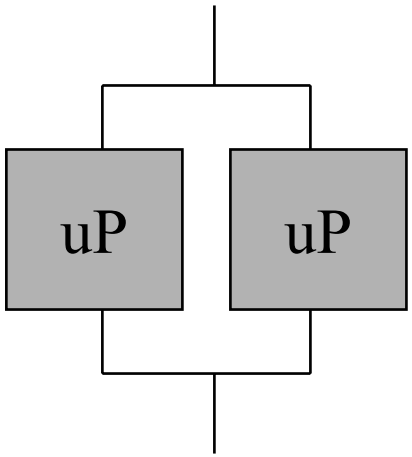
Improving Performance with AltiVec

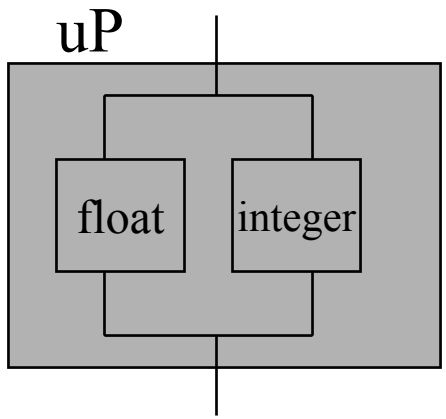
Motorola PowerPC Customer Support
Umair Yousufi
May 21, 2001

Parallelism takes different forms

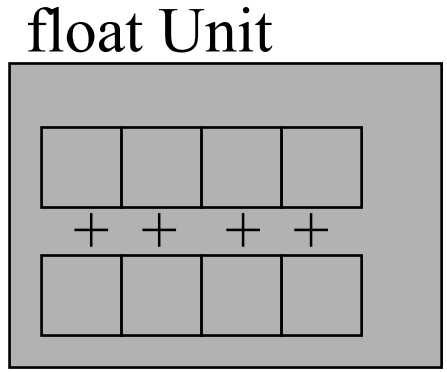
- AltiVec consists of instruction-level parallelism
- However, all current AltiVec-enabled PowerPCs use superscalar parallelism, and are capable of running in SMP configurations



SMP



SuperScalar



Instruction level

SMART NETWORKS DEVELOPER FORUM

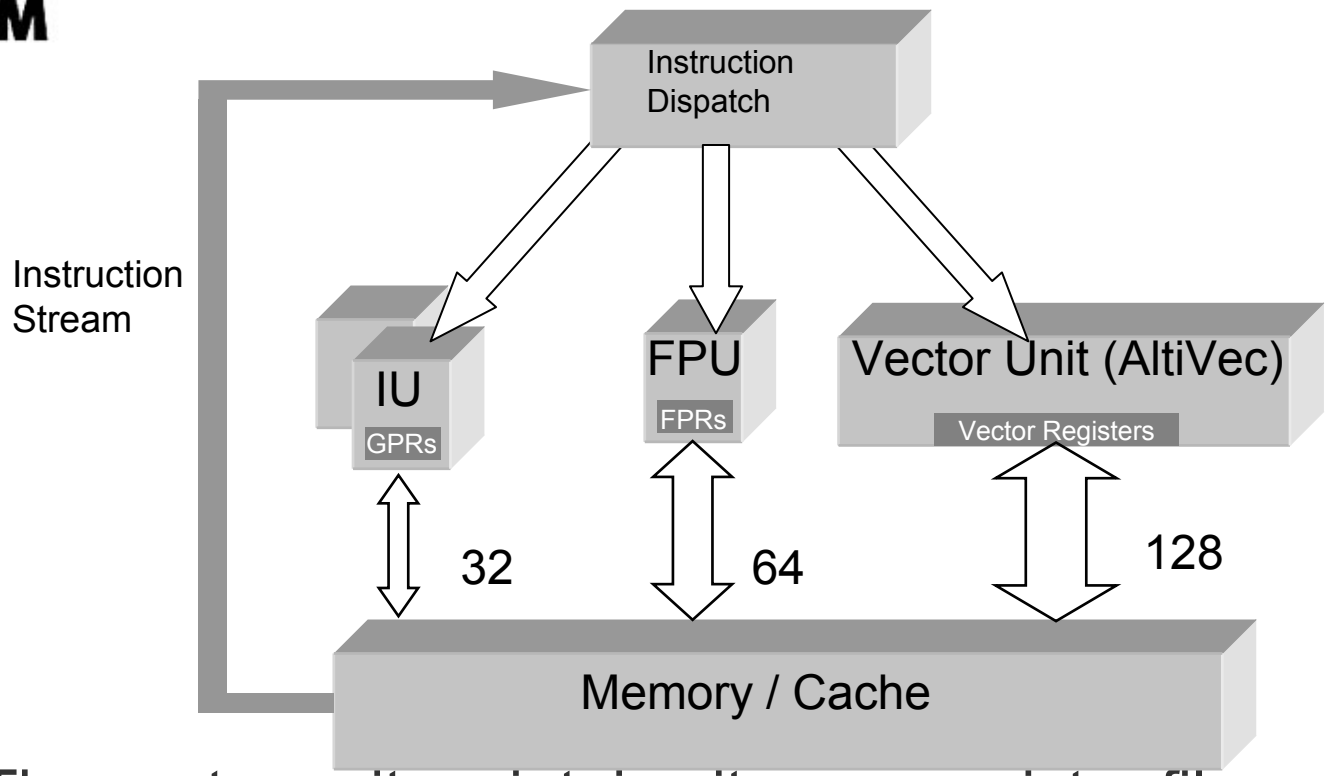
Agenda

- AltiVec Description
- General Examples
  - FIR
  - RMAX
  - Copying
- Networking Examples
  - Packet Flow

AltiVec is a type of instruction parallelism

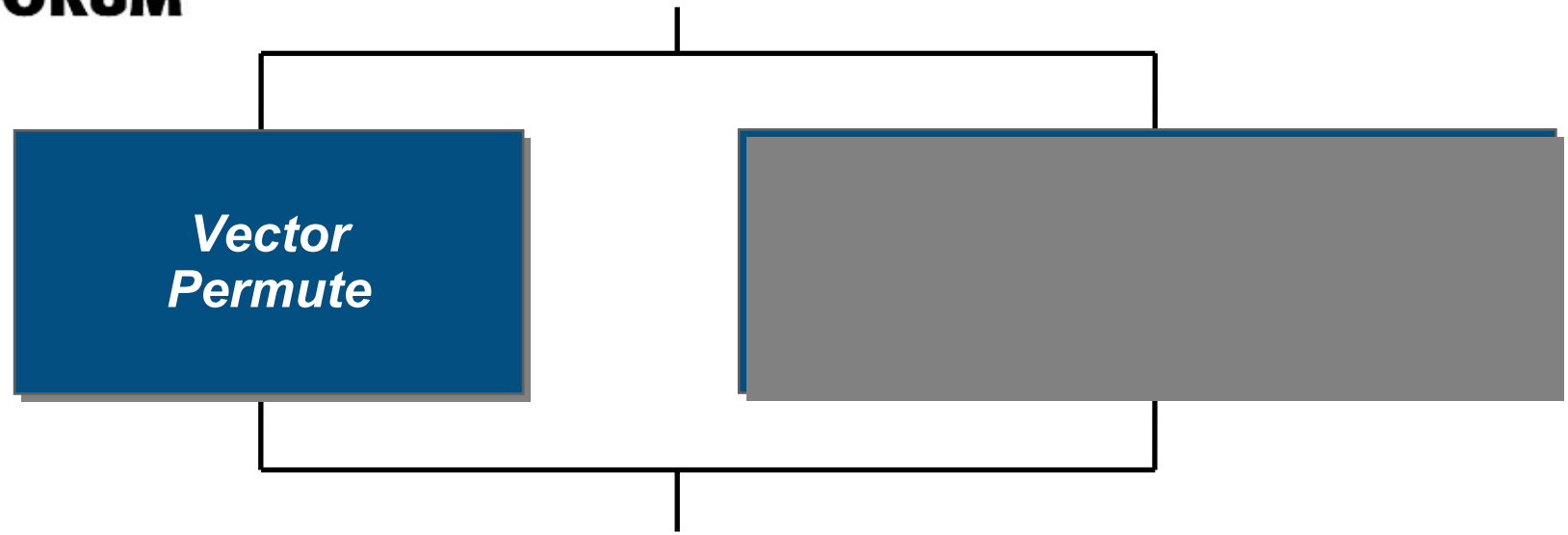
- Single instruction/ single data stream (SISD) - a sequential computer.
- Single instruction/ multiple data streams (SIMD) - e. g. an array processor. (This is AltiVec)
- Multiple instruction/ single data stream (MISD) - unusual.

Vectors use another Execution Unit



- The vector unit maintains its own register file
  - …so no penalty for mingling AltiVec with regular instructions
  - …but must go through memory to transfer data from vector registers to general purpose or floating-point registers
    - not a major concern

The Vector Unit has subunits

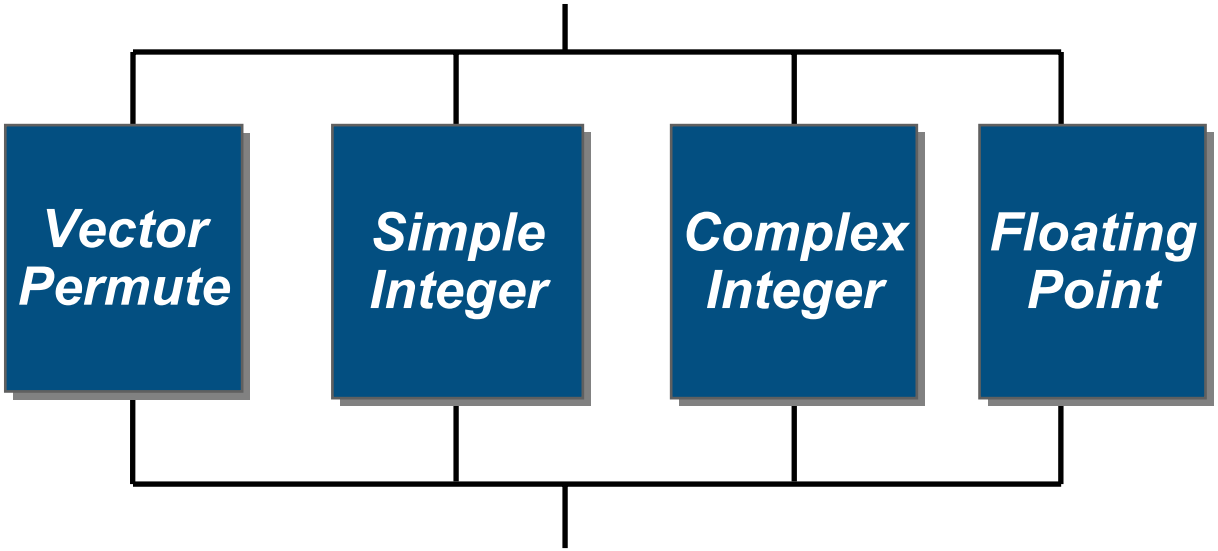


- Subunits within Vector Unit are implementation-dependent
  - — Above shows the 7400 AltiVec execution unit.

  Q: Can two instructions go to a 7400 AltiVec unit in a single clock?

  A: Yes, but only if one uses the ALU subunit and the other uses the Permute subunit

AltiVec subunits in 7450

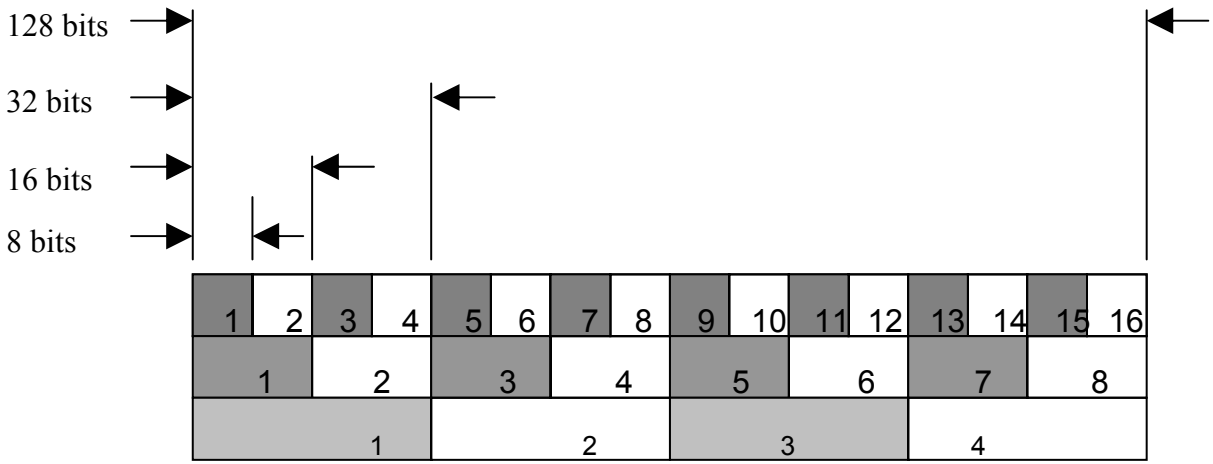


- Each functionality is broken into it's own subunit.

  Q: ***Now*** how many instructions can go to the 7450 Vector Unit unit in a single clock?

AltiVec uses common data types

- AltiVec operates on 128 bits
- Bit interpretation depends on instruction
- Other types include:
  - Pixel
  - Bool

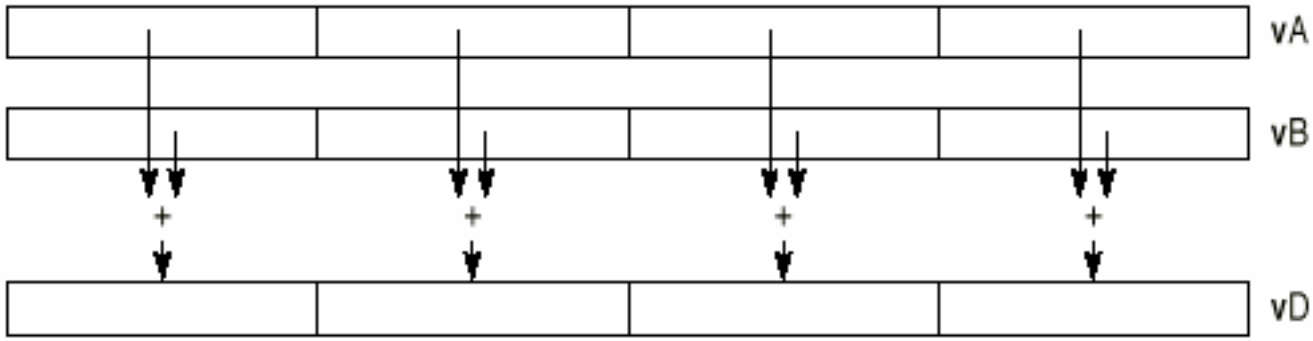


Number in Vector	Bit Length	Type	Possible Range
16	8	char	0 …255
		signed char	-128…127
8	16	short	0...64K
		signed short	-32K…32K
4	32	int	$0…2^{32}$
		unsigned int	$-2^{31}…2^{31}$
		float	IEEE Float

VEC_ADD is a typical AltiVec instruction

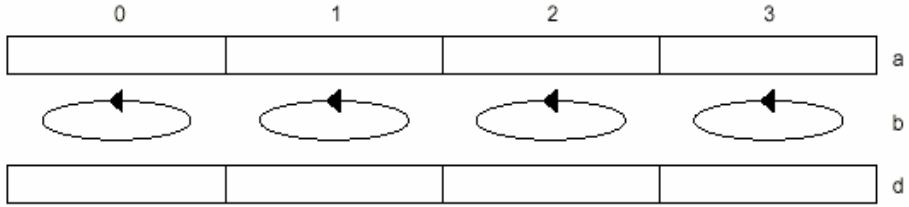
- Code would look like this:

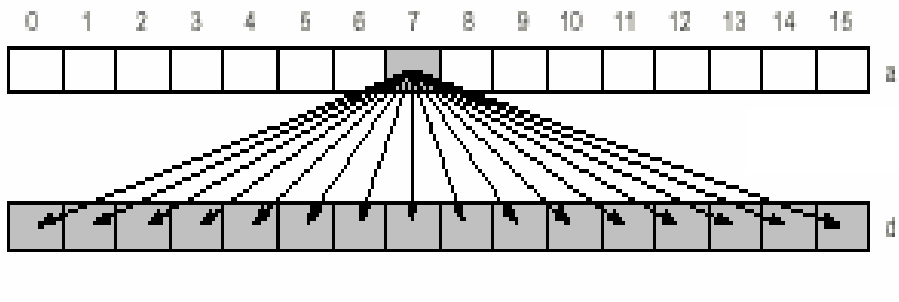
```
vector float vD, vA, vB;
vD = vec_add(vA, vB);
```

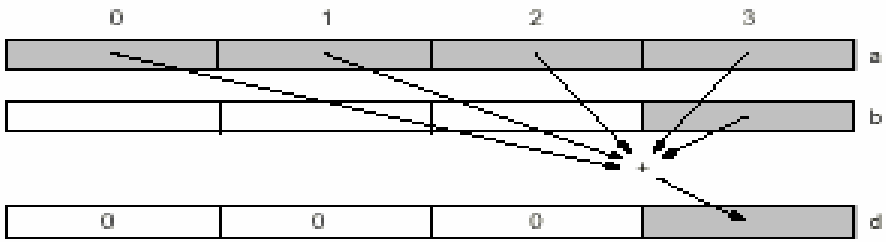


- Use the same instruction for types ***char***, ***short***, and ***int***
  - Compiler checks type and maps to correct assembly instruction

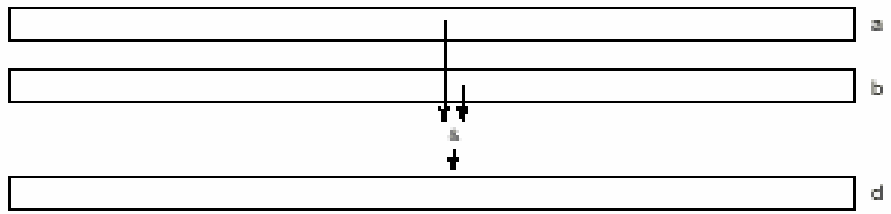
Which is which?







- 1) Logical AND *(vec_and)*
- 2) Intra-element summation *(vec_sums)*
- 3) Rotate Left *(vec_rl)*
- 4) Vector splat *(vec_splat)*



An FIR Filter is a common DSP algorithm

- **In words**:

  A Finite Impulse Response (FIR) filter produces an output, y(n), that is the weighted sum of the current and past inputs, x(n).

- **In symbols:**

  x = current and past inputs

  b = coefficients of filter

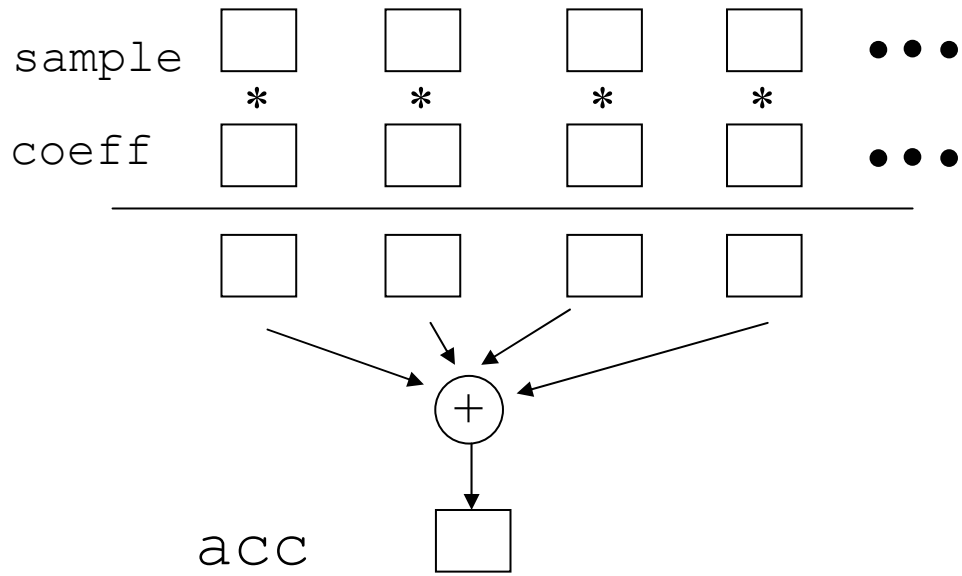
$$y_n = \sum_{i=0}^{N} (b_i) * (x_{n-i})$$

- **In real world applications**:
  - Used to implement filters (e.g. lowpass, bandpass) without introducing phase distortion

An FIR in C is straightforward

- The scalar version:

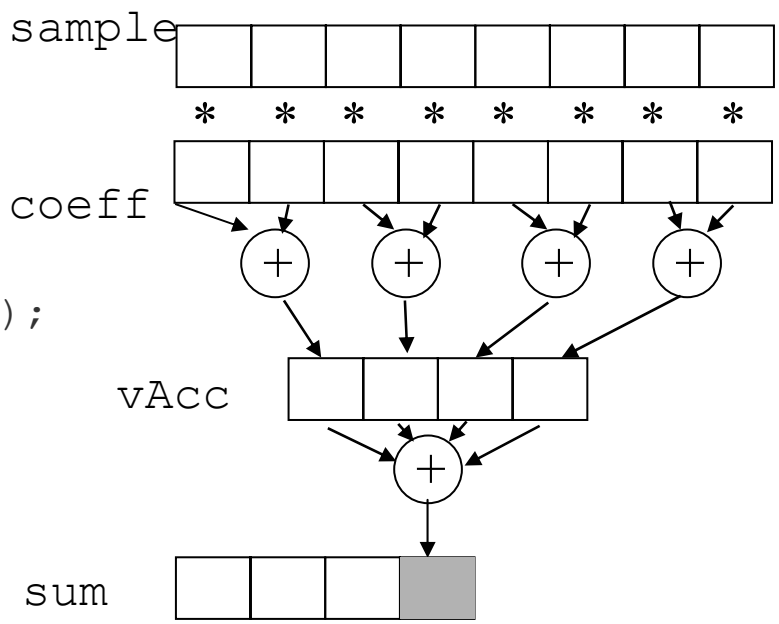
```
for (i = 0; i < N; i++)
    {
        acc = acc + (sample[i] * coeff[i]);
        //... just a multiply-accumulate
    }
    return round (L_tmp);
```



Vector conversion of an FIR is easy

- The vector version:

```
for (i = 4; i <= N/8; i++)
{
        vAcc.v = vec_msums(vSample[i], vCoeff[i], vAcc.v);
       //vec_msums is a fused multiply accumulate instruction
}
sum.v = (vector signed short)
         vec_sums(vAcc.v,
                  (vector signed int)(0));
sum.v = (vector signed short)
        vec_sums(vAcc.v,
                 (vector signed int)(sum.v));
return sum.s[6];
```



Vector FIR performance is 20x faster

- FIR is exactly the functionality AltiVec excels at:

Vectors in Data Set	Unoptimized scalar*	Vector*	SPEEDUP
800	51204	2226	23.00

(*clocks - lower is better)

- Q: Why is the speed up not 8x exactly?
- A: A number of reasons:
  - AltiVec vec_madd performs more than 8 math operations
  - Computations speed up more than Loads/Stores
  - Speedup can vary with size (esp. if larger than cache)
  - Compiler may optimize code differently

Vector FIR introduces vectorization topics

- Unions
  - Some snippets of code

```
vAcc.v = vec_msums(vSample[i], vCoeff[i], vAcc.v);
return sum.s[6];
```

  - The union is defined as

```
union vec_i
{
        int i[4];
        vector signed int v;
};
```

  - Provides 2 key advantages
    - Automatically ensures alignment
    - Allows individual access of vector elements

Vector FIR introduces vectorization topics

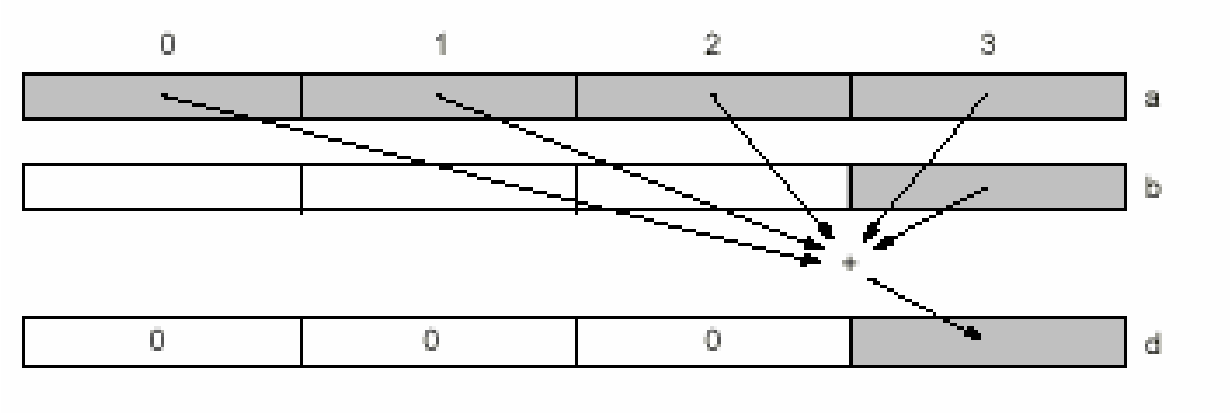
- Q: What if N is not a multiple of 8?
- A: Scalar code (not shown) must handle the remainder

- Q: What if data is not aligned?
- A: Two options:
  - 1. Handle non-aligned data at beginning with scalar code, continue with vector, then finish at end with scalar code
  - 2. Use permute instructions to align data on the fly

  - Best solution depends on the actual algorithm
  - There is a slowdown associated, but the penalty can be amortized in longer loops.

Intra-element AltiVec operations are unique

- vec_sums is a unique AltiVec instruction. It operates on elements ***within*** a vector
  - Most instructions are point-wise ***inter***-element operations



- Limited, because it does not operate on all data types.
- Functions relying heavily on intra-element calculations are difficult to vectorize and may not yield any speed gains even if vectorized.
  - Workarounds: 1. Reorganize data 2.Restructure algorithm order

RMAX searches arrays for maximum

- Scalar Version

```
signed short rmax (signed short vec[], int N)
{
    for (i = 0; i < N; i++)
    {
        if (searchArray[i] > max)
        {
            max = searchArray[i];
        }
    }
    return max;
}
```

- Description
  - Returns the ***value*** of the largest value in an array
- Used to search databases
- No notion of smarter algorithmic searches
  - Binary search, hashing, etc...

SMART NETWORKS DEVELOPER FORUM

Vector RMAX looks longer

```
signed short vrmax(vector signed short v1[], int N) {
        N = N/8;
        maxVec.v = v1[0];
        for(i=1; i<N; i++)
        maxVec.v = vec_max(v1[i], maxVec.v);
        //The largest eight maximum values are now located in maxVec
        //Shift right 8 bytes to split maxVec in half and compare
        maxVecTop.v = vec_sro(maxVec.v, (vector unsigned char )(64));
        maxVec.v = vec_max(maxVec.v, maxVecTop.v);
        //The largest 4 values are in the upper half of maxVec
        //Shift right 4 bytes to split maxVec in half again and compare
        maxVecTop.v = vec_sro(maxVec.v, (vector unsigned char)(32));
        maxVec.v = vec_max(maxVec.v, maxVecTop.v);
        //The largest two values are in the top of maxVec.  Extract
        //with scalar code.
        if (maxVec.s[6] > maxVec.s[7])
           return (maxVec.s[6]);
        else
           return (maxVec.s[7]);
}
```

Despite more instructions, vRMAX is faster

- Vec_max contributes greatly to the speedup

Vectors in Data Set	Unoptimized scalar*	Vector*	SPEEDUP
500	60041	2569	23.37

(*clocks - lower is better)

- Other variations on RMAX
  - Finding the index of the largest value instead of just the value
    - You'll see that our use of vec_max does not record the index
    - The scalar version of this adds just one line
    - This has been implemented and the performance is about 9x

VEC_MAX extracts largest values from vectors

- Start by adjusting loop inde. This is common with AltiVec esp. if the external interface must stay the same

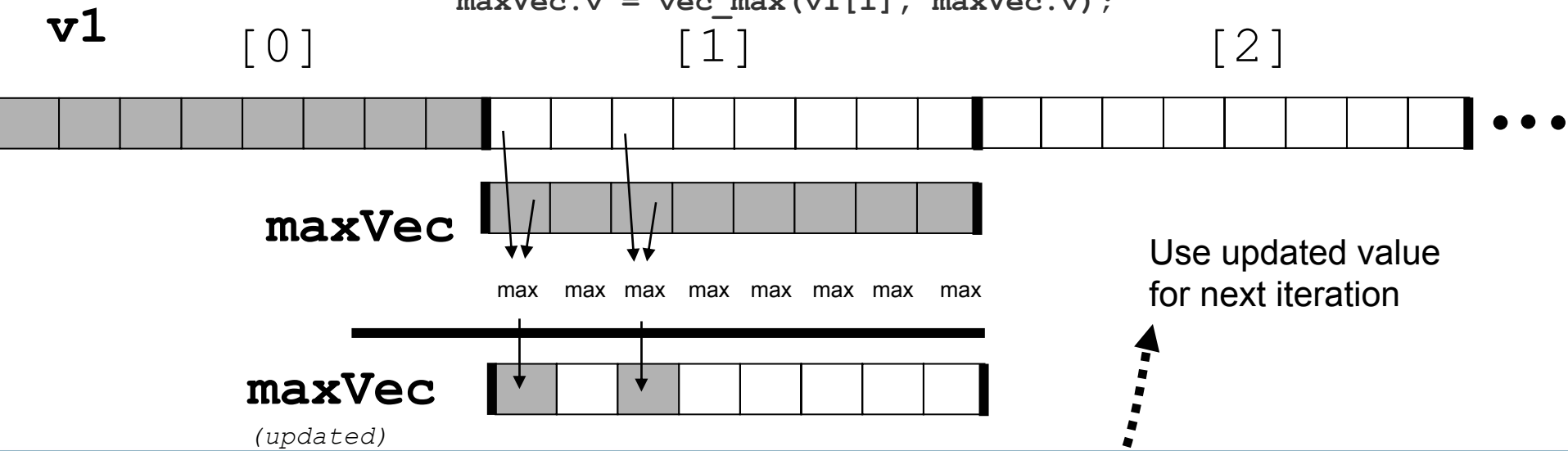
```
N = N/8;
```

- Then assign maxVec to v1[0]

```
maxVec.v = v1[0];
```

- Now begin main loop

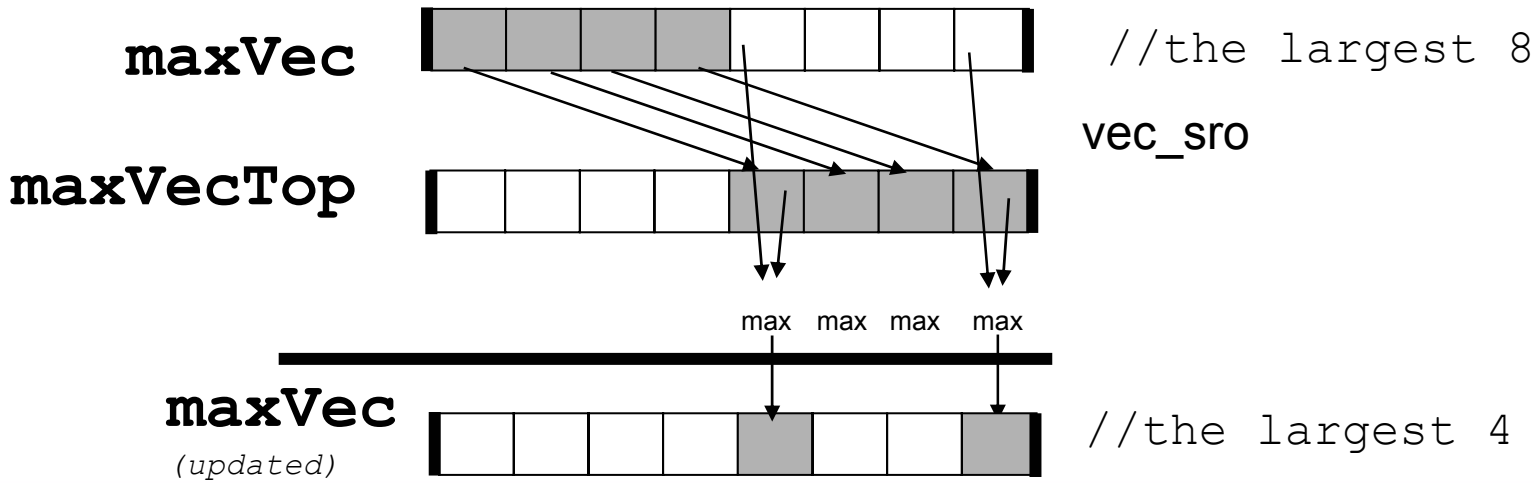
```
for(i=1; i<N; i++)
        maxVec.v = vec_max(v1[i], maxVec.v);
```



After main loop, must do fine tuning

- After the main loop completes, **maxVec** holds the largest eight numbers from the array.
- Find Maximum value **within** maxVec
  - ***What does this sound like***?
  - Common technique:
    - Copy elements to another array and do point-wise operations

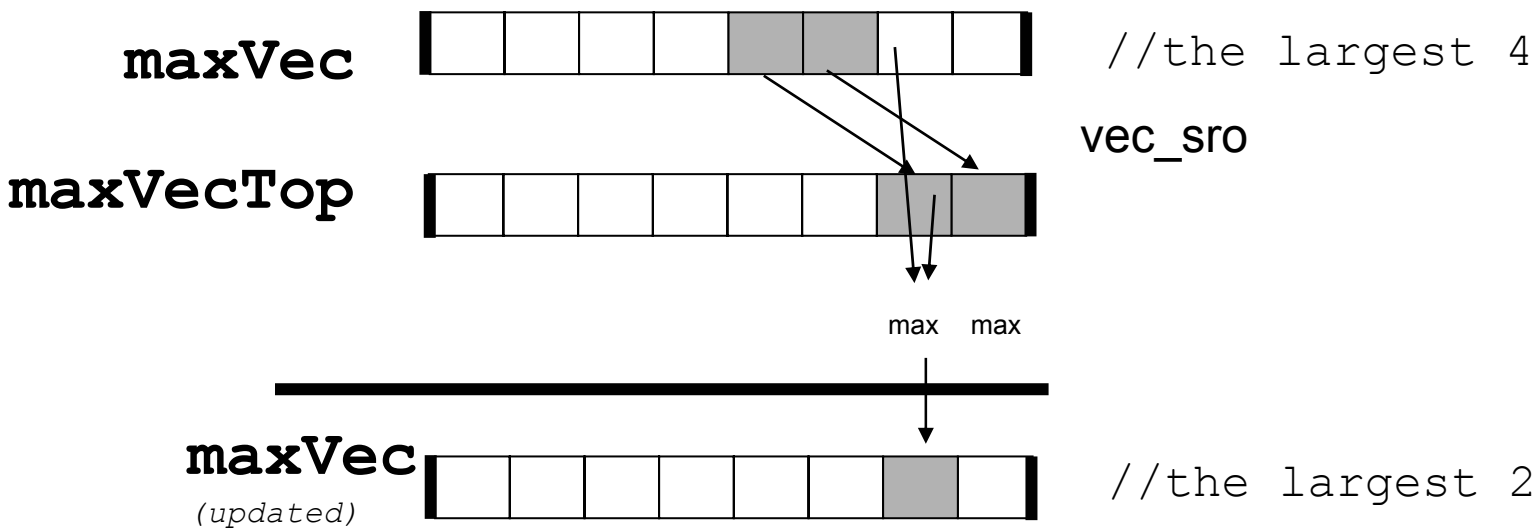
```
maxVecTop.v = vec_sro(maxVec.v, (vector unsigned char )(64)); //SR8byte
maxVec.v = vec_max(maxVec.v, maxVecTop.v);
```



Scalar code finishes RMAX

- Can go one step further with vector code first

```
maxVecTop.v = vec_sro(maxVec.v, (vector unsigned char )(32));  //SR4byte
maxVec.v = vec_max(maxVec.v, maxVecTop.v);
```



- Scalar code can select between last values

```
if (maxVec.s[6] > maxVec.s[7])
          return (maxVec.s[6]);
       else
          return (maxVec.s[7]);
```

RMAX (index)

- Scalar Version

```
for (i = 0; i < N; i++)
    {
        if (searchArray[i] > max)
        {
            max = searchArray[i];
            max_idx = i;
        }
    }
    return max_idx;
```

- Description
  - Returns the ***index*** of the largest value in an array
- No notion of smarter algorithmic searches
  - Binary search, hashing, etc...

Non-vector functions benefit

- A trivial scalar copy:

```
for (i = 0; i < N; i++)
    out[i] = inp[i];
```

- An optimized **scalar** copy:

```
union double_u
{
        short s[4];
        double f;
};
void sblock (union double_u *in, int N, union double_u *out)
{
    unsigned int i;
    N = N / 4;
    for (i = 0; i < N; i++)
        out[i].f = in[i].f;            //64-bits!
}
```

Copying takes advantage of 128bit pathways

- The optimized scalar uses a union to enforce a 64-bit load/store

- Vector version takes this one step further and uses 128-bit load/store

```
void vblock(vector signed short inp[], int N, vector signed
   short out[])
{
        N = N >> 3;

      for (i=0; i< N; i++)
            out[i] = inp[i];                    //128-bits!
}
```

Copying performance matches load size

Vectors in Data Set	Unoptimized scalar	Optimized Scalar	Vector	SPEEDUP
1000	23895	6087	2738	8.73

- Speed gains almost exactly match sizes of relative data types
- Unrolling the loops would likely increase performance of all copies

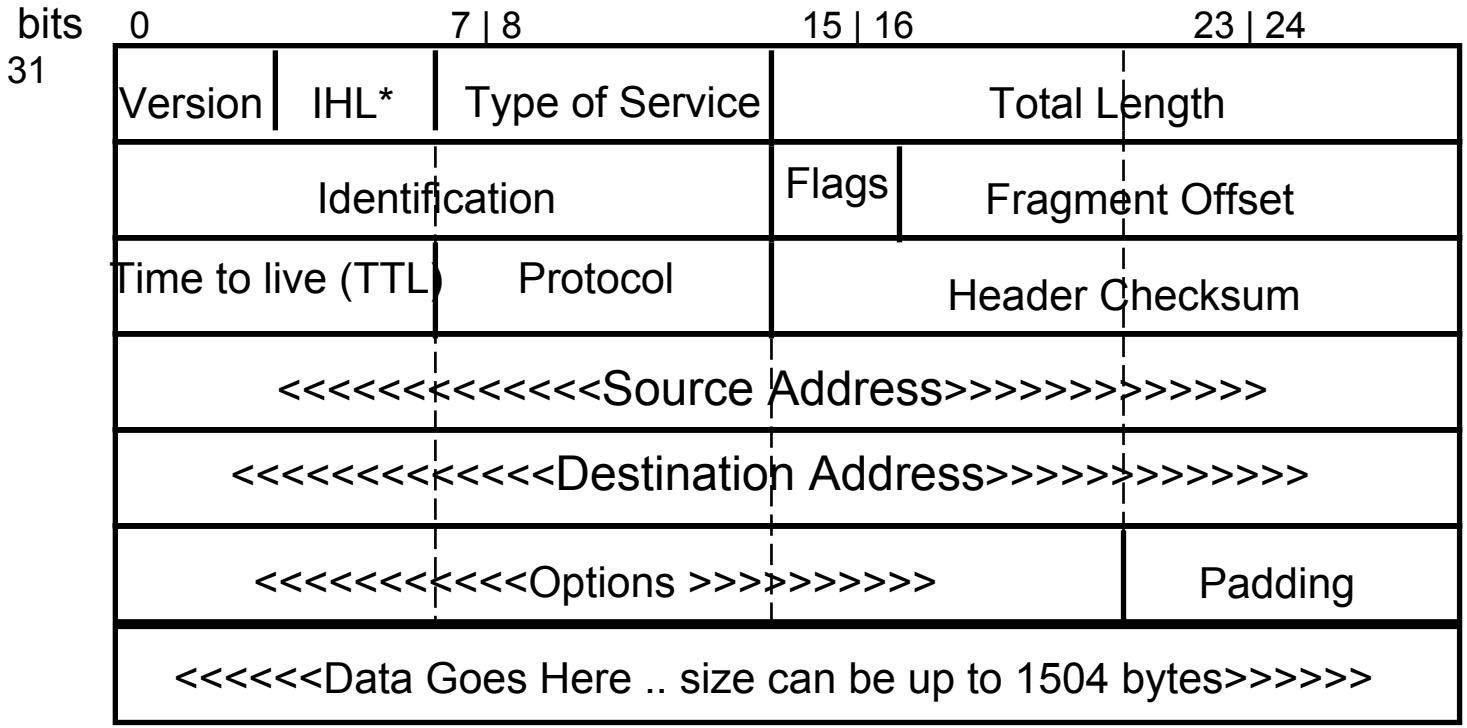
Packet flow is a networking algorithm

- Packet flow: the process of receiving, processing and forwarding packets.
  - RFC1812

- Packet (aka datagrams): a piece of a message transmitted over a packet-switching network.
  - Contains:
    - destination address
    - data

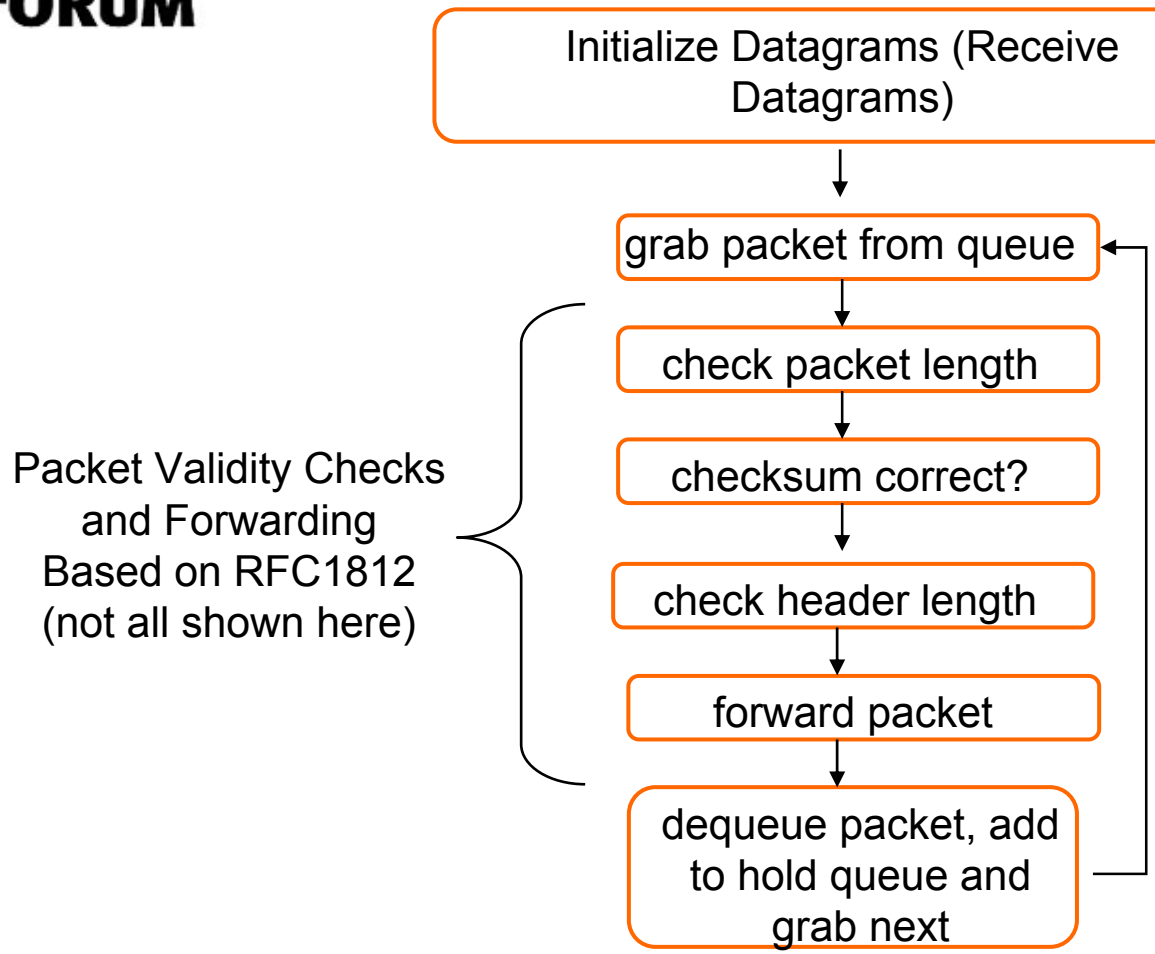
- A packet-switching network is a network in which a message is divided into small packets and sent over the network in possibly different paths.

A packet at a glance



*IHL = IP Header Length

Packet Flow checks headers first



Data Reorganization is added to basic vectorization strategy

- Operations performed on a packet are independent from the operations performed on other packet.
  - this allows vectorization
- Since most of the operations (if not all) are performed on a byte size elements, sixteen elements (packets) can be processed in parallel at a time.
- Requires an ***internal*** reorganization of data

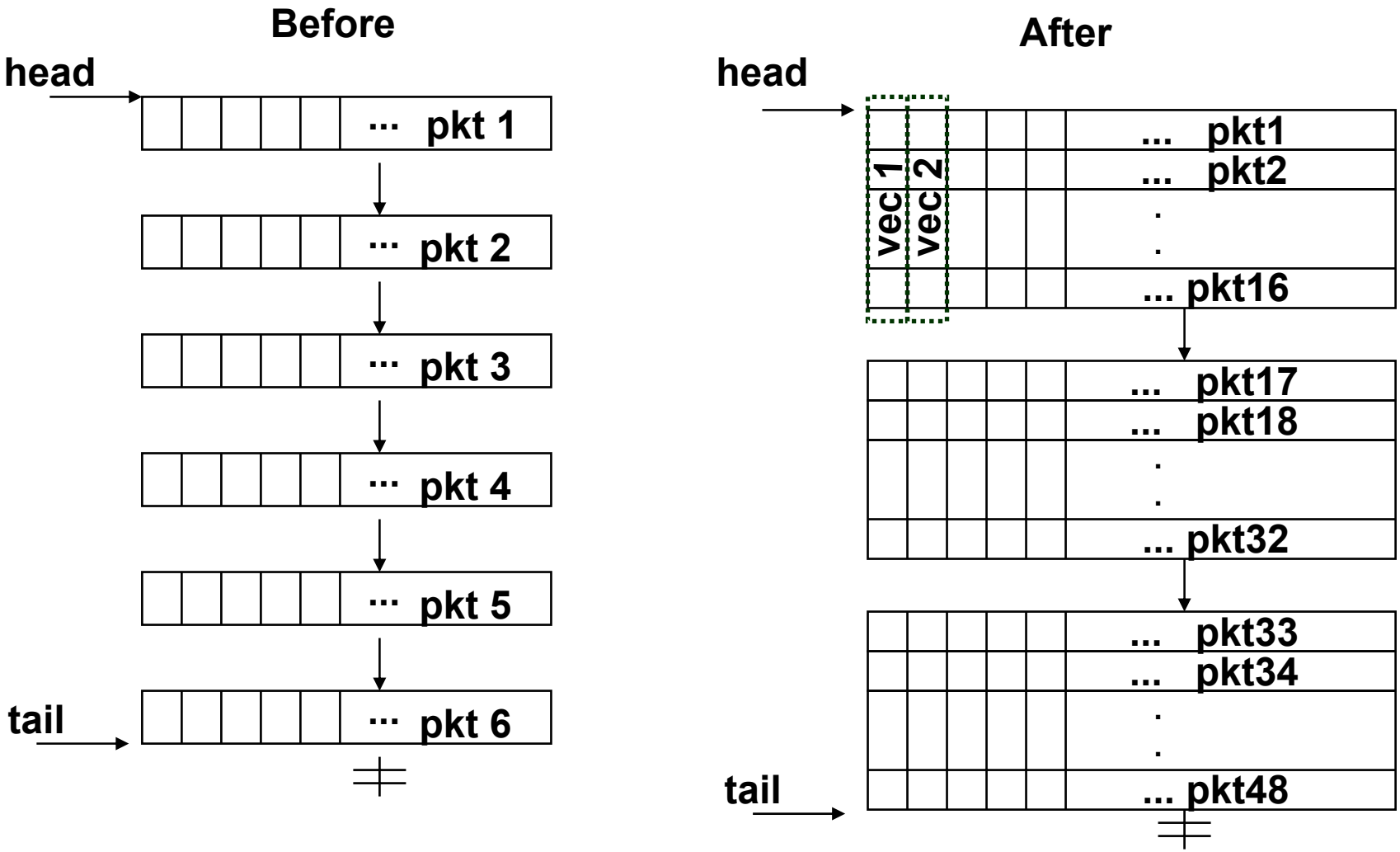
SMART NETWORKS DEVELOPER FORUM

Vector version has caveats

- Packet length restrictions
- Buffering
- Bad packet identifications

SMART NETWORKS DEVELOPER FORUM

Reorganized data looks transposed



MOTOROLA

SMART NETWORKS DEVELOPER FORUM

Original Packets are in rows

```
00: 45|08|00|30|00|01|00|00|0f|06|ef|72|c0|a0|9d|85|
    c0|a0|9d|86|00|00|00|00|00|00|00|00|00|00|00|00|
    00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|
                        .. length = 48
01: 45|08|00|30|00|02|00|00|0f|06|ef|71|c0|a0|9d|85|
    c0|a0|9d|86|00|00|00|00|00|00|00|00|00|00|00|00|
    00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|
                        .. length = 48


11: 45|08|05|e0|00|12|00|00|0f|06|e9|b1|c0|a0|9d|85|
    c0|a0|9d|86|00|00|00|00|00|00|00|00|00|00|00|00|
    00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|
                      .. length = 1504
12: 45|08|05|e0|00|13|00|00|0f|06|e9|b0|c0|a0|9d|85|
    c0|a0|9d|86|00|00|00|00|00|00|00|00|00|00|00|00|
    00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|
                      .. length = 1504
```

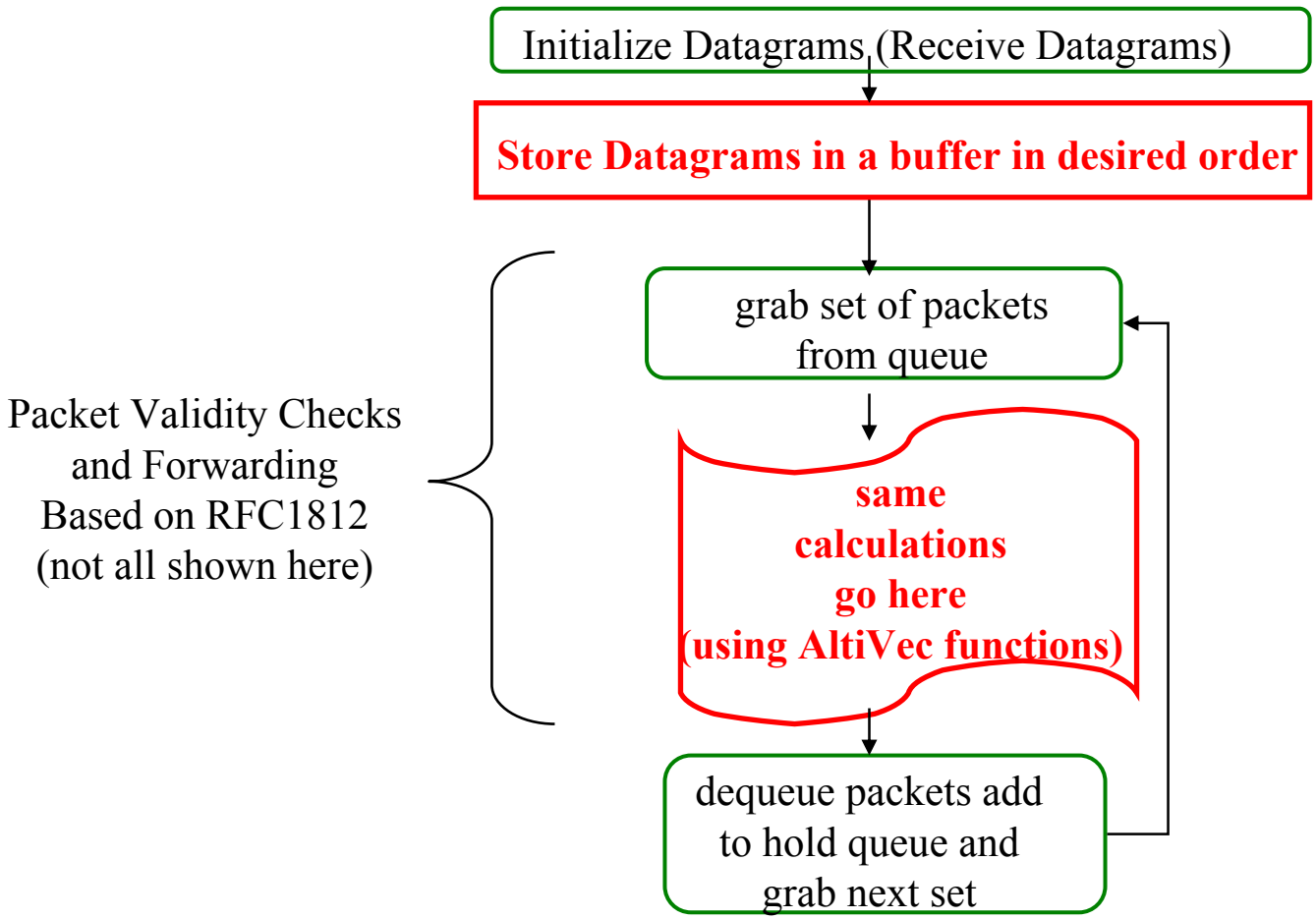
SMART NETWORKS DEVELOPER FORUM

Reordered packets are in columns

```
00: 45|45|45|45|45|45|45|45|45|45|45|45|45|45|45|45|
    08|08|08|08|08|08|08|08|08|08|08|08|08|08|08|08|
    00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|
    30|30|30|30|30|30|30|30|30|30|30|30|30|30|30|30|
    00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|
    01|02|03|04|05|06|07|08|09|0a|0b|0c|0d|0e|0f|10|
    00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|
    00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|
    0f|0f|0f|0f|0f|0f|0f|0f|0f|0f|0f|0f|0f|0f|0f|0f|
```

```
01: 45|45|45|45|45|45|45|45|45|45|45|45|45|45|45|45|
    08|08|08|08|08|08|08|08|08|08|08|08|08|08|08|08|
    05|05|05|05|05|05|05|05|05|05|05|05|05|05|05|05|
    e0|e0|e0|e0|e0|e0|e0|e0|e0|e0|e0|e0|e0|e0|e0|e0|
    00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|
    11|12|13|14|15|16|17|18|19|1a|1b|1c|1d|1e|1f|20|
    00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|
    00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|
    0f|0f|0f|0f|0f|0f|0f|0f|0f|0f|0f|0f|0f|0f|0f|0f|
    06|06|06|06|06|06|06|06|06|06|06|06|06|06|06|06|
```

Reorganization precedes vectorization



SMART NETWORKS DEVELOPER FORUM

Check #1: Total length >= 20 bytes

Description: Header validity check #1: Total length >= 20 bytes (minimum legal IP datagram)?

Scalar:

```
if ( ((*(byteptr+2)) << 8) + (*(byteptr+3)) < 20 )
  {
   printf( "Error: packet length too small to hold minimum 20-
   byte IP header\n");
    badpkt |= 1;
  }
```

SMART NETWORKS DEVELOPER FORUM

Vector implementation looks longer

```
miscChar1 = loadVector((unsigned char*)(vbyteptr+2));
miscChar2 = loadVector((unsigned char*)(vbyteptr+3));

miscShrt1 = vec_mulo(miscChar1, (vector unsigned char)(1));
miscShrt2 = vec_mulo(miscChar2, (vector unsigned char)(1));

iplen1 = vec_sl(miscShrt1, (vector unsigned short)(8));
iplen1 = vec_add(iplen1,miscShrt2);

miscShrt1 = vec_mule(miscChar1, (vector unsigned char)(1));
miscShrt2 = vec_mule(miscChar2, (vector unsigned char)(1));

iplen2 = vec_sl(miscShrt1, (vector unsigned short)(8));
iplen2 = vec_add(iplen2, miscShrt2);
if( vec_any_lt(iplen1, (vector unsigned short)(20)) ||
    vec_any_lt(iplen2, (vector unsigned short)(20)) )
{
  printf( "Error: packet length too small to hold minimum 20-byte IP header\n");
  badpkt |= 1;
}
```

SMART NETWORKS DEVELOPER FORUM

Check #2: IP Checksum Correct

- **Description**: Header validity check #2: IP Checksum correct?

- **Scalar**:

```
//count is computed as header length in 16 bit words
for ( count = ((*(hdrptr)) & 0xf) * 2; count > 0; count--, addr++ )
  sum += *addr;

//Fold the 32-bit accumulator value down to a 16-bit sum
while ( sum >> 16 )
  sum = (sum & 0xffff) + (sum >> 16);

The one's compliment of the 16 bit checksum should be all FF's
  return ( (unsigned int)(sum) ) == ((unsigned int)(-1));
```

Vector IP Checksum Correct - declaration of variables

```
int count;
vector unsigned char PERMUTER1 =
  (vector unsigned char)(0x00,0x10,0x01,0x11,0x02,0x12,0x03,0x13,
                         0x04,0x14,0x05,0x15,0x06,0x16,0x07,0x17);

vector unsigned char PERMUTER2 =
  (vector unsigned char)(0x08,0x18,0x09,0x19,0x0a,0x1a,0x0b,0x1b,
                         0x0c,0x1c,0x0d,0x1d,0x0e,0x1e,0x0f,0x1f);

vector unsigned short sum = (vector unsigned short)(0);
vector unsigned short addr1, addr2, tmpSum;
vector unsigned char* addr = vhdrptr;
```

Vector IP Checksum Correct - Algorithm

```
for ( count = ((*((unsigned char*)vhdrptr)) & 0xf) * 2;
      count > 0; count--, addr+=2 )
 {
      addr1 = (vector unsigned short)vec_perm(*addr, *(addr+1),
   PERMUTER1);
      addr2 = (vector unsigned short)vec_perm(*addr, *(addr+1),
   PERMUTER2);
      tmpSum = vec_adds(addr1, addr2);
      sum = vec_adds(tmpSum, sum);
 }

return (vec_all_eq(sum, (vector unsigned short)(-1)));
```

DST works for large, uniform data

- DST = data stream touch
- If you know exactly what (and when!) data will be needed, DST can help prefetch data into the L1 cache to prevent lengthy memory accesses.
- DST uses idle bus cycles, so it doesn't negatively impact other higher priority uses of the bus

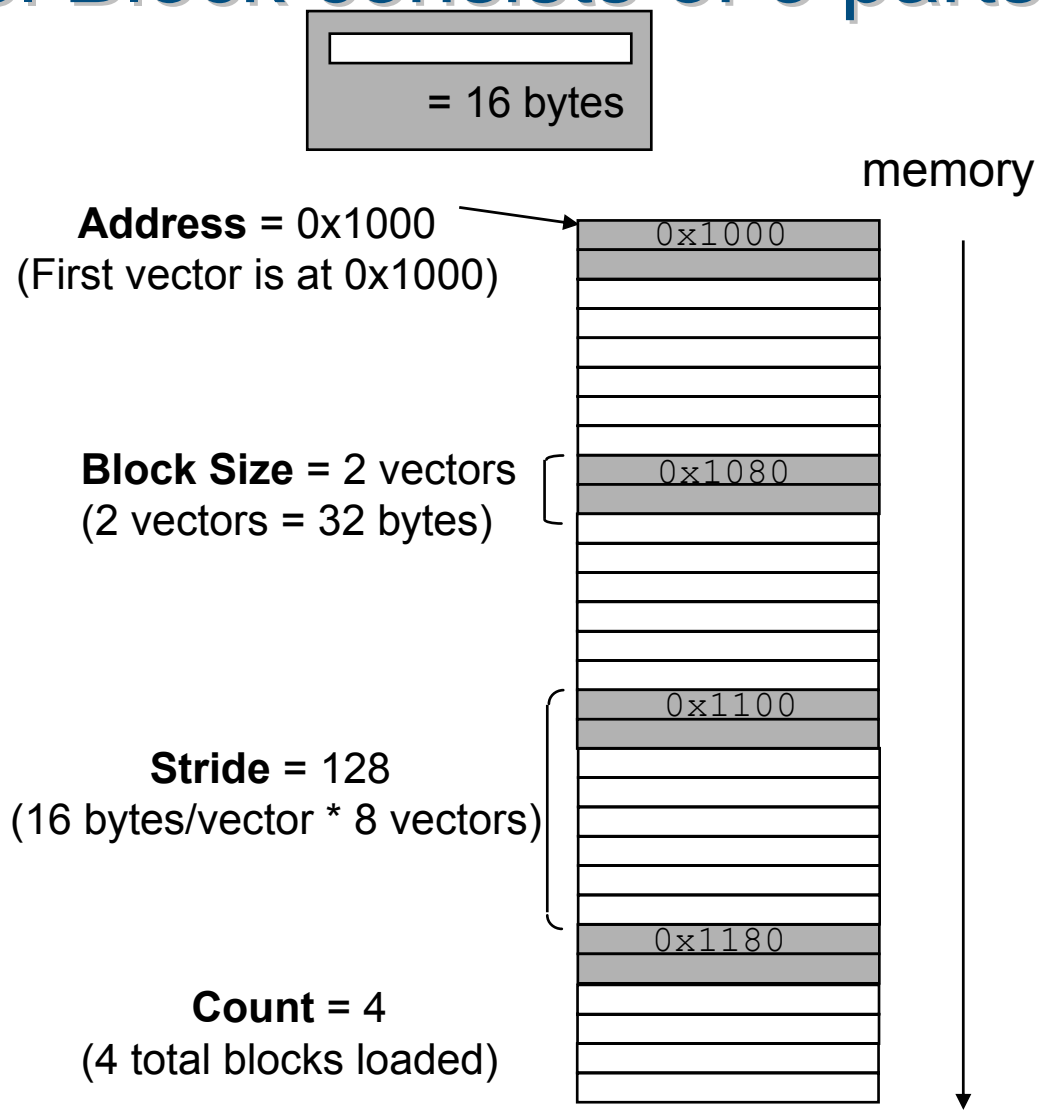
Format of the DST instruction

```
vec_dst(int *addr, dstControlBlock, streamID);
```

- ***Addr*** is where the prefetching should begin
- ***dstControlBlock*** must be generated with the user's desired parameters
- ***streamID*** is a value between 0-3
  - This allows upto 4 prefetch streams running concurrently

DST Control Block consists of 3 parts

- Control block has three parts
  - ***Block Size***
    - Size of the blocks to be loaded into the cache
  - ***Stride***
    - Bytes between beginning of one block to beginning of another
  - ***Block Count***
    - Total Number of blocks to prefetch



Functions exist to generate Control Block

- Returns proper control word:

```
inline UInt32 GetPrefetchConstant(      int blockSizeInVectors,
                                        int blockCount,
                                        int blockStride )
{
  return        ((blockSizeInVectors << 24) & 0x1F000000) |
                ((blockCount << 16) & 0x00FF0000)         |
                (blockStride & 0xFFFF);
}
```

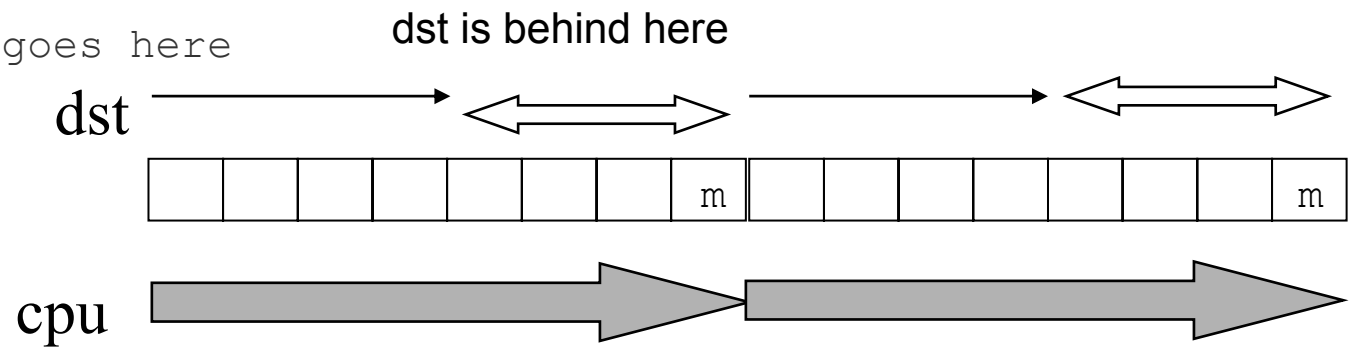
- The stride must be larger than the block size, otherwise redundant loads will occur
- Why is stride needed?

DST often falls behind CPU

- This falls into a “producer/consumer” problem
  - CPU consumes the data
  - DST must produce it
- What does CPU do if DST falls behind?
  - CPU doesn’t wait
  - It fetches the data anyway
- What happens to the DST stream after it falls behind?
  - It keeps trying to prefetch, only to find the data is already in the L1!

Reissue DST to catch up

```
// m*n = size of total array
for (i=0; i<n; i++)
{
   dst(addr, cBlock, iD)  //Reissue every m operations
   for (j=0; j<m; j++)
   {
        //code goes here
   }
}
```



- Should stagger DST. Current DST should fetch for **next** iterations
  - Going too far ahead means risking your data may get thrown out

DST performance varies with many factors

- Difficult to know when to reissue, even with sim_g4:

Normal*	Vec Stream*	Speedup
40,098,642	112,063,652	1.79469943

* sums/second - higher is better

- Adding streams sometimes helps, sometimes doesn't:

Num Streams	Num Cycles**	Speedup
0	3996516	n/a
1	3245525	1.23
2	2624000	1.52
3	3137354	1.27
4	2763743	1.45

** lower is better