Lecture 13 (part 2)
Data Level Parallelism (1)

EEC 171 Parallel Architectures

John Owens
UC Davis

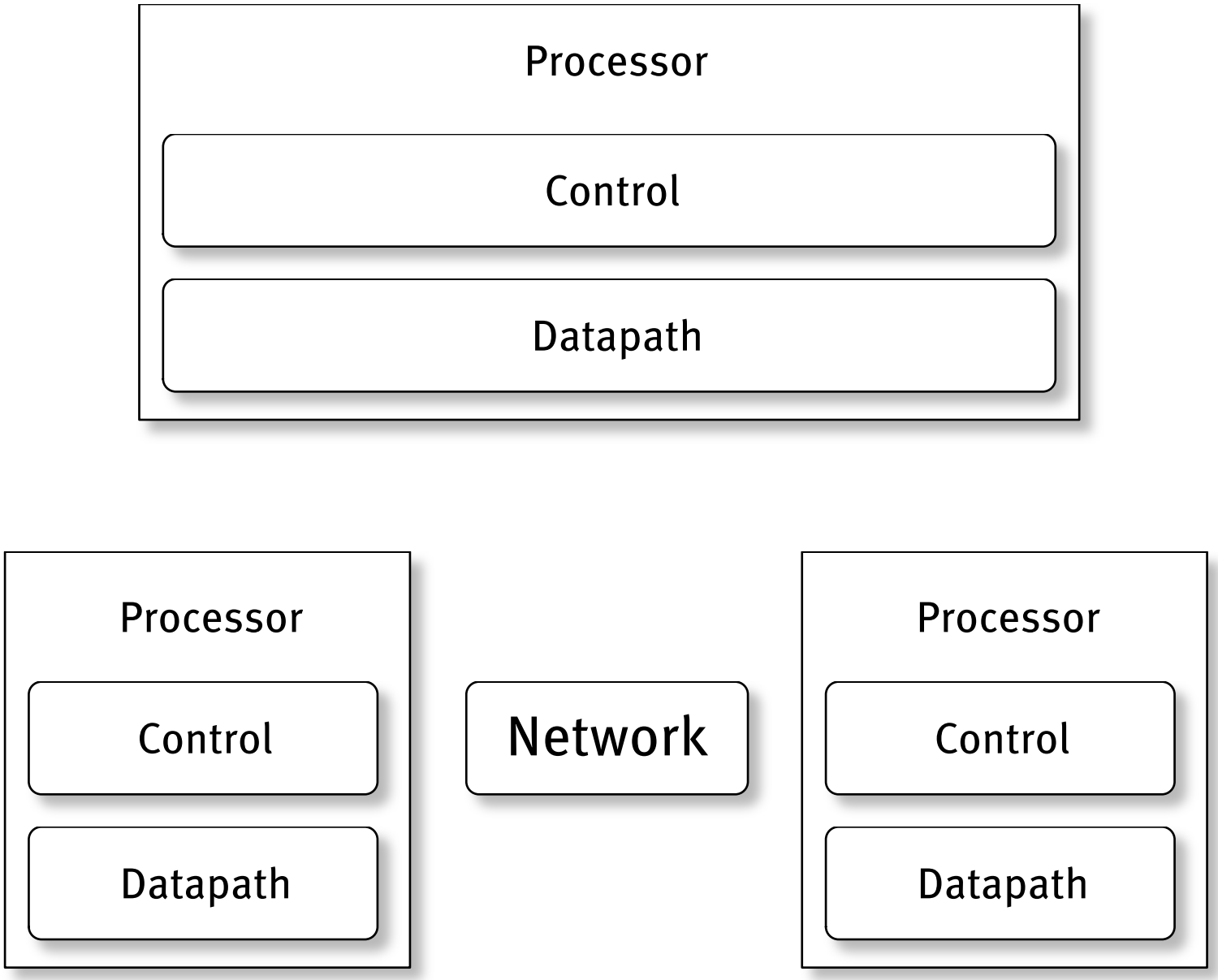
Credits

- © John Owens / UC Davis 2007–9.
- Thanks to many sources for slide material: Computer Organization and Design (Patterson & Hennessy) © 2005, Computer Architecture (Hennessy & Patterson) © 2007, Inside the Machine (Jon Stokes) © 2007, © Dan Connors / University of Colorado 2007, © Kathy Yelick / UCB 2007, © Wen-Mei Hwu/David Kirk, University of Illinois 2007, © David Patterson / UCB 2003–7, © John Lazzaro / UCB 2006, © Mary Jane Irwin / Penn State 2005, © John Kubiatowicz / UCB 2002, © Krste Asinovic/Arvind / MIT 2002, © Morgan Kaufmann Publishers 1998.

DLP Outline

- Today:
  - SIMD instructions
- Upcoming:
  - Vector machines
  - Vector efficiency
  - Massively parallel machines
  - GPUs
  - Stream processors
  - Algorithms and programming models

What We Know


Processor
Control
Datapath
Processor
Control
Datapath
Network
Processor
Control
Datapath

Cook analogy

- We want to prepare food for several banquets, each of which requires many dinners.
- We have two positions we can fill:
  - The boss (control), who gets all the ingredients and tells the chef what to do
  - The chef (datapath), who does all the cooking

Cook analogy

- ILP is analogous to:
  - One ultra-talented boss with many hands
  - One ultra-talented chef with many hands
- TLP is analogous to:
  - Building multiple kitchens, each with a boss and a chef
  - Putting more bosses and chefs in the same kitchen

Cook analogy

- DLP is analogous to:
  - One boss
  - Lots of cloned chefs

Flynn's Classification Scheme

- SISD – single instruction, single data stream
  - Uniprocessors
- SIMD – single instruction, multiple data streams
  - single control unit broadcasting operations to multiple datapaths
- MISD – multiple instruction, single data
  - no such machine (although some people put vector machines in this category)
- MIMD – multiple instructions, multiple data streams
  - aka multiprocessors (SMPs, MPPs, clusters, NOWs)

Performance beyond single thread ILP

- There can be much higher natural parallelism in some applications (e.g., database or scientific codes)
- Explicit Thread Level Parallelism or Data Level Parallelism
- Thread: process with own instructions and data
  - Thread may be a subpart of a parallel program (“thread”), or it may be an independent program (“process”)
  - Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute
- Data Level Parallelism: Perform identical operations on data, and (possibly) lots of data

Continuum of Granularity

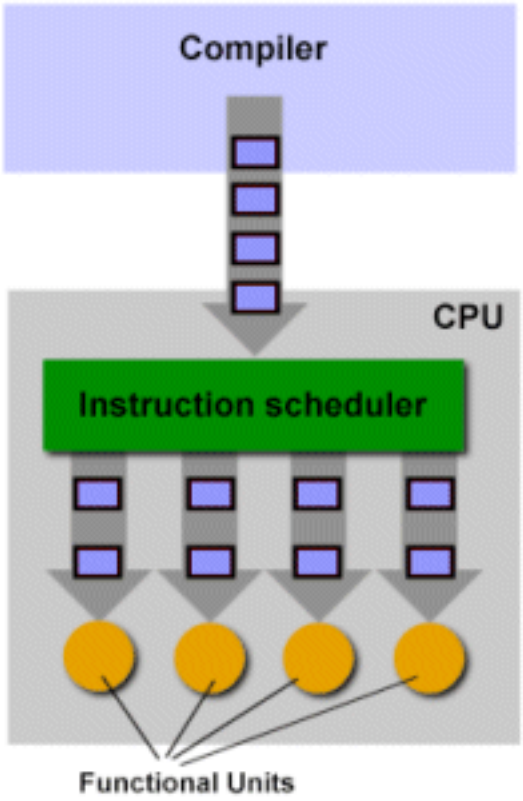
- "Coarse"
  - Each processor is more powerful
  - Usually fewer processors
  - Communication is more expensive between processors
  - Processors are more loosely coupled
  - Tend toward MIMD
- "Fine"
  - Each processor is less powerful
  - Usually more processors
  - Communication is cheaper between processors
  - Processors are more tightly coupled
  - Tend toward SIMD

ILP vs. TLP

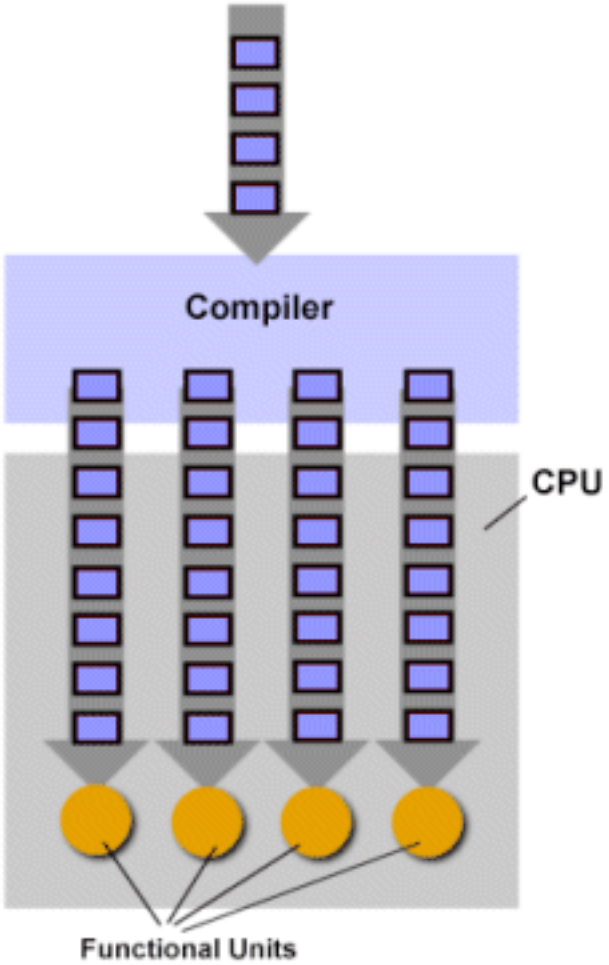
- “SIMD is about exploiting parallelism in the data stream, while superscalar SISD is about exploiting parallelism in the instruction stream.”

http://arstechnica.com/articles/paedia/cpu/simd.ars

What We Know

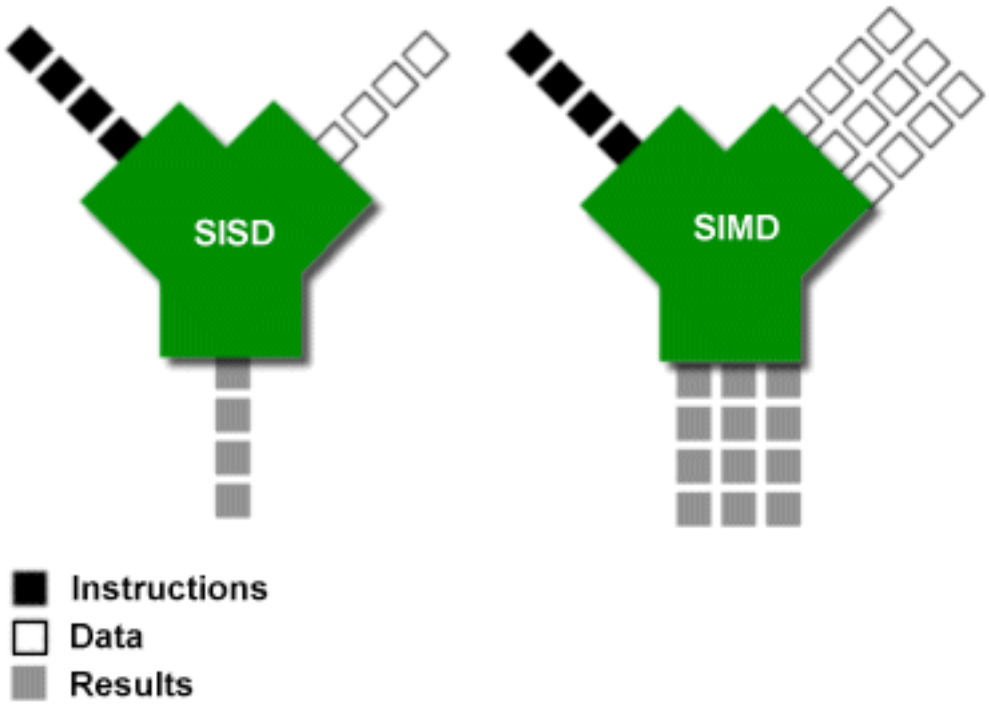


Dynamic Superscalar Instruction Scheduling

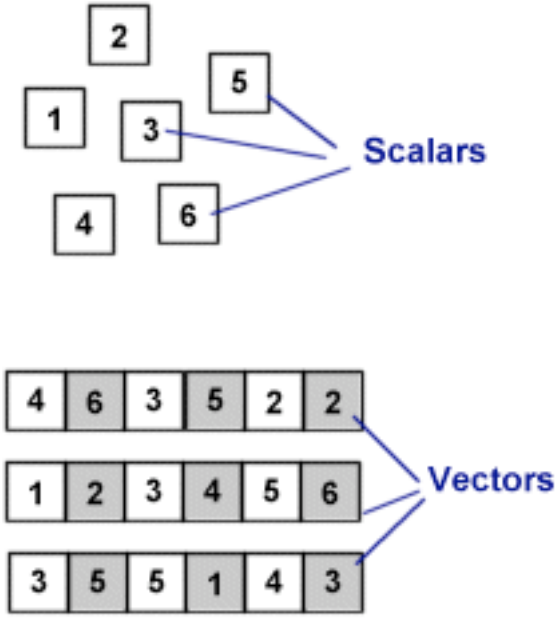


VLIW Instruction Scheduling

What's New



Scalar vs. Vector



- “The basic unit of SIMD love is the vector, which is why SIMD computing is also known as vector processing. A vector is nothing more than a row of individual numbers, or scalars.”

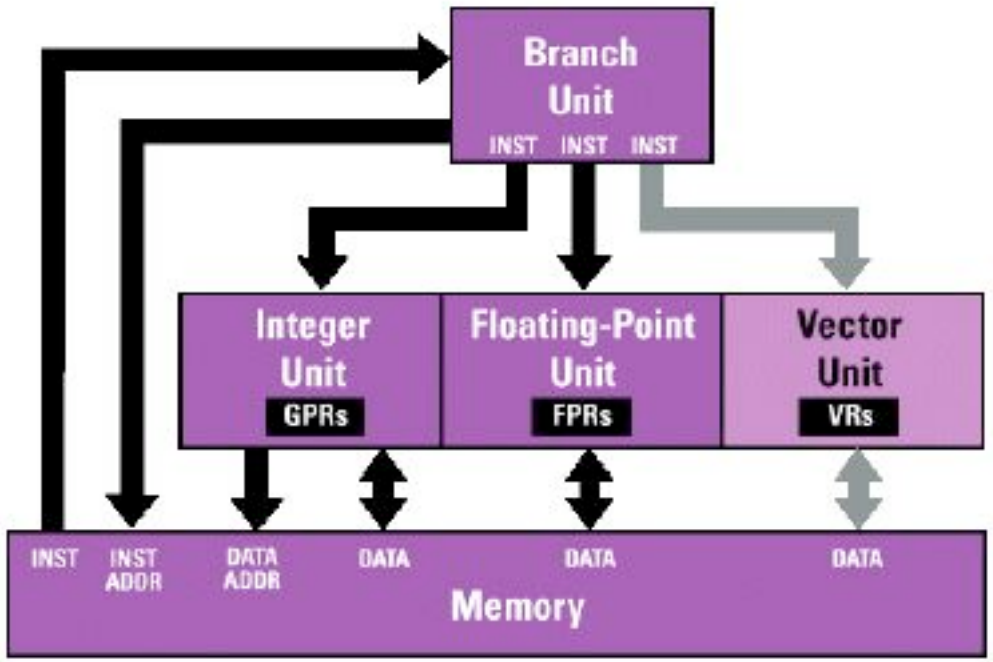
SIMD architectures by Jon "Hannibal" Stokes
http://arstechnica.com/articles/paedia/cpu/simd.ars

Representing Vectors

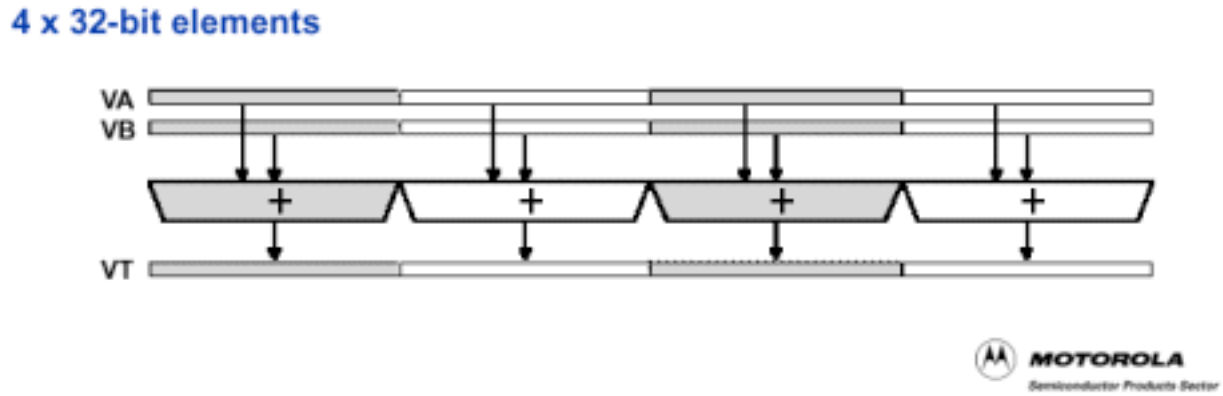
- Multiple items within same data word
- Multiple data words

Motorola AltiVec

High-level structural overview for PowerPC with AltiVec technology

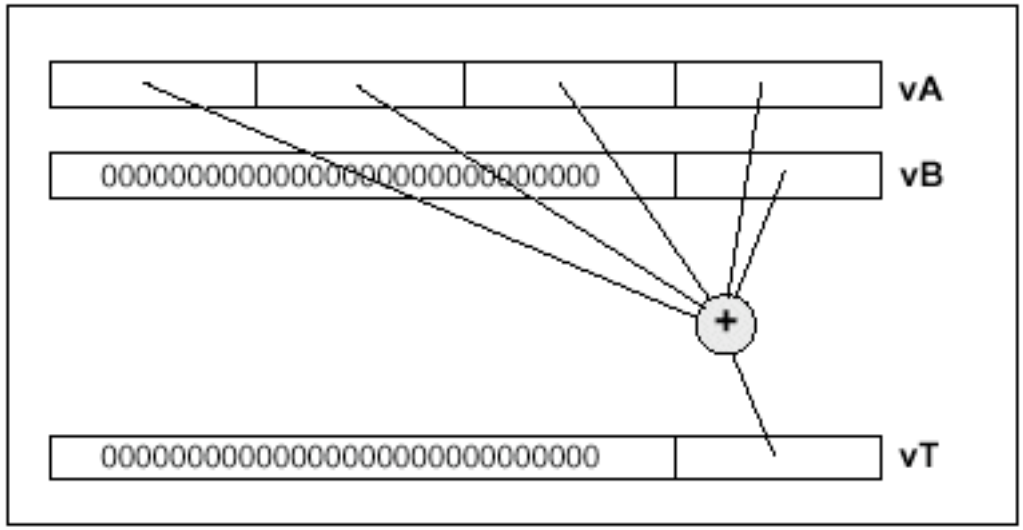


Motorola AltiVec



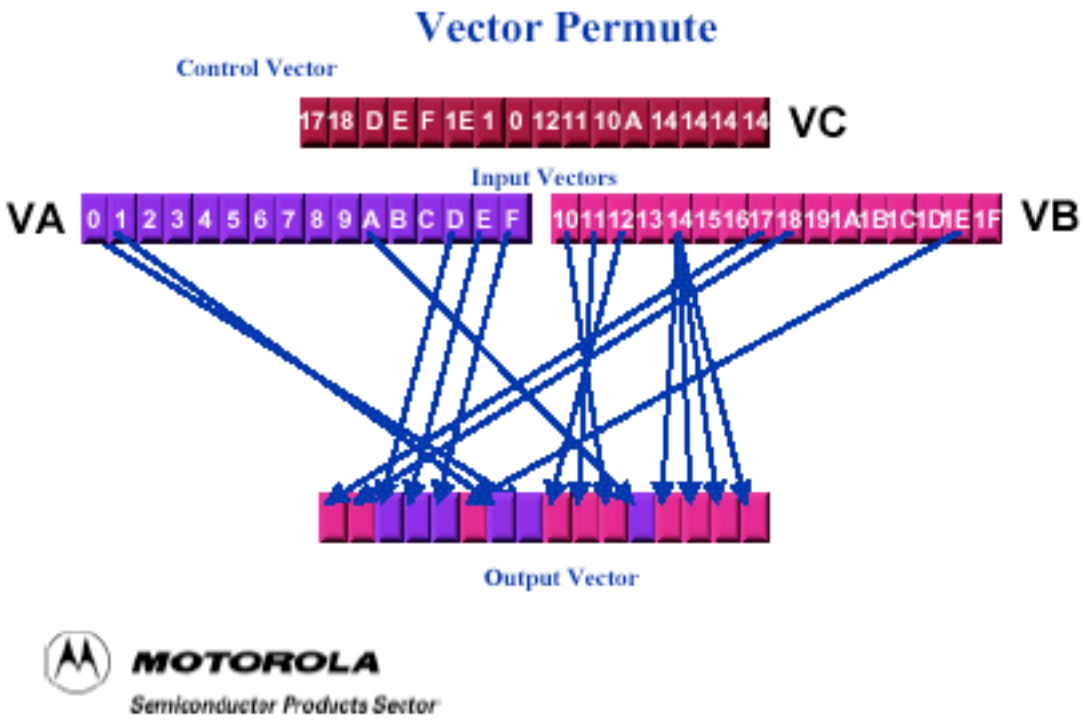
- Intra element arithmetic and non-arithmetic functions.
  - integer instructions
  - integer arithmetic instructions
  - integer compare instructions
  - integer rotate and shift instructions
  - floating-point instructions
  - floating-point arithmetic instructions
  - floating-point rounding and conversion instructions
  - floating-point compare instruction
  - floating-point estimate instructions
  - memory access instructions

Motorola AltiVec



- Inter Element Arithmetic: Between elements in vector
  - Example: Sum elements across vector, store in accumulator
  - alignment support instructions
  - permutation and formatting instructions
- pack instructions
- unpack instructions
- merge instructions
- splat instructions
- permute instructions
- shift left/right instructions

Motorola AltiVec

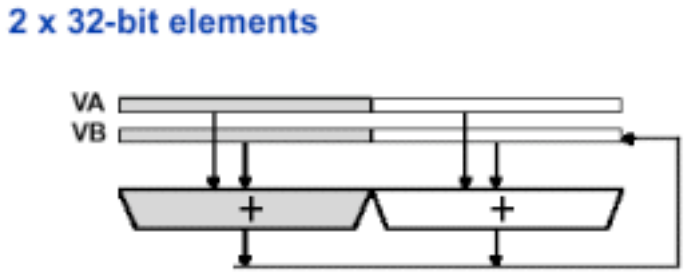


- Inter Element Non-Arithmetic (vector permute)

AltiVec Architecture

- 32 128b-wide new architectural registers
  - 16 8b elements, 8 16b elements, 4 32b (FP/int) elements
- 2 fully-pipelined, parallel AltiVec units:
  - Vector Permute Unit (pack, unpack, permute, load/store)
  - Vector ALU (all math)
  - Latency: 1 cycle for simple instrs, 3–4 for complex
  - No interrupts except load and store

MMX Instructions



- 57 new instructions
- 2 operands per instruction (like x86)
  - No single-cycle permutes

Intel MMX Datatypes

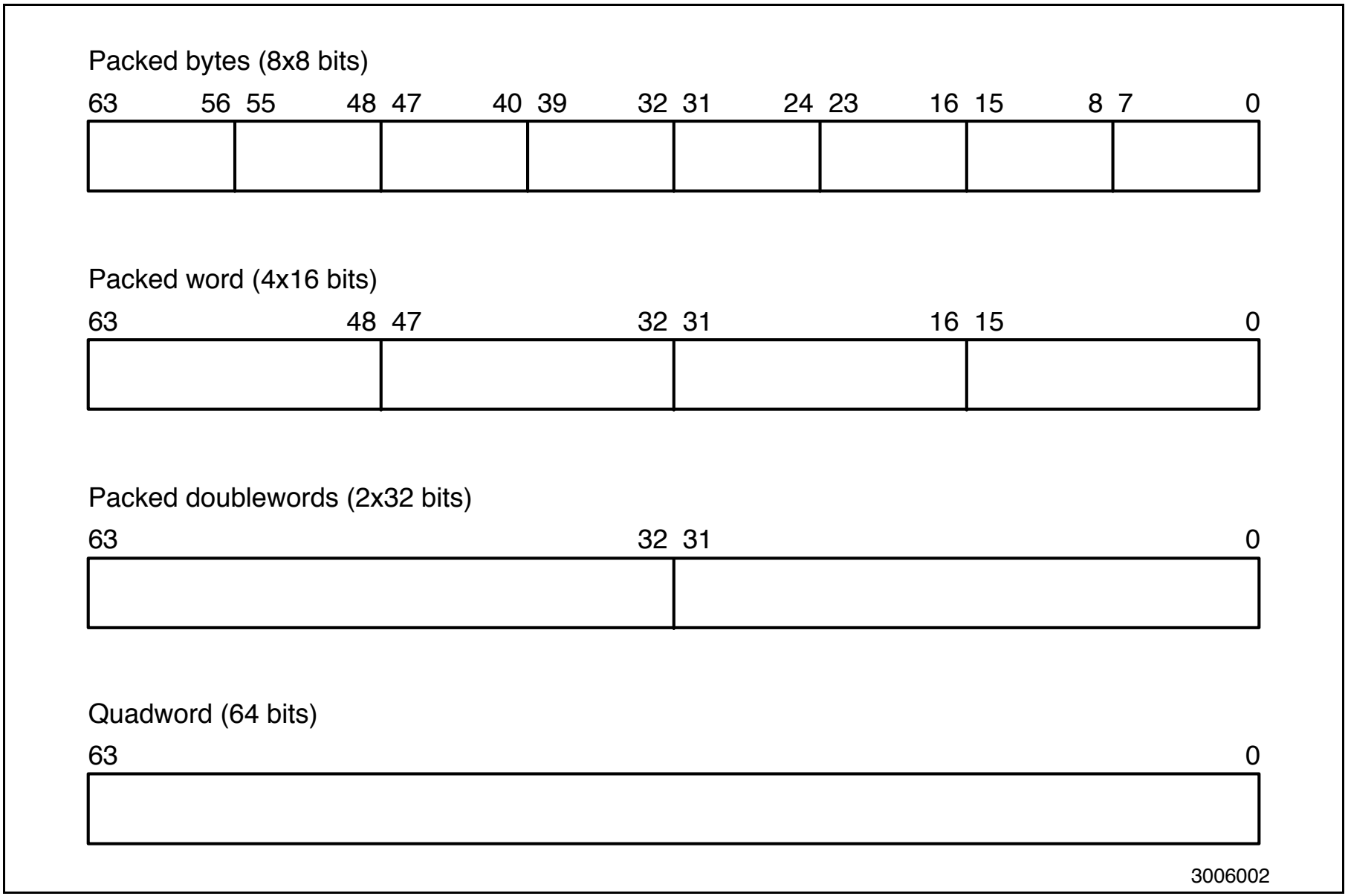


Figure 8-2. MMX™ Data Types

Intel MMX

Table 8-2. MMX™ Instruction Set Summary

Category		Wraparound	Signed Saturation	Unsigned Saturation
Arithmetic	Addition	PADDB, PADDW, PADDD	PADDSB, PADDSW	PADDUSB, PADDUSW
	Subtraction	PSUBB, PSUBW, PSUBD	PSUBSB, PSUBSW	PSUBUSB, PSUBUSW
	Multiplication	PMULL, PMULH		
	Multiply and Add	PMADD		
Comparison	Compare for Equal	PCMPEQB, PCMPEQW, PCMPEQD		
	Compare for Greater Than	PCMPGTPB, PCMPGTPW, PCMPGTPD		
Conversion	Pack		PACKSSWB, PACKSSDW	PACKUSWB
	Unpack High	PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ		
	Unpack Low	PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ		

		Packed	Full Quadword
Logical	And		PAND
	And Not		PANDN
	Or		POR
	Exclusive OR		PXOR
Shift	Shift Left Logical	PSLLW, PSLLD	PSLLQ
	Shift Right Logical	PSRLW, PSRLD	PSRLQ
	Shift Right Arithmetic	PSRAW, PSRAD	

		Doubleword Transfers	Quadword Transfers
Data Transfer	Register to Register	MOVD	MOVQ
	Load from Memory	MOVD	MOVQ
	Store to Memory	MOVD	MOVQ
Empty MMX™ State		EMMS	

MMX Details

- 8 MMX registers (64b each)
- Aliased with FP registers
  - “EMMS” (exit MMX) switches between them
  - Why is this a good idea?
  - Why is this a bad idea?
- Supports saturating arithmetic
- Programmed with intrinsics

MMX Example

To give you a better understanding of what can be done with MMX I've written a small function that blends two 32-bit ARGB pixels using 4 8-bit factors, one for each channel. To do this in C++ you would have to do the blending channel by channel. But with MMX we can blend all channels at once.

The blending factor is a one byte value between 0 and 255, as is the channel components. Each channel is blended using the following formula.

res = (a*fa + b*(255-fa))/255

```
; DWORD LerpARGB(DWORD a, DWORD b, DWORD f);
global _LerpARGB

_LerpARGB:

    ; load the pixels and expand to 4 words
    movd        mm1, [esp+4]    ; mm1 = 0 0 0 0 aA aR aG aB
    movd        mm2, [esp+8]    ; mm2 = 0 0 0 0 bA bR bG bB
    pxor        mm5, mm5        ; mm5 = 0 0 0 0 0 0 0 0
    punpcklbw   mm1, mm5        ; mm1 = 0 aA 0 aR 0 aG 0 aB
    punpcklbw   mm2, mm5        ; mm2 = 0 bA 0 bR 0 bG 0 bB

    ; load the factor and increase range to [0-256]
    movd        mm3, [esp+12]   ; mm3 = 0 0 0 0 faA faR faG faB
    punpcklbw   mm3, mm5        ; mm3 = 0 faA 0 faR 0 faG 0 faB
    movq        mm6, mm3        ; mm6 = faA faR faG faB [0 - 255]
    psrlw       mm6, 7          ; mm6 = faA faR faG faB [0 - 1]
    paddw       mm3, mm6        ; mm3 = faA faR faG faB [0 - 256]

    ; fb = 256 - fa
    pcmpeqw     mm4, mm4        ; mm4 = 0xFFFF 0xFFFF 0xFFFF 0xFFFF
    psrlw       mm4, 15         ; mm4 =   1   1   1   1
    psllw       mm4, 8          ; mm4 = 256 256 256 256
    psubw       mm4, mm3        ; mm4 = fbA fbR fbG fbB

    ; res = (a*fa + b*fb)/256
    pmullw      mm1, mm3        ; mm1 = aA aR aG aB
    pmullw      mm2, mm4        ; mm2 = bA bR bG bB
    paddw       mm1, mm2        ; mm1 = rA rR rG rB
    psrlw       mm1, 8          ; mm1 = 0 rA 0 rR 0 rG 0 rB

    ; pack into eax
    packuswb    mm1, mm1        ; mm1 = 0 0 0 0 rA rR rG rB
    movd        eax, mm1        ; eax = rA rR rG rB

    ret
```

http://www.programmersheaven.com/2/mmx

OpenEXR

Original OpenEXR Image

... with original Exposure setting of zero (0):

(click for larger image)

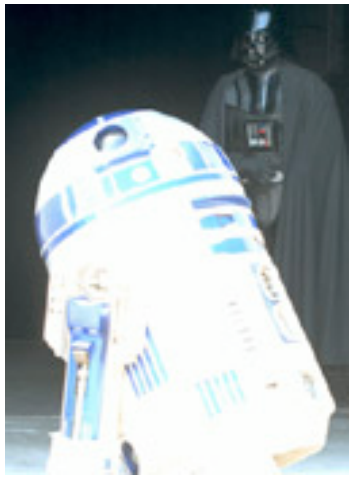
Adjust 3 Stops Brighter

... details emerge from the shadows.

(click for larger image)

Adjust 5 Stops Brighter

... and even more details emerge from the shadows.



(click for larger image)

http://www.openexr.com/

- 16-bit FP format (1 sign, 10 mantissa, 5 exponent)
- Natively supported (paired “half”) in NVIDIA GPUs

SSE (Streaming SIMD Extensions)

- Intel 4-wide floating point
- Pentium III: 2 fully-pipelined, SIMD, single-precision FP units
- 8 128-bit registers added to ISA
- In HW, (historically) broke 4-wide instructions into 2 2-wide instructions
  - Interrupts are a problem
- MMX + SSE: added 10% to PIII die

Multimedia Extensions

- Very short vectors added to existing ISAs for micros
- Usually 64-bit registers split into 2x32b or 4x16b or 8x8b
- Newer designs have 128-bit registers (Altivec, SSE2)
- Limited instruction set:
  - no vector length control
  - no strided load/store or scatter/gather
  - unit-stride loads must be aligned to 64/128-bit boundary
- Limited vector register length:
  - requires superscalar dispatch to keep multiply/add/load units busy
  - loop unrolling to hide latencies increases register pressure
- Trend towards fuller vector support in microprocessors