Multi-media Extensions in Super-pipelined Micro-architectures. A New Case for SIMD Processing ?

Marco Ferretti

Dipartimento di Informatica e Sistemistica
University of Pavia, Italy
marco.ferretti@unipv.it

Talk Outline

- Introductory remarks
- SIMD Processing - an old paradigm
- Media Processing
- Multimedia extensions in GPP
- Instruction classes
- A brief tour
- Practical issues
- Concluding remarks

Introductory remarks

- General purpose computing vs embedded domain processing
- Workload: media processing - still images, video, audio, graphics
- Off-line (games); On-line (most Internet based applications)

Introductory remarks

- GPP microprocessors vs Video Processors
  - ISA architectures for GPP aimed at abstract model of computation
  - Video Processors: evolution of DSPs, include a RISC or DSP core plus dedicated functional units and I/O specific devices

    Programmable, support specific semantics for media operations (motion est., DCT)

    Coarse parallelism (multiple units), fine paralleism (VLIW micro-architecture)

    MVP (TI), MSP (Samsung), TM-1 (Phlips), Mpact (Chromatic)

Introductory remarks

- Multimedia Extensions in GPP:
  - MAX® and MAX2® (HP)
  - VIS® (SUN)
  - MMX® and SSE® (INTEL)
  - MDMX® (MIPS)
  - ALTIVEC® (Motorola)
  - MVI® (Compaq)
  - 3DNow!® (AMD)

SIMD Processing - old paradigm

- Image processing
  - low level (pixels, local/global operations)
  - intermediate level
- Regular *data structures*
- Local neighborhood *operations*
- Parallel architectures
- Programming paradigm
    - PE oriented languages (expose topology)
    - Collection oriented languages (sets are primitives)

Media Processing

- Images, audio, video and graphics
  - *Streaming* audio and video over the Internet
  - 3D graphics for games and user interfaces
- *Kernels* applied to huge quantities of data (image pixels)
  - IDCT on 8x8 blocks of coefficients
  - 8x8 matrix transpose
  - Motion estimation on 16x16 blocks
  - 3x3, nxn filtering

Media Processing

- *Kernels* applied to huge quantities of data (graphics: vertexes in geometry and ligthting)
  - vertex transformation (matrix multiplication)
  - clipping (compare and branch)
  - displaying to screen (perspective division)

Media Processing

- MPEG-1 decompression on HP 735

Task	MOI	%Time
Header decode	0.6	0.1
Huffman decode	55.3	7.5
Inv. Quantization	8.7	2.4
IDCT	206.5	38.7
Motion comp.	79.9	18.3
Display	188.7	33.0
Total	539.7	100

- (1994) 99Mhz PA-RISC with 256K I and D cache achieved 18.7 fps (352x240 Y, 176x120 C_b and C_r)

Media Processing

- Predictable structure in algorithms
  - scan input data
    - apply kernel to current block
    - update block address / get next block
  - kernel
    - load from memory a block
    - process the block
    - output block to device or to memory
  - block
    - process pixels within the block

Media Processing

- Data Parallelism
  - within the block
    - pixels are subject to the same operation(s)
  - blocks are processed by the same kernel
    - often no dependency among blocks
    - multiple blocks can be *processed* concurrently
    - blocks are *accessed* serially

Media Processing

- Levels of SIMD processing
  - *internal*: within block multiple data (pixels, vertexes) can be worked on by the same operation

    micro-architecture level

  - *external*: multiple blocks (more iterations on blocks as atomic data) are retrieved and dispatched to different processors/cores

    system level (parallel programming)

Media Processing

- SIMD in Media Processing
  - streaming data access
  - blocks are retrieved, transformed, output and seldom re-used
  - lay out of data in memory must be taylored to block SIMD processing at the micro-architecture level

Multimedia Extensions in GPP

- Current GPP characteristics (*enablers*)
  - system level busses
    - 64-bit transactions load large data chunks from main memory into the cache hierarchy (typ. 32 bytes)
    - multiple concurrent read/write transactions
  - huge internal pathways from caches to microarcitecture's core
  - 32/64-bit microarchitectures
  - on-chip: more room than functions (larger caches ?)

Multimedia Extensions in GPP

- Current GPP characteristics (*enablers*)
  - super-pipelines n
  - super-scalar execution y
  - out-of-order dynamic disp. y
  - predicate execution (shortly) ?

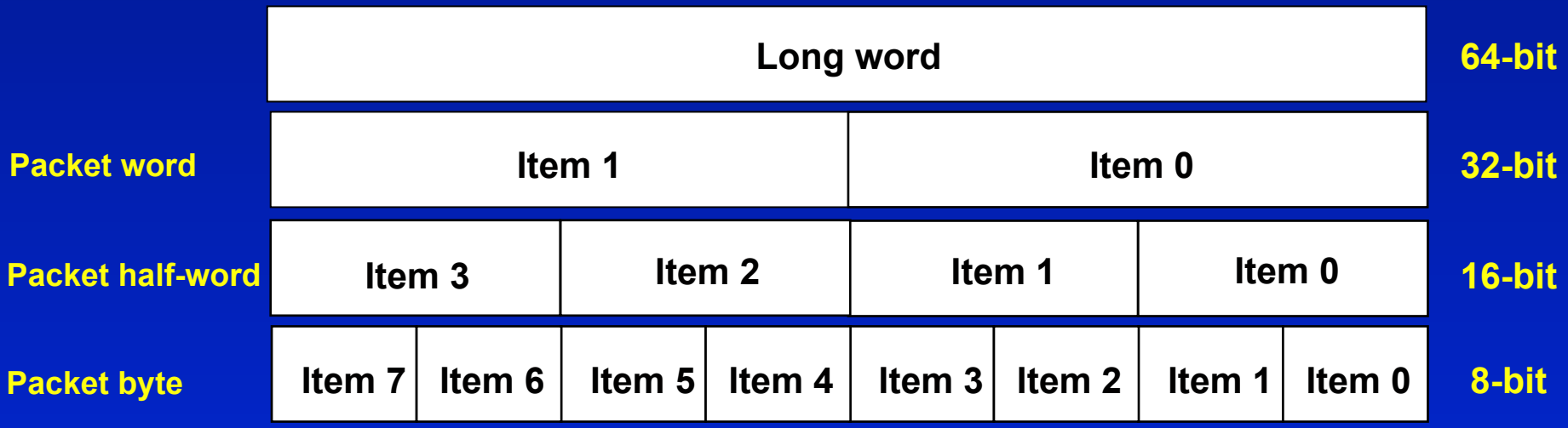
Multimedia Extensions in GPP

- SIMD support *within* the microarchitecture

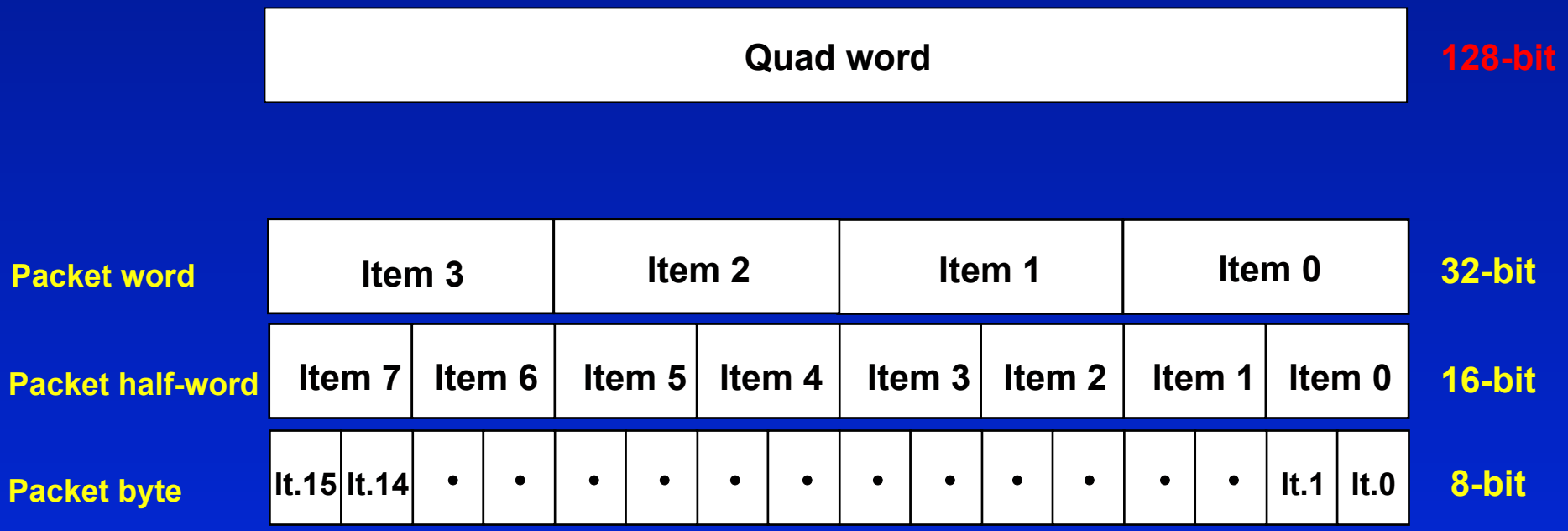
subword parallelism

  - use all data that have already travelled the long route from main memory to the functional units of the microarchitecture
  - capitalize on available hw within functional units

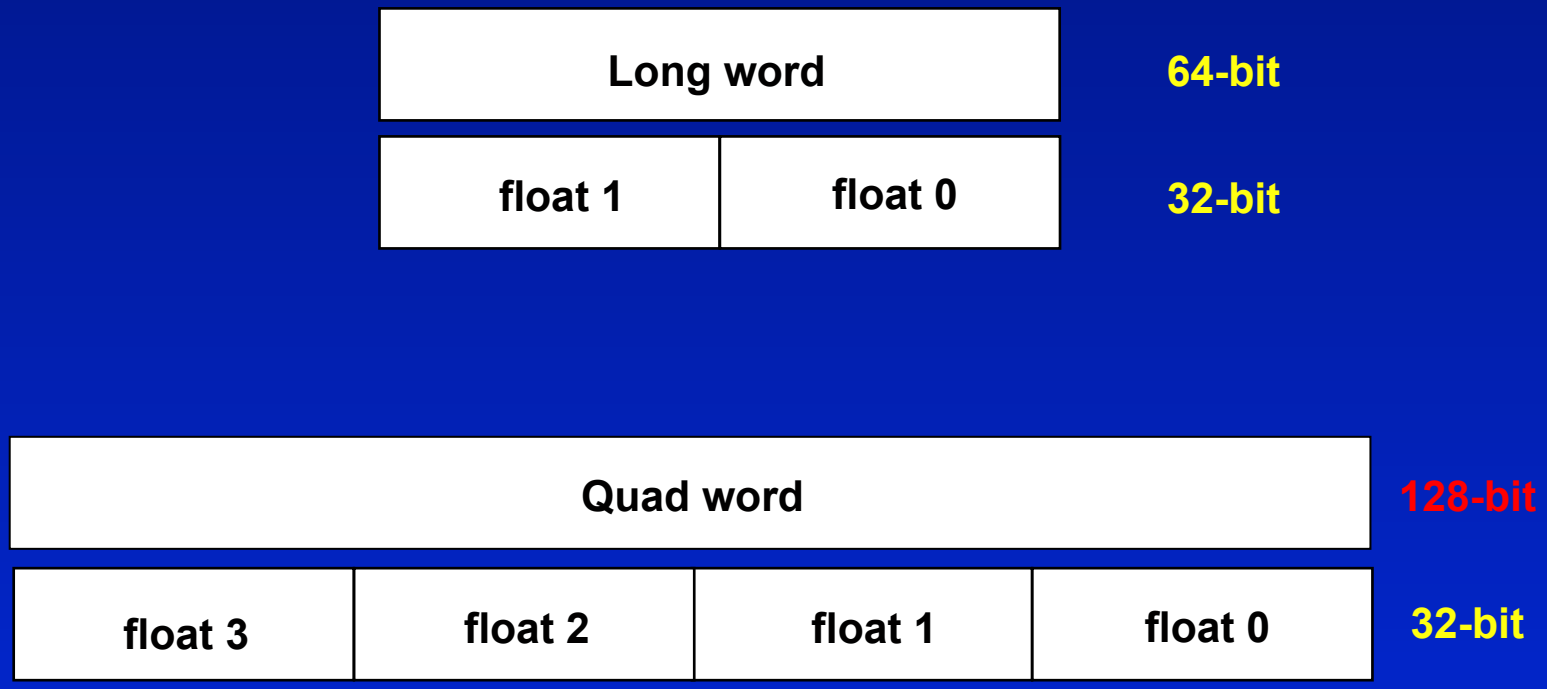
Packed Integer Data Type



Packed Integer Data Type



Packed Float Data Type



Multimedia Extensions in GPP

- Subword parallelism implementation issues
  - die are usage (minimum/substantial)
  - register support
  - degree of sub-word SIMD mode supported
  - type of ISA modification (orthogonal vs. specialized instructions)

Multimedia Extensions in GPP

- Die-area usage - *conservative approach*
  - no extra state (register sharing)
  - few instructions (decoding logic unaffected)
  - minor changes to functional units
  - no new functional unit for specialized ops.
  - Optymized support of a few, well chosen media kernels without addressing general purpose processing

Multimedia Extensions in GPP

- Die-area usage - *progressive approach*
  - silicon area within die used for new processing modes rather than caches
  - rich instruction set
  - new state (dedicated registers)
  - new functional units
  - integer vs float packed data types

Multimedia Extensions in GPP

- Register support (*conservative*)
  - use of existing registers in either data path
  - mapping on integer data path: pressure on address computation (minimal), easy extensions of packed integer computations
  - mapping on float data path: possible mix of contexts (float registers for integer packed data, NaN) and required switching coded in applications with substantial penalties, partitioned coding

Multimedia Extensions in GPP

- Register support (*progressive*)
  - new state requires OS kernel modifications
  - concurrent dispatching of multimedia instructions and legacy ones
  - optimal use of multiple issue in micro-architecture
  - mandatory to support graphics effectively

Multimedia Extensions in GPP

- Degree of sub-word parallelism (image/video)
  - 1 byte-per-pixel: typical of input data, the finest data subdivision, hardly used through a processing chain
  - 2 bytes: good dynamic range, good support to fixed point arithmetic
  - 4 bytes: seldom necessary, much close to single precision floating point

Multimedia Extensions in GPP

- Degree of sub-word parallelism (audio)
  - 2 bytes: correct precision for fixed point arithmetic
- Degree of sub-word parallelism (graphics)
  - 4 bytes: mandatory for single precision floating point operations

Multimedia Extensions in GPP

- Degree of sub-word parallelism (audio)
  - 2 bytes: correct precision for fixed point arithmetic
- Degree of sub-word parallelism (graphics)
  - 4 bytes: mandatory for single precision floating point operations
- The actual level of SIMD data micro-parallel execution depends on the functional units ultimately (64 to 64, 128 to 64, 128 to 128)

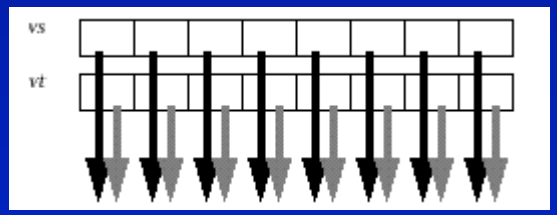
Multimedia Extensions in GPP

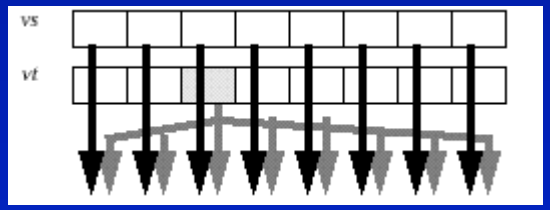
- Extending an existing ISA
  - orthogonal, classic types of instructions
  - specific processing
    - Sum od Absolute Differences (SAD) for block comparison in motion estimation
    - 3D arrays indexes to memory addresses computation
  - pathlength issue
    - inherently improved thanks to the sub-word model (1 op more data)
    - data reformatting causes overhead

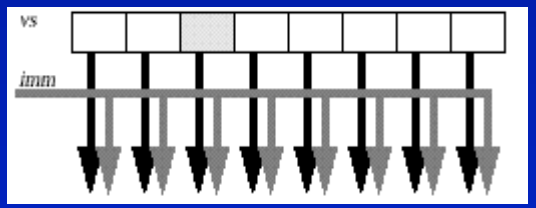
Multimedia Extensions in GPP

- Extending an existing ISA
  - efficient execution: even in CISC ISA (Intel) multimedia instructions map "directly" to micro-operations (one to two); in RISC it's granted
  - latency minimized (in integer functional units almost always 1, maximum 3)
  - multiple issue: data dependencies are minimal in multimedia kernels
  - loop unrolling

Data Types







Instruction classes

- Arithmetic
- Data reformatting
- Conditional execution
- Reduction
- Memory access & cache management
- Special instructions

Instruction classes - Arithmetic

- Logical
- Modulo vs Saturation arithmetic
- Multiplication & Multiply-Add

- Floating point modes
- Approx. Reciprocal and SQRT

Instruction classes - Arithmetic

- Modulo vs Saturation
  - modulo arithmetic (C standard) is best for addresses (and *cryptography* !)
  - subword processing cannot tolerate exceptions for overflow (underflow) *within* the word
  - pixel ops. **require** saturation
- Signed vs Unsigned modes

Instruction classes - Arithmetic

- Modulo vs Saturation

mm1	1A00h	2A00h	3A00h	4A00h
mm2	F700h	F700h	F700h	F700h

PADDW mm1,mm2

mm1	1100h	2100h	3100h	4100h

PADDWUS mm1,mm2

mm1	FFFFh	FFFFh	FFFFh	FFFFh

Instruction classes - Arithmetic

- Multiplication
  - area three times larger than adder, latency about three times longer, result depth doubles
  - many different approaches
    - no mult — use shift-and-add (HP conservative impl.)
    - n x n to n — 8 x 8 to 8, 16 x 16 to 16
    - n x 2n to 2n — 8 x 16 to 16
    - n x n to 2n — 8 x 8 to 16, 16 x 16 to 32

Instruction classes - Arithmetic

- Multiply-ADD
  - basic operation of many signal processing kernels (filters, convolution, …)
  - inherited from MAC in DSP
  - non MAC version (Intel integer) 16 x 16 to 32
  - MAC versions
    - 8 x 8 + to 28 16 x 16 + to 48 (MIPS accumulator)
    - 8 x 8 + to 16 16 x 16 + to 16 (Altivec orthogonal impl.)
      16 x 16 + to 32

Instruction classes - Arithmetic

- Shift-and-add
  - approximates multiplication by constants
    - one operand is shifted by 1,2 or 3 bits, then added to second one
    - HP MAX only

$$SQRT(2) \cong 1.4142_{10} = 1.01101010_2$$

T = 1.01 X	T = X + X >> 2	PshRadd X,2,X,T
S = 1.0101 X	S = X + T >> 2	PshRadd T,2,X,S
T = 1.0110101 X	T = T + S >> 3	PshRadd S,3,T,T

Instruction classes - Arithmetic

- Floating point: single precision
- Two modes
  - IEEE compliance guarantees precision floating point and portability (Java subset)
  - Exceptions (underflow, NaN) cause context switches, extremely lengthy
  - Flush-to-zero (with masked exceptions) makes applications much faster at a minimum detriment of precision

Instruction classes - Arithmetic

- Floating point
  - perspective transformation (division)
  - 3D lighting - distance from ligth source (SQRT, division)
- Hardware look-up versions of reciprocal approximates
  - (11 vs 23 bits in mantissa) with two ops.
  - 22/23 bits with 1 Newton-Rampson iteration in 3 / 5 ops.
- pipelined, low latency (2 cycles in INTEL SSE instead of 36)

Instruction classes - Data Reformatting

- Converting among different precisions
  - packing (2n to n bits, low or high)
  - unpacking
- Conversions comply with signed - unsigned representation
- Usually outside loops that process kernels
  - exceptions due to precision escalation

Instruction classes - Data Reformatting

- Rearranging subwords
  - PERMUTE
    - PERMUTE with REPLICATION
    - PERMUTE using one or two source registers
  - MERGING
    - at different boundaries (bytes, half word, word)
- Simplify block matrix transposition
  - used in IDCT, in AoS to SoA conversion in graphics
  - 4x4 matrix transposition in 8 single-cycle ops.

Instruction classes - Reduction

- Hard in Block-based loop programming
- Inter-register
  - averaging two data items (motion compensation)
    - fairly commom
  - sum-across
  - sum-of-products
    - advanced version of Multiply-ADD; simplyfies dot products

Instruction classes - Conditional exec.

- Data dependent branches are killers to super-pipelined micro-architectures
- Image processing: often data dependent *contexts* are shallow
  - image overlay

```
for (i=0; i<image_size; i++) {
 If (x[i]==Background) new_image[i]=y[i];
                  else new_image[i]=x[i];
                                  }
```

Instruction classes - Conditional exec.

- General conditional assignement

  if cond(a_i,b_i) then c_i=s_i else c_i=t_i i=1,n

  parallel subword compare instruction generate MASKS or Condition BITS

- *Self* conditional assignement

  if cond(a_i,b_i) then c_i=a_i else c_i=b_i i=1,n

  max and min with saturation arithmetic

Instruction classes - Conditional exec.

- Mask mode

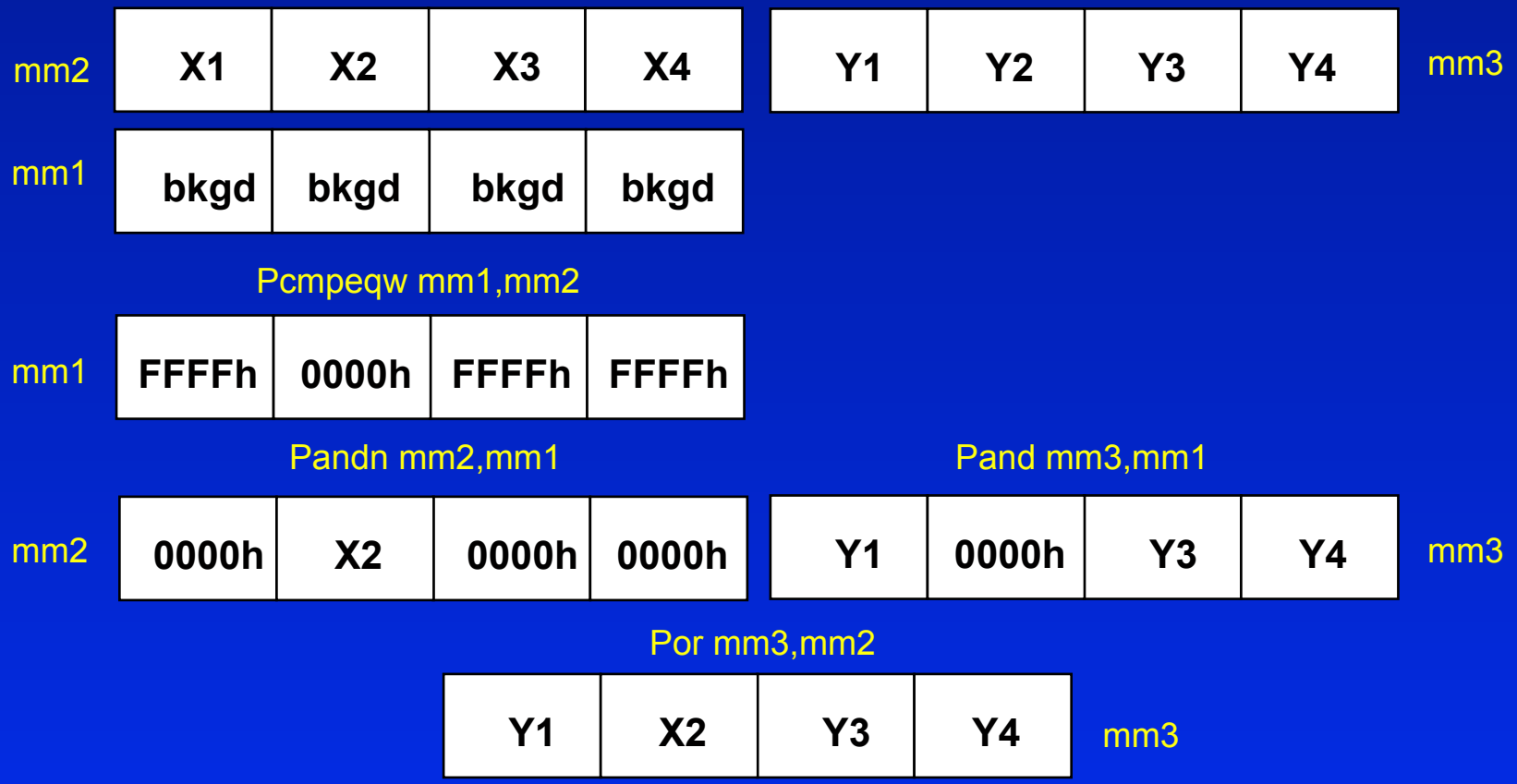
mm2	X1	X2	X3	X4
mm1	v1	v2	v1	v1

Pcmpeqw mm1,mm2

mm1	FFFFh	0000h	FFFFh	FFFFh

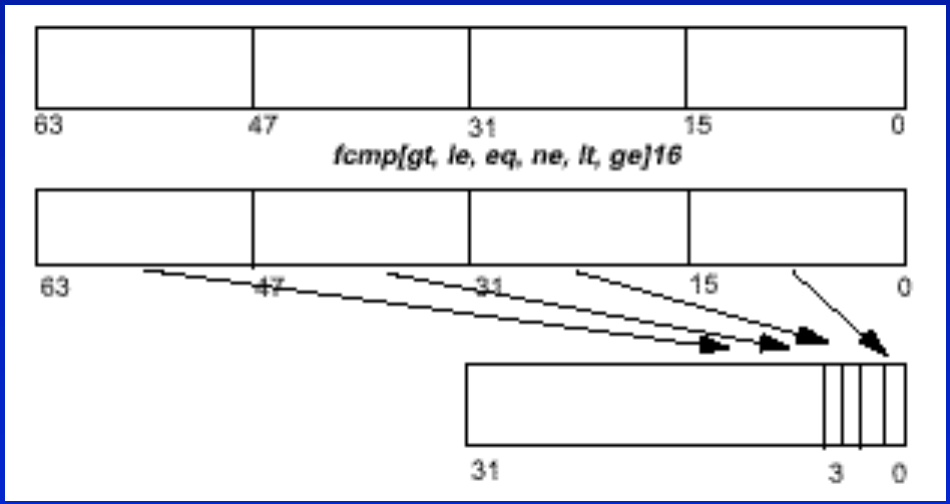
Instruction classes - Conditional exec.

- Mask mode: implementation of image overlay



Instruction classes - Conditional exec.

- Bit mode



typically used for *partial store*

Instruction classes - Conditional exec.

- *MAX* and *Min* with saturation arithmetic

  if cond(a_i,b_i) then c_i=a_i else c_i=b_i i=1,n

  MAX cond(a_i,b_i) $a_i > b_i$; MIN cond(a_i,b_i) $a_i < b_i$

Ra	40	5	78	200
Rb	51	34	15	243

Hsub,us Ra,Rb,Rc

Rc	0	0	63	0

Hadd Rc,Rb,Rc

Rc	51	34	78	243

Hsub,us Ra,Rc,Rc

Rc	40	5	15	200

Instruction classes - Conditional exec.

- SIMD ‘global OR’ support
  - the result of a parallel compare (MASK or BIT mode) is *reduced* to a single bit
  - current implementation for MASK compare into a status bit
  - all ones / all zeros

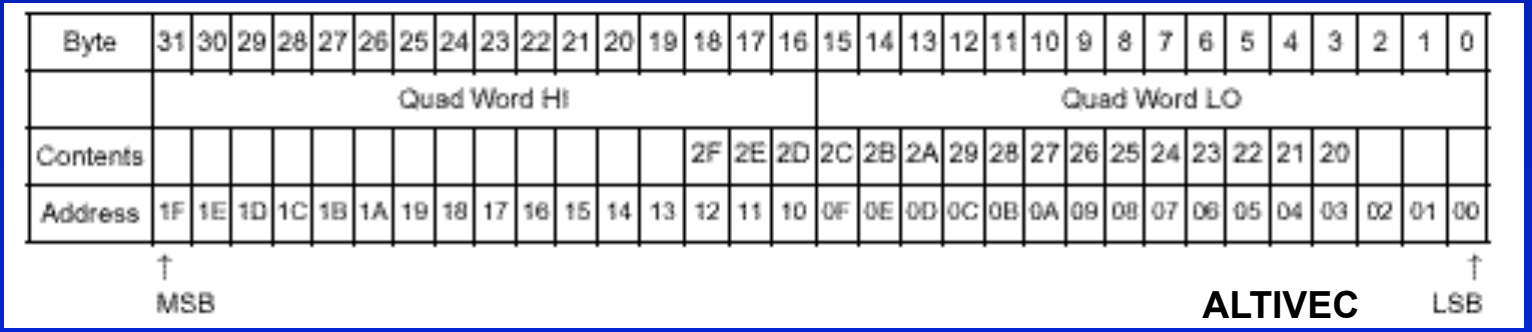
Instruction classes - Memory access

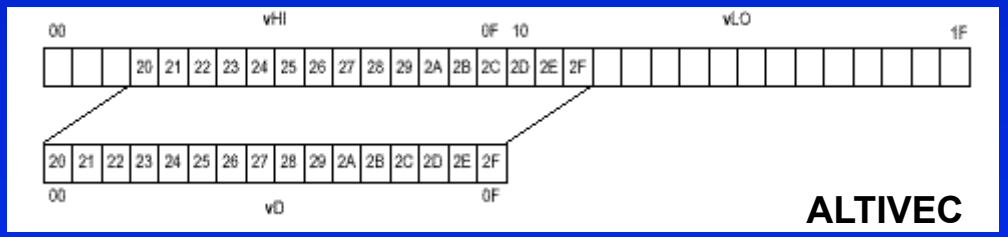
- Effective access to the memory hierarchy is the pre-condition for SIMD within the micro-arch.

- Data alignement
- Partial store vs Write Combining store
- Block store

Instruction classes - Memory access

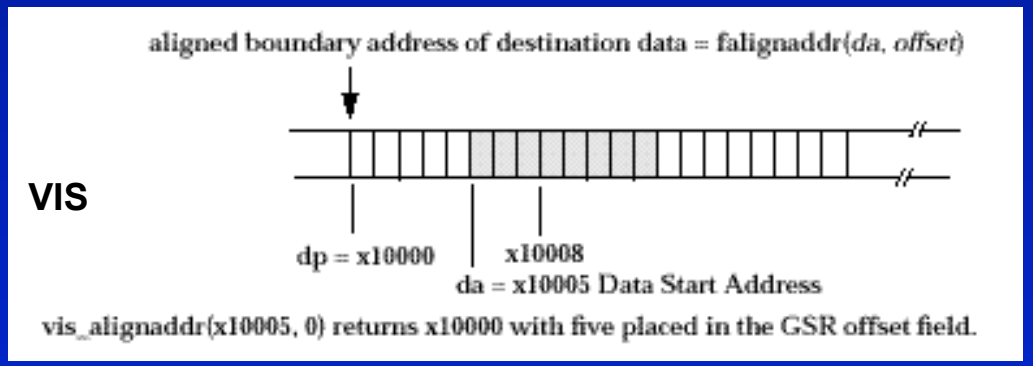
- Data alignement
  - Data aligned in memory to multiples of subword length are retrieved at maximum throughput
    - misaligned accesses fixes: microcode, sw through fault or sw no fault

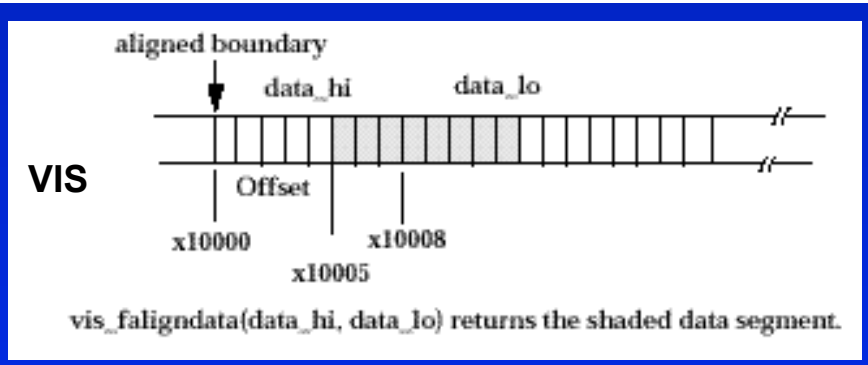




Instruction classes - Memory access

- Data alignement
  - support for re-alignement with special registers





Instruction classes - Memory access

- Partial store
  - move sub-words conditionally to memory
    - multi-channel images; respecting image boundaries
  - implemented as read-modified-store only on cacheable memory segments
- WC semantics
  - stores conditioned by masks throughout the memory chain
  - eliminates unnecessary read-for-ownership that pollutes caches

Instruction classes - Memory access

- Block load/store
  - only in SPARC VIS
  - transferring data from/to memory to/from *multiple registers*
  - alignement at the block size

Instruction classes - Cache manag.

- Proper use of memory hierarchy mandatory in media processing
- Space and time locality as supported by cache hierarchy model unsuited to media processing
- Block based media processing vs 1D array layout of data fetched from memory into caches

prefetching

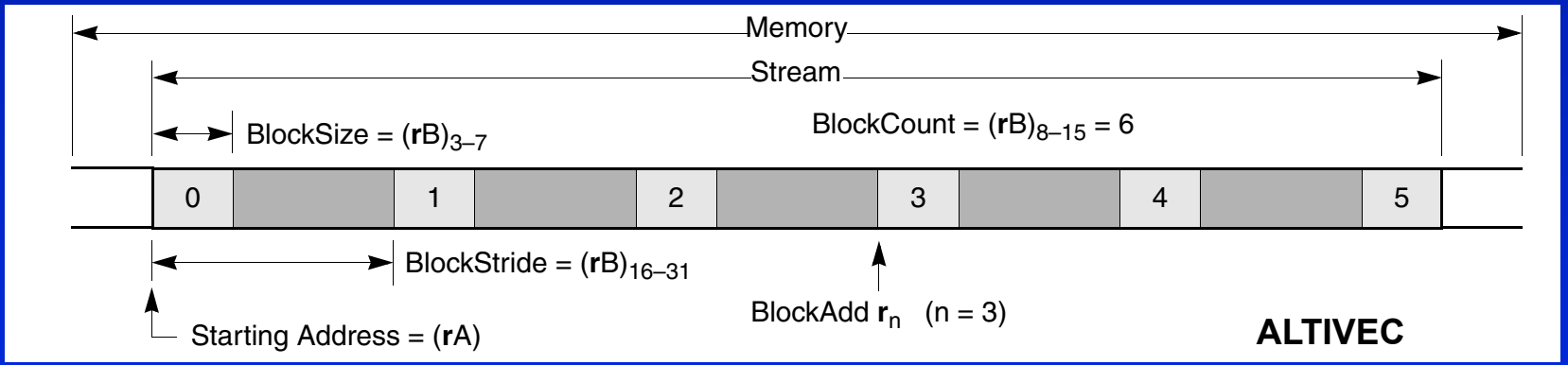
cacheability hints

Instruction classes - Cache manag.

- Prefetching
  - proper understanding of cache/memory subsystem to correctly place prefetching in software loops
  - do not alter state, retired soon
  - 1) simple implementation
    - block = cache line
    - no stride
    - matches 1D signal processing, row image processing

Instruction classes - Cache manag.

- Prefetching
  - 2) advanced: definition of *data stream*
    - starting address
    - block (1-32 quadwords)
    - number of blocks in the stream
    - stride (displ. In bytes, signed)



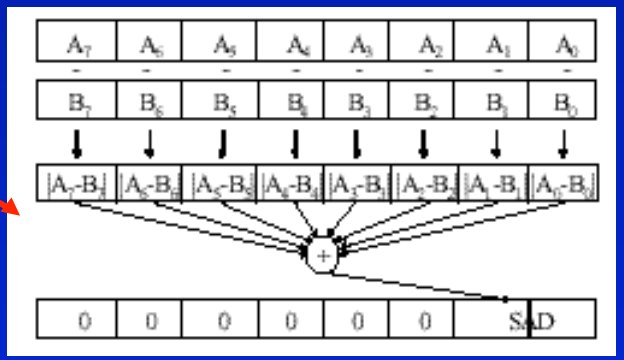
  - suites 2D near neighbor

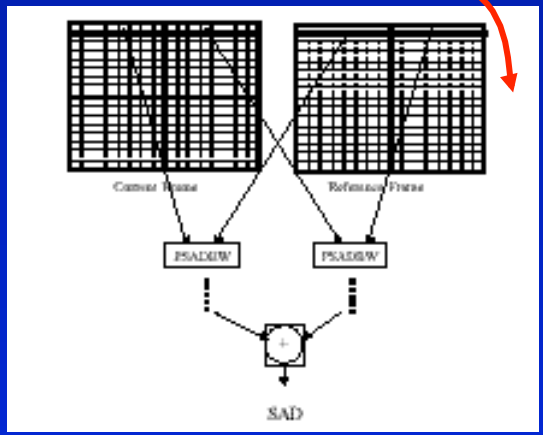
Instruction classes - Cache manag.

- Cacheability hints
  - allows to prevent cache polluting for non persistent loads and streaming stores
  - no cache for streaming stores
  - cache level
    - full cache hierarchy for “standard” locality
    - L1 for transient load
  - specified at the instruction level
  - combined with prefetching

Instruction classes - Special instr.

- Non orthogonal, task specific
- Sum of Absolute Difference (SAD)
  - easier than SSD in motion estimation

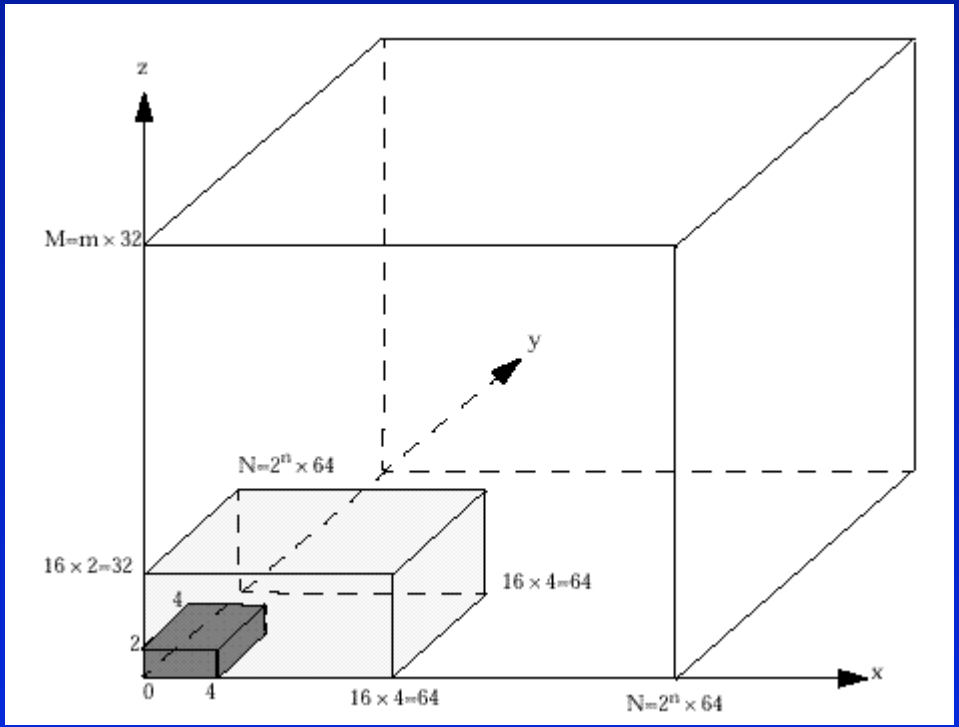




- dedicated functional unit
- three uops in INTEL PIII using integer multiplier Wallace tree
- four ALTIVEC instructions

Instruction classes - Special instr.

- Voxels address computation (VIS only)



Data layout for 16-bit:
X then Y then Z
4x4x2 = 64 bytes (2 cache lines)
64x64x32 = 256K, good TLB hit rate

ARRAY16 instruction returns address of given (x,y,z) point for subsequent loading

Tour: Extensions at a Glance

General	MAX2	VIS	MMX	MDMX	Altivec	SSE
Data Formats	16	(8),16,32	8,16,32	8,16,(24) 32,(48)	8,16,32	32
# instr.	17	81	57	35	162	70
# regist./depth	31 i / 64	32 f / 64	8 f / 64	32 f / 64	32 s / 128	8 s / 128
paired/single	n	n	n	y	n	y

Tour: Extensions at a Glance

Arithmetic	MAX2	VIS	MMX	MDMX	Altivec	SSE
saturation	y	n	y	y	y	n.a.
accumulator	n	n	n	y	y	n
multiply	n	8x16 to16	16x16 to16	8x8 to16 i 16x16 to 32 i 32x32 to 32 f	8x8 to 8 i 8x8 to 16 i 16x16 to 16 i	32x32 to 32 f
multiply-add	n	n	16x16 to 32	8x8 to 24 i 16x16 to 48 i 32x32 to 32 f	8x8to16 i 16x16to16 i 16x16to32 i 32x32to32 f	n
shift-add	y	n	n	n	n	n
shift	y	n	y	y i	y i	n
division/SQRT	n	n	n	n	y	y

Tour: Extensions at a Glance

	MAX2	VIS	MMX	MDMX	Altivec	SSE
Reduction						
average	y	n	y	n	y	n
vector sum	n	n	n	n	y	n
Compare						
max-min	sat.	sat.	y	y	y	y
compare	n	16, 32 bit	8, 16, 32 mask	16, 32 cc	8, 16, 32 (cc) mask	32 mask
cond. assign.					y	

Tour: Extensions at a Glance

Data Manag.	MAX2	VIS	MMX	MDMX	Altivec	SSE
Pack	n	32 to 16 16 to 8 32 to 8	32 to 16 16 to 8	64 to 32	32 to 16 16 to 8 RBG	n
Unpack	n	8 to 16	8 to 16 16 to 32 32 to 64		8 to 16 16 to 32	n
Mix	16, 32	8	n	n	8, 16, 32	32
Shuffle	16 (rep)	n	16	8, 16	8 (rep,alltoall)	32 (rep)

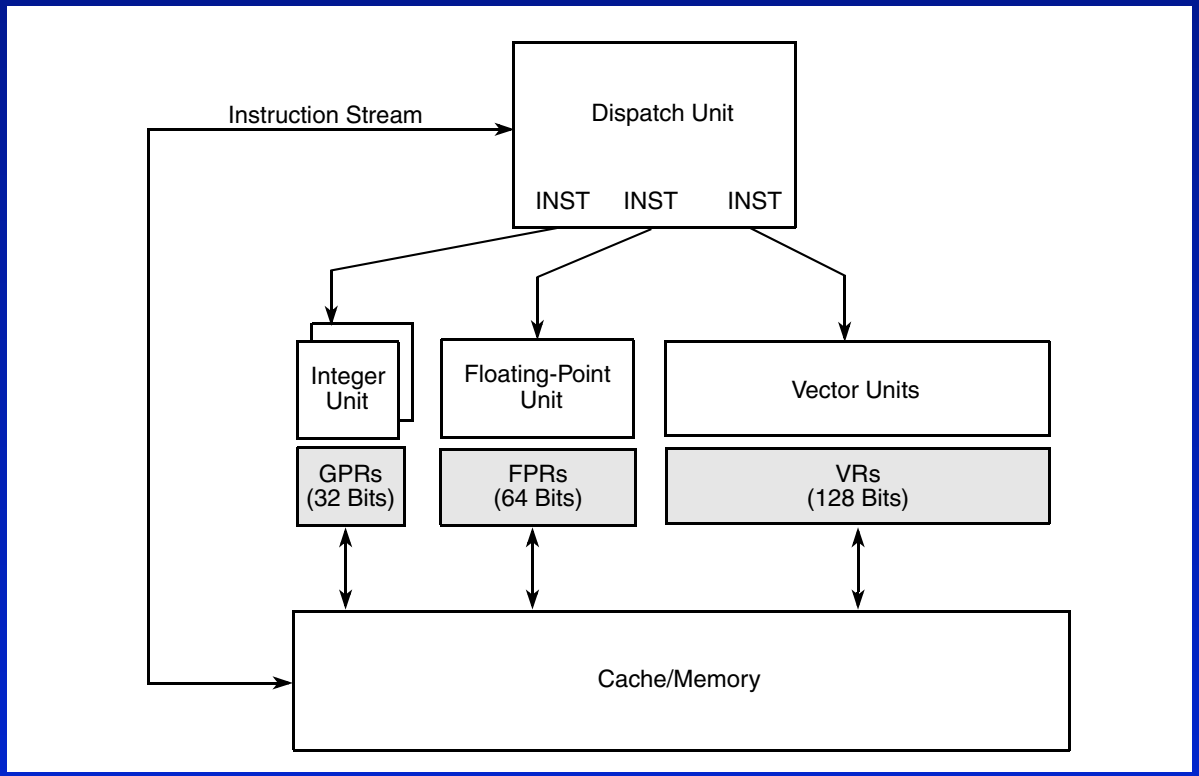
Tour: Extensions at a Glance

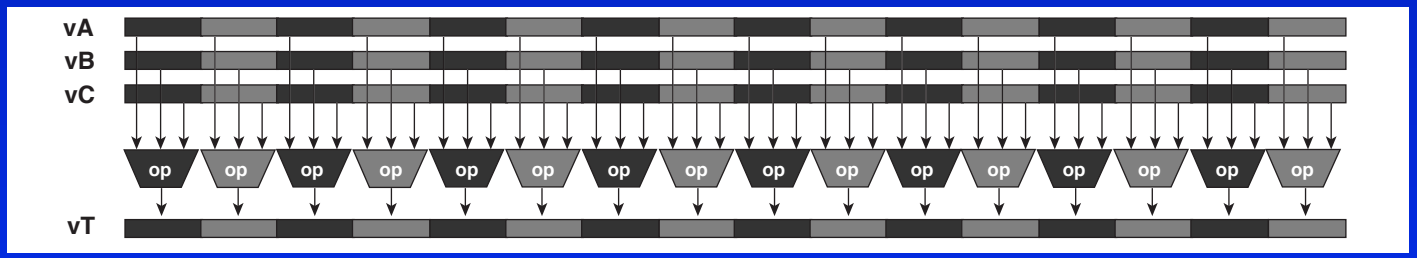
Memory	MAX2	VIS	MMX	MDMX	Altivec	SSE
unalign	y	y	n	y	n	y
block load/store	n	y	n	n	n	n
partial store	8	8,16,32	8 (wc)		8,16,32	32 (wc)
cache hints	y	n	y (w)		y (r,w)	y (w)
prefetch	y (r,w)		y (t,s)		y (t,s)	y (t,s)

Tour: Extensions at a Glance

Special	MAX2	VIS	MMX	MDMX	Altivec	SSE
SAD	n	8	8	n	n	n
Array addr.	n	y	n	n	n	n
Vector const.	n	n	n	y	n	n

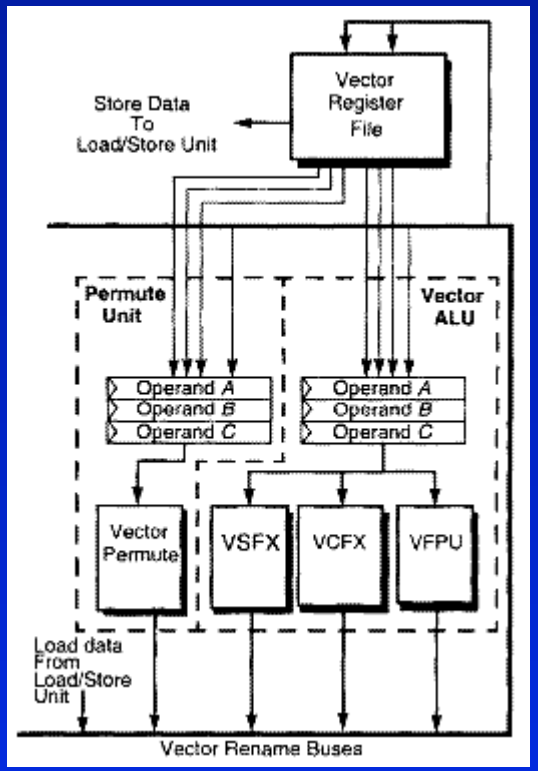
ALTIVEC (Motorola)





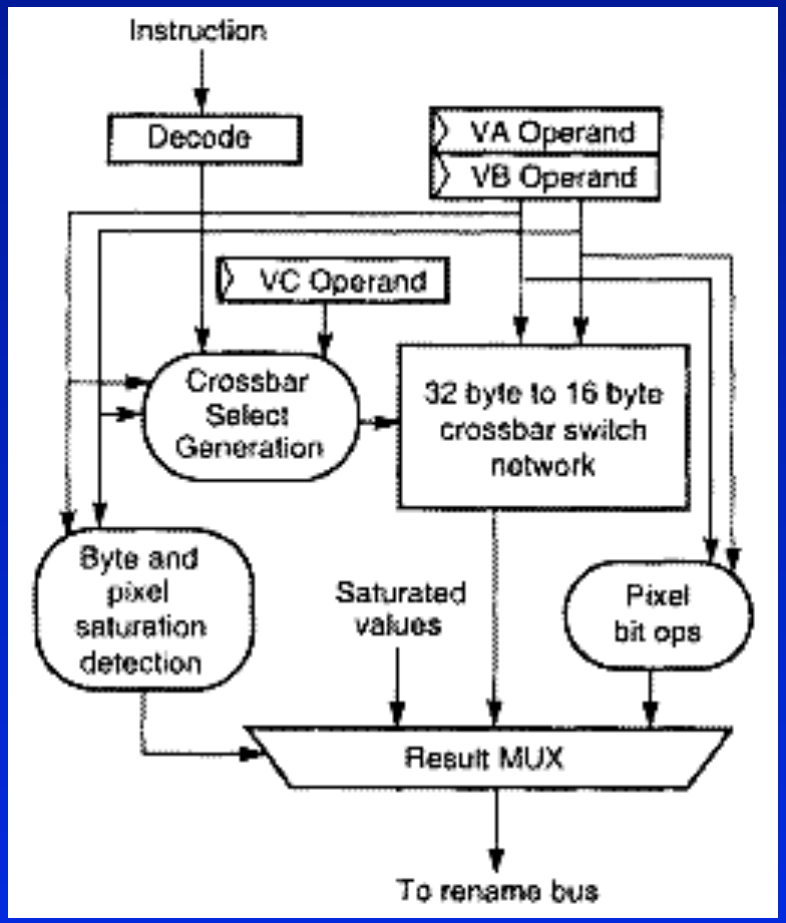
ALTIVEC (Motorola)

- VPERM and VALU dispatched concurrently
  - VPERM 32 to 16 crossbar
  - VALU has three subunits
    - 1 cycle VSFX integer
    - 3 cycles VCFX mult, mul-sum, sum across
    - 4 cycles VFPU 4 float ops.



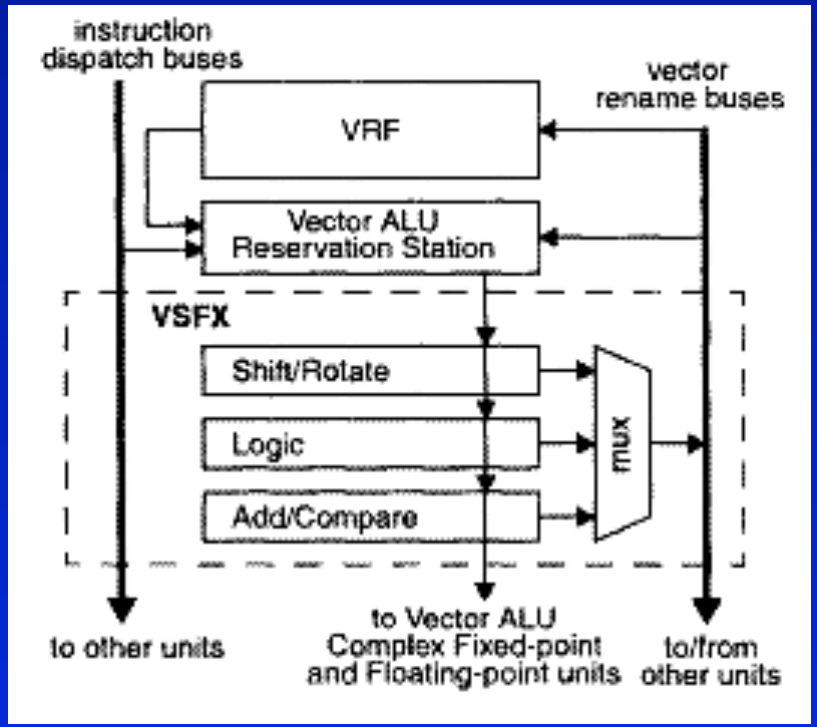
ALTIVEC (Motorola)

- VPERM



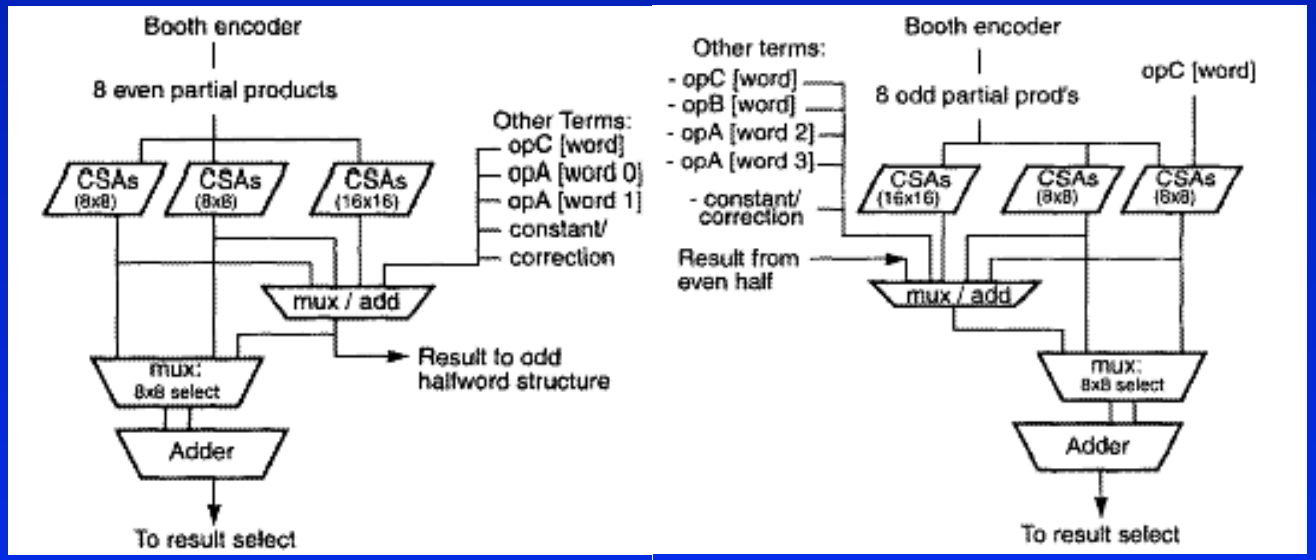
ALTIVEC (Motorola)

- VSFX



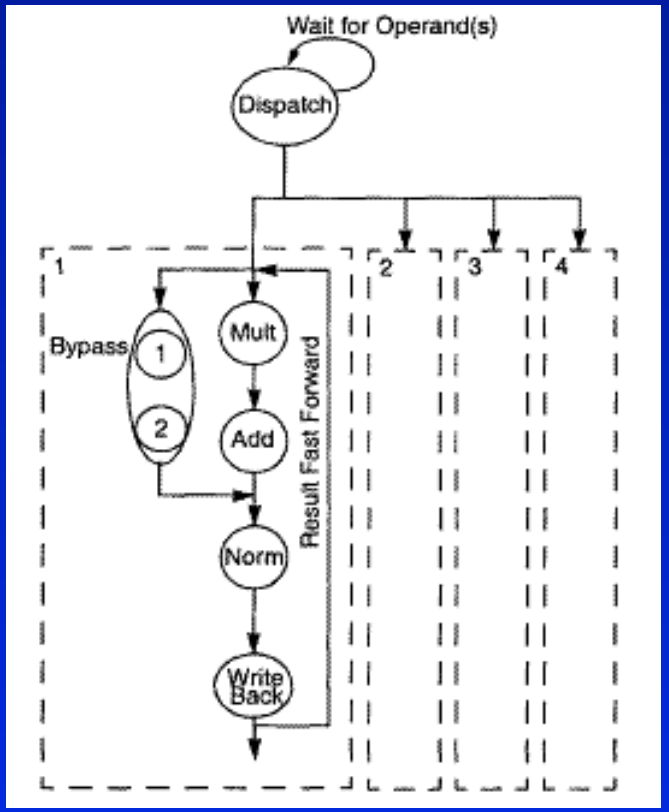
ALTIVEC (Motorola)

- VCFX
  - four 32-bit data paths
  - each partitioned into even-odd 16-bit MAD units
  - partial products accumulated form multiply, multiply add and sum across



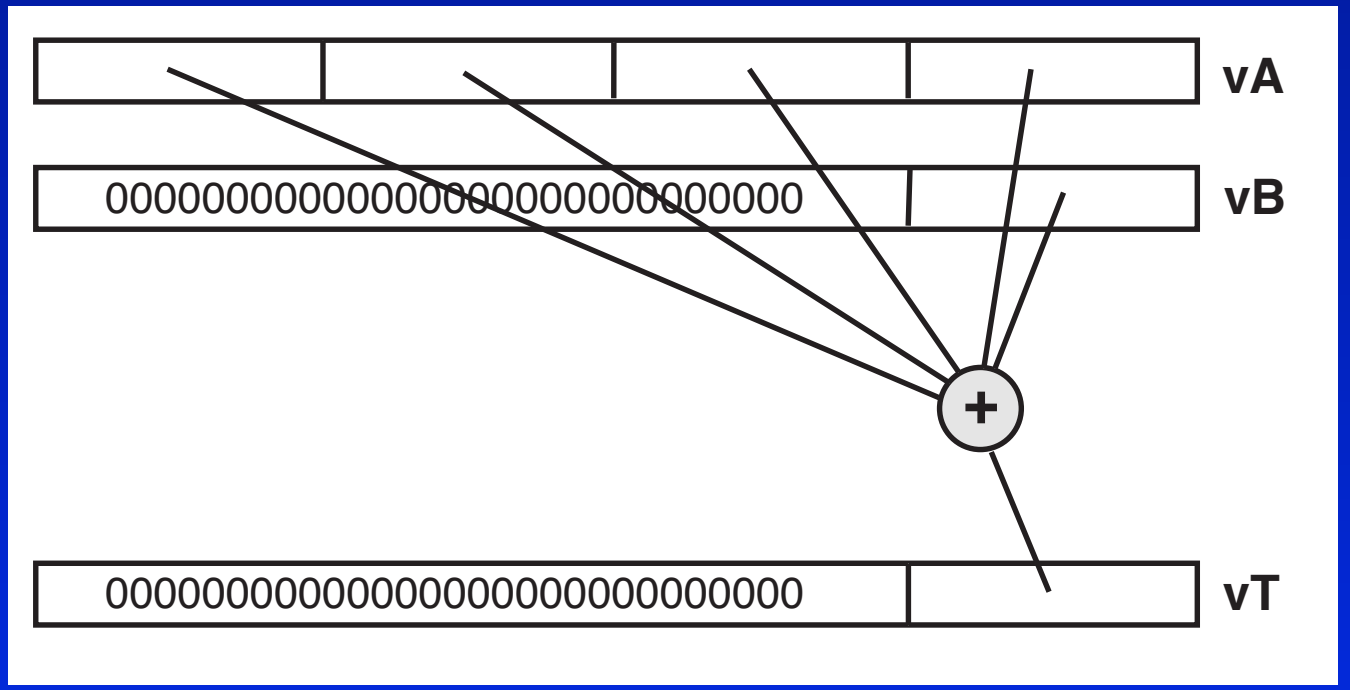
ALTIVEC (Motorola)

- VFPU
  - JAVA mode
    - exception handling
  - non Java mode
    - 4 cycles
    - NaN forced to zeros



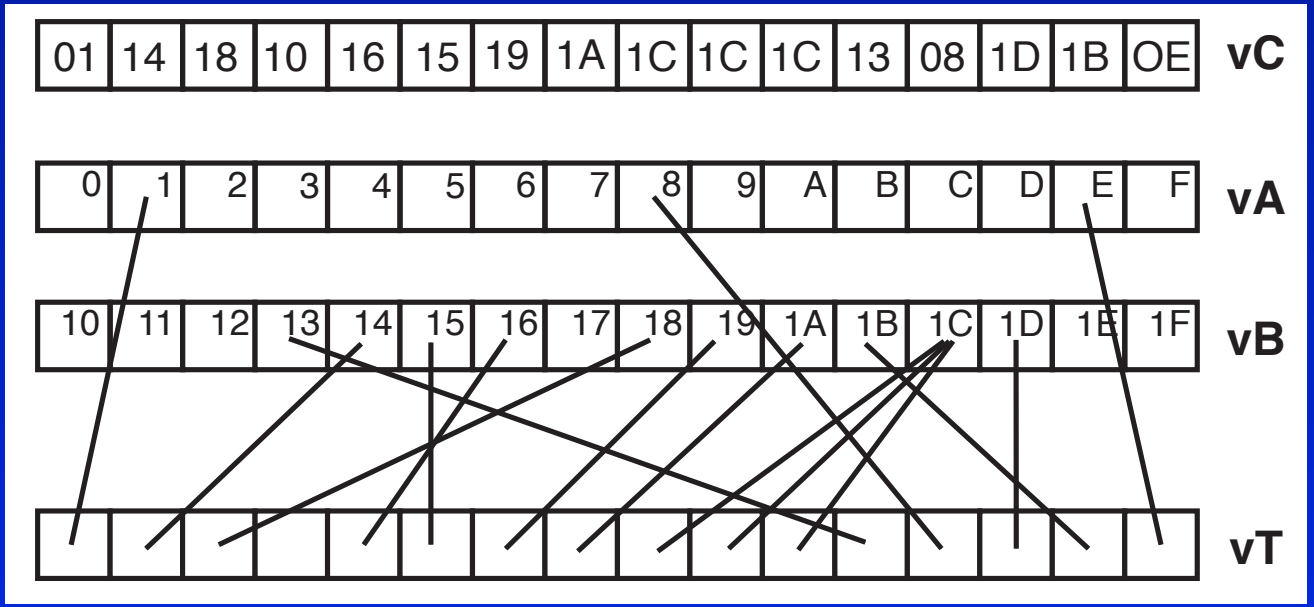
ALTIVEC (Motorola)

- SUM ACROSS (reduction)



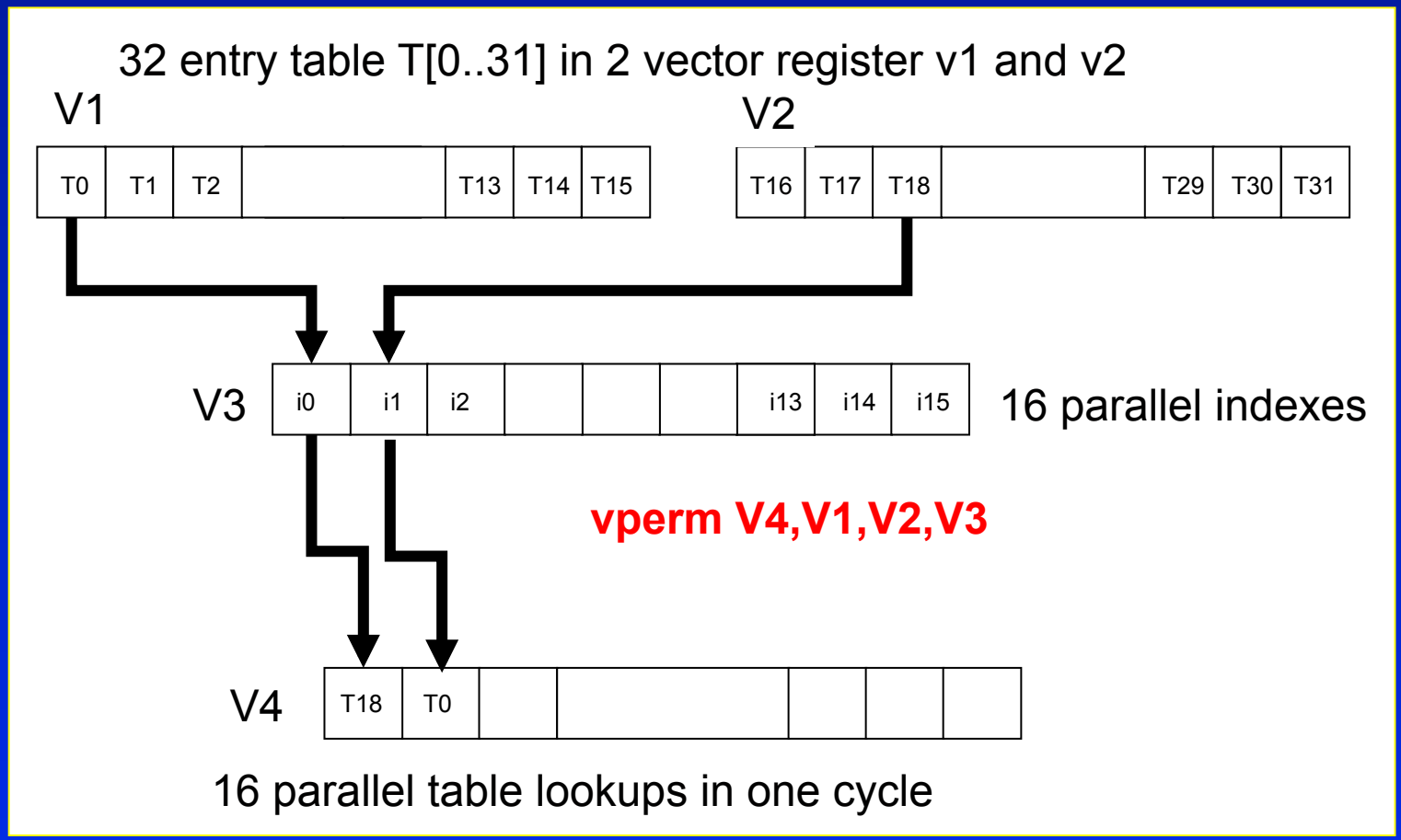
ALTIVEC (Motorola)

- PERMUTE (All to All)



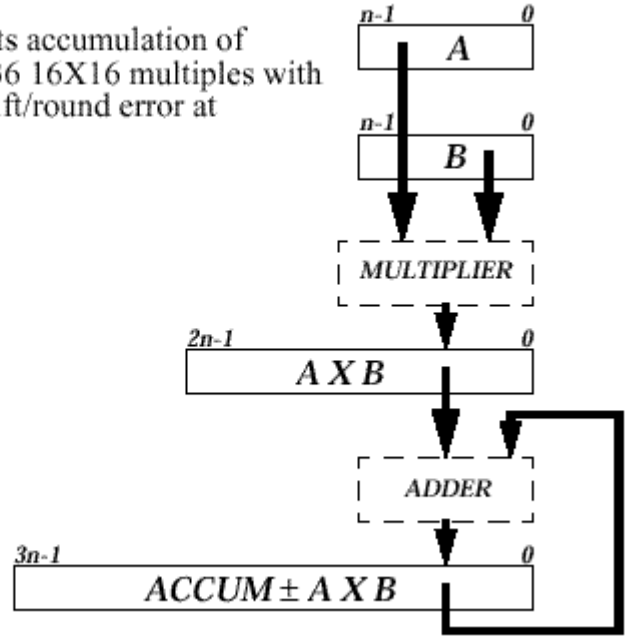
ALTIVEC (Motorola)

- Table lookup

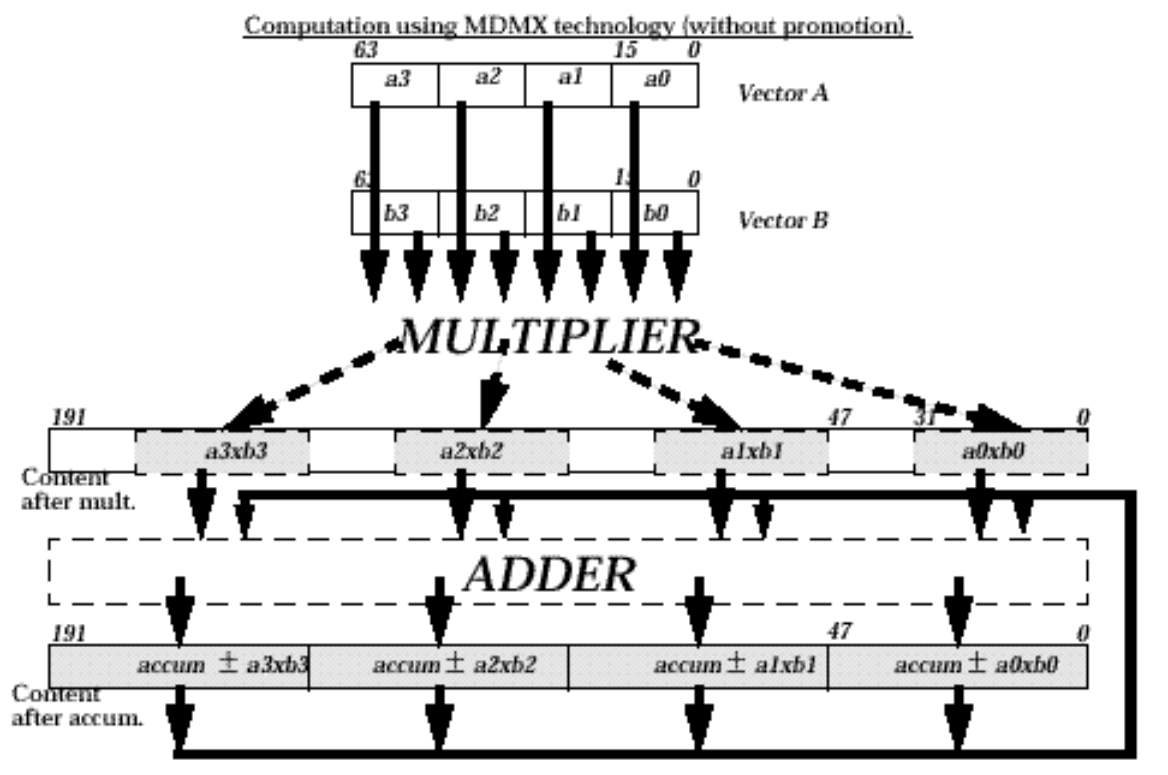


MDMX (MIPS)

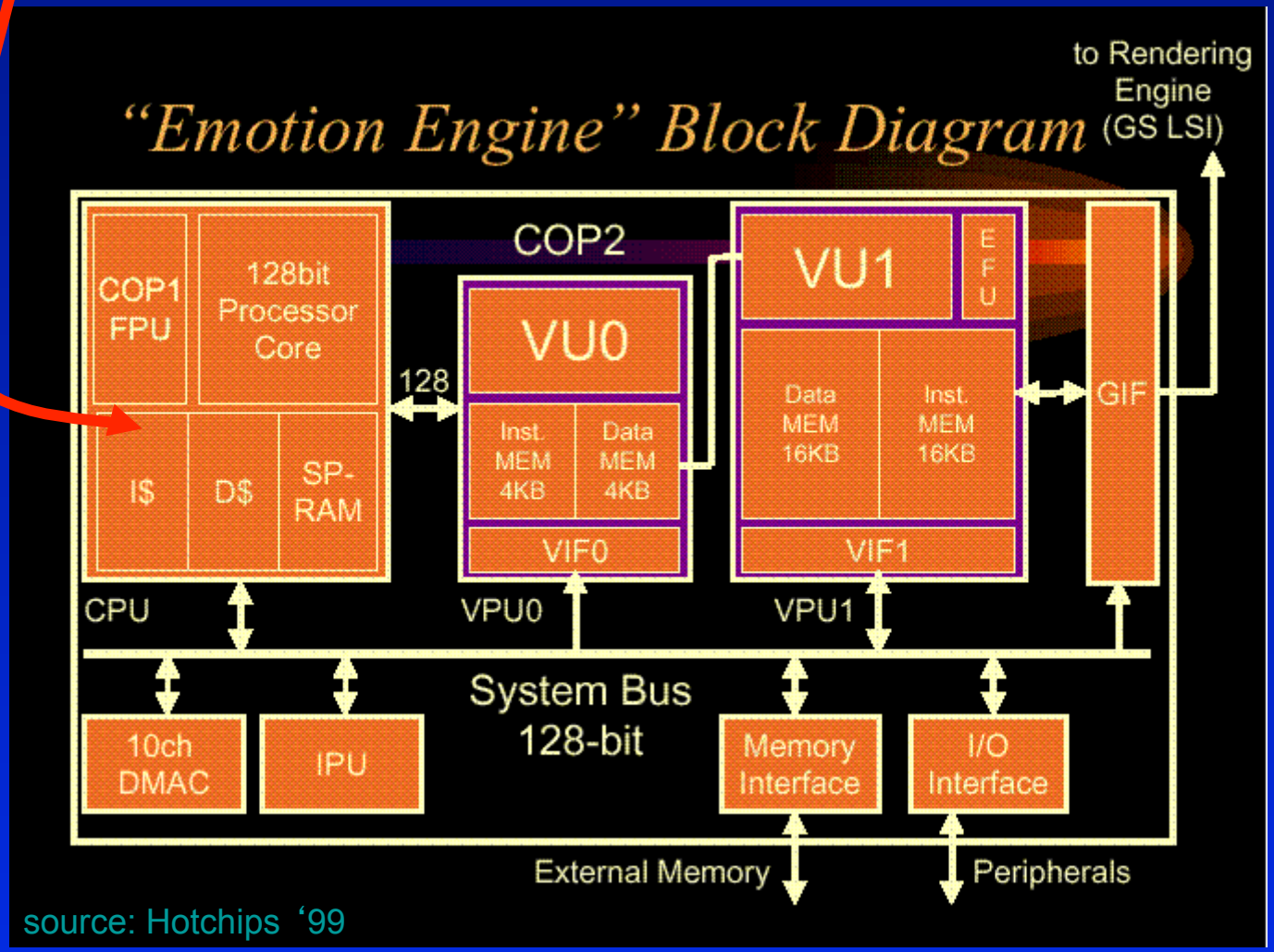
One slice permits accumulation of 256 8x8 or 65536 16X16 multiples with only a single shift/round error at the end.



MDMX (MIPS)

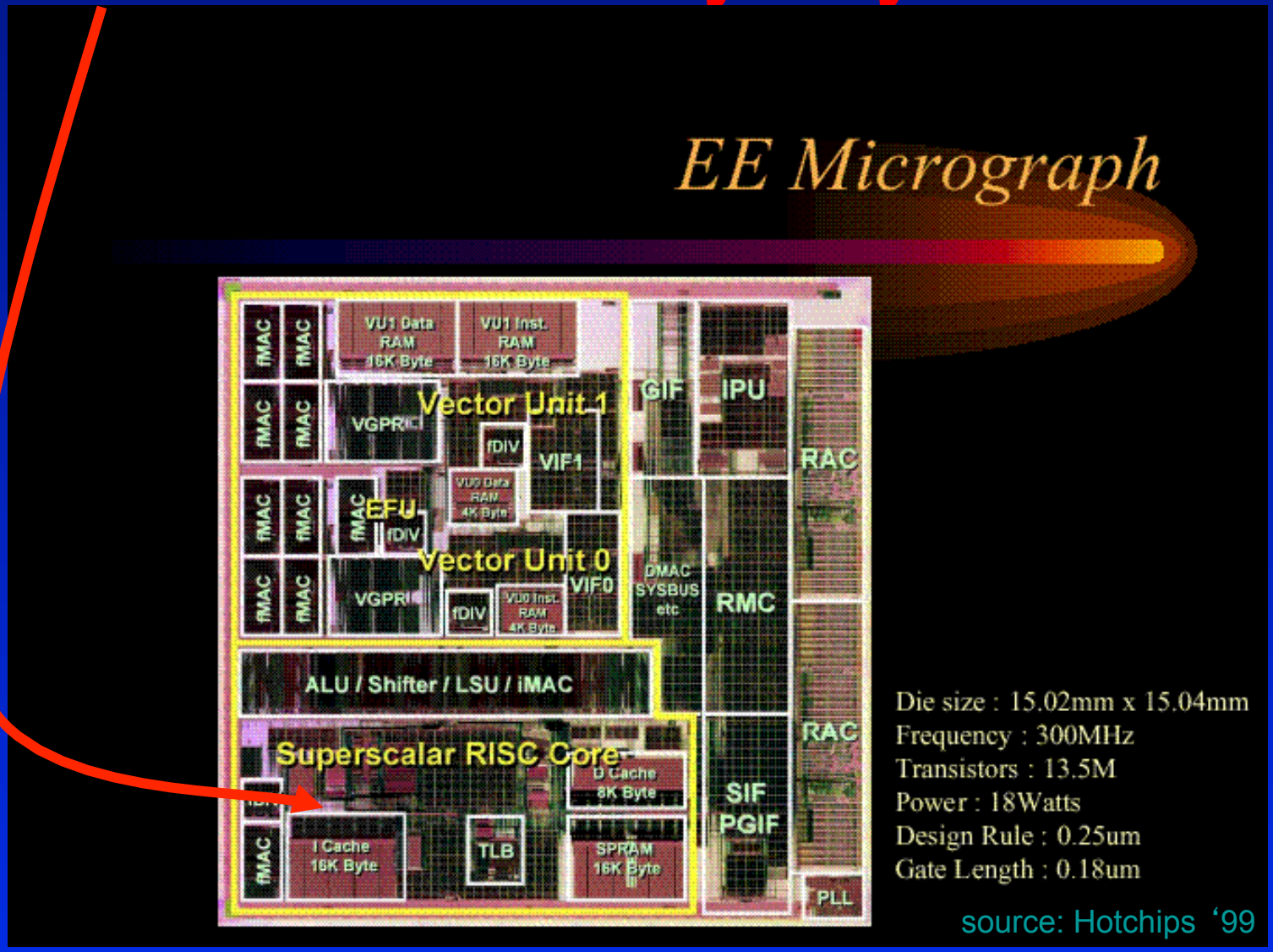

Computation using MDMX technology (without promotion).
63
15
0
a3
a2
a1
a0
Vector A
b3
b2
b1
b0
Vector B
MULTIPLIER
191
47
0
a3xb3
a2xb2
a1xb1
a0xb0
Content after mult.
ADDER
accum ± a3xb3
accum ± a2xb2
accum ± a1xb1
accum ± a0xb0
Content after accum.

MIPS in Toshiba-Sony Playstation 2

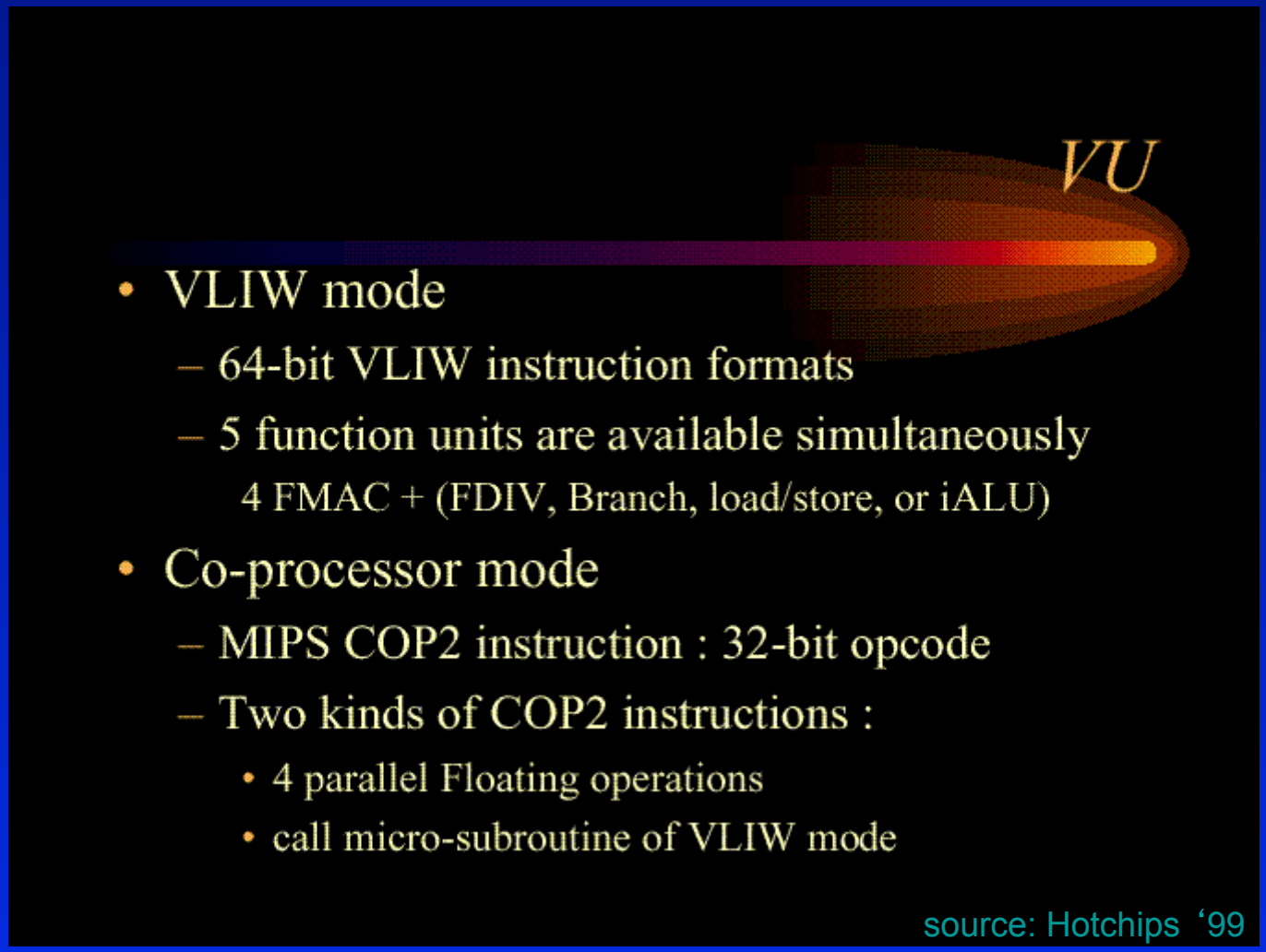


source: Hotchips '99

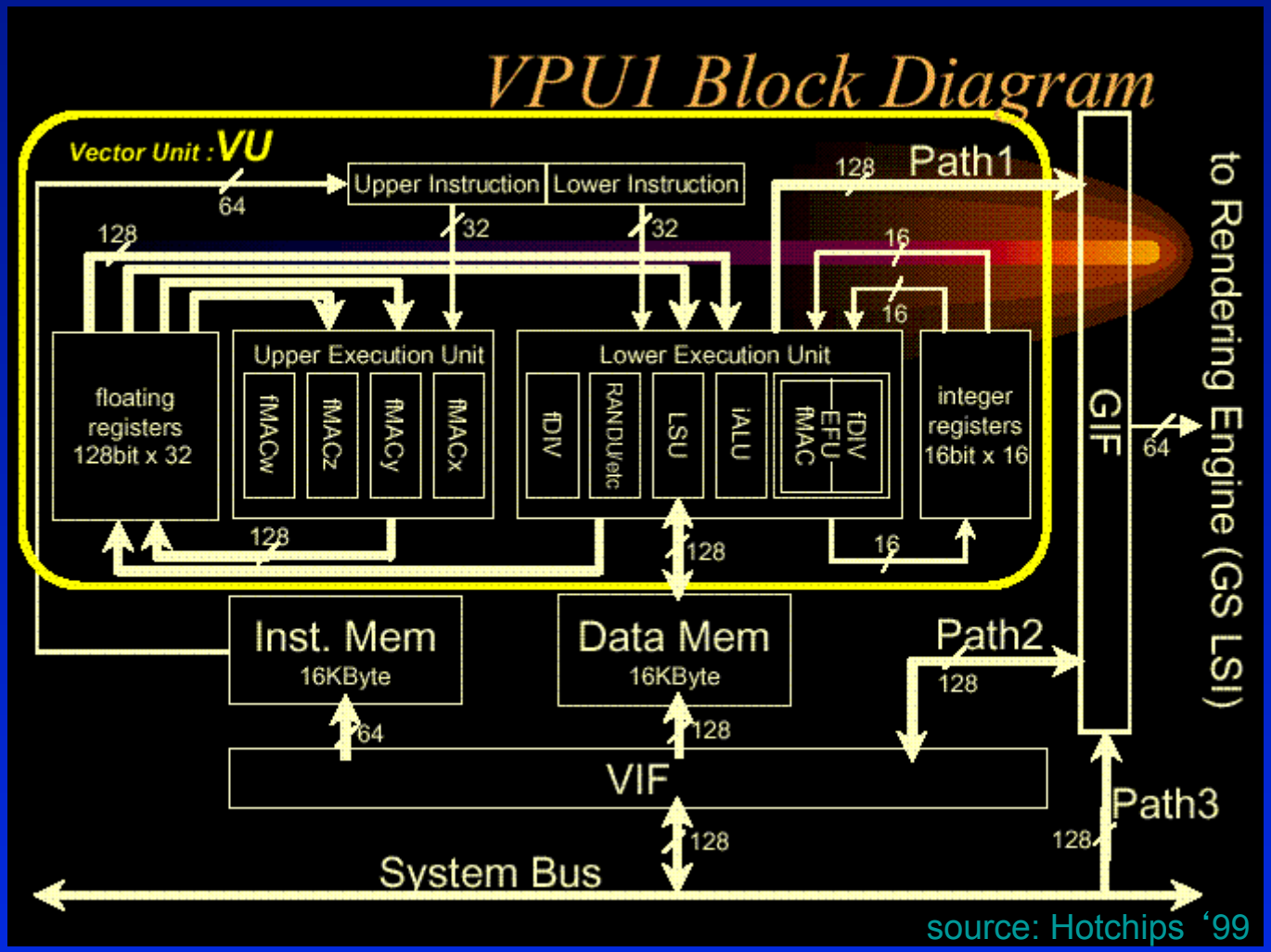
MIPS in Toshiba-Sony Playstation 2



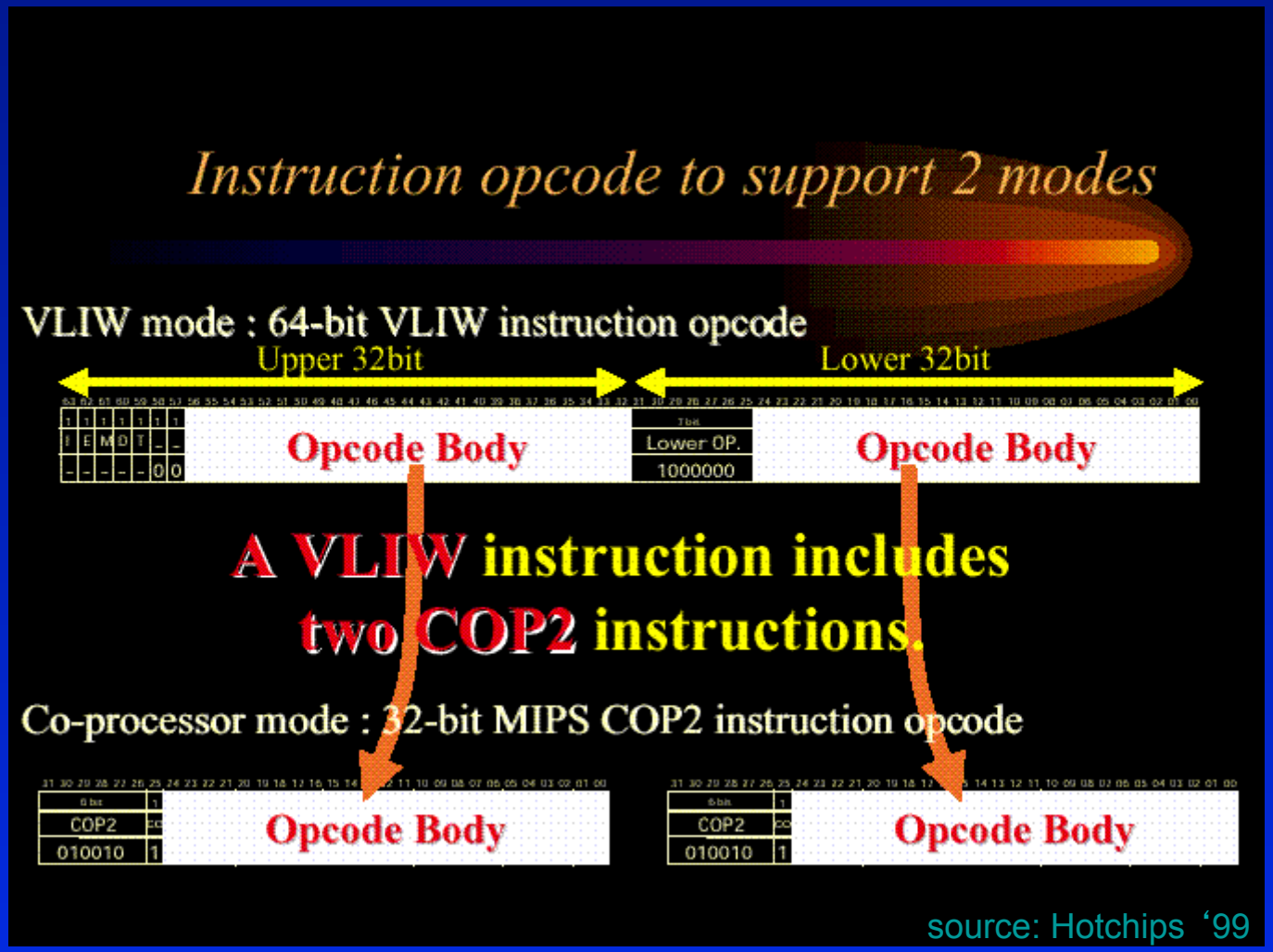
MIPS in Toshiba-Sony Playstation 2



MIPS in Toshiba-Sony Playstation 2

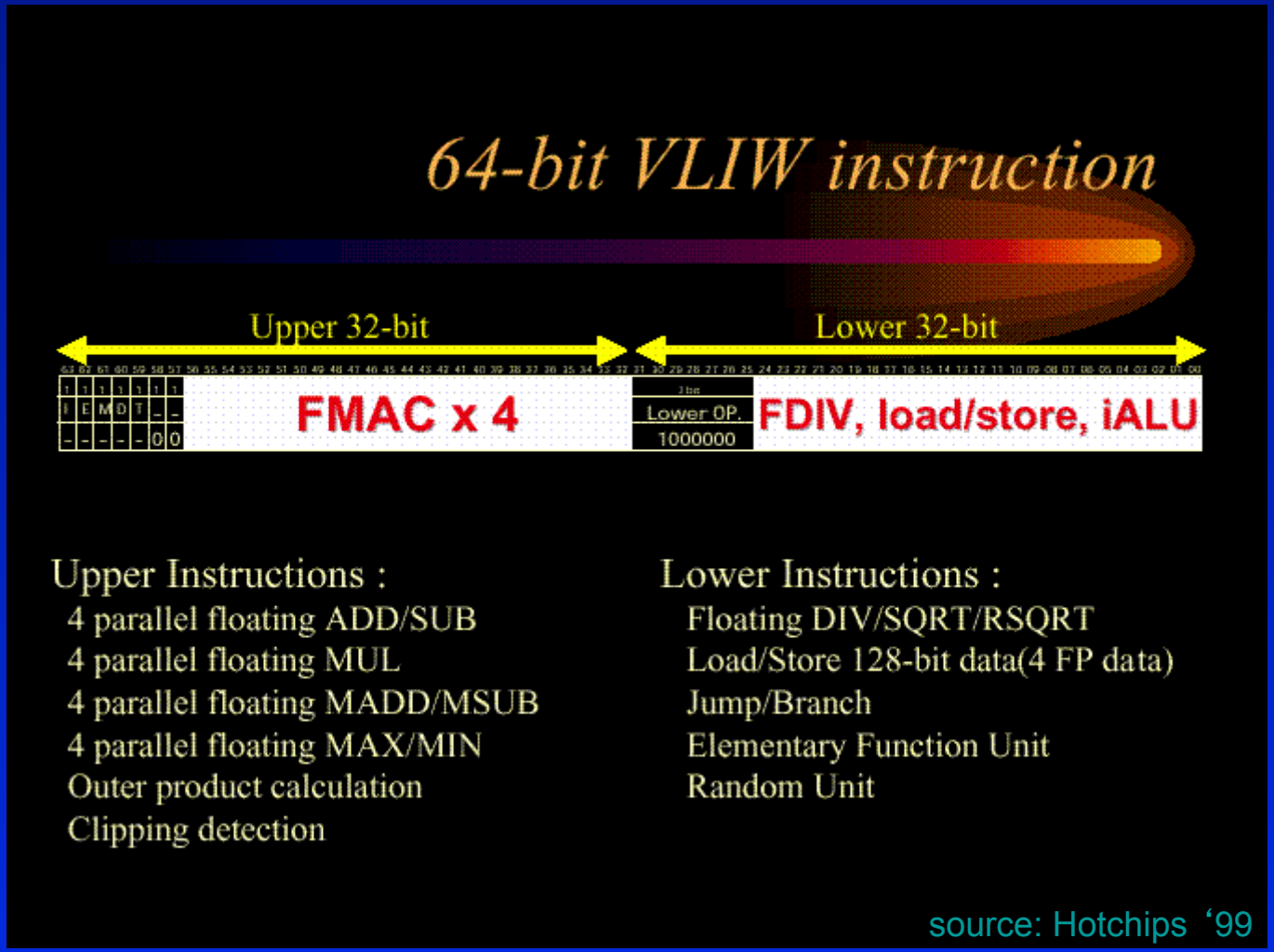


MIPS in Toshiba-Sony Playstation 2



source: Hotchips '99

MIPS in Toshiba-Sony Playstation 2



Upper Instructions :
- 4 parallel floating ADD/SUB
- 4 parallel floating MUL
- 4 parallel floating MADD/MSUB
- 4 parallel floating MAX/MIN
- Outer product calculation
- Clipping detection

Lower Instructions :
- Floating DIV/SQRT/RSQRT
- Load/Store 128-bit data(4 FP data)
- Jump/Branch
- Elementary Function Unit
- Random Unit

source: Hotchips '99

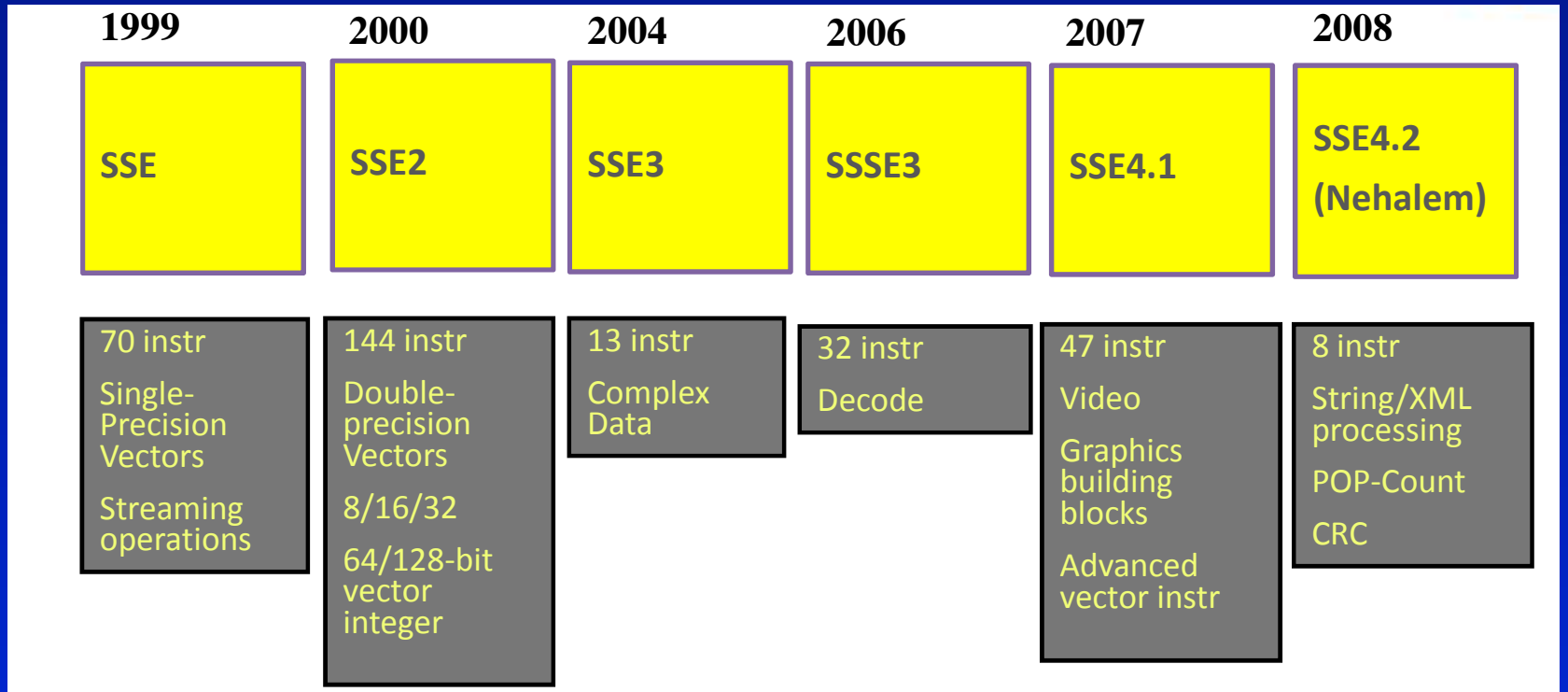
Practical Issues

- Benchmarking
  - SPECmedia informal proposal on MPEG2 (see URL in ref. [36])
  - practical comparisons are rare and scope-limited (our work on MAX2 and MMX see ref. [32])
  - MMX more throuroughlly covered (see ref. [33])
- Actual applications
  - image/video compression (standards)
  - speech processing
  - printer software chain
  - cryptography

Practical Issues

- Development environment
  - Art of Computer (assembly) Programming
  - Chip vendors' libraries
    - optimize single algorithms, not suited to chaining algorithms
  - *Intrinsics*
    - aliasing C variables to registers causes unnecessary memory accesses
  - Compilers
    - simple loops, no effective conditional execution

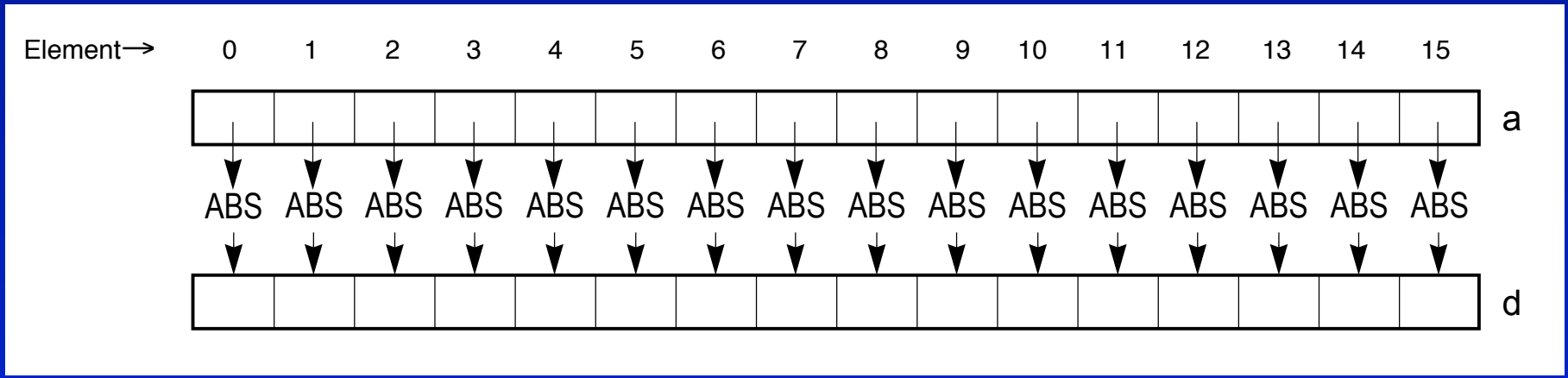
(MMX) SSEx INTEL



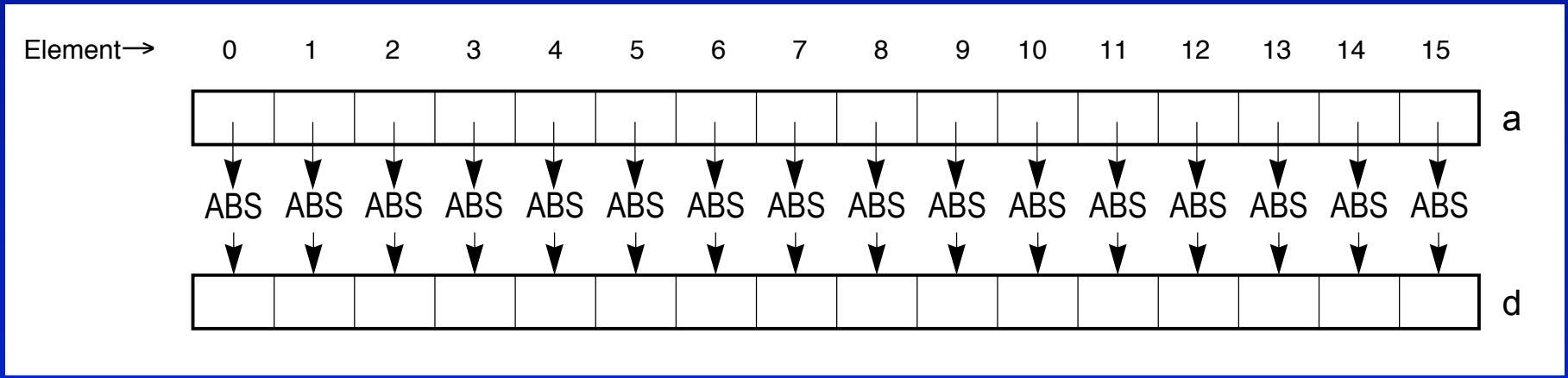
Continued by

- Intel® AES New Instructions - Intel® AES-NI (2009)
- Intel® Advanced Vector Extensions – Intel® AVX (2010/11)

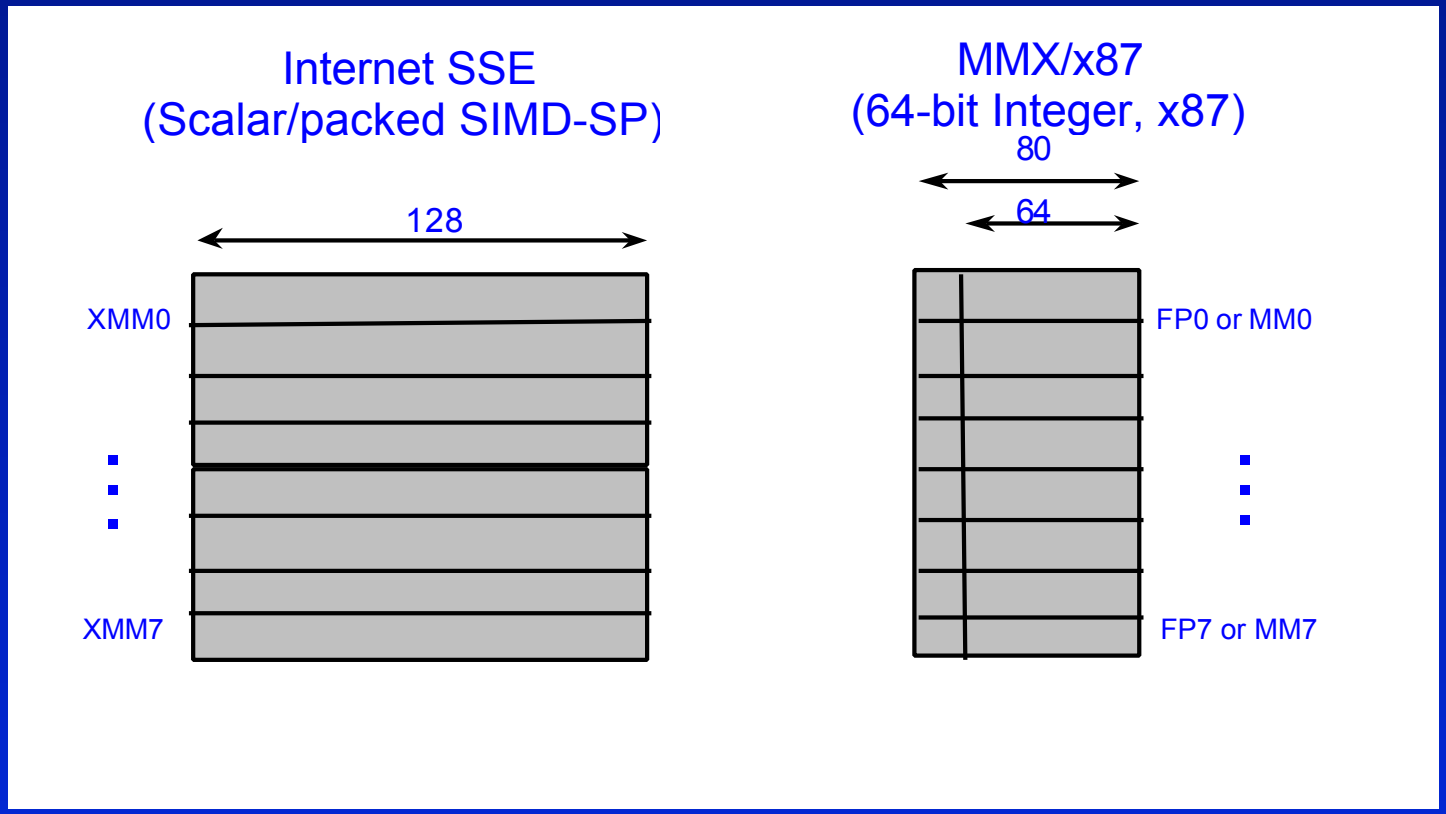
SSE (INTEL)


Element→
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
a
ABS ABS ABS ABS ABS ABS ABS ABS ABS ABS ABS ABS ABS ABS ABS ABS
d

SSE (INTEL)


Element→
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
a
ABS ABS ABS ABS ABS ABS ABS ABS ABS ABS ABS ABS ABS ABS ABS ABS
d

MMX SSE2 (INTEL)



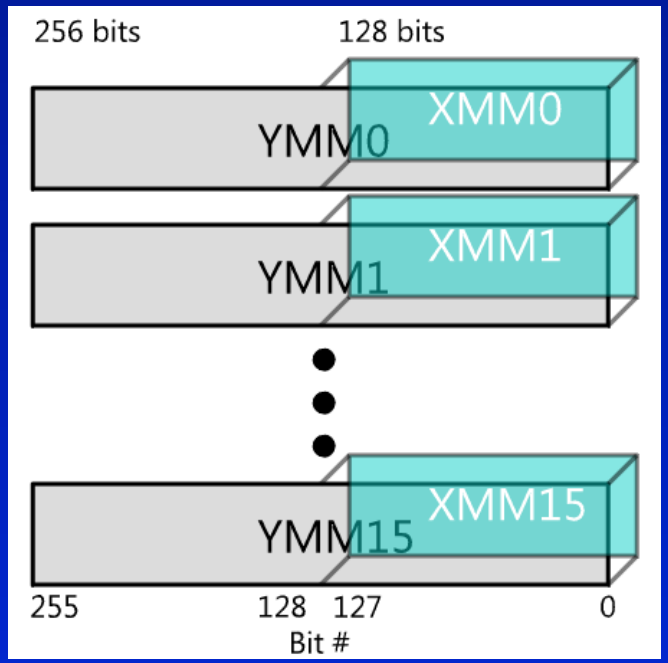
- REGISTERS

AVX (INTEL)

Intel® Advanced Vector Extensions (Intel® AVX) is a set of instructions for doing Single Instruction Multiple Data (SIMD) operations on Intel® architecture CPUs. These instructions extend previous SIMD offerings (MMX™ instructions and Intel® Streaming SIMD Extensions (Intel® SSE)) by adding the following new features:

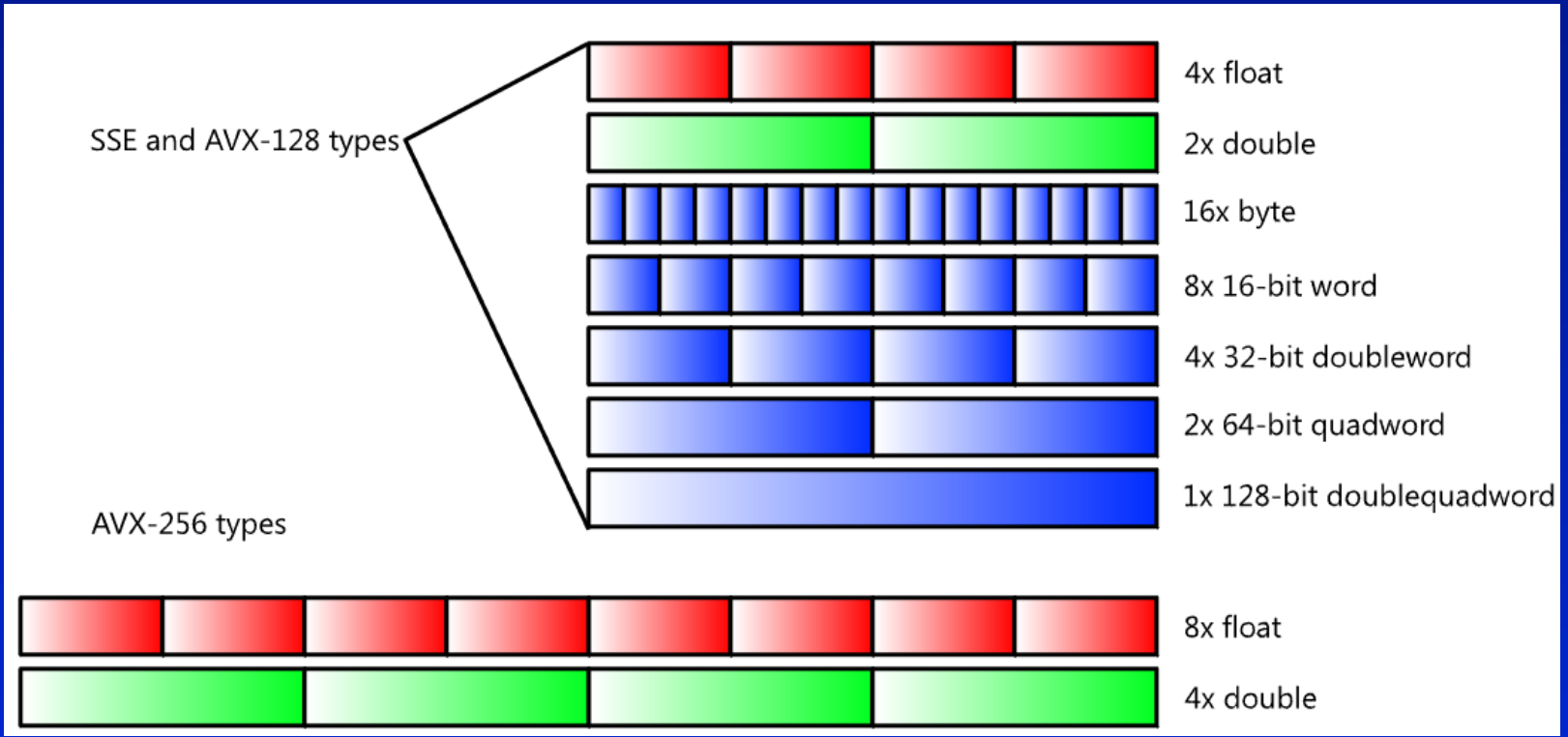
- The 128-bit SIMD registers have been expanded to 256 bits. Intel® AVX is designed to support 512 or 1024 bits in the future.
- Three-operand, nondestructive operations have been added. Previous two-operand instructions performed operations such as A = A + B, which overwrites a source operand; the new operands can perform operations like A = B + C, leaving the original source operands unchanged.
- A few instructions take four-register operands, allowing smaller and faster code by removing unnecessary instructions.
- Memory alignment requirements for operands are relaxed.
- A new extension coding scheme (VEX) has been designed to make future additions easier as well as making coding of instructions smaller and faster to execute.

AVX (INTEL)



- XMM registers overlay the YMM registers

AVX (INTEL)



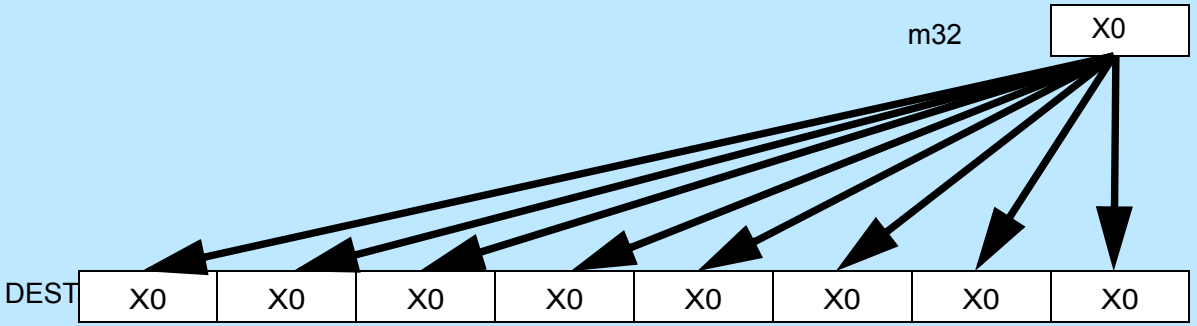
- AVX and SSE data types

AVX (INTEL)

Instruction	Description
VBROADCASTSS, VBROADCASTSD, VBROADCASTF128	Copy a 32-bit, 64-bit or 128-bit memory operand to all elements of a XMM or YMM vector register.
VINSERTF128	Replaces either the lower half or the upper half of a 256-bit YMM register with the value of a 128-bit source operand. The other half of the destination is unchanged.
VEXTRACTF128	Extracts either the lower half or the upper half of a 256-bit YMM register and copies the value to a 128-bit destination operand.
VMASKMOVPS, VMASKMOVPD	Conditionally reads any number of elements from a SIMD vector memory operand into a destination register, leaving the remaining vector elements unread and setting the corresponding elements in the destination register to zero. Alternatively, conditionally writes any number of elements from a SIMD vector register operand to a vector memory operand, leaving the remaining elements of the memory operand unchanged.
VPERMILPS, VPERMILPD	Shuffle 32-bit or 64-bit vector elements, with a register or memory operand as selector.
VPERM2F128	Shuffle the four 128-bit vector elements of two 256-bit source operands into a 256-bit destination operand, with an immediate constant as selector.
VZEROALL	Set all YMM registers to zero and tag them as unused. Used when switching between 128-bit use and 256-bit use.
VZEROUPPER	Set the upper half of all YMM registers to zero. Used when switching between 128-bit use and 256-bit use.

- New instructions

AVX (INTEL)



VPBROADCASTD Operation (VEX.256 encoded version)

VPBROADCASTD: __m256i _mm256_broadcastd_epi32(__m128i);

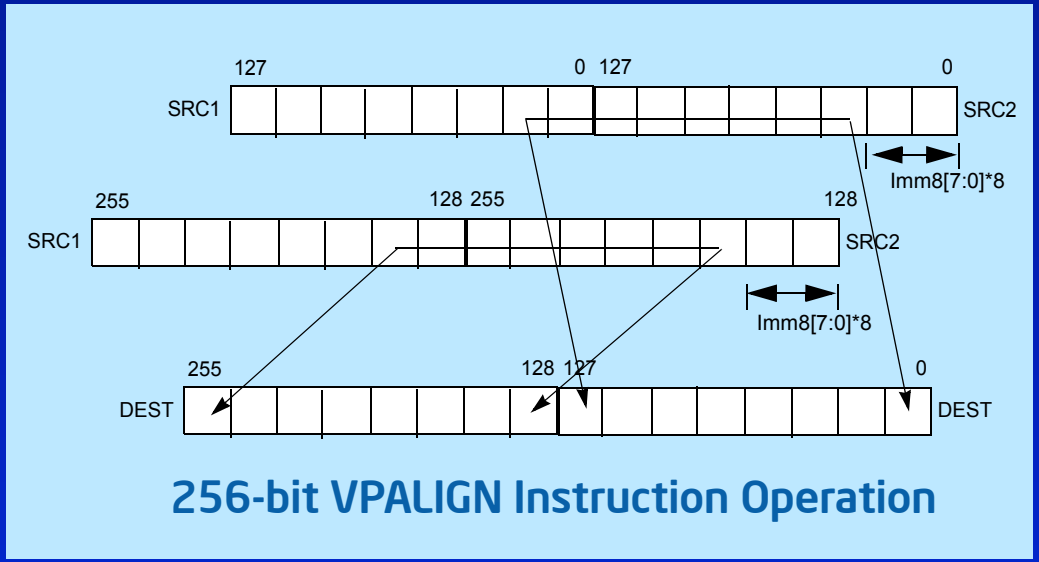
AVX (INTEL)

VPMASKMOV — Conditional SIMD Integer Packed Loads and Stores

VPMASKMOVD - 256-bit load

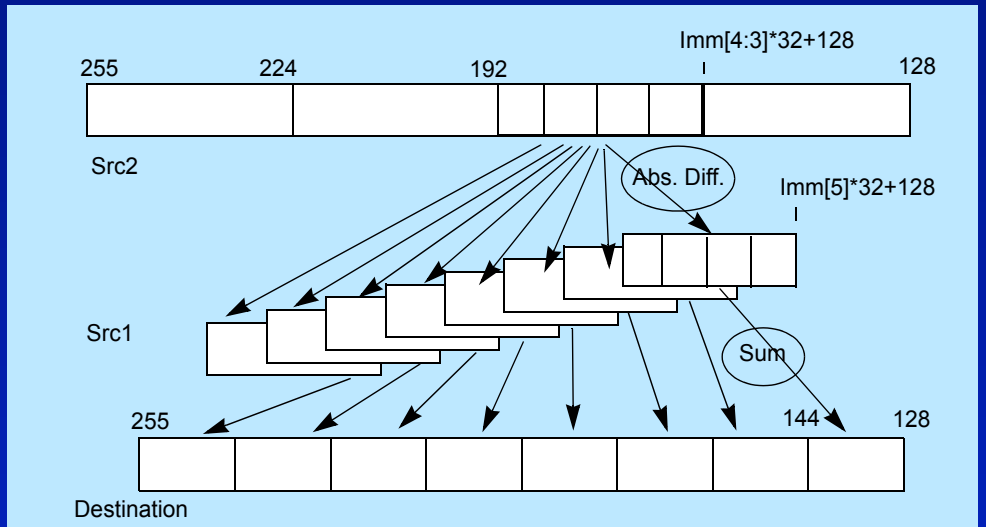
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0

AVX (INTEL)

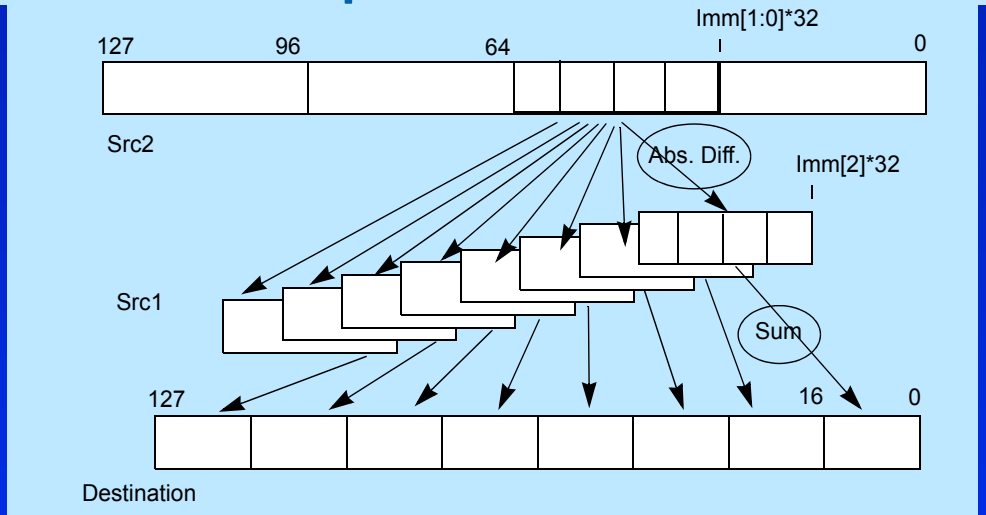


256-bit VPALIGN Instruction Operation

AVX (INTEL)



MPSADBW — Multiple Sum of Absolute Differences

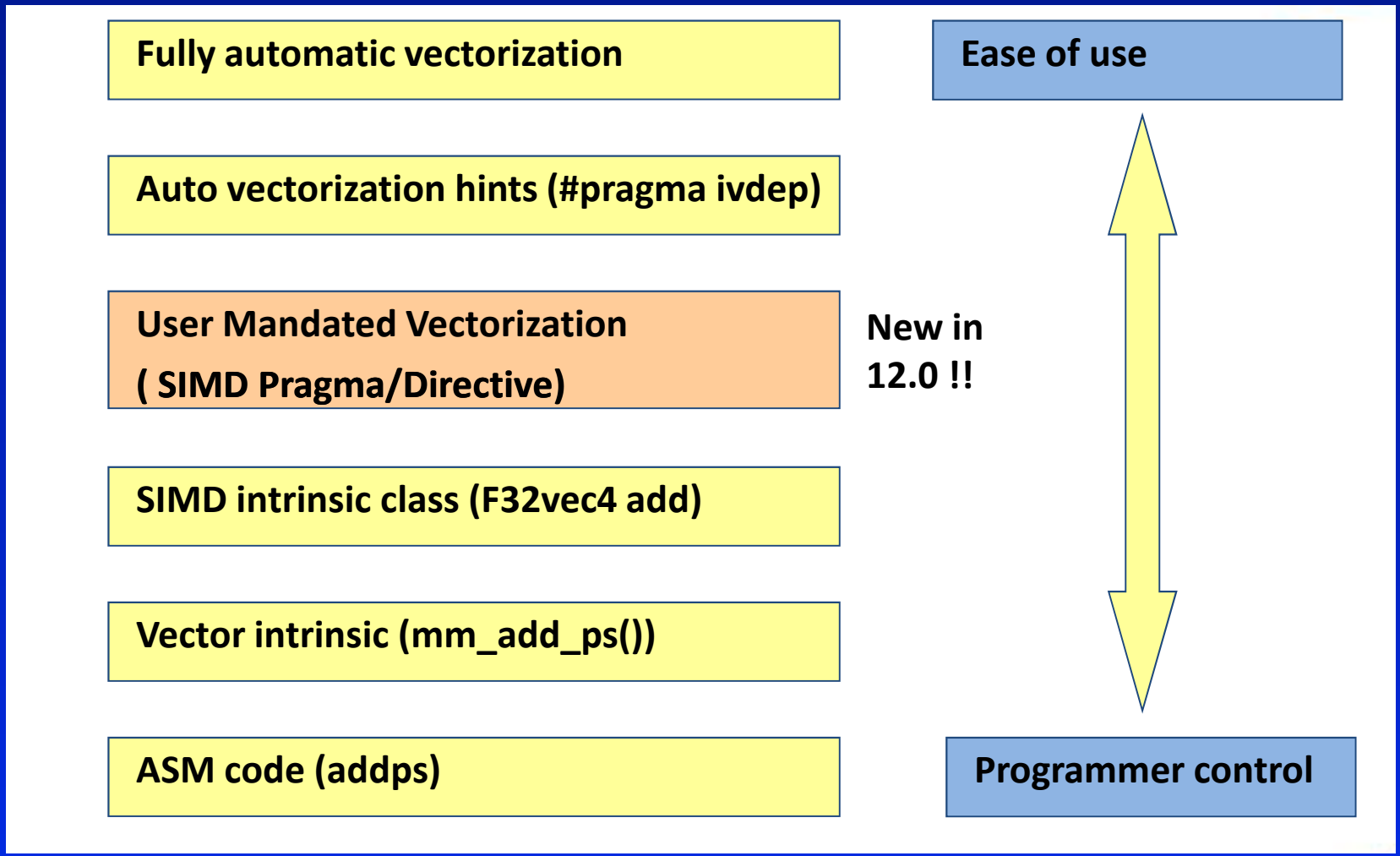


AVX (INTEL)

FMA	Each [z] is the string 132 or 213 or 231, giving the order the operands A,B,C are used in: 132 is A=AC+B 213 is A=AB+C 231 is A=BC+A
VFMADD[z][P/S][D/S]	Fused multiply add A = r1 * r2 + r3 for packed/scalar of double/single
VFMADDSUB[z]P[D/S]	Fused multiply alternating add/subtract of packed double/single A = r1 * r2 + r3 for odd index, A = r1 * r2-r3 for even
VFMSUBADD[z]P[D/S]	Fused multiply alternating subtract/add of packed double/single A = r1 * r2-r3 for odd index, A = r1 * r2+r3 for even
VFMSUB[z][P/S][D/S]	Fused multiply subtract A = r1 * r2-r3 of packed/scalar double/single
VFNMADD[z][P/S][D/S]	Fused negative multiply add of packed/scalar double/single A = -r1 * r2+r3
VFNMSUB[z][P/S][D/S]	Fused negative multiply subtract of packed/scalar double/single A = -r1 * r2-r3

- Future FMA: Fused Multiply Add

SSE AVX (INTEL)

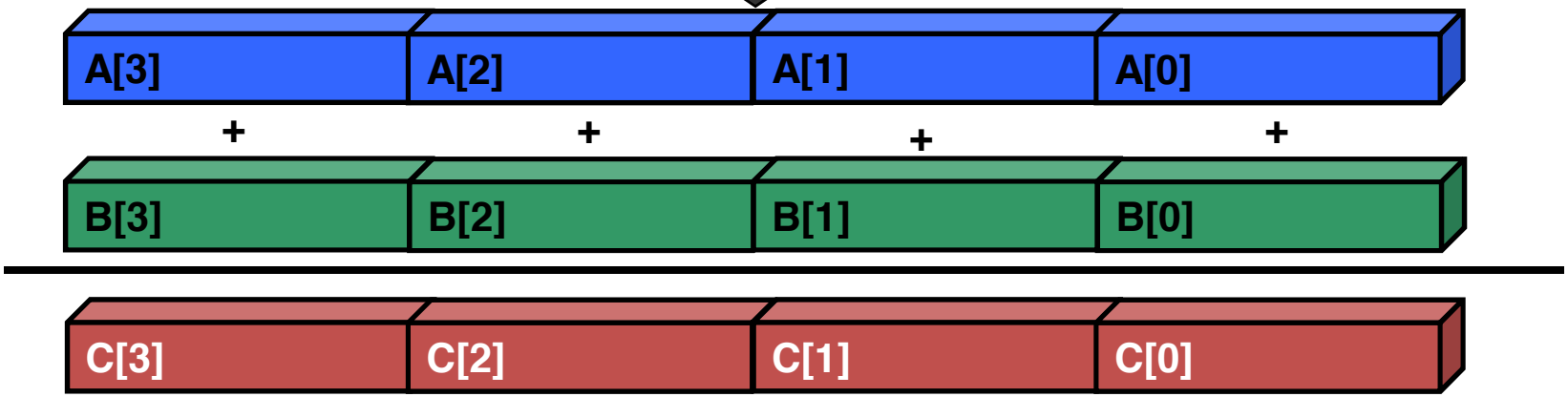


Automatic Vectorization

Transforming sequential code to exploit the vector (SIMD, SSE) processing capabilities

- Manually by explicit source code modification
- Automatically by tools like a compiler

```
for (i=0;i<MAX;i++)
   c[i]=a[i]+b[i];
```



(MMX) SSE INTEL

```
static double A[1000], B[1000],
              C[1000];
void add() {
  int i;
  for (i=0; i<1000; i++)
    if (A[i]>0)
      A[i] += B[i];
    else
      A[i] += C[i];
}
```

```
.B1.2::
  movaps    xmm2, A[rdx*8]
  xorps     xmm0, xmm0
  cmpltpd   xmm0, xmm2
  movaps    xmm1, B[rdx*8]
  andps     xmm1, xmm0
  andnps    xmm0, C[rdx*8]
  orps      xmm1, xmm0
  addpd     xmm2, xmm1
  movaps    A[rdx*8], xmm2
  add       rdx, 2
  cmp       rdx, 1000
  jl        .B1.2
```

SSE2

```
.B1.2::
  vmovaps    ymm3, A[rdx*8]
  vmovaps    ymm1, C[rdx*8]
  vcmpgtpd   ymm2, ymm3, ymm0
  vblendvpd  ymm4, ymm1,B[rdx*8], ymm2
  vaddpd     ymm5, ymm3, ymm4
  vmovaps    A[rdx*8], ymm5
  add        rdx, 4
  cmp        rdx, 1000
  jl         .B1.2
```

AVX

```
.B1.2::
  movaps    xmm2, A[rdx*8]
  xorps     xmm0, xmm0
  cmpltpd   xmm0, xmm2
  movaps    xmm1, C[rdx*8]
  blendvpd  xmm1, B[rdx*8], xmm0
  addpd     xmm2, xmm1
  movaps    A[rdx*8], xmm2
  add       rdx, 2
  cmp       rdx, 1000
  jl        .B1.2
```

SSE4.1

AVX (INTEL)

- Intrinsics

```
_mm256_op_suffix(data_type param1, data_type param2, data_type param3)
```

where `_mm256` is the prefix for working on the new 256-bit registers; `_op` is the operation, like `add` for addition or `sub` for subtraction; and `_suffix` denotes the type of data to operate on, with the first letters denoting packed (**p**), extended packed (**ep**), or scalar (**s**). The remaining letters are the

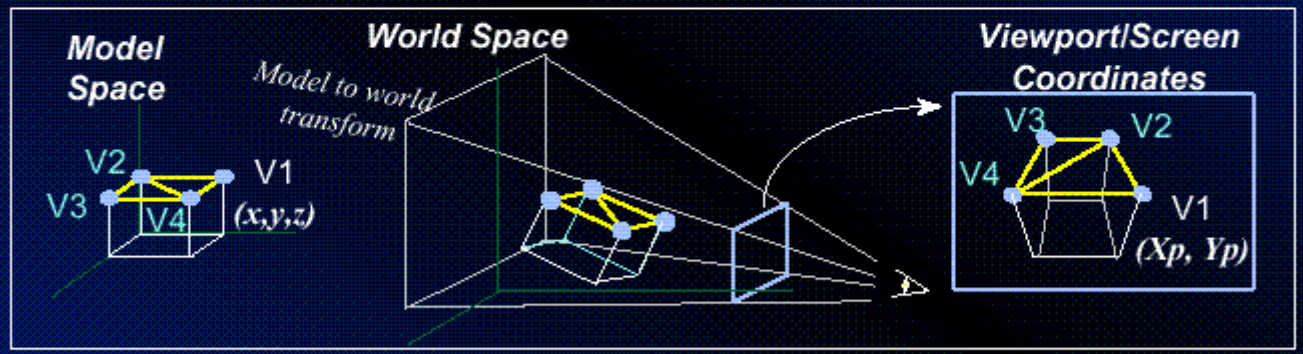
Marking	Meaning
`[s/d]`	Single- or double-precision floating point
`[i/u]nnn`	Signed or unsigned integer of bit size *nnn,* where *nnn* is 128, 64, 32, 16, or 8
`[ps/pd/sd]`	Packed single, packed double, or scalar double
`epi32`	Extended packed 32-bit signed integer
`si256`	Scalar 256-bit integer

Type	Meaning
`__m256`	256-bit as eight single-precision floating-point values, representing a YMM register or memory location
`__m256d`	256-bit as four double-precision floating-point values, representing a YMM register or memory location
`__m256i`	256-bit as integers, (bytes, words, etc.)
`__m128`	128-bit single precision floating-point (32 bits each)
`__m128d`	128-bit double precision floating-point (64 bits each)

AVX (INTEL)

3D Geometry is Data Parallel

- Compute x, y, z in parallel per vertex
- Compute multiple vertices in parallel



SIMD FP is best option to deliver > 2x perf. gain

Foil 6