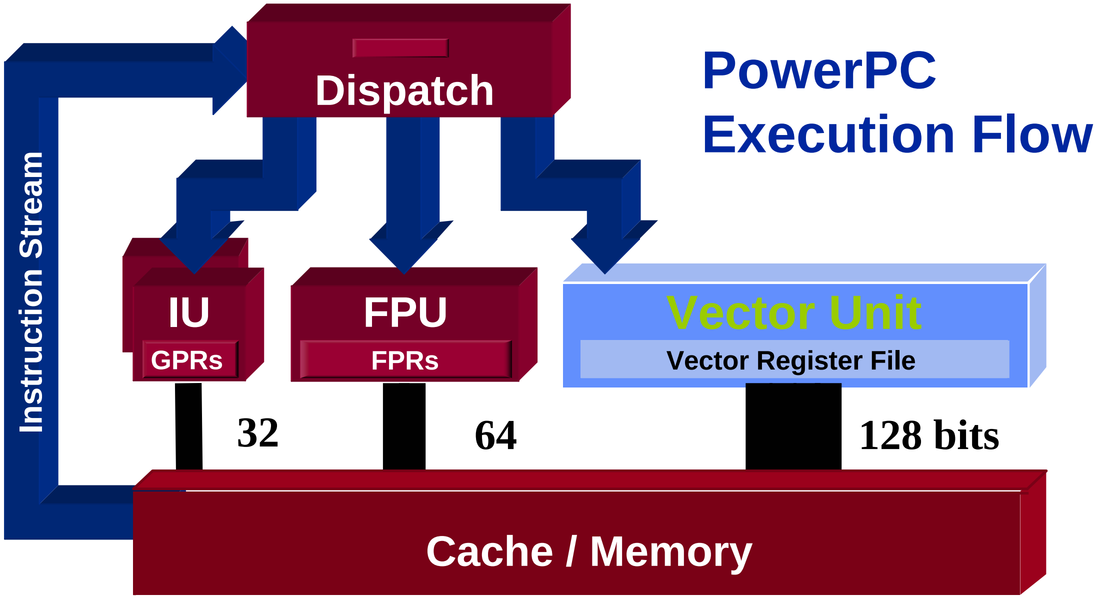
- A *vector architecture* allows the simultaneous processing of ***multiple data items*** in parallel
- Operations are performed on multiple data elements by a ***single instruction***
  - Referred to as Single Instruction Multiple Data (*SIMD*) parallel processing



What is AltiVec?

- **SIMD extension to PowerPC Architecture**
  - … no tradeoffs … just additions
- **Provides a high-performance RISC microprocessor with DSP-like compute power**
  - Allows highly parallel operations for the simultaneous execution of up to *16 operations* in a single clock cycle
- **Offers a *programmable solution* for controller and signal processing functions**
  - …which can easily migrate via software upgrades to follow changing standards and customer requirements

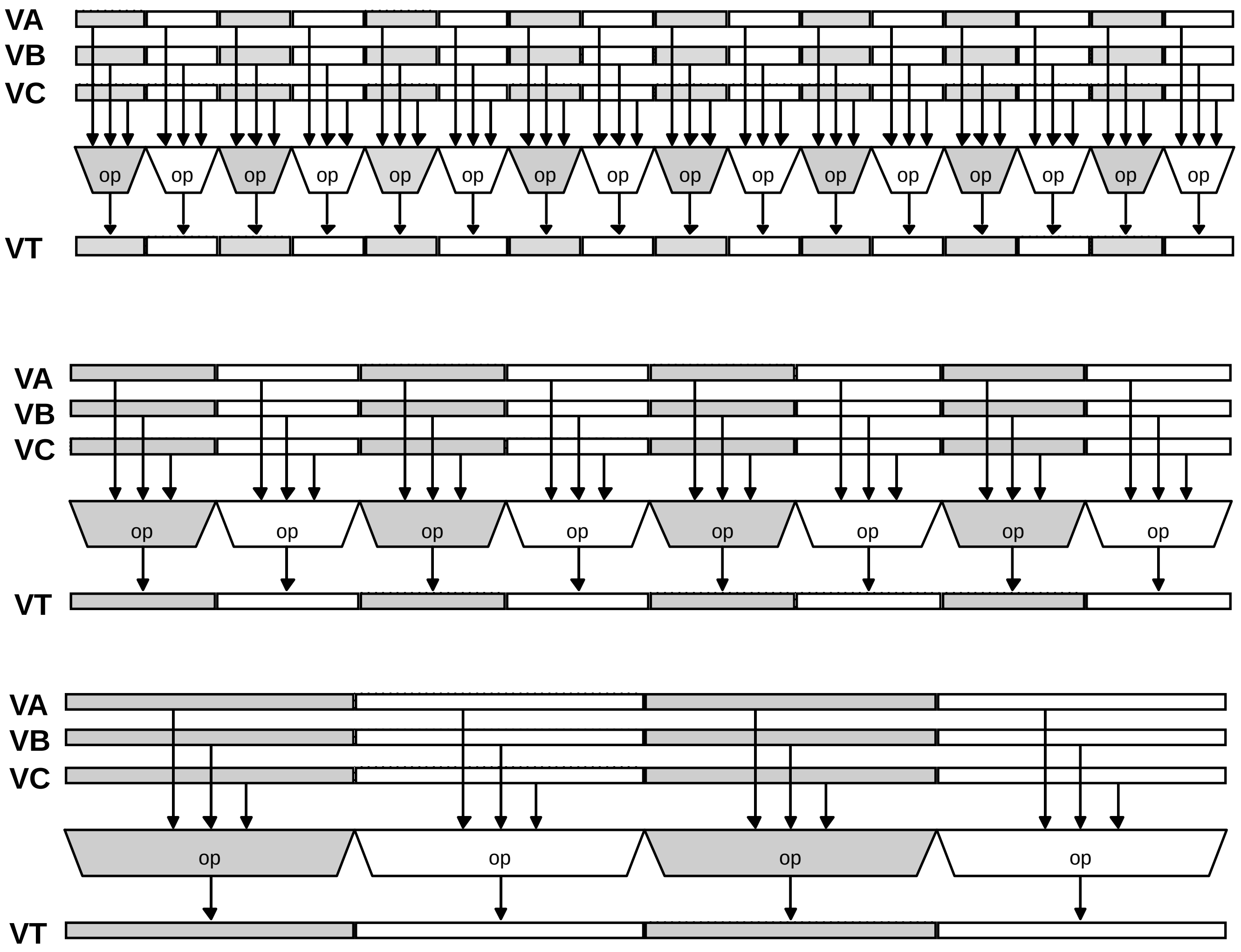
AltiVec’s Vector Execution Unit



PowerPC Execution Flow

- Concurrent with PowerPC integer and floating-point units
- Separate, dedicated 32 128-bit vector registers
- Approximately 11% of the silicon area
- No penalty for mixing integer, floating point and AltiVec operations

SIMD Intra-element Instructions



AltiVec Instruction Set Features

- **162 new instructions added to the PowerPC ISA**
  - Intra and inter-element *arithmetic* instructions
  - Intra and inter-element *conditional* instructions
  - Powerful Permute, Shift and Rotate, Splat, Pack/Unpack and Merge instructions

- **4-operand, non-destructive instructions**
  - Up to three source operands and a single destination operand
  - Supports advanced “multiply-add/sum” and permute primitives

- **All instructions fully pipelined with single-cycle throughput**
  - Simple ops: 1 cycle latency
  - Compound ops: 3-4 cycle latency
  - No restriction on issue with scalar instructions

Enabling AltiVec in your applications

- **Let Compiler do the job**
  - Currently there are no practical *Autovectorizers* out-there
  - It is a guess how much performance will be actually extracted
  - and they still require certain expertise to work with
- **Using C intrinsic**
  - There is a standardized list of *intrinsics* supported in all PowerPC enabled compilers
  - Can actually guarantee good level of control, up to the level of register assignment
- **Using Assembly Language programming**
  - The most effective, and the most laborious way
  - Can provide 100% performance extraction

The ways…

Method 0 – Where to start

- **There are two possible starting points:**
  - Mathematical description of the algorithm
  - Existing C code written for serial execution
- **Designing algorithms in Vector form from scratch is the best approach…**
- **… but the practice shows that much more often it is not the case**
  - Most of the time starting point is C code written for “serial” execution
- **C code is an excellent starting point**
  - It guarantees model execution
- **… but you should remember that vectorization is not trivial**
  - Vectorized code often needs completely different approach
- **Finally set your goals clear**
  - Do you want the fastest code possible?
  - Do you want the most compact code possible?
  - Combination of both?

Method 1 – Beware of your Bottlenecks

- **One of the first things to realize is whether the algorithm is Compute or I/O bound**
  - If the code entirely bounded by **memory** performance it would not help to reduce the latency of computations
  - If the code is **computation** intensive, we probably want to reduce that latency and then revisit memory bandwidth
  - If the code is **control** intensive, it might not be the best candidate for vectorization
    - unless you can use predication and convert control dependency to data dependency
    - or you can use **MULTICHANNEL** processing
- **If memory is the bottleneck the question is –**
  - Is it streaming data case (system bus is the bottleneck)
  - … or is it cache resident scenario (same data processed more than once, and then flushed back to main memory)
- **If system bus is the bottleneck…**
  - Is it saturated?
    - The theoretical **60x** bus throughput is *640 Mbytes/sec @ 100MHz*
    - The theoretical **MPX** bus throughput is *800 Mbytes/sec @ 100MHz*
  - If you have already reached your bus capacity, and you are sure you are doing just the necessary work – what else can you do?

Method 1 – Beware of your Bottlenecks

- **If we are dealing with *cache resident data*, you should strongly consider using AltiVec**
  - **AltiVec has FOUR TIMES the bandwidth**
    - One vector load brings 128 bit of data
    - One FP load - 64 bit
    - Once scalar load only 32 bit
  - **Beware, that there is no direct "link" between GPRs and Vector Registers (VRs)**
  - **To "copy" a value between the two you need to store-load it**
    - This means that you do need to keep majority of computations in one place
  - **One exception from this rule is memcpy/memmove/memset kind of operation**
    - If you copy large amounts of data between two locations, it might be useful to use AltiVec just as a transport manager
- **If *computations* are the critical path…**
  - **Is there true data dependency between values being computed?**
  - **If there is not, you could be almost certain that AltiVec will deliver performance improvement**

Method 2 – Look at the loops

- The easiest way to reduce computation latency is to restructure loops
  - This is the main optimization performed by autovectorizers
- Loop Unrolling
  - Do multiple iterations in one pass

```
int res[n],a[n],b[n];

for (i=0; i<n; i++){
   res[i] = a[i] + b[i];
}
```

```
int res[n],a[n],b[n];

for (i=0; i<n/4; i+=4){
   res[i]     = a[i]     + b[i];
   res[i+1] = a[i+1] + b[i+1];
   res[i+2] = a[i+2] + b[i+2];
   res[i+3] = a[i+3] + b[i+3];
}
```

```
vector int vres[q_n],va[g_n],vb[q_n];

// q_n == n/4
for (i=0; i<q_n; i+=4){
   vres[i]     = vec_add(va[i],vb[i]);
}
```

- Change the amount of computations in the loop
  - Traditionally you might want to remove loop invariant computations from the loops and do several smaller loops as opposed to single large one…
  - But paradoxically it might sometimes HELP to move MORE computations into loops

Method 3 – Look at data dependency

- True Data dependency is often preventing vectorization
  - as well as some classical code optimizations
- But in some cases it could be eliminated
- Let us consider Dot Product of two Matrixes
  - X,Y vectors size N

$$\text{Dot\_Product}(x[n], y[n]) = \sum_{i=1}^{n} x[i] * y[i]$$

```
int DotProduct( int *X, int *Y, int length ){
          int temp = 0;

          // N Iterations
          for( int i = 0; i < length; i++) {
               temp = X[i]*Y[i] + temp;
          }

          return temp;
}
```

Method 3 – Look at data dependency

- Same function could be written in vector form
  - Note that v1 and v2 are size of N/4
  - so is the “length” == four times fewer iterations

```
int VectorDotProduct( vector int *v1, vector int *v2, int length ){
        vector int temp = (vector int) vec_splat_u32(0);
        int result;
        // Loop over the length of the vectors multiplying like terms and summing
        // Number of iterations is N/4
        for( int i = 0; i < length; i++)
                        temp = vec_madd( v1[i], v2[i], temp);
        // Still have four ints splat across a vector
        // Add across the vector
        temp = vec_add( temp, vec_sld( temp, temp, 4 ));     // Vector Shift Left Double
        temp = vec_add( temp, vec_sld( temp, temp, 8 ));
        vec_ste( temp, 0, &result );

        return result;
}
```

Method 3 – Look at data dependency

- But is this the best possible way of doing it?
  - vec_madd takes 4 cycles to complete…

```
int VectorDotProduct( vector int *v1, vector int *v2, int length ){
        vector int temp = (vector int) vec_splat_u32(0);
        int result;
        // Loop over the length of the vectors multiplying like terms and summing
        // Number of iterations is N/4
        for( int i = 0; i < length; i++)
                        temp = vec_madd( v1[i], v2[i], temp); // true data dependency
                                                              // only 1 madd every 4 cycles

        temp = vec_add( temp, vec_sld( temp, temp, 4 ));      // Vector Shift Left Double
        temp = vec_add( temp, vec_sld( temp, temp, 8 ));
        vec_ste( temp, 0, &result );

        return result;
}
```

Method 3 – Look at data dependency

- **Now eliminate the data dependency…**

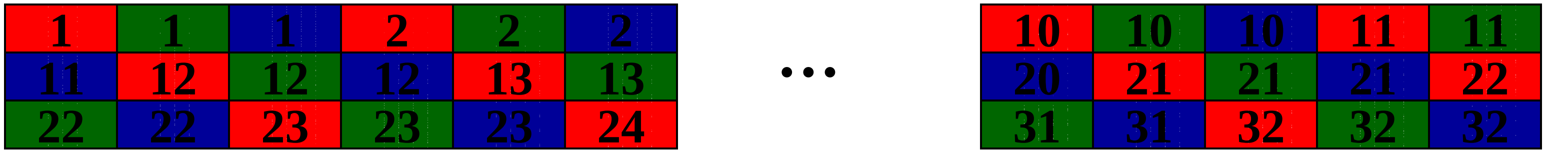
```
int FastVectorDotProduct( vector float *v1, vector float *v2, int length ){
   vector float temp = (vector float) vec_splat_s8(0);
   vector float temp2 = temp;     vector float temp3 = temp;
   vector float temp4 = temp;     vector float result;
   for( int i = 0; i < length; i += 4){                    //Loop over the length of the vectors,
      temp   = vec_madd( v1[i], v2[i], temp);              //this time doing 4 vectors in parallel
      temp2 = vec_madd( v1[i+1], v2[i+1], temp2);          // to fill the pipeline
      temp3 = vec_madd( v1[i+2], v2[i+2], temp3);
      temp4 = vec_madd( v1[i+3], v2[i+3], temp4);
   }
   //Sum our temp vectors
   temp        = vec_add( temp, temp2 );
   temp3       = vec_add( temp3, temp4 );
   temp        = vec_add( temp, temp3 );
   //Add across the vector
   temp        = vec_add( temp, vec_sld( temp, temp, 4 ));
   temp        = vec_add(temp, vec_sld( temp, temp, 8 ));
   //Copy the result to the stack so we can return it via the IPU
   vec_ste( temp, 0, &result );
   return result;
}
```

Many thanks to Ian Ollmann

Method 4 – Look at your Data Layout

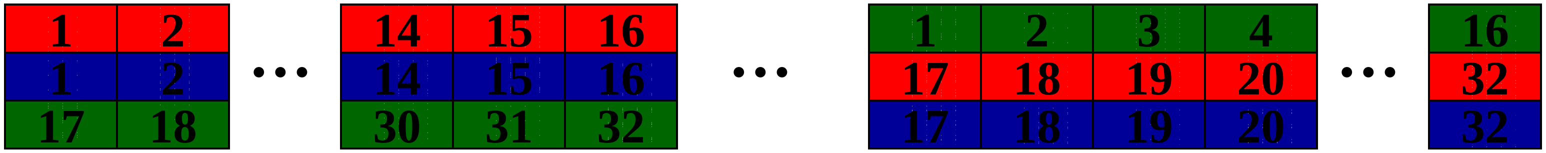
- Often algorithm calls for loading of blocks of data in certain order
  - Images (pixels orders) are good example
  - Let us look at RGB to YCbCr conversion

Layout 1



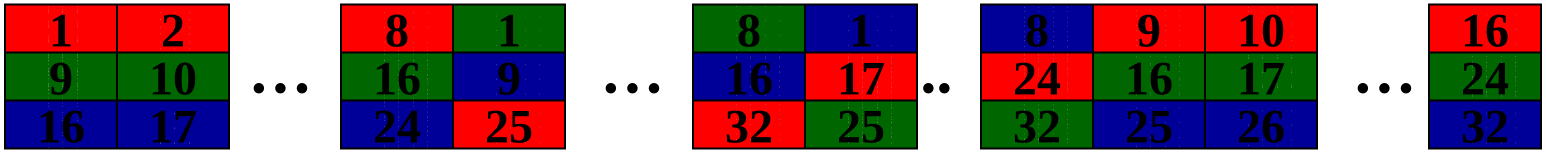
🟥 1	🟩 1	🟦 1	🟥 2	🟩 2	🟦 2		🟥 10	🟩 10	🟦 10	🟥 11	🟩 11
🟦 11	🟥 12	🟩 12	🟦 12	🟥 13	🟩 13	…	🟦 20	🟥 21	🟩 21	🟦 21	🟥 22
🟩 22	🟦 22	🟥 23	🟩 23	🟦 23	🟥 24		🟩 31	🟦 31	🟥 32	🟩 32	🟦 32

Layout 2



🟥 1	🟥 2		🟥 14	🟥 15	🟥 16		🟩 1	🟩 2	🟩 3	🟩 4		🟩 16
🟦 1	🟦 2	…	🟦 14	🟦 15	🟦 16	…	🟥 17	🟥 18	🟥 19	🟥 20	…	🟥 32
🟩 17	🟩 18		🟩 30	🟩 31	🟩 32		🟦 17	🟦 18	🟦 19	🟦 20		🟦 32

Layout 3



🟥 1	🟥 2		🟥 8	🟩 1		🟩 8	🟦 1		🟦 8	🟥 9	🟥 10		🟥 16
🟩 9	🟩 10	…	🟩 16	🟦 9	…	🟦 16	🟥 17	..	🟥 24	🟩 16	🟩 17	…	🟩 24
🟦 16	🟦 17		🟦 24	🟥 25		🟥 32	🟩 25		🟩 32	🟦 25	🟦 26		🟦 32

Method 4 – Look at your Data Layout

- One vector load from Layout 1 yields this:

1	1	1	2	2	2		5	5	5	6

- One solution (and maybe the only) is to get 3 vectors worth, and then use vec_perm instruction

1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	6
6	6	7	7	7	8	8	8	9	9	9	10	10	10	11	11
11	12	12	12	13	13	13	14	14	14	15	15	15	16	16	16

0	3	6	9	12	15	18	21	1	4	7	10	13	16	19	22
2	5	8	11	14	17	20	24	x	x	x	x	x	x	x	x

1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	x	x	x	x	x	x	x	x

Method 4 – Look at your Data Layout

- One vector load from Layout 2 yields 16 bytes of one “color”

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- Which means that only three vector loads will yield 16 full pixels
  - In three vector registers

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- This is the fastest way to get data “in” BUT only if processing is done on “chars” and no extra precision is needed

Method 4 – Look at your Data Layout

- **The actual formula for RGB to YCbCr conversion is:**

$$Y = \frac{77}{256}\left(219 \cdot \frac{R_Y}{256} + 16\right) + \frac{150}{256}\left(219 \cdot \frac{G_Y}{256} + 16\right) + \frac{29}{256}\left(219 \cdot \frac{B_Y}{256} + 16\right)$$

$$Cb = -\frac{44}{256}\left(219 \cdot \frac{R_Y}{256} + 16\right) - \frac{87}{256}\left(219 \cdot \frac{G_Y}{256} + 16\right) + \frac{131}{256}\left(219 \cdot \frac{B_Y}{256} + 16\right) + 128$$

$$Cr = \frac{131}{256}\left(219 \cdot \frac{R_Y}{256} + 16\right) - \frac{110}{256}\left(219 \cdot \frac{G_Y}{256} + 16\right) - \frac{21}{256}\left(219 \cdot \frac{B_Y}{256} + 16\right) + 128$$

- **Which is equivalent to…**

$$Y = \frac{8432}{32768}\left(R_Y\right) + \frac{16425}{32768}\left(G_Y\right) + \frac{3176}{32768}\left(B_Y\right) + 16$$

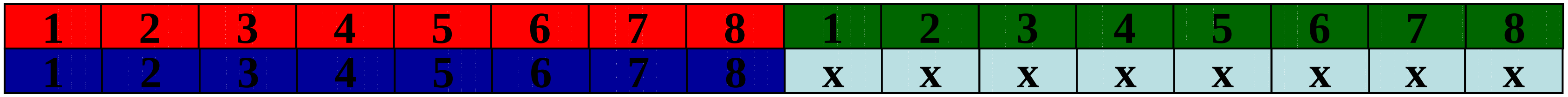
$$Cb = -\frac{4818}{32768}\left(R_Y\right) - \frac{9527}{32768}\left(G_Y\right) + \frac{14345}{32768}\left(B_Y\right) + 128$$

$$Cr = \frac{14345}{32768}\left(R_Y\right) - \frac{12045}{32768}\left(G_Y\right) - \frac{2300}{32768}\left(B_Y\right) + 128$$

- **Which needs 16 bits precision for accurate computation…**

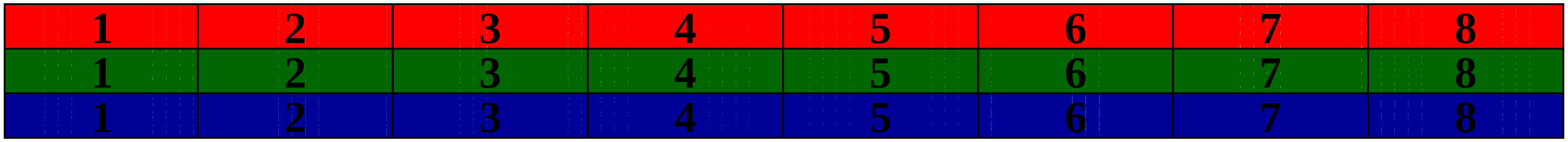
Method 4 – Look at your Data Layout

- **This could be done by “unpacking” bytes to shorts:**



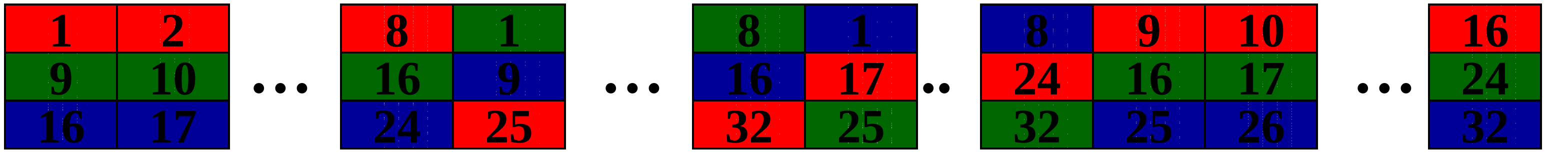
1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	x	x	x	x	x	x	x	x

- **vec_unpack_2sh and vec_unpack_2sl**



1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

- **On which the computations are performed… and packed back to bytes by vec_packsu()**
- **This means that by “reverse engineering” the necessary order of bytes back to memory we will get Layout3, so there is no need for permute instructions after vector loads**



1	2
9	10
16	17

…

8	1
16	9
24	25

…

8	1
16	17
32	25

..

8	9	10
24	16	17
32	25	26

…

16
24
32

Method 4 – Look at your Data Layout

- **Do the global Data Layout analysis**
  - Caches are only working well on data streams which exhibit spatial and time locality
  - Remember that two vector loads == one cache line
    - If loading from multiple cache lines, do one load from each line, then go back and load the second half (also works for scalar access)
    - Do not group too many loads and stores
      - 8-16 vector stores in a row can overflow CSQ (completed store queue) and cause processor to stall
      - but remember store merging – put two stores to the same cache line together
- **See if "in place" computations are possible**
  - sometimes reduces memory traffic in half

Method 5 – Look at the Data Types

- **Similarly to the optimization method 4 we can see that Data Type analysis can affect algorithm mapping**
- **In addition to "normal" or forward data type analysis…**
  - If you multiply two bytes, you better use short as result
- **… there could be a "reverse" data analysis**
  - If the actual precision of the result being used is LESS then the precision provided by extended data types, maybe simple rounding will suffice
- **A good example of this rule is use of double precision floating point in many embedded algorithms**
  - Often original algorithms are developed for "generic" conditions, which might not meet exact use of the algorithm in this specific instance
  - In this case it is a variation of the Method 0 – know (profile) your application and possible data set

Method 5 – Look at the Data Types

- One case of data type consideration (and partially data layout) is aligning allocated data to quad word boundaries
  - Different compilers have different means of achieving it, but all of them DO
    - Here is GCC example…

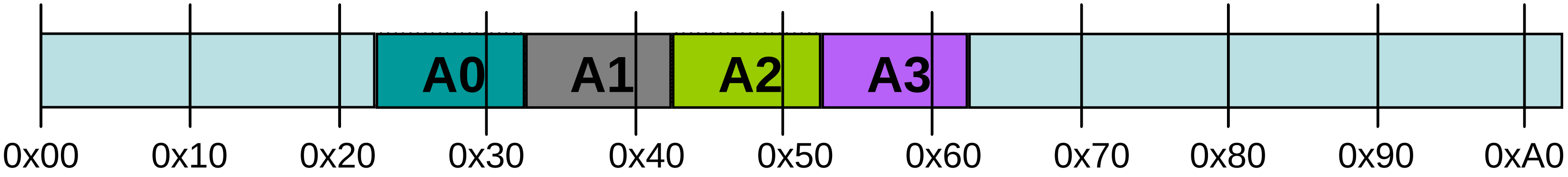
```
typedef union{
    vector unsigned int vec;
    int elements[4];
}LongVector __attribute__ ((aligned (16))) ;

 unsigned char 8bitBuf[] __attribute__ ((aligned (16))) ={
 #include "attribute_table.txt"
 };
```

- In this example every variable of data type LongVector will be aligned on quad-word boundaries and 8bitBuf is already aligned
- Data Alignment is <u>absolutely</u> critical for mapping algorithms on AltiVec

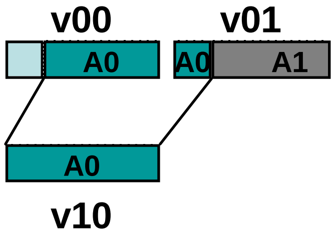
Method 5 – Look at the Data Types

- **Why?**

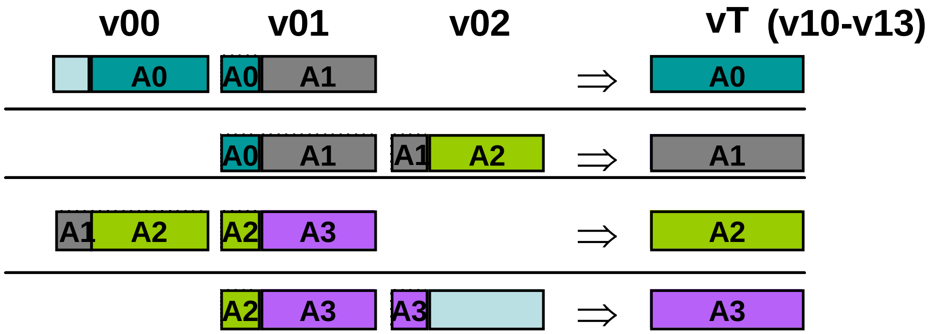


For a series of array elements: A0, A1, A2, A3

```
lvx     v00 ,&A0
lvx     v01 ,&A1
lvsl    v02 ,&A0
vperm   v10 ,v00,v01,v02
```



```
lvx     v00 ,&A0
lvx     v01 ,&A1
lvsl    v02 ,&A0
vperm   v10 ,v00,v01,v02
lvx     v00 ,&A2
vperm   v11 ,v01,v00,v02
lvx     v01 ,&A3
vperm   v12 ,v00,v01,v02
lvx     v00 ,&(A3+16)
vperm   v13 ,v01,v00,v02
```



Method 5 – Look at the Data Types

- Loading Unaligned Data requires getting ***twice*** (or more) the data you really need

```
vector unsigned char vectorLoadUnaligned( vector unsigned char *v ){
    vector unsigned char permuteVector = vec_lvsl( 0, (int*) v );
    vector unsigned char low  = vec_ld( 0, v );
    vector unsigned char high = vec_ld( 16, v );
    return vec_perm( low, high, permuteVector );
}
```

Method 5 – Look at the Data Types

- **Store Unaligned Data is even 'better' ...**
  - **You need to *LOAD* in order to be able to Store!**

```
void vectorStoreUnaligned( vector unsigned char v, vector unsigned char *where){
   vector unsigned char  permuteVector = vec_lvsr( 0, (int*) where );
   vector unsigned char  low,high,tmp,mask;
   vector     signed char  ones                = vec_splat_s8( -1 );
   vector     signed char   zeroes             = vec_splat_s8( 0 );

   vector unsigned char low   = vec_ld ( 0, where );                //Load the surrounding area
   vector unsigned char high  = vec_ld ( 16, where );
   //Make a mask for which parts of the vectors to swap out
   mask = vec_perm( zeros, ones, permuteVector );
   tmp    = vec_perm( tmp, tmp, permuteVector );                    //Right rotate our input data
   low    = vec_sel( tmp, low, mask );                    // Insert masked data to aligned vector
   high  = vec_sel( high, v, mask );

   vec_st ( low, 0, where );                              //Store aligned results
   vec_st ( high, 16, where );
}
```

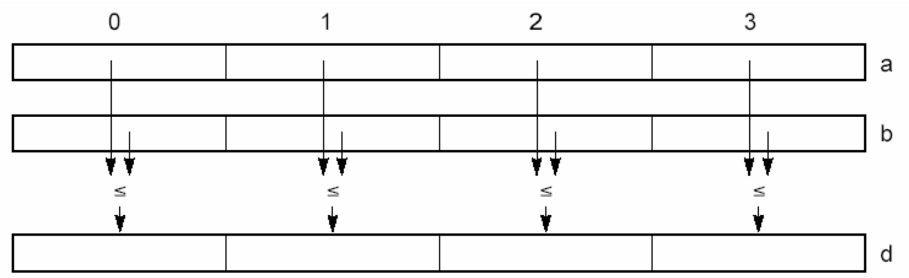
Method 6 – Eliminate Branching

- **You cannot proceed at full speed unless you know exactly where you are going…**
- **When processor encounters a branch instruction, and condition data is not available processor guesses…**
  - And if it guessed wrong, it will back track to the decision point and start over
- **There are some general guidelines on how processor will guess…**
  - Static branch prediction: **Forward branch – not taken**, backward branch is taken
    - Which means in if-then-else place LIKELY section in "then"
  - Dynamic branch prediction – after one or two invocations of the same branch instructions enough **HISTORY** is accumulated to make good prediction next time around…
    - But branch predictor is vulnerable to aliasing…
- **Try to avoid branches even if it means more computations – it is likely to be faster!**

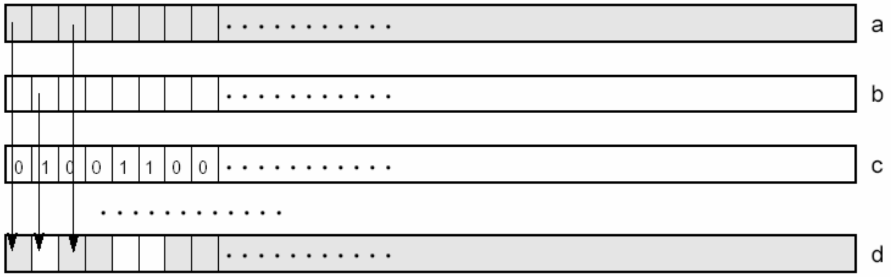
Method 6 – Eliminate Branching

- A simple example of finding maximum of two numbers
  - Assuming it is used to compare two arrays

```
int Max( int a, int b ) {
   int result;
   if( a < b )    result = b;
   else           result = a;
   return result;
}
```



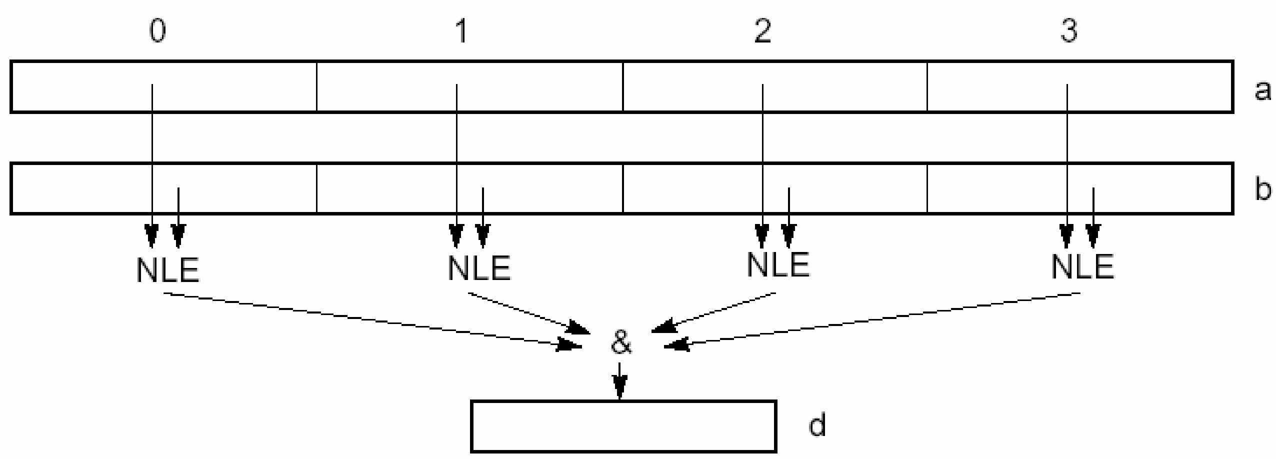
```
//Return the maximum of two vector integers:    result = (a & ~mask) | (b & mask);
vector signed int Max(vector signed int a, vector signed int b ){
    vector bool    int mask = vec_cmplt ( a, b );                //If ( a < b)...
    vector signed int result = vec_sel( a, b, mask );            //Select a or b
    return result;
}
```



Method 6 – Eliminate Branching

- **Some compare instructions write their results to integer unit registers**
  - ***vec_any_xx*() functions that return 1 if any element satisfies the test**
  - ***vec_all_xx()* functions that return 1 if all of the elements in the vector satisfy the test**

```
//Return true if the second and third floats in v are greater than 0.0
Boolean AreSecondAndThirdElementsPositive( vector float v ){
   vector unsigned int compare = (vector unsigned int)( QNaN, 0, 0, QNaN);
   return vec_all_nle( v, (vector float) compare );
}
```

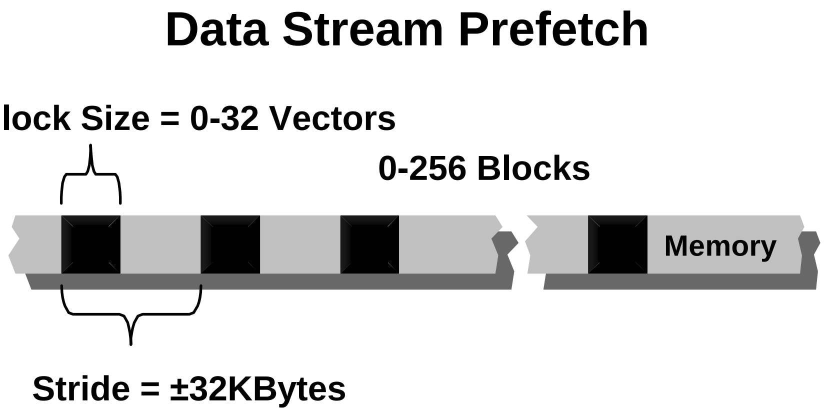


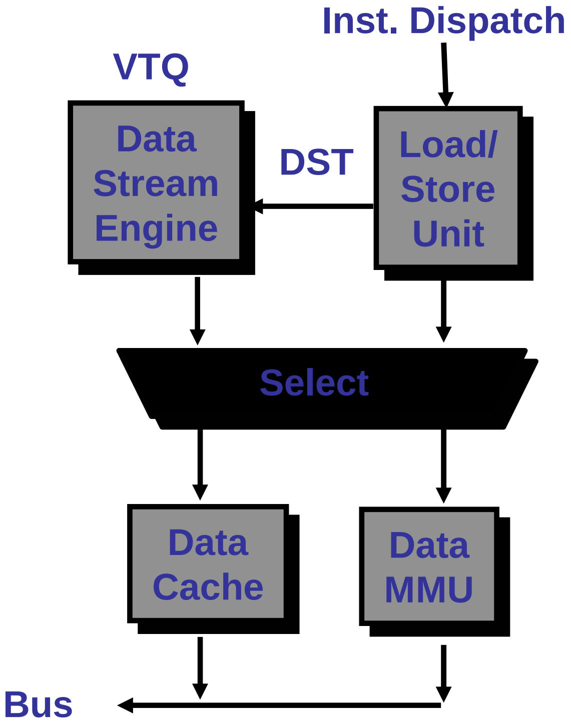
Method 7 – Look at Memory data rate

- **Memory bandwidth is the natural (and actually desirable) limit of productivity for I/O based applications**
  - **If it is a streaming application we are mainly talking about system bus throughput**
    - Maximum Actual 60x bus throughput is about 640 Mbytes/sec @ 100MHz
    - Maximum Actual MPX bus throughput is about 800 Mbytes/sec @ 100MHz
  - **If we are reusing data already fetched, we are probably dealing with Cache hierarchy**
    - Approximate L1 cache scalar throughput is around 3.7 Gbytes/sec @ 1GHz
    - AltiVec provides 4x the bandwidth accessing it at 15 GBytes/sec @ 1GHz
- **For the case when system bus is the bottleneck, we need to guarantee that it is used all the time**
  - **In AltiVec it is achieved with Data Stream Touch instruction (dstx)**
- **For the cases when we mainly depend on Cache performance, we need to improve locality and eliminate unnecessary requests**
  - **This is done in part with careful layout**
  - **and dcba/dcbt instructions**

Method 7 – Look at Memory data rate

- AltiVec allows up to *four* prefetch streams, independent and asynchronous
  - addressed by a two bit ID tag
  - *Vec_dst*(a,b,c); where *a* is initial address, *b* is control constant, *c* is ID tag



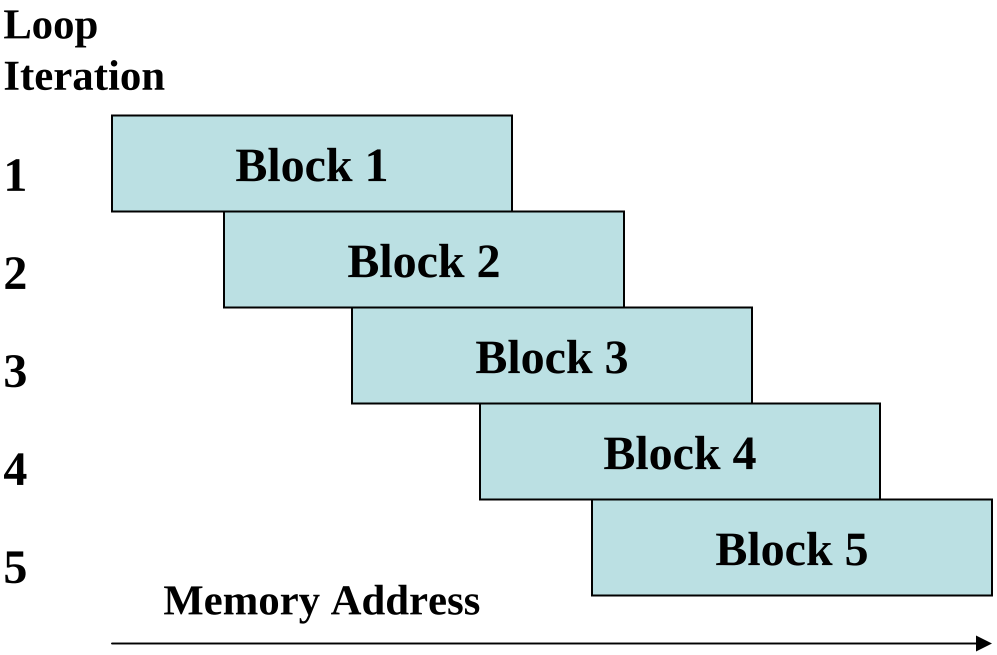


Method 7 – Look at Memory data rate

- One to four cache streams could be set using
  - *vec_dst* (Data Stream Touch for load) and…
  - *vec_dstst* (Vector Data Stream Touch for Store == load+store)
- Transient variants *vec_dstt*() and *vec_dststt*()
  - Mark their blocks to be flushed straight to RAM instead of L2 and L3 caches
- Use *vec_dst()* and *vec_dstt()* just to read a block of data
- Use *vec_dstst()* and *vec_dststt()* to read *and modify* a block of data
- Do not use prefetch for write only
  - All writes go into a Store Miss Merge Queue
  - When making two adjacent vector stores to the same cache line, they could be merged into one cache line store, so no memory read is needed
    - This is also more efficient then using dcbz (less an instruction)

Method 7 – Look at Memory data rate

- Prefetch thread is a low priority process
  - There are multiple events that can delay or stop prefetch
- Prefetch streams will also stop silently if they step on memory that is either unmapped or would cause a protection violation
- Prefetch engine could be shared by user and OS



Method 7 – Look at Memory data rate

- **To improve Cache dependent case, we might want to "help" it**
  - **AltiVec is capable of 'hinting' Cache on future usage of data**
- ***vec_ldl*() and *vec_stl*() (Vector Load/Store Indexed LRU) mark the new cache blocks as those *least recently used***
  - They will be the first to be flushed when more space in the cache is needed
  - They also mark their cache blocks as *transient*, which means that they will be flushed directly to memory rather than take up space in the L2
- **Next step is to use cache management instructions – dcba/dcbt/dcbz**
  - dcba (Data Cache Block Allocate) – allocates cache block without fetching it from memory
    - this is telling Cache "I will soon store into this cache block"
  - dcbt (Data Cache Block Touch) – fetches the cache block from memory
    - this means "I will soon load from this location"
  - dcbz (Data Cache Block Clear to Zero) – same as dcba, but zeroes out allocated block
- **There are many more dcbx instructions for a variety of scenarios**

Method 8 – Get rid of memory accesses…

- **If you do not like it… do not do it!**
- **All the color conversions do the simple calculation for each color of output space using values of colors from input space**
  - cmyk_ycck_convert() from GNU GhostScript
  - The equation looks like the following
    - OutColor1 = C1 * InColor1 + C2 * InColor2 + C3 * InColor3
      - where C1, C2 and C3 are the constants
- **To speed up the calculation original GNU code uses the pre calculated table to exchange C1 * InColor1 with Table[InColor1]**
  - The same trick is used to limit range of the output value (to one byte length)
- **AltiVec version calculate the output value "as is" using original equation and "mradd" instructions.**
  - It is match faster than memory byte access operations
  - And we do such calculation for eight pixels simultaneously
- **The result is 5x Faster…**

Method 8 – Get rid of memory accesses…

- GNU optimized implementation with Lookup Table

```
while (--num_rows >= 0) {
  …
  for (col = 0; col < num_cols; col++) {

              r = MAXJSAMPLE - GETJSAMPLE(inptr[0]);
              g = MAXJSAMPLE - GETJSAMPLE(inptr[1]);
              b = MAXJSAMPLE - GETJSAMPLE(inptr[2]);
              /* K passes through as-is */
              outptr3[col] = inptr[3];
              inptr += 4;
              /* Y */
              outptr0[col] = ((ctab[r+R_Y_OFF]   + ctab[g+G_Y_OFF]   + ctab[b+B_Y_OFF])   >> SCALEBITS);
              /* Cb */
              outptr1[col] = ((ctab[r+R_CB_OFF] + ctab[g+G_CB_OFF] + ctab[b+B_CB_OFF]) >> SCALEBITS);
              /* Cr */
              outptr2[col] = ((ctab[r+R_CR_OFF] + ctab[g+G_CR_OFF] + ctab[b+B_CR_OFF]) >> SCALEBITS);
  }
}
```

Method 8 – Get rid of memory accesses…

```
while (--num_rows >= 0)     {
  source0 = vec_ld(0, inptr);
  for (col = 0; col < num_cols; source0 = source4, col+=16) {
              …
              /*              Perform 16-bit arithmetic conversion of R,G,B to Y,Cb,Cr
                              Y  =  0.29900 * R + 0.58700 * G + 0.11400 * B
                              Cb = -0.16874 * R - 0.33126 * G + 0.50000 * B  + CENTERJSAMPLE
                              Cr =  0.50000 * R - 0.41869 * G - 0.08131 * B  + CENTERJSAMPLE    */
              y0   = vec_mradds(ss_ry,  r0, vec_mradds(ss_gy,  g0, vec_mradds(ss_by,  b0, ss_zero)));
              y1   = vec_mradds(ss_ry,  r1, vec_mradds(ss_gy,  g1, vec_mradds(ss_by,  b1, ss_zero)));
              cb0 = vec_mradds(ss_rcb, r0, vec_mradds(ss_gcb, g0, vec_mradds(ss_bcb, b0, ss_center)));
              cb1 = vec_mradds(ss_rcb, r1, vec_mradds(ss_gcb, g1, vec_mradds(ss_bcb, b1, ss_center)));
              cr0  = vec_mradds(ss_rcr, r0, vec_mradds(ss_gcr, g0, vec_mradds(ss_bcr, b0, ss_center)));
              cr1  = vec_mradds(ss_rcr, r1, vec_mradds(ss_gcr, g1, vec_mradds(ss_bcr, b1, ss_center)));

              /*              Pack results into 8-bit format and store to non-interleaved data streams:
                              output arrays are aligned, but the size of the image
                              could be unaligned with vector size (num_cols % 16 != 0) !             */
              y   = vec_packsu(y0,  y1);
              cb = vec_packsu(cb0, cb1);
              cr  = vec_packsu(cr0, cr1);
              …
              *outptr0++ = y;
              *outptr1++ = cb;
              *outptr2++ = cr;
              *outptr3++ = k;
  }
}
```

Method 9 – Get rid of computations…

- **Well, the same idea, do not like it, don't do it…**
  - You have some control over where the bottleneck is, so move it around a bit
- ***Data slicing* is a very effective technique to be used in AltiVec**
- **It is best explained with an example**
- **Let's consider Byte-wise Bit Reversal algorithm**
  - **For each byte return bit reversal version of the input:**

```
unsigned char reverse (unsigned char in){
  unsigned char out =  ((in & 0x01)<<7) |
                ((in & 0x02) <<5) |
                ((in & 0x04) <<3) |
                ((in & 0x08) <<1) |
                ((in & 0x10) >>1) |
                ((in & 0x20) >>3) |
                ((in & 0x40) >>5) |
                ((in & 0x80) >>7);
  return out;
}
```

- **This straightforward method yields 0.10 Bytes/Cycle**

Method 9 – Get rid of computations…

- Alternative implementation could be *Big Lookup Table*:
  - 256 entry byte table holding the “reversed” values
  - So, the computation for each byte is converted into a single “load”
    - reversed[j] = big_lookup[in[i]];

```
unsigned char big_lookup[256] = {
  0x00,0x80,0x40,0xc0,0x20,0xa0,0x60,0xe0,0x10,0x90,0x50,0xd0,0x30,0xb0,0x70,0xf0,
  0x08,0x88,0x48,0xc8,0x28,0xa8,0x68,0xe8,0x18,0x98,0x58,0xd8,0x38,0xb8,0x78,0xf8,
  0x04,0x84,0x44,0xc4,0x24,0xa4,0x64,0xe4,0x14,0x94,0x54,0xd4,0x34,0xb4,0x74,0xf4,
  0x0c,0x8c,0x4c,0xcc,0x2c, 0xac,0x6c,0xec,0x1c,0x9c,0x5c,0xdc,0x3c,0xbc,0x7c,0xfc,
  0x02,0x82,0x42,0xc2,0x22,0xa2,0x62,0xe2,0x12,0x92,0x52,0xd2,0x32,0xb2,0x72,0xf2,
  0x0a,0x8a,0x4a,0xca,0x2a,0xaa,0x6a,0xea,0x1a,0x9a,0x5a,0xda,0x3a,0xba,0x7a,0xfa,
  0x06,0x86,0x46,0xc6,0x26,0xa6,0x66,0xe6,0x16,0x96,0x56,0xd6,0x36,0xb6,0x76,0xf6,
  0x0e,0x8e,0x4e,0xce,0x2e,0xae,0x6e,0xee,0x1e,0x9e,0x5e,0xde,0x3e,0xbe,0x7e,0xfe,
  0x01,0x81,0x41,0xc1,0x21,0xa1,0x61,0xe1,0x11,0x91,0x51,0xd1,0x31,0xb1,0x71,0xf1,
  0x09,0x89,0x49,0xc9,0x29,0xa9,0x69,0xe9,0x19,0x99,0x59,0xd9,0x39,0xb9,0x79,0xf9,
  0x05,0x85,0x45,0xc5,0x25,0xa5,0x65,0xe5,0x15,0x95,0x55,0xd5,0x35,0xb5,0x75,0xf5,
  0x0d,0x8d,0x4d,0xcd,0x2d,0xad,0x6d,0xed,0x1d,0x9d,0x5d,0xdd,0x3d,0xbd,0x7d,0xfd,
  0x03,0x83,0x43,0xc3,0x23,0xa3,0x63,0xe3,0x13,0x93,0x53,0xd3,0x33,0xb3,0x73,0xf3,
  0x0b,0x8b,0x4b,0xcb,0x2b,0xab,0x6b,0xeb,0x1b,0x9b,0x5b,0xdb,0x3b,0xbb,0x7b,0xfb,
  0x07,0x87,0x47,0xc7,0x27, 0xa7,0x67,0xe7,0x17,0x97,0x57,0xd7,0x37,0xb7,0x77,0xf7,
  0x0f, 0x8f,0x4f, 0xcf, 0x2f, 0xaf, 0x6f, 0xef, 0x1f, 0x9f, 0x5f, 0xdf,0x3f,0xbf,0x7f,0xff
};
```

- This method yields 0.19 Bytes/Cycle or 2x faster then original

Launched by Motorola
freescale™
semiconductor

Method 9 – Get rid of computations…

- Another method is *Small Lookup Table*
  - based on splitting each byte into two nibbles
  - looking up values for both of them independently, and merging result later

```
unsigned char small_lookup_l[16] __attribute__ ((aligned (16))) = {
  0x00,0x08,0x04,0x0c,0x02,0x0a,0x06,0x0e,0x01,0x09,0x05,0x0d,0x03,0x0b,0x07,0x0f
};

unsigned char small_lookup_h[16] __attribute__ ((aligned (16))) = {
  0x00,0x80,0x40,0xc0,0x20,0xa0,0x60,0xe0,0x10,0x90,0x50,0xd0,0x30,0xb0,0x70,0xf0
};
```

```
reversed[j] = small_lookup_l[(in[j]&0xf0)>>4] | small_lookup_h[(in[j]&0x0f)];
```

- This method uses less memory, but runs a bit slower: 0.11 Bytes/Cycle

Method 9 – Get rid of computations…

- The true advantages comes from observation that small tables will fit into two ALtiVec registers…
- … and ALL lookups are completely independent, so 16 of them could be performed in parallel !!!

```
void reverse_vector(vector unsigned char *in,vector unsigned char *out, int num_elements){
  int i;
  vector unsigned char st_l, st_h;
  vector unsigned char four =  vec_splat_u8(4);
  vector unsigned char v_in,vl,vh, v_out;

  st_l = vec_ld (0,(vector unsigned char *) small_lookup_l);
  st_h = vec_ld (0,(vector unsigned char *) small_lookup_h);

  for(i=0; i<num_elements; i+=16){
       v_in    = vec_ld (i,in);
      vh       = vec_sr(v_in,four);
      vh       = vec_perm(st_l,st_l,vh);
      vl       = vec_perm(st_h,st_h,v_in);
      v_out  = vec_or(vh,vl);
      vec_st(v_out,i,out);
  }
```

- This method for the same conditions gets
  - 2.7 Bytes/Cycle
- It is 30x faster then original Scalar
- …and 15x faster then BigLookupTable

Method 10 – Constant Generations

- Often a “standard” C declaration of an initialized variable is interpreted by compiler as declare + store…

```
vector signed int        zero_vec_32 = { 0,0,0,0 };
vector float             zero_vec_fp  = { 0.0,0.0,0.0,0.0 };
```

- In this case it is much more practical to generate these constants

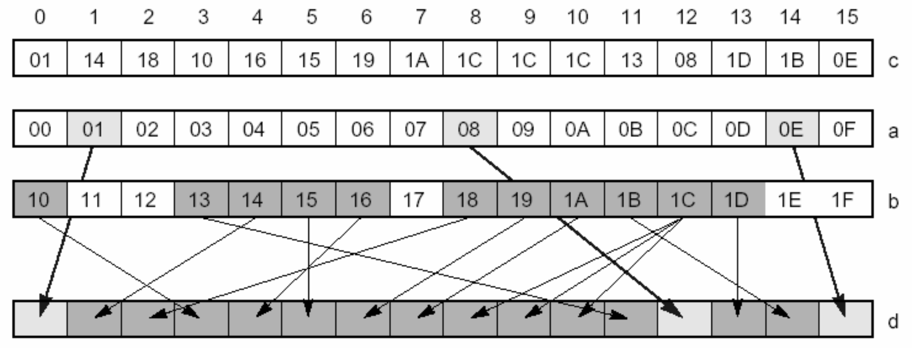
```
vector signed int        zero_vec_32 = vec_splat_s32(0);
vector float              zero_vec_fp = vec_ctf( vec_splat_u32(0), 0 ));
                                 // Vector Convert from Fixed-Point Word
```

- These are trivial cases, but what about “fancy” constants…

```
vector float vec_neg_zero( void ){  //Generate a vector full of –0.0.
        vector unsigned int result = vec_splat_u32(-1); // Vector Splat
        return (vector float ) vec_sl( result, result ); // Vector Shift Left
}
```

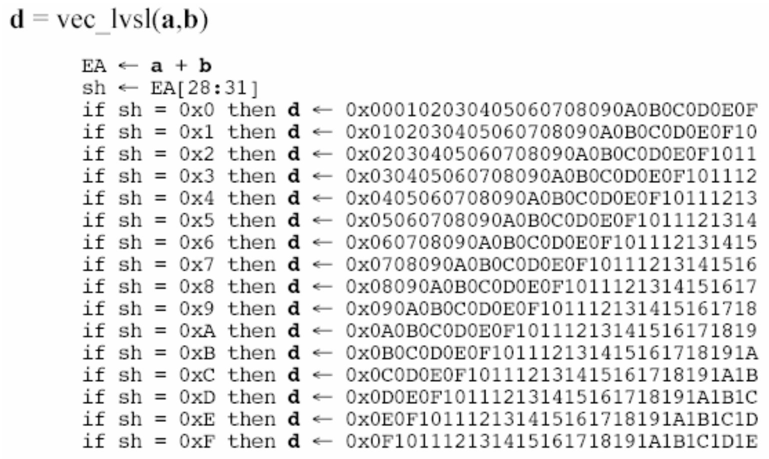
Method 10 – Constant Generations

- A very common reason for constant generation is their use for vector permute instruction



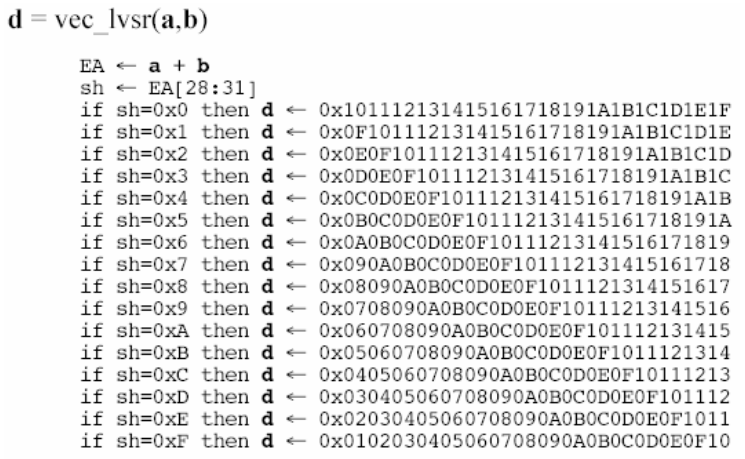
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
01	14	18	10	16	15	19	1A	1C	1C	1C	13	08	1D	1B	0E	c
00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	a
10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	b
																d

```
vector unsigned char
 vectorLoadUnaligned( vector unsigned char *v ){
    vector unsigned char permuteVector =
                  vec_lvsl( 0, v );
    vector unsigned char low  = vec_ld( 0, v );
    vector unsigned char high = vec_ld( 16, v );
    return vec_perm( low, high, permuteVector );
}
```



d = vec_lvsl(**a**,**b**)

```
EA ← a + b
sh ← EA[28:31]
if sh = 0x0 then d ← 0x000102030405060708090A0B0C0D0E0F
if sh = 0x1 then d ← 0x0102030405060708090A0B0C0D0E0F10
if sh = 0x2 then d ← 0x02030405060708090A0B0C0D0E0F1011
if sh = 0x3 then d ← 0x030405060708090A0B0C0D0E0F101112
if sh = 0x4 then d ← 0x0405060708090A0B0C0D0E0F10111213
if sh = 0x5 then d ← 0x05060708090A0B0C0D0E0F1011121314
if sh = 0x6 then d ← 0x060708090A0B0C0D0E0F101112131415
if sh = 0x7 then d ← 0x0708090A0B0C0D0E0F10111213141516
if sh = 0x8 then d ← 0x08090A0B0C0D0E0F1011121314151617
if sh = 0x9 then d ← 0x090A0B0C0D0E0F101112131415161718
if sh = 0xA then d ← 0x0A0B0C0D0E0F10111213141516171819
if sh = 0xB then d ← 0x0B0C0D0E0F101112131415161718191A
if sh = 0xC then d ← 0x0C0D0E0F101112131415161718191A1B
if sh = 0xD then d ← 0x0D0E0F101112131415161718191A1B1C
if sh = 0xE then d ← 0x0E0F101112131415161718191A1B1C1D
if sh = 0xF then d ← 0x0F101112131415161718191A1B1C1D1E
```



d = vec_lvsr(**a**,**b**)

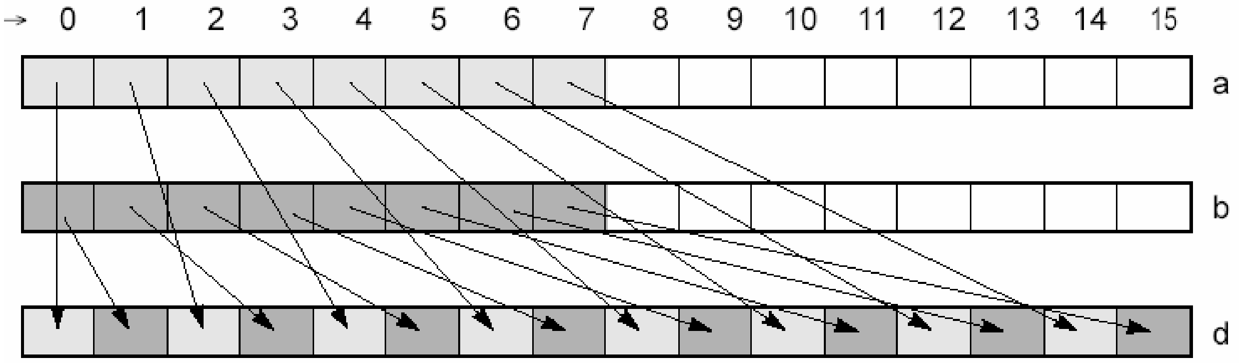
```
EA ← a + b
sh ← EA[28:31]
if sh=0x0 then d ← 0x101112131415161718191A1B1C1D1E1F
if sh=0x1 then d ← 0x0F101112131415161718191A1B1C1D1E
if sh=0x2 then d ← 0x0E0F101112131415161718191A1B1C1D
if sh=0x3 then d ← 0x0D0E0F101112131415161718191A1B1C
if sh=0x4 then d ← 0x0C0D0E0F101112131415161718191A1B
if sh=0x5 then d ← 0x0B0C0D0E0F101112131415161718191A
if sh=0x6 then d ← 0x0A0B0C0D0E0F10111213141516171819
if sh=0x7 then d ← 0x090A0B0C0D0E0F101112131415161718
if sh=0x8 then d ← 0x08090A0B0C0D0E0F1011121314151617
if sh=0x9 then d ← 0x0708090A0B0C0D0E0F10111213141516
if sh=0xA then d ← 0x060708090A0B0C0D0E0F101112131415
if sh=0xB then d ← 0x05060708090A0B0C0D0E0F1011121314
if sh=0xC then d ← 0x0405060708090A0B0C0D0E0F10111213
if sh=0xD then d ← 0x030405060708090A0B0C0D0E0F101112
if sh=0xE then d ← 0x02030405060708090A0B0C0D0E0F1011
if sh=0xF then d ← 0x0102030405060708090A0B0C0D0E0F10
```

Method 10 – Constant Generations

- **But there are number of cases that are much more complex…**

```
vector unsigned char vec_perm1 =
                        (vector unsigned char){0,1,2,3,16,17,18,19,4,5,6,7,20,21,22,23};
```

```
vector unsigned char a = vec_lvsl(0, 0);
vector unsigned char b = vec_lvsr(0, 0);
vector unsigned char vec_perm1 =
  (vector unsigned char)vec_mergeh((vector unsigned int)a, (vector unsigned int)b);
```



Method 10 – Constant Generations

```
00: (1) splat 00
01: (1) splat 01
02: (1) splat 02
03: (1) splat 03
04: (1) splat 04
05: (1) splat 05
06: (1) splat 06
07: (1) splat 07
08: (1) splat 08
09: (1) splat 09
0A: (1) splat 0A
0B: (1) splat 0B
0C: (1) splat 0C
0D: (1) splat 0D
0E: (1) splat 0E
0F: (1) splat 0F
10: (2) splat 08, add self
11: (3) splat 02, splat 0F, add together
12: (2) splat 09, add self
13: (3) splat 04, splat 0F, add together
14: (2) splat 0A, add self
15: (3) splat 06, splat 0F, add together
16: (2) splat 0B, add self
17: (3) splat 08, splat 0F, add together
18: (2) splat 0C, add self
19: (3) splat 0A, splat 0F, add together
1A: (2) splat 0D, add self
1B: (3) splat 0C, splat 0F, add together
1C: (2) splat 0E, add self
1D: (3) splat 0E, splat 0F, add together
1E: (2) splat 0F, add self
1F: (2) splat FB, srl self
20: (3) splat 01, splat 05, rol by
21: (3) splat 09, splat 05, rol by
22: (3) splat 04, rol self, average signed
23: (3) splat F2, sl self, rol by
24: (3) splat 09, splat 02, rol by
25: (4) splat 06; splat FB, srl self; add together
26: (4) splat 07; splat FB, srl self; add together
27: (3) splat 0D, add self, add together
28: (2) splat 0A, rol self
29: (4) splat 01; splat 0A, rol self; add together
2A: (3) splat 0E, add self, add together
2B: (3) splat F3, srl self, subtract
2C: (3) splat 0B, splat 02, rol by
2D: (3) splat 0F, add self, add together
2E: (3) splat F2, srl self, add together
2F: (3) splat F2, splat 04, rol by
30: (3) splat 03, splat 04, rol by
31: (4) splat 09; splat 0A, rol self; add together
32: (3) splat 0A, rol self, add together
33: (3) splat 0C, rol self, nor together
34: (3) splat 0D, splat 02, rol by
35: (3) splat F1, srl self, average signed
36: (4) splat 0E; splat 0A, rol self; add together
37: (3) splat F3, add self, rol by
38: (3) splat 07, splat 03, rol by
39: (3) splat F2, add self, srl by
3A: (3) splat FA, rol self, srl by
3B: (3) splat F6, add self, rol by
3C: (2) splat F2, srl self
3D: (3) splat F4, splat 06, rol by
3E: (2) splat FA, srl self
3F: (3) splat F3, splat 04, rol by
40: (2) splat 04, rol self
41: (3) splat 05, splat 06, rol by
42: (3) splat 09, splat 06, rol by
43: (3) splat 0D, splat 06, rol by
44: (3) splat 04, rol self, add together
45: (3) splat 07, rol self, average unsigned
46: (4) splat 06; splat 04, rol self; add together
47: (3) splat 0E, sl self, average unsigned
48: (3) splat 09, splat 03, rol by
49: (3) splat 0E, rol self, average unsigned
4A: (3) splat F2, srl self, subtract
4B: (3) splat 0F, rol self, average unsigned
4C: (3) splat 0C, rol self, subtract
4D: (3) splat 0B, rol self, subtract
4E: (3) splat F2, add self, rol self
4F: (2) splat F4, rol self
50: (3) splat 05, splat 04, rol by
51: (4) splat 02; splat F4, rol self; add together
52: (3) splat 0D, rol self, nor together
53: (3) splat 05, rol self, average unsigned
54: (3) splat F3, rol self, subtract
55: (3) splat F5, sl self, subtract
56: (4) splat 07; splat F4, rol self; add together
57: (3) splat 0D, rol self, average unsigned
58: (2) splat 0B, rol self
59: (4) splat 01; splat 0B, rol self; add together
5A: (3) splat 05, rol self, nor together
5B: (3) splat 0B, rol self, or together
5C: (4) splat 04; splat 0B, rol self; add together
5D: (3) splat F5, add self, rol by
5E: (3) splat F2, splat 05, rol by
5F: (3) splat F5, splat 04, rol by
60: (3) splat 03, splat 05, rol by
61: (3) splat 0B, splat 05, rol by
62: (4) splat 0A; splat 0B, rol self; add together
63: (3) splat 0B, rol self, add together
64: (4) splat 0C; splat 0B, rol self; add together
65: (3) splat 05, rol self, subtract
66: (3) splat 0C, rol self, average unsigned
67: (3) splat F3, sl self, nor self
68: (3) splat 0D, splat 03, rol by
69: (3) splat F1, srl self, add together
6A: (4) splat F1, srl self; splat F2; add together
6B: (3) splat F3, sl self, xor together
6C: (3) splat 0D, rol self, subtract
6D: (3) splat 0D, sl self, subtract
6E: (4) splat F1, srl self; splat F6; add together
6F: (3) splat F6, splat 04, rol by
70: (3) splat 07, splat 04, rol by
71: (3) splat F1, add self, srl by
72: (3) splat 0E, sl self, subtract
73: (4) splat F1, srl self; splat FB; add together
74: (4) splat F1, srl self; splat FC; add together
75: (3) splat 0E, rol self, subtract
76: (3) splat F6, sl self, add together
77: (3) splat F7, add self, rol by
78: (2) splat F1, srl self
79: (3) xor self, splat F1, average unsigned
7A: (3) xor self, splat F3, average unsigned
7B: (3) xor self, splat F5, average unsigned
7C: (2) splat F9, srl self
7D: (3) xor self, splat F9, average unsigned
7E: (3) xor self, splat FB, average unsigned
7F: (3) xor self, splat FD, average unsigned
80: (2) splat 06, sl self
81: (2) splat 06, rol self
82: (3) splat 04, cmpeq self, average unsigned
83: (2) splat 07, rol self
84: (3) splat 08, cmpeq self, average unsigned
85: (3) splat 0A, cmpeq self, average unsigned
86: (3) splat 0C, cmpeq self, average unsigned
87: (2) splat 0F, rol self
88: (3) splat F1, splat 03, sl by
89: (3) splat F7, sl self, subtract
8A: (3) splat 07, rol self, add together
8B: (3) splat 0E, rol self, subtract
8C: (4) splat 05; splat 0F, rol self; add together
8D: (3) splat 0E, rol self, xor together
8E: (3) splat F4, add self, rol by
8F: (3) splat F1, splat 03, rol by
90: (3) splat 09, splat 04, rol by
91: (3) splat 0E, rol self, add together
92: (3) splat F3, rol self, add together
93: (3) splat F2, add self, rol by
94: (3) splat 0D, rol self, subtract
95: (3) splat F5, sl self, add together
96: (3) splat 0F, rol self, add together
97: (3) splat F2, splat 03, rol by
98: (2) splat F3, sl self
99: (4) splat 01; splat F3, sl self; add together
9A: (3) splat F4, sl self, average unsigned
9B: (3) splat 05, rol self, subtract
9C: (3) splat FA, srl self, average unsigned
9D: (3) splat FD, sl self, add together
9E: (3) splat F4, splat 05, rol by
9F: (2) splat F3, rol self
A0: (2) splat 05, rol self
A1: (2) splat 0D, rol self
A2: (3) splat F4, rol self, average unsigned
A3: (3) splat FA, sl self, rol by
A4: (3) splat 0B, rol self, nor together
A5: (3) splat 05, rol self, add together
A6: (4) splat 05; splat 0D, rol self; add together
A7: (3) splat F4, splat 03, rol by
A8: (3) splat F5, splat 03, sl by
A9: (4) splat 08; splat 0D, rol self; add together
AA: (4) splat 09; splat 0D, rol self; add together
AB: (3) splat F5, sl self, subtract
AC: (3) splat 0D, rol self, xor together
AD: (3) splat 0D, rol self, or together
AE: (3) splat 0D, rol self, add together
AF: (3) splat F5, splat 03, rol by
B0: (3) splat 0B, splat 04, rol by
B1: (4) splat F6, rol self; splat F4; add together
B2: (4) splat F6, rol self; splat F5; add together
B3: (3) splat 0B, rol self, subtract
B4: (3) splat 0C, rol self, subtract
B5: (3) splat F1, srl self, average unsigned
B6: (3) splat F2, srl self, subtract
B7: (3) splat F6, splat 03, rol by
B8: (3) splat F7, splat 03, sl by
B9: (3) splat F3, add self, rol self
BA: (3) splat F2, sl self, add together
BB: (3) splat 04, rol self, nor together
BC: (3) splat F2, splat 06, rol by
BD: (2) splat F6, rol self
BE: (2) splat F5, rol self
BF: (2) splat FD, rol self
C0: (2) splat 0C, rol self
C1: (3) splat 07, splat 06, rol by
C2: (3) splat 0B, splat 06, rol by
C3: (3) splat 0F, splat 06, rol by
C4: (3) splat F1, splat 02, sl by
C5: (3) splat F1, add self, rol by
C6: (3) splat F1, rol self, sl by
C7: (3) splat F1, splat 02, rol by
C8: (2) splat F2, sl self
C9: (3) splat 0E, rol self, average signed
CA: (4) splat 02; splat F2, sl self; add together
CB: (2) splat F2, rol self
CC: (3) splat F3, splat 02, sl by
CD: (4) splat 02; splat F2, rol self; add together
CE: (3) splat F2, srl self, xor together
CF: (2) splat FC, rol self
D0: (3) splat 0D, splat 04, rol by
D1: (4) splat 02; splat FC, rol self; add together
D2: (4) splat 03; splat FC, rol self; add together
D3: (3) splat F4, splat 02, rol by
D4: (3) splat F5, splat 02, sl by
D5: (3) splat 0A, rol self, nor together
D6: (3) splat F2, add self, add together
D7: (3) splat F5, splat 02, rol by
D8: (2) splat FB, sl self
D9: (3) splat F3, add self, add together
DA: (3) splat F5, rol self, average signed
DB: (3) splat F6, splat 02, rol by
DC: (3) splat F7, splat 02, sl by
DD: (3) splat F2, sl self, average signed
DE: (3) splat F6, splat 05, rol by
DF: (2) splat FB, rol self
E0: (2) splat F0, add self
E1: (3) splat F0, splat F1, add together
E2: (2) splat F1, add self
E3: (2) splat F1, rol self
E4: (2) splat F2, add self
E5: (3) splat F0, splat F5, add together
E6: (2) splat F3, add self
E7: (3) splat F0, splat F7, add together
E8: (2) splat F4, add self
E9: (3) splat F0, splat F9, add together
EA: (2) splat F5, add self
EB: (2) splat FA, rol self
EC: (2) splat F6, add self
ED: (3) splat F0, splat FD, add together
EE: (2) splat F7, add self
EF: (3) splat F0, cmpeq self, add together
F0: (1) splat F0
F1: (1) splat F1
F2: (1) splat F2
F3: (1) splat F3
F4: (1) splat F4
F5: (1) splat F5
F6: (1) splat F6
F7: (1) splat F7
F8: (1) splat F8
F9: (1) splat F9
FA: (1) splat FA
FB: (1) splat FB
FC: (1) splat FC
FD: (1) splat FD
FE: (1) splat FE
FF: (1) splat FF
```

00: (1) splat 00
01: (1) splat 01
...
10: (2) splat 08, add self
11: (3) splat 02, splat 0F, add together
12: (2) splat 09, add self
…
3D: (3) splat F4, splat 06, rol by
3E: (2) splat FA, srl self
3F: (3) splat F3, splat 04, rol by
…
B1: (4) splat F6, rol self; splat F4; add together
B2: (4) splat F6, rol self; splat F5; add together
B3: (3) splat 0B, rol self, subtract
…
FE: (1) splat FE
FF: (1) splat FF

32 sequences consist of 1 instruction
44 sequences consist of 2 instructions
152 sequences consist of 3 instructions
28 sequences consist of 4 instructions

Many thanks to Holger Bettag

Put it all together

- **Step back and take a 10,000 foot view**
- **There is a logical sequence to be observed in implementation of these methods…**
  - One can look at the optimization process as on moving the bottleneck around the processor –
    - if computation takes longer then anything else – speed them up
    - if system bus is underutilized – use prefetching
    - if bus is 100% full, computations are at the minimum… reduce the code and data size?
- **But the truly superior goal is to reach computational entropy –**
  - get rid of all the unnecessary computations through *algorithm* modifications
  - and balance added memory bandwidth with real data I/O
  - use predictability of the data streams to the full extent
- **Concentrate your effort, in large applications work with 10% of the code which accounts for 90% of execution time**

General Coding Strategy

- **Use Vector algorithms**
  - Aim for high throughput
- **Align your Data**
  - 16 bytes
  - Never hurts scalar code
  - Keep all data in close proximity
    - Helps to improve memory performance
  - Try not to mix different data types in the same vector
- **Do more work**
  - On a cold cache assume having 40 cycles for each 32byte chunk of data
  - You are likely to achieve TOP performance when processing time exactly equal to the fetch time
  - Use prefetch and 'hinting' instructions

More ways…

AltiVec Library Offering

- Telecomm
  - FFT/IFFT, FIR, Autocorrelation, Convolution Encoder/Viterbi Decoder (GSM,3G), Error Correction Codes (CRC 8,12,16,24)
  - Voice Over IP (G723, G729) elements
- MultiMedia
  - DCT/IDCT, MPEG2, MPEG4, H.26x, AC3, MP3, JPEG/JPEG 2000, Quantization/Dequantization, SAD
  - Voice Recognition, Pattern Recognition,
- Printer
  - GhostScript Library elements, Color Management routines, Color Conversion (RGB to YCbCr), Scaling/Rotation, Filtering routines, FS Dithering
- Networking
  - OSPF, QOS, NAT, Route Lookup, IP Reassembly, TCP/IP,
  - Encryption (AES, DES, 3DES, MD5, SHA, RSA, Kasumi)
  - Wireless network (802.11), LZO
- LibC (means could be “Linked” at compilation)
  - Link level support for standard C functions (memcpy, strcmp etc.)
- Mathematical primitives (Extension of LibC+)
  - Matrix math, LargeNumber Lib
  - Math.h - Log, Exp, Sin, Cos, Sqrt
- OS enablement
  - Linux (TCP/IP),
  - VxWorks elements

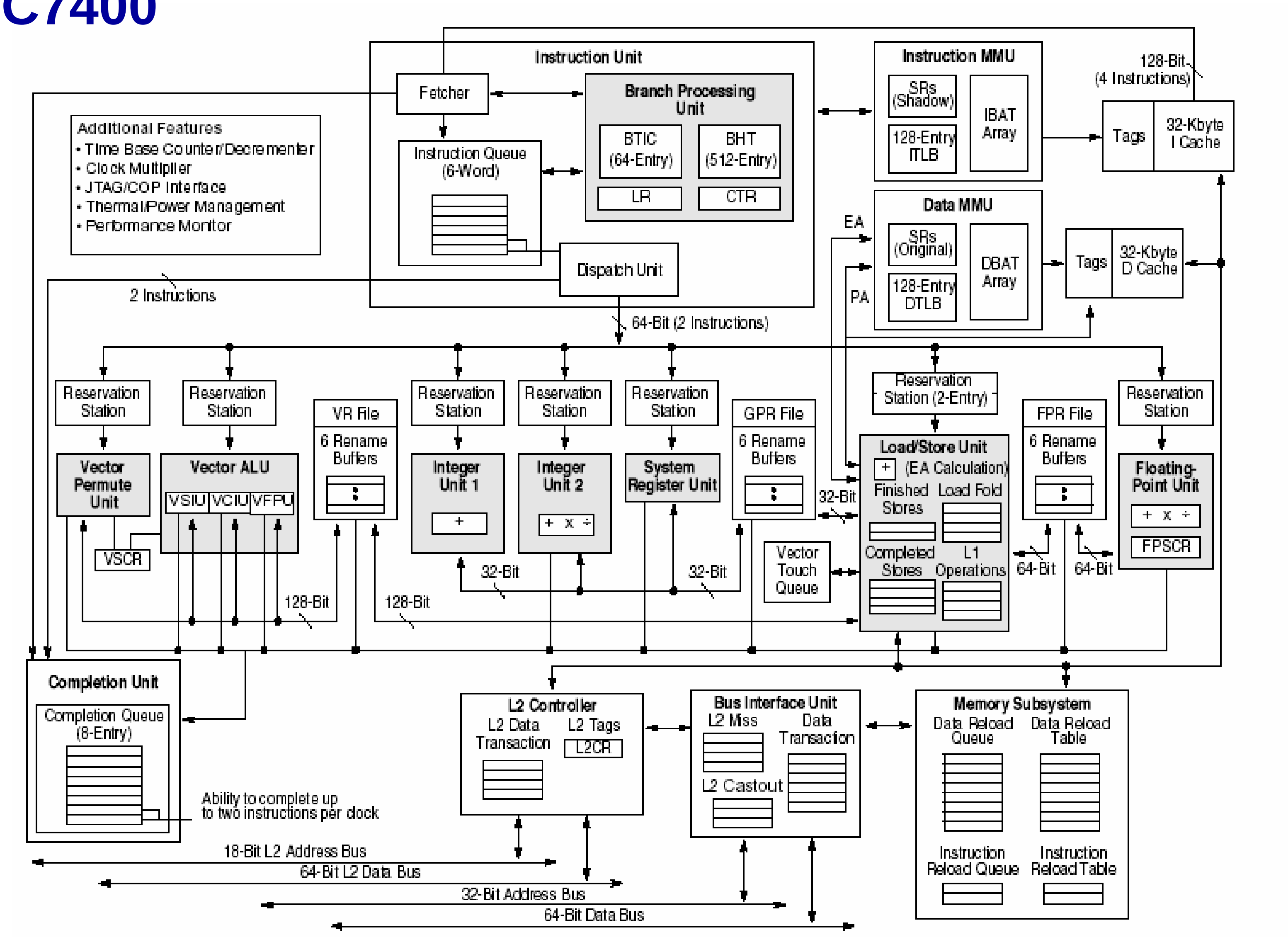
Legend

Green color	- available on the AV web site now (motorola.com/altivec)
Orange color	- available upon request
Light blue	- available soon

To summarize:

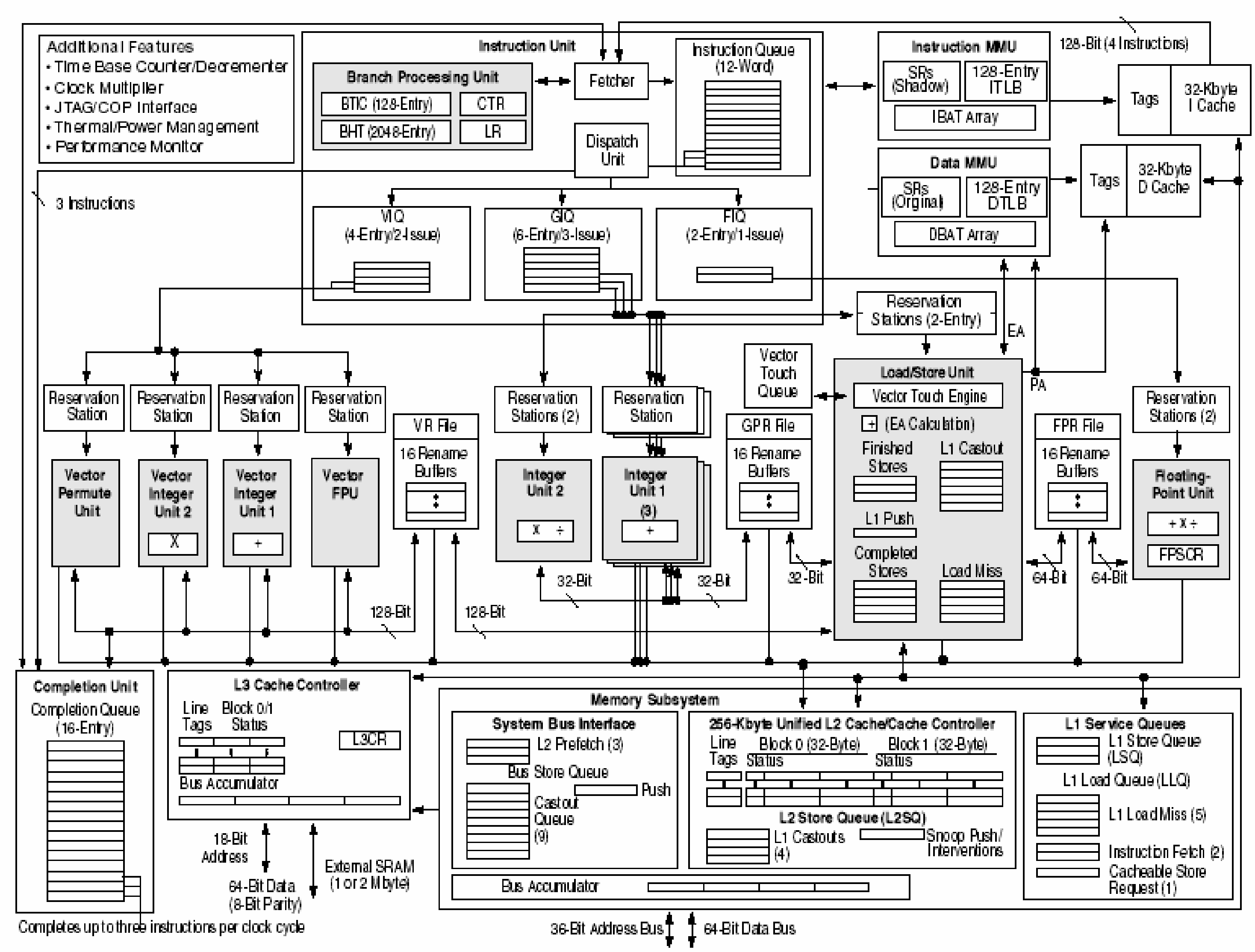
- AltiVec™ Technology transparently adds SIMD functionality to a high speed RISC engine
- AltiVec enables a broad range of embedded and computing applications
- C level programming offers certain level of comfort while providing powerful way to extract parallelism from applications
- You must think in terms of Vector Processing throughout design cycle of an application
  - AltiVec is not a pixie dust to be sprinkled on an existing code
- Given that – 2x-4x-11x speedup is possible

MPC7400



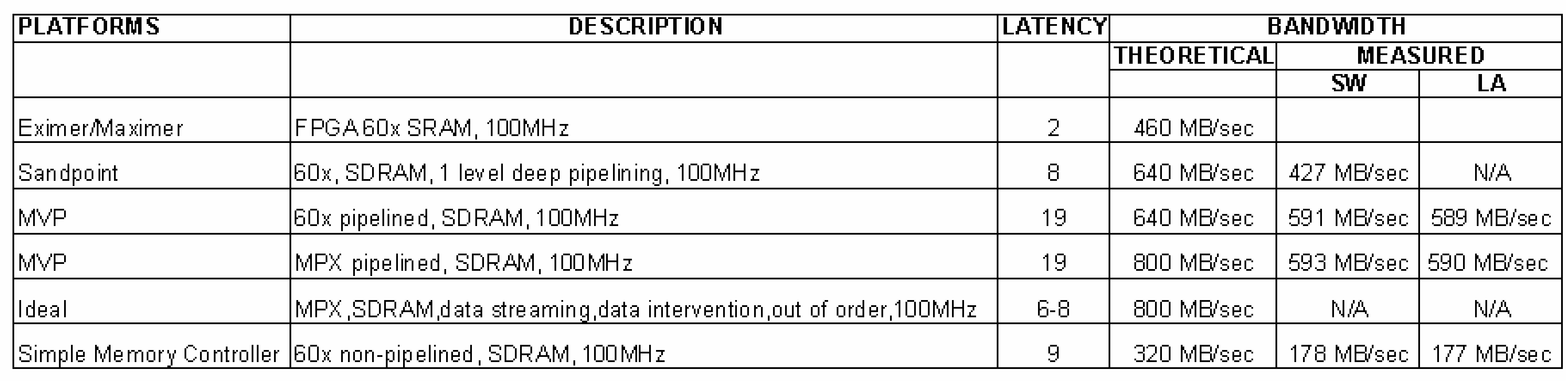
Additional Features
- Time Base Counter/Decrementer
- Clock Multiplier
- JTAG/COP Interface
- Thermal/Power Management
- Performance Monitor

MPC7450


Additional Features
• Time Base Counter/Decrementer
• Clock Multiplier
• JTAG/COP Interface
• Thermal/Power Management
• Performance Monitor
3 Instructions
Instruction Unit
Branch Processing Unit
BTIC (128-Entry)
CTR
BHT (2048-Entry)
LR
Fetcher
Dispatch Unit
Instruction Queue (12-Word)
VIQ (4-Entry/2-Issue)
GIQ (6-Entry/3-Issue)
FIQ (2-Entry/1-Issue)
Instruction MMU
SRs (Shadow)
128-Entry ITLB
IBAT Array
128-Bit (4 Instructions)
Tags
32-Kbyte I Cache
Data MMU
SRs (Original)
128-Entry DTLB
DBAT Array
Tags
32-Kbyte D Cache
Reservation Stations (2-Entry)
EA
PA
Reservation Station
Reservation Station
Reservation Station
Reservation Station
Vector Permute Unit
Vector Integer Unit 2
X
Vector Integer Unit 1
+
Vector FPU
VR File
16 Rename Buffers
Reservation Stations (2)
Integer Unit 2
X ÷
Reservation Station
Integer Unit 1 (3)
+
Vector Touch Queue
GPR File
16 Rename Buffers
Load/Store Unit
Vector Touch Engine
+ (EA Calculation)
Finished Stores
L1 Castout
L1 Push
Completed Stores
Load Miss
FPR File
16 Rename Buffers
Reservation Stations (2)
Floating-Point Unit
+ X ÷
FPSCR
128-Bit
128-Bit
32-Bit
32-Bit
32-Bit
64-Bit
64-Bit
Completion Unit
Completion Queue (16-Entry)
L3 Cache Controller
Line Tags
Block 0/1 Status
L3CR
Bus Accumulator
18-Bit Address
64-Bit Data (8-Bit Parity)
External SRAM (1 or 2 Mbyte)
Memory Subsystem
System Bus Interface
L2 Prefetch (3)
Bus Store Queue
Push
Castout Queue (9)
Bus Accumulator
256-Kbyte Unified L2 Cache/Cache Controller
Line Tags
Block 0 (32-Byte) Status
Block 1 (32-Byte) Status
L2 Store Queue (L2SQ)
L1 Castouts (4)
Snoop Push/ Interventions
L1 Service Queues
L1 Store Queue (LSQ)
L1 Load Queue (LLQ)
L1 Load Miss (5)
Instruction Fetch (2)
Cacheable Store Request (1)
Completes up to three instructions per clock cycle
36-Bit Address Bus
64-Bit Data Bus

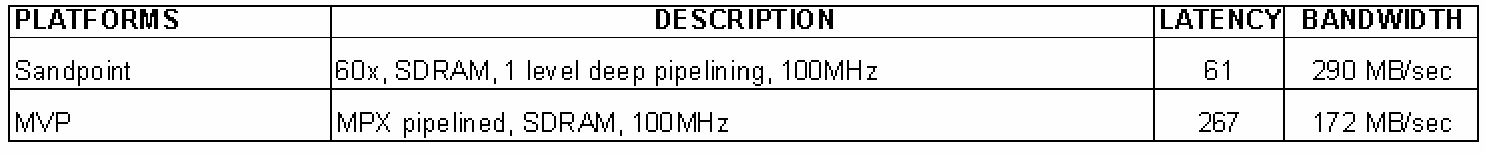
60x vs. MPX

- **Bandwidth**



PLATFORMS	DESCRIPTION	LATENCY	BANDWIDTH		
			THEORETICAL	MEASURED	
				SW	LA
Eximer/Maximer	FPGA 60x SRAM, 100MHz	2	460 MB/sec		
Sandpoint	60x, SDRAM, 1 level deep pipelining, 100MHz	8	640 MB/sec	427 MB/sec	N/A
MVP	60x pipelined, SDRAM, 100MHz	19	640 MB/sec	591 MB/sec	589 MB/sec
MVP	MPX pipelined, SDRAM, 100MHz	19	800 MB/sec	593 MB/sec	590 MB/sec
Ideal	MPX ,SDRAM,data streaming,data intervention,out of order,100MHz	6-8	800 MB/sec	N/A	N/A
Simple Memory Controller	60x non-pipelined, SDRAM, 100MHz	9	320 MB/sec	178 MB/sec	177 MB/sec

- **PCI I/O**



PLATFORMS	DESCRIPTION	LATENCY	BANDWIDTH
Sandpoint	60x, SDRAM, 1 level deep pipelining, 100MHz	61	290 MB/sec
MVP	MPX pipelined, SDRAM, 100MHz	267	172 MB/sec