Computação Paralela

Mest. Engenharia Computacional
Mest. Int. Engenharia Computacional

Ano letivo 2022/2023

Rui Costa, Nuno Lau

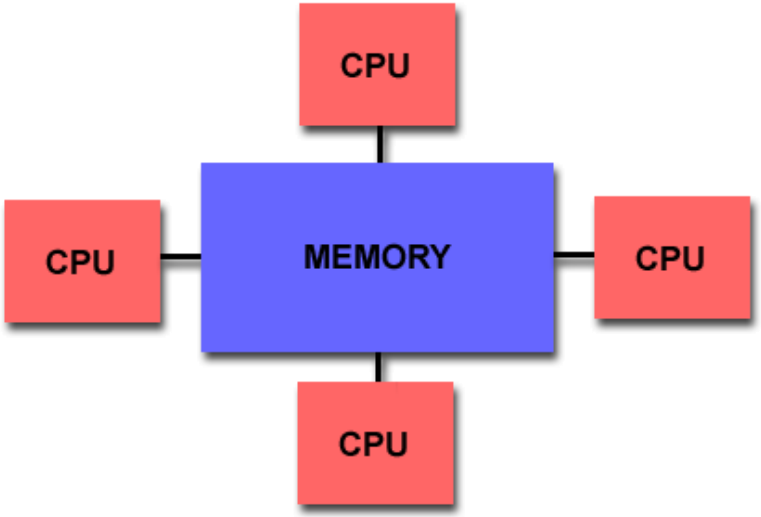
Parallel Computer Memory Architectures

Memory architectures

- Shared Memory
- Distributed Memory
- Hybrid Distributed-Shared Memory

Shared Memory

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.



- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: ***UMA*** and ***NUMA***.

Shared Memory : UMA vs. NUMA

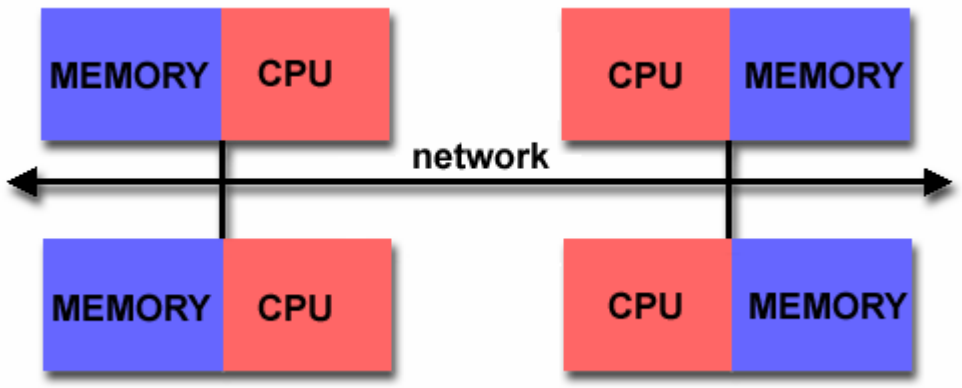
- **Uniform Memory Access (UMA):**
  - Most commonly represented today by Symmetric Multiprocessor (SMP) machines
  - Identical processors
  - **Equal access and access times to memory**
  - Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.
- **Non-Uniform Memory Access (NUMA):**
  - Often made by physically linking two or more SMPs
  - One SMP can directly access memory of another SMP
  - **Not all processors have equal access time to all memories**
  - Memory access across link is slower
  - If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

Shared Memory: Pro and Con

- **Advantages**
  - Global address space provides a user-friendly programming perspective to memory
  - Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs
- **Disadvantages:**
  - Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
  - Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
  - Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Distributed Memory

- Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a **communication network** to connect inter-processor memory.
- **Processors have their own local memory**. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of **cache coherency does not apply**.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can can be as simple as Ethernet.



Distributed Memory: Pro and Con

- **Advantages**
  - Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
  - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
  - Cost effectiveness: can use commodity, off-the-shelf processors and networking.
- **Disadvantages**
  - The programmer is responsible for many of the details associated with data communication between processors.
  - It may be difficult to map existing data structures, based on global memory, to this memory organization.
  - Non-uniform memory access (NUMA) times

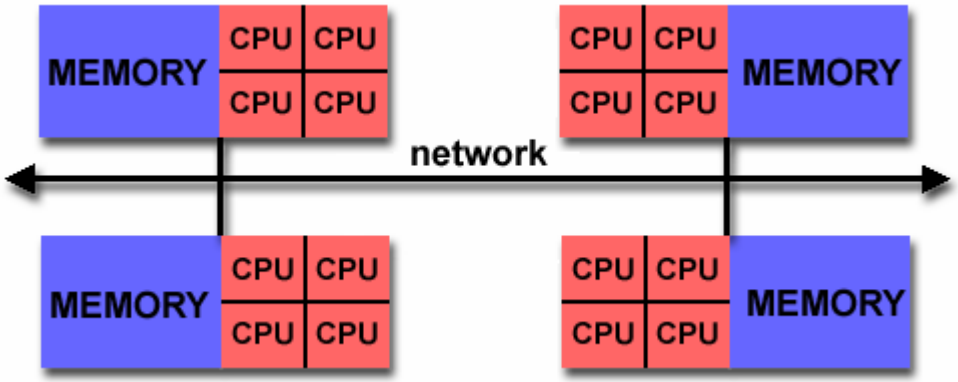
Hybrid Distributed-Shared Memory

Summarizing a few of the key characteristics of shared and distributed memory machines

Comparison of Shared and Distributed Memory Architectures			
Architecture	CC-UMA	CC-NUMA	Distributed
Examples	SMPs Sun Vexx DEC/Compaq SGI Challenge IBM POWER3	Bull NovaScale SGI Origin Sequent HP Exemplar DEC/Compaq IBM POWER4 (MCM)	Cray T3E Maspar IBM SP2 IBM BlueGene
Communications	MPI Threads OpenMP shmem	MPI Threads OpenMP shmem	MPI
Scalability	to 10s of processors	to 100s of processors	to 1000s of processors
Draw Backs	Memory-CPU bandwidth	Memory-CPU bandwidth Non-uniform access times	System administration Programming is hard to develop and maintain
Software Availability	many 1000s ISVs	many 1000s ISVs	100s ISVs

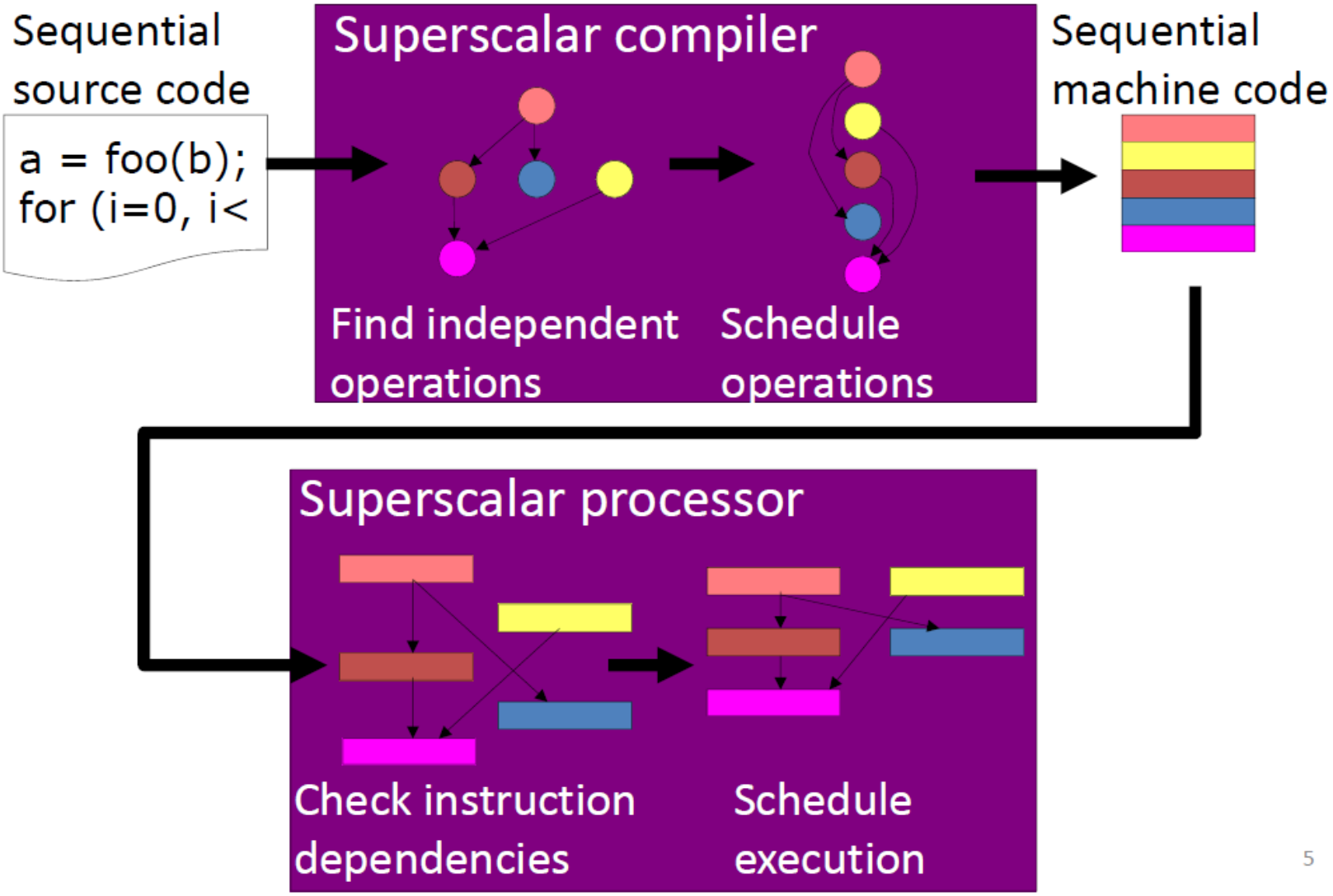
Hybrid Distributed-Shared Memory

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.



- The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global.
- The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.
- Advantages and Disadvantages: whatever is common to both shared and distributed memory architectures.

Superscalar Compiler and Processor



Multimedia extensions

- Extensions to the base ISA that allow a form of vector computation
- “Vectors” are implemented in dedicated registers
  - MMX: 64 bits
  - SSE: 128 bits
  - AVX: 256 bits
  - AVX-512: 512 bits
- Registers of N bits may be used as vectors of 2x(N/2) elements, 4x(N/4) elements, etc.
- One multimedia instruction applies simultaneously to all elements of a register

Multimedia extensions

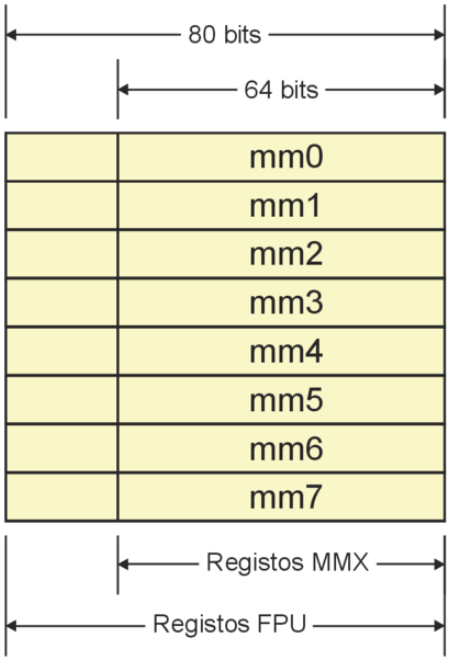
- MMX
  - 57 instructions added to Pentium
- Extended MMX
- SSE
  - Pentium III
  - 71 instructions (52 FP SIMD, 19 MMX)
- SSE2
  - 144 new instructions (Pentium 4)
  - 128 bit registers
  - Cache-control
- SSE3
  - 13 new instructions (Pentium 4 - 2004)
  - Horizontal processing of values in a register
- SSSE3
  - 32 new instructions (Core)
- SSE4
  - 54 instructions (Penryn - 2008)

Multimedia extensions

- 3D Now!
  - AMD
  - 45 instructions (21 FP SIMD, 19 MMX, 5 DSP)
- AltiVec
  - Motorola
  - 162 instructions
- AVX
  - Sandy Bridge – 2011
  - 256 bit registers
  - Instructions with 3 operands
- AVX2
  - Haswell New Instructions, 2013
  - Broadcast/permute operations on data elements
  - Vector shift instructions with variable-shift count per data element
  - Instructions to fetch non-contiguous data elements from memory
- AVX-512
  - Proposed in 2013, First Processor 2016
  - 512 bit registers
  - Several extensions: Foundation, Prefetch, Vector Neural Network, etc.

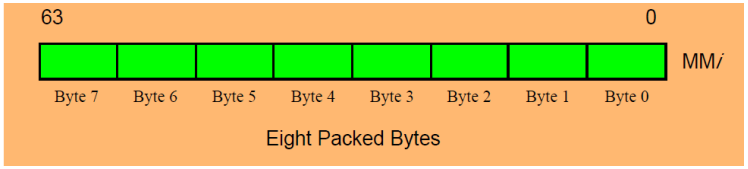
MMX registers

- 8 registers of 64 bits
  - MM0, MM1, …,MM7
  - Are implemented on the same hardware as the FP registers: ST0, …, ST7
    - This allowed to mantain compatibility with existing operating systems

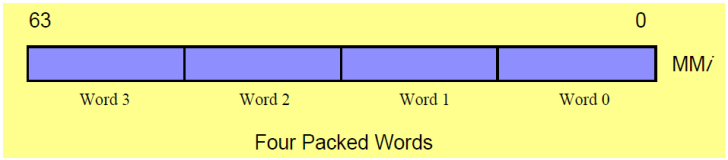


MMX data types

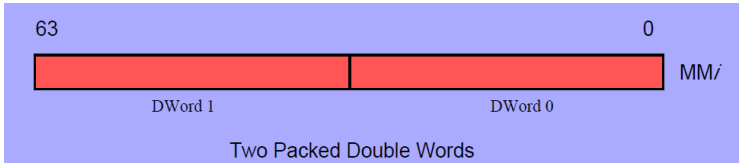
- 8 bytes array



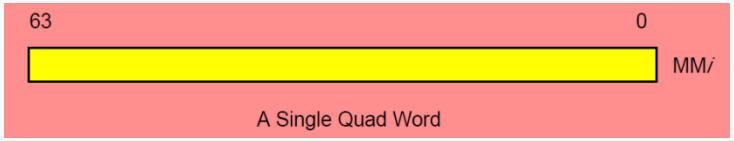
- 4 words array (16 bit/word)



- 2 double words array (32 bits)



- Quadword (64 bits)

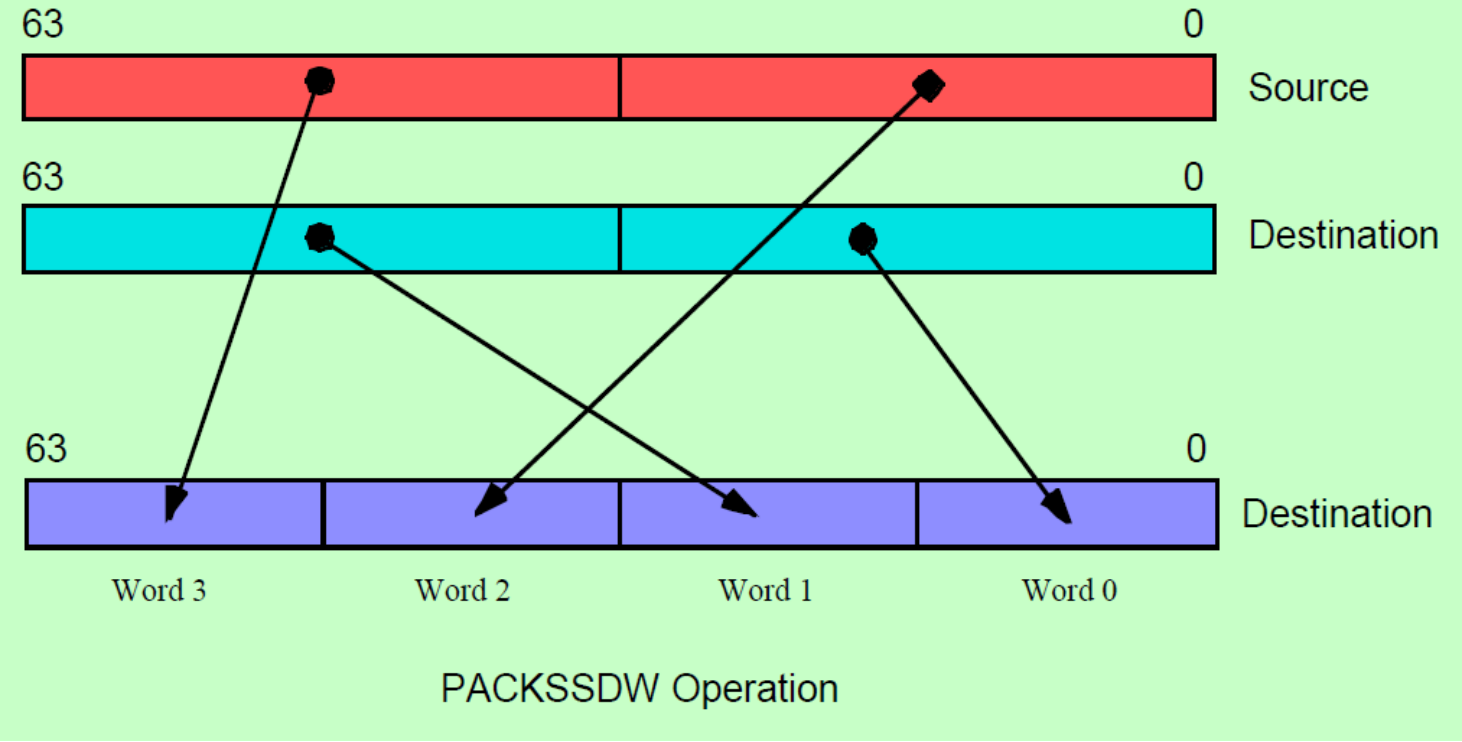


MMX instructions

- Data transfer
  - Move data between registers, memory and MMX registers
  - `movd` (32 bits) and `movq` (64 bits)
- Conversion
  - Converts larger data types to smaller data types and vice-versa
- Packed arithmetic
- Comparisons
- Logic operations
- Shift and Rotate
- EMMS
  - Prepares the processor to execute FP code again

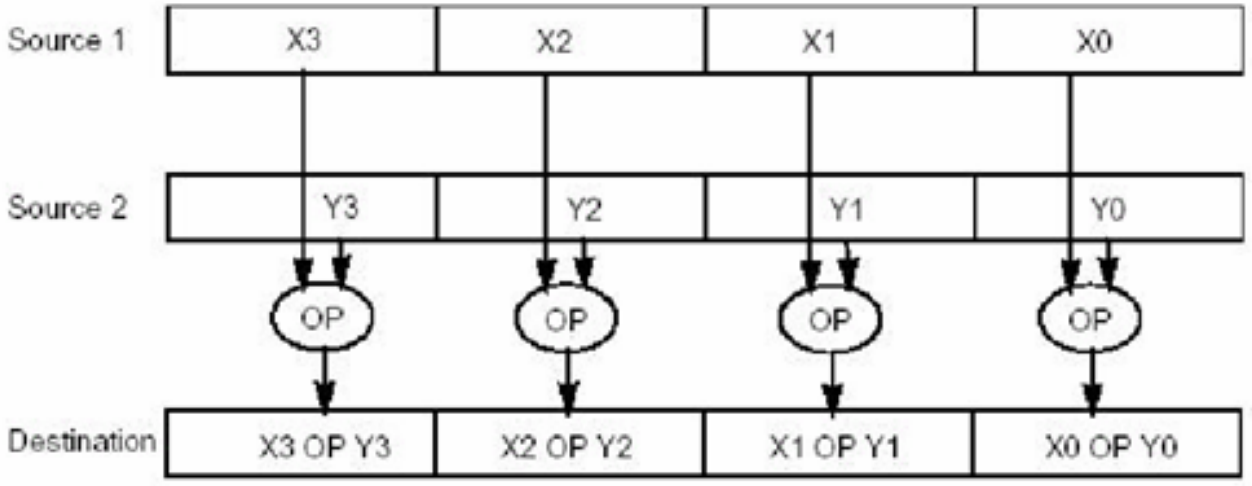
MMX instructions

- Convertion



MMX instructions

- Packed arithmetic
  - SIMD: *Single Instruction Multiple Data*
  - `paddb` (8bit), `paddw` (16bit), `paddd` (32bit)
  - `paddb`, `paddsb` *(signed saturation)*, `paddusb` *(unsigned saturation)*

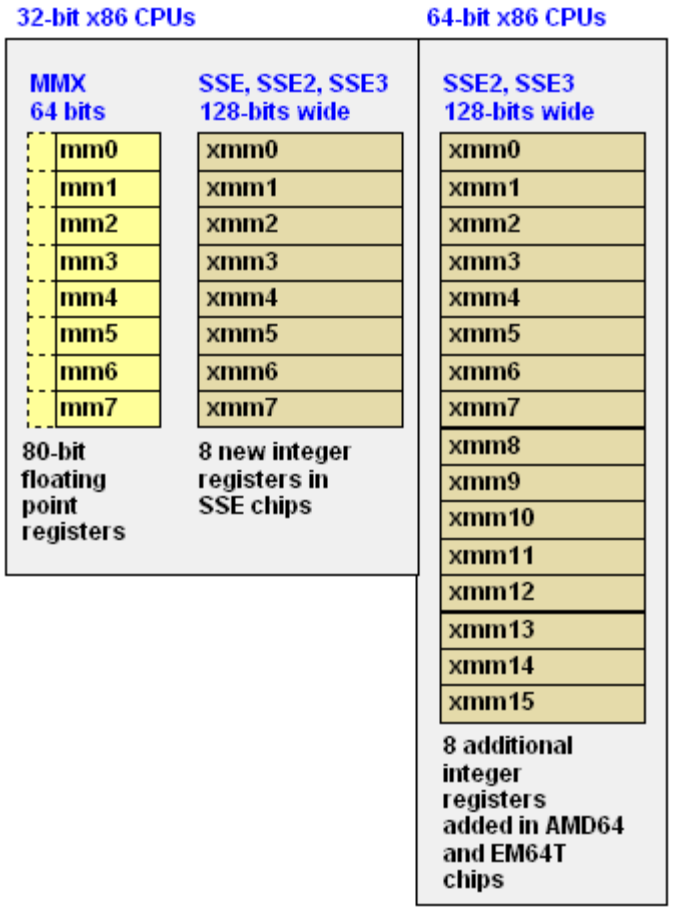


MMX instructions

Packed Arithmetic	**Wrap Around**	**Signed Sat**	**Unsigned Sat**
Addition	PADD	PADDS	PADDUS
Subtraction	PSUB	PSUBS	PSUBUS
Multiplication	PMULL/H		
Multiply & add	PMADD		
Shift right Arithmetic	PSRA		
Compare	PCMPcc		
Conversions	**Regular**	**Signed Sat**	**Unsigned Sat**
Pack		PACKSS	PACKUS
Unpack	PUNPCKL/H		
Logical Operations	**Packed**	**Full 64-bit**	
And		PAND	
And not		PANDN	
Or		POR	
Exclusive or		PXOR	
Shift left	PSLL	PSLL	
Shift right	PSRL	PSRL	
Transfers and Memory Operations	**32-bit**	**64-bit**	
Register-register move	MOVD	MOVQ	
Load from memory	MOVD	MOVQ	
Store to memory	MOVD	MOVQ	
Miscellaneous			
Empty multimedia state	EMMS		

SSE/SSE2 data types

- 8 registers of 128 bits
  - XMM0, …, XMM7
- 4 FPs simple precision (SSE)
- 2 FPs double precision (SSE2)
- 16 bytes (SSE2)
- 8 words (SSE2)
- 4 double words (SSE2)
- 1 inteiro 128bit (SSE2)



Sum arrays

universidade de aveiro
ua

Code:

```
int A[size], B[size], C[size];
…
for (i = 0 ; i < size ; i++)
    C[i] = A[i] + B[i];
```

Parallelism?

Sum arrays

```
movdqa (%eax,%edx,4), %xmm0    # load A[i] to A[i+3]
movdqa (%ebx,%edx,4), %xmm1    # load B[i] to B[i+3]
paddd  %xmm0, %xmm1            # CCCC = AAAA + BBBB
movdqa %xmm1, (%ecx,%edx,4)    # store C[i] to C[i+3]
addl   $4, %edx                # i += 4
```

- movdqa (%eax,%edx,4), %xmm0
  - **mov**: transferência
  - **dq**: double quad
  - **a**: align
  - **(%eax,%edx,4)**: eax + 4*edx

Sum elements of an array

universidade de aveiro
ua

Code:

```
int A[size], total=0;
…
for (i = 0 ; i < size ; i++)
     total += A[i];
```

Parallelism?

Sum elements of an array

Restructured code:

```
int A[size], temp[4], total;
temp[0]=temp[1]=temp[2]=temp[3]=0;
for (i = 0 ; i < size ; i+=4) {
      temp[0] += A[i];   temp[1] += A[i+1];
      temp[2] += A[i+2]; temp[3] += A[i+3];
}
total = temp[0]+temp[1]+temp[2]+temp[3];
```

Parallelism?

Internal product of 2 vectors

universidade de aveiro
ua

Code:

```
int A[size], B[size], iprod;
…
for (i = 0 ; i < size ; i++)
     iprod += A[i]*b[i];
```

Parallelism?

Internal product of 2 vectors

Loop unrolled code:

```
int A[size], B[size], iprod=0;
for (i = 0 ; i < size ; i+=3) {
    iprod += A[i]*B[i];
    iprod += A[i+1]*B[i+1];
    iprod += A[i+2]*B[i+2];
}
```

Parallelism?

Internal product of 2 vectors

universidade de aveiro
ua

Loop unrolled code:

```
int A[size], B[size], iprod=0;
for (i = 0 ; i < size ; i+=3) {
    temp0 = A[i]*B[i];
    temp1 = A[i+1]*B[i+1];
    temp2 = A[i+2]*B[i+2];
    iprod += temp0 + temp1 + temp2
}
```

Parallelism?

SSE with unrolled loop

- Internal product of 2 vectors (**a** and **b**)

```
length6 = (size/24)*24
for(; i < length6; i += 24){
   __asm__ volatile
   (// instruction comment
    "\n\t movdqa 0x00(%0),%%xmm2 \t#" "\n\t movdqa 0x10(%0),%%xmm3 \t#"
    "\n\t movdqa 0x20(%0),%%xmm4 \t#" "\n\t movdqa 0x30(%0),%%xmm5 \t#"
    "\n\t movdqa 0x40(%0),%%xmm6 \t#" "\n\t movdqa 0x50(%0),%%xmm7 \t#"

    "\n\t mulps 0x00(%1),%%xmm2 \t#"  "\n\t mulps 0x10(%1),%%xmm3 \t#"
    "\n\t mulps 0x20(%1),%%xmm4 \t#"  "\n\t mulps 0x30(%1),%%xmm5 \t#"
    "\n\t mulps 0x40(%1),%%xmm6 \t#"  "\n\t mulps 0x50(%1),%%xmm7 \t#"

    "\n\t addps %%xmm2,%%xmm0 \t#"    "\n\t addps %%xmm3,%%xmm0 \t#"
    "\n\t addps %%xmm4,%%xmm0 \t#"    "\n\t addps %%xmm5,%%xmm0 \t#"
    "\n\t addps %%xmm6,%%xmm0 \t#"    "\n\t addps %%xmm7,%%xmm0 \t#"
   :
   :"r" (a+i), // %0
    "r" (b+i)  // %1
   );
}
```

SSE with loop unrolling

- Comparison with other libraries