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 parallelism (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 lighting)
  - vertex transformation (matrix multiplication)
  - clipping (compare and branch)
  - displaying to screen (perspective division)
Media Processing

- MPEG-1 decompression on HP 735

<table>
<thead>
<tr>
<th>Task</th>
<th>MOI</th>
<th>%Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header decode</td>
<td>0.6</td>
<td>0.1</td>
</tr>
<tr>
<td>Huffman decode</td>
<td>55.3</td>
<td>7.5</td>
</tr>
<tr>
<td>Inv. Quantization</td>
<td>8.7</td>
<td>2.4</td>
</tr>
<tr>
<td>IDCT</td>
<td>206.5</td>
<td>38.7</td>
</tr>
<tr>
<td>Motion comp.</td>
<td>79.9</td>
<td>18.3</td>
</tr>
<tr>
<td>Display</td>
<td>188.7</td>
<td>33.0</td>
</tr>
<tr>
<td>Total</td>
<td>539.7</td>
<td>100</td>
</tr>
</tbody>
</table>

- (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
    
    system level
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 tailored 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 microarchitecture’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

<table>
<thead>
<tr>
<th>Packet word</th>
<th>Long word</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item 1</td>
<td>Item 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Packet half-word</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Item 3</td>
<td>Item 2</td>
</tr>
</tbody>
</table>

| Packet byte      | |
|------------------|--|--|--|--|--|--|--|
| Item 7           | Item 6 | Item 5 | Item 4 | Item 3 | Item 2 | Item 1 | Item 0 |

- Packet word: 64-bit
- Packet half-word: 32-bit
- Packet byte: 16-bit
- Packet byte: 8-bit

---

Marco Ferretti, University of Pavia

CINI - FNM ASS
# Packed Integer Data Type

<table>
<thead>
<tr>
<th>Packet word</th>
<th>Item 3</th>
<th>Item 2</th>
<th>Item 1</th>
<th>Item 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet half-word</td>
<td>Item 7</td>
<td>Item 6</td>
<td>Item 5</td>
<td>Item 4</td>
</tr>
<tr>
<td>Packet byte</td>
<td>It.15</td>
<td>It.14</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

- **Quad word**: 128-bit
- **Packet word**: 32-bit
- **Packet half-word**: 16-bit
- **Packet byte**: 8-bit
Packed Float Data Type

- Long word (64-bit)
  - float 1
  - float 0 (32-bit)

- Quad word (128-bit)
  - float 3
  - float 2
  - float 1
  - float 0 (32-bit)
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.
  - Optimized 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 of 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

<table>
<thead>
<tr>
<th>mm1</th>
<th>1A00h</th>
<th>2A00h</th>
<th>3A00h</th>
<th>4A00h</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm2</td>
<td>F700h</td>
<td>F700h</td>
<td>F700h</td>
<td>F700h</td>
</tr>
<tr>
<td>mm1</td>
<td>1100h</td>
<td>2100h</td>
<td>3100h</td>
<td>4100h</td>
</tr>
<tr>
<td>mm1</td>
<td>FFFFh</td>
<td>FFFFh</td>
<td>FFFFh</td>
<td>FFFFh</td>
</tr>
</tbody>
</table>
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

\[
\begin{align*}
\text{SQRT}(2) & \approx 1.4142_{10} = 1.01101010_2 \\
T &= 1.01 \times X \quad T = X + X \gg 2 \quad \text{PshRadd} \; X,2,X,T \\
S &= 1.0101 \times X \quad S = X + T \gg 2 \quad \text{PshRadd} \; T,2,X,S \\
T &= 1.0110101 \times X \quad T = T + S \gg 3 \quad \text{PshRadd} \; S,3,T,T
\end{align*}
\]
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 light 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 common
  - sum-across
  - sum-of-products
    - advanced version of Multiply-ADD; simplifies 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

```c
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 assignment
  
  \[
  \text{if } \text{cond}(a_i, b_i) \text{ then } c_i = s_i \text{ else } c_i = t_i \quad i=1,n
  \]

*parallel subword compare* instruction generate MASKS or Condition BITS

- Self conditional assignment
  
  \[
  \text{if } \text{cond}(a_i, b_i) \text{ then } c_i = a_i \text{ else } c_i = b_i \quad i=1,n
  \]

max and min with saturation arithmetic
Instruction classes - Conditional exec.

- Mask mode

<table>
<thead>
<tr>
<th>m1</th>
<th>mm2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>X1</td>
</tr>
<tr>
<td>X2</td>
<td>X2</td>
</tr>
<tr>
<td>X3</td>
<td>X3</td>
</tr>
<tr>
<td>X4</td>
<td>X4</td>
</tr>
</tbody>
</table>

Pcmpeqw mm1,mm2

<table>
<thead>
<tr>
<th>mm1</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFFh</td>
</tr>
<tr>
<td>0000h</td>
</tr>
<tr>
<td>FFFFh</td>
</tr>
<tr>
<td>FFFFh</td>
</tr>
</tbody>
</table>
Instruction classes - Conditional exec.

- **Mask mode: implementation of image overlay**

<table>
<thead>
<tr>
<th>mm2</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
<th>X4</th>
<th>mm3</th>
<th>Y1</th>
<th>Y2</th>
<th>Y3</th>
<th>Y4</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm1</td>
<td>bkgd</td>
<td>bkgd</td>
<td>bkgd</td>
<td>bkgd</td>
<td>mm1</td>
<td>FFFFh</td>
<td>0000h</td>
<td>FFFFh</td>
<td>FFFFh</td>
</tr>
<tr>
<td>mm2</td>
<td>0000h</td>
<td>X2</td>
<td>0000h</td>
<td>0000h</td>
<td>mm2</td>
<td>Y1</td>
<td>0000h</td>
<td>Y3</td>
<td>Y4</td>
</tr>
<tr>
<td>mm3</td>
<td>mm3</td>
<td>Y1</td>
<td>X2</td>
<td>Y3</td>
<td>Y4</td>
<td>mm3</td>
<td>mm3</td>
<td>mm3</td>
<td>mm3</td>
</tr>
</tbody>
</table>

- Pcmpeqw mm1,mm2
- Pandn mm2,mm1
- Pand mm3,mm1
- Por mm3,mm2
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 $c_i = b_i$ \hspace{1cm} i=1,n

  **MAX** cond($a_i, b_i$) \hspace{1cm} $a_i > b_i$ ; **MIN** cond($a_i, b_i$) \hspace{1cm} $a_i < b_i$

<table>
<thead>
<tr>
<th>Ra</th>
<th>40</th>
<th>5</th>
<th>78</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb</td>
<td>51</td>
<td>34</td>
<td>15</td>
<td>243</td>
</tr>
<tr>
<td>Rc</td>
<td>0</td>
<td>0</td>
<td>63</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hadd Rc,Rb,Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rc</td>
</tr>
<tr>
<td>51</td>
</tr>
<tr>
<td>34</td>
</tr>
<tr>
<td>78</td>
</tr>
<tr>
<td>243</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hsub,us Ra,Rb,Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rc</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>200</td>
</tr>
</tbody>
</table>

Marco Ferretti, University of Pavia

CINI - FNM ASS -48-
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 alignment
- Partial store vs Write Combining store
- Block store
Instruction classes - Memory access

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

aligned boundary address of destination data = falignaddr(da, offset)

dp = x10000  x10008
da = x10005 Data Start Address

vis_alignaddr(x10005, 0) returns x10000 with five placed in the GSR offset field.

aligned boundary

vis_faligndata(data_hi, data_lo) returns the shaded data segment.
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*
  - alignment 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 management.

- 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

<table>
<thead>
<tr>
<th>General</th>
<th>MAX2</th>
<th>VIS</th>
<th>MMX</th>
<th>MDMX</th>
<th>Altivec</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Formats</strong></td>
<td>16</td>
<td>(8),16,32</td>
<td>8,16,32</td>
<td>8,16,(24)</td>
<td>8,16,32</td>
<td>32</td>
</tr>
<tr>
<td><strong># instr.</strong></td>
<td>17</td>
<td>81</td>
<td>57</td>
<td>35</td>
<td>162</td>
<td>70</td>
</tr>
<tr>
<td><strong># regist./depth</strong></td>
<td>31 i / 64</td>
<td>32 f / 64</td>
<td>8 f / 64</td>
<td>32 f / 64</td>
<td>32 s / 128</td>
<td>8 s / 128</td>
</tr>
<tr>
<td><strong>paired/single</strong></td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>n</td>
<td>y</td>
</tr>
</tbody>
</table>
# Tour: Extensions at a Glance

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>MAX2</th>
<th>VIS</th>
<th>MMX</th>
<th>MDMX</th>
<th>Altivec</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>saturation</td>
<td>y</td>
<td>n</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y.a.</td>
</tr>
<tr>
<td>accumulator</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>y</td>
<td>n</td>
</tr>
<tr>
<td>multiply</td>
<td>n</td>
<td>8x16 to 16</td>
<td>16x16 to 16</td>
<td>8x8 to 16 i</td>
<td>8x8 to 8 i</td>
<td>32x32 to 32 f</td>
</tr>
<tr>
<td>multiply-add</td>
<td>n</td>
<td>n</td>
<td>16x16 to 32</td>
<td>8x8 to 24 i</td>
<td>8x8 to 16 i</td>
<td>n</td>
</tr>
<tr>
<td>shift-add</td>
<td>y</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>shift</td>
<td>y</td>
<td>n</td>
<td>y</td>
<td>y i</td>
<td>y i</td>
<td>n</td>
</tr>
<tr>
<td>division/SQRT</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>y</td>
</tr>
</tbody>
</table>
# Tour: Extensions at a Glance

<table>
<thead>
<tr>
<th>Reduction</th>
<th>MAX2</th>
<th>VIS</th>
<th>MMX</th>
<th>MDMX</th>
<th>Altivec</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>average</td>
<td>y</td>
<td>n</td>
<td>y</td>
<td>n</td>
<td>y</td>
<td>n</td>
</tr>
<tr>
<td>vector sum</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>n</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compare</th>
<th>MAX2</th>
<th>VIS</th>
<th>MMX</th>
<th>MDMX</th>
<th>Altivec</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>max-min</td>
<td>sat.</td>
<td>sat.</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>compare</td>
<td>n</td>
<td>16, 32</td>
<td>8, 16, 32</td>
<td>16, 32</td>
<td>8, 16, 32</td>
<td>32</td>
</tr>
<tr>
<td>cond. assign.</td>
<td></td>
<td>bit</td>
<td>mask</td>
<td>cc</td>
<td>(cc) mask</td>
<td></td>
</tr>
</tbody>
</table>
# Tour: Extensions at a Glance

<table>
<thead>
<tr>
<th>Data Manag.</th>
<th>MAX2</th>
<th>VIS</th>
<th>MMX</th>
<th>MDMX</th>
<th>Altivec</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pack</td>
<td>n</td>
<td>32 to 16, 16 to 8, 32 to 8</td>
<td>32 to 16, 16 to 8</td>
<td>64 to 32</td>
<td>32 to 16, 16 to 8 RBG</td>
<td>n</td>
</tr>
<tr>
<td>Unpack</td>
<td>n</td>
<td>8 to 16, 16 to 32, 32 to 64</td>
<td>8 to 16, 16 to 32, 32 to 64</td>
<td>8 to 16, 16 to 32</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>Mix</td>
<td>16, 32</td>
<td>8</td>
<td>n</td>
<td>n</td>
<td>8, 16, 32</td>
<td>32</td>
</tr>
<tr>
<td>Shuffle</td>
<td>16 (rep)</td>
<td>n</td>
<td>16</td>
<td>8, 16</td>
<td>8 (rep, alloallo)</td>
<td>32 (rep)</td>
</tr>
</tbody>
</table>
## Tour: Extensions at a Glance

<table>
<thead>
<tr>
<th>Memory</th>
<th>MAX2</th>
<th>VIS</th>
<th>MMX</th>
<th>MDMX</th>
<th>Altivec</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>unalign</td>
<td>y</td>
<td>y</td>
<td>n</td>
<td>y</td>
<td>n</td>
<td>y</td>
</tr>
<tr>
<td>block load/store</td>
<td>n</td>
<td>y</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>partial store</td>
<td>8</td>
<td>8,16,32</td>
<td>8 (wc)</td>
<td>8,16,32</td>
<td>32 (wc)</td>
<td></td>
</tr>
<tr>
<td>cache hints</td>
<td>y</td>
<td>n</td>
<td>y (w)</td>
<td>y (r,w)</td>
<td>y (w)</td>
<td></td>
</tr>
<tr>
<td>prefetch</td>
<td>y (r,w)</td>
<td>y (t,s)</td>
<td>y (t,s)</td>
<td>y (t,s)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Marco Ferretti, University of Pavia

CINI - FNM ASS
Tour: Extensions at a Glance

<table>
<thead>
<tr>
<th>Special</th>
<th>MAX2</th>
<th>VIS</th>
<th>MMX</th>
<th>MDMX</th>
<th>Altivec</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAD</td>
<td>n</td>
<td>8</td>
<td>8</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Array addr.</td>
<td>n</td>
<td>y</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Vector const.</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

Marco Ferretti, University of Pavia

CINI - FNM ASS -66-
ALTIVEC (Motorola)

Instruction Stream

Dispatch Unit

INST INST INST

Integer Unit

Floating-Point Unit

Vector Units

GPRs (32 Bits)

FPRs (64 Bits)

VRs (128 Bits)

Cache/Memory

vA vB vC vT
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)

![Diagram showing vector addition]

- Each vector operand is 1:8D bit wide.
- These 1:8D bit operands are loaded into registers.
- The registers hold the data source for the ALTI vecor arithmetic operations.
- The result of these operations is stored in the destination vector.
- ALTI vecor arithmetic operations provide support for high precision scientific calculations often not optimally performed in communications.
- This can be contrasted with the operation of other machines where hundreds or even thousands of operations are performed in parallel, simultaneously executing up to 1E operations in a single instruction.

This new engine provides highly parallel operations allowing for the simultaneous execution of up to 1E operations in a single instruction. This is in contrast to existing integer and floating point units. This new engine includes separate vector execution units that operate concurrently with an existing PowerPC ar.

The ALTI vector technology offers 1E-way parallelism for 8D bit signed and unsigned integers. It expands the current presentation of four operand, 16-element, intra-element operations to accommodate vector parallelism in computations on the elements contained in the vector.

The ALTI vecor technology offers support for 8D element operations and provides better support for scientific computations. A variety of operations, including saturation and modular arithmetic, are supported. ALTI is analogous to the addition of floating point numbers and other operations.

These operations correspond to fields of the destination vector operand. Intra-element arithmetic operations are performed on the elements contained in the vector.

Data elements are loaded into single instruction operands. All operands are available for multimedia and other operations.

PowerPC architecture provides better support for multimedia and other operations, offering highly parallel operations that dramatically accelerate operations at the next level of operation.
ALTIVEC (Motorola)

- PERMUTE (All to All)

![Diagram showing the PERMUTE operation (All to All) in the ALTIVEC architecture.](image-url)
ALTIVEC (Motorola)

- **Table lookup**

32 entry table $T[0..31]$ in 2 vector registers $v1$ and $v2$

- $vperm V4,V1,V2,V3$

- 16 parallel indexes

- 16 parallel table lookups in one cycle
One slice permits accumulation of 256 8x8 or 65536 16x16 multiples with only a single shift/round error at the end.
MDMX (MIPS)
MIPS in Toshiba-Sony Playstation 2

"Emotion Engine" Block Diagram

- COP1 FPU
- COP2
- VU0
- VU1
- VIF0
- VIF1
- CPU
- VPU0
- VPU1
- System Bus 128-bit
- 10ch DMAC
- IPU
- Memory Interface
- I/O Interface
- External Memory
- Peripherals

source: Hotchips '99
MIPS in Toshiba-Sony Playstation 2

EE Micrograph

Die size : 15.02mm x 15.04mm
Frequency : 300MHz
Transistors : 13.5M
Power : 18Watts
Design Rule : 0.25um
Gate Length : 0.18um

source: Hotchips ‘99

Marco Ferretti, University of Pavia
MIPS in Toshiba-Sony Playstation 2

- VLIW mode
  - 64-bit VLIW instruction formats
  - 5 function units are available simultaneously
    4 FMAC + (FDIV, Branch, load/store, or iALU)
- Co-processor mode
  - MIPS COP2 instruction : 32-bit opcode
  - Two kinds of COP2 instructions :
    - 4 parallel Floating operations
    - call micro-subroutine of VLIW mode

source: Hotchips ‘99
MIPS in Toshiba-Sony Playstation 2

VPU1 Block Diagram

Source: Hotchips '99
MIPS in Toshiba-Sony Playstation 2

**Instruction opcode to support 2 modes**

**VLIW mode:** 64-bit VLIW instruction opcode
- Upper 32bit
- Lower 32bit

**A VLIW instruction includes two COP2 instructions.**

**Co-processor mode:** 32-bit MIPS COP2 instruction opcode

*source: Hotchips '99*
MIPS in Toshiba-Sony Playstation 2

64-bit VLIW instruction

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 thoroughly 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
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

ABS  ABS  ABS  ABS  ABS  ABS  ABS  ABS  ABS  ABS  ABS  ABS  ABS  ABS  ABS

a

d
SSE (INTEL)

Vector Absolute Value Saturated

\[ \delta = \text{ABS} \]

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

ABS   ABS   ABS   ABS   ABS   ABS   ABS   ABS   ABS   ABS   ABS   ABS   ABS   ABS   ABS

\[ \delta = 0 \]
The Internet Streaming SIMD Extensions mentioned, 128-bit execution is actually performed in 64-bit chunks and yet the peak rate of one 128-bit operation can be sustained if, as commonly occurs, instructions alternate between different execution units (i.e., add-multiply-add-multiply). Implementing a 256-bit wide SIMD unit would require doubling the width of execution units in order to still attain peak throughput in the same manner. Increasing SIMD-width beyond 128 bits would also require an increase in memory bandwidth in order to feed the wider execution units. There is a cost to this additional bandwidth, which may not follow Moore’s Law progression, required by other application areas. Also, since the primary focus for the extensions has been on 3D geometry, greater than 4-wide parallelism may offer diminishing returns, since triangular strip lengths in current desktop 3D applications tend to be fairly small (i.e., on the order of 20 vertices per strip).

Related to this decision were the following two issues:

- **State Space**: overlap or new registers
- **Pentium® III processor implementation**

**State Space**

There were two choices: overlap the new state with the MMX/x87 FP registers or add a new state. One big advantage of the first choice is that it would not require any operating system (OS) changes, just like the MMX technology extension. However, there were many disadvantages with this choice. First, we could only implement four 4-wide 128-bit registers in the existing space since we only had eight 80-bit registers, or we could go to a 2-wide format, thus sacrificing potential performance gains. Second, we would be forced to share the state with MMX registers, which was an issue for the already register-starved IA-32 architecture. The complexity of adding another set of overlapped state was overwhelming.

Adding a new state had the advantage of reducing implementation complexity and easing programming model issues. SIMD-FP and MMX or x87 instructions can be used concurrently. This clearly eased OS Vendor and ISV concerns. The disadvantage of the second approach was that Intel had a dependency of not being able to use new features without OS support. However, Intel worked around this by implementing the new state store and restore instructions in an earlier implementation. Thus by the time the Pentium III processor was released, the new OS’s supported this new state.

To ensure no unusual corner cases, all of the new state was separated from the x87-FP state. Figure 1 shows the new 128-bit registers. There is a new control/status register MXCSR which is used to mask/unmask numerical exception handling, to set rounding modes, to set flush-to-zero mode, and to view status flags.

**Figure 1: The Internet SSE 128-bit registers**

There is also a new interrupt vector to handle SIMD-FP numeric exceptions.

**Pentium® III Processor Implementation**

The Pentium III processor implements each 4-wide computational macro-instruction as two 64-bit micro-instructions. However, since the processor is a superscalar implementation (i.e., two execution ports), a full 4-wide SIMD operation can also be done every clock cycle (assuming instructions alternate between execution units). With this approach, applications can theoretically achieve a full 4x performance gain; 2x is the realized gain on real applications in part because of micro-instruction pressure within the microarchitecture. A future 128-bit implementation can deliver a higher level of performance scaling.

**Scalar Versus Packed Operations**

We considered the inclusion of scalar floating instructions in the new SIMD-FP mode because applications often require both scalar and packed operations. It is possible to use x87-FP for scalar and the new registers just for SIMD-FP. However, this approach results in a cumbersome programming paradigm, since x87-FP is a stack register model while the SIMD-FP is a flat register model. Passing parameters would either require more conversion instructions or would be through memory, as currently implemented. Additionally, the results generated via x87-FP operations might differ from SIMD-FP results, due to differences between how computation is performed in the two paradigms (32 bit in SIMD-FP versus 80 bit in x87 FP).
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)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBROADCASTSS, VBROADCASTSD, VBROADCASTF128</td>
<td>Copy a 32-bit, 64-bit or 128-bit memory operand to all elements of a XMM or YMM vector register.</td>
</tr>
<tr>
<td>VINSERTF128</td>
<td>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.</td>
</tr>
<tr>
<td>VEXTRACTF128</td>
<td>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.</td>
</tr>
<tr>
<td>VMASKMOVPS, VMASKMOVPD</td>
<td>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.</td>
</tr>
<tr>
<td>VPERMILPS, VPERMILPD</td>
<td>Shuffle 32-bit or 64-bit vector elements, with a register or memory operand as selector.</td>
</tr>
<tr>
<td>VPERM2F128</td>
<td>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.</td>
</tr>
<tr>
<td>VZEROALL</td>
<td>Set all YMM registers to zero and tag them as unused. Used when switching between 128-bit use and 256-bit use.</td>
</tr>
<tr>
<td>VZEROUPPER</td>
<td>Set the upper half of all YMM registers to zero. Used when switching between 128-bit use and 256-bit use.</td>
</tr>
</tbody>
</table>

### New instructions

Marco Ferretti, University of Pavia  
CINI - FNM ASS -93-
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)

<table>
<thead>
<tr>
<th>FMA</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
<td></td>
</tr>
<tr>
<td>Fused multiply add A = r1 * r2 + r3 for packed/scalar of double/single</td>
<td></td>
</tr>
<tr>
<td>Fused multiply alternating add/subtract of packed double/single A = r1 * r2 + r3 for odd index, A = r1 * r2-r3 for even</td>
<td></td>
</tr>
<tr>
<td>Fused multiply alternating subtract/add of packed double/single A = r1 * r2-r3 for odd index, A = r1 * r2+r3 for even</td>
<td></td>
</tr>
<tr>
<td>Fused multiply subtract A = r1 * r2-r3 of packed/scalar double/single</td>
<td></td>
</tr>
<tr>
<td>Fused negative multiply add of packed/scalar double/single A = -r1 * r2+r3</td>
<td></td>
</tr>
<tr>
<td>Fused negative multiply subtract of packed/scalar double/single A = -r1 * r2-r3</td>
<td></td>
</tr>
</tbody>
</table>

- **Future FMA: Fused Multiply Add**

Marco Ferretti, University of Pavia  
CINI - FNM ASS  
-98-
SSE AVX (INTEL)

- Fully automatic vectorization
- Auto vectorization hints (#pragma ivdep)
- User Mandated Vectorization (SIMD P pragma/D irective)
- SIMD intrinsic class (F32vec4 add)
- Vector intrinsic (mm_add_ps())
- ASM code (addps)

Ease of use
New in 12.0 !!

Programmer control

Marco Ferretti, University of Pavia  CINI - FNM ASS  -99-
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

\[
\text{for } (i=0; i<\text{MAX}; i++) \\
c[i] = a[i] + b[i];
\]
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];
}
AVX (INTEL)

## Intrinsic Function

```c
_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

<table>
<thead>
<tr>
<th>Marking</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[s/d]</td>
<td>Single- or double-precision floating point</td>
</tr>
<tr>
<td>[i/u]nnn</td>
<td>Signed or unsigned integer of bit size <code>nnn</code>, where <code>nnn</code> is 128, 64, 32, 16, or 8</td>
</tr>
<tr>
<td>[ps/pd/sd]</td>
<td>Packed single, packed double, or scalar double</td>
</tr>
<tr>
<td>ep132</td>
<td>Extended packed 32-bit signed integer</td>
</tr>
<tr>
<td>s1256</td>
<td>Scalar 256-bit integer</td>
</tr>
</tbody>
</table>

### Type

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>__m256</td>
<td>256-bit as eight single-precision floating-point values, representing a YMM register or memory location</td>
</tr>
<tr>
<td>__m256d</td>
<td>256-bit as four double-precision floating-point values, representing a YMM register or memory location</td>
</tr>
<tr>
<td>__m256i</td>
<td>256-bit as integers, (bytes, words, etc.)</td>
</tr>
<tr>
<td>__m128</td>
<td>128-bit single precision floating-point (32 bits each)</td>
</tr>
<tr>
<td>__m128d</td>
<td>128-bit double precision floating-point (64 bits each)</td>
</tr>
</tbody>
</table>
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