High Performance Computing

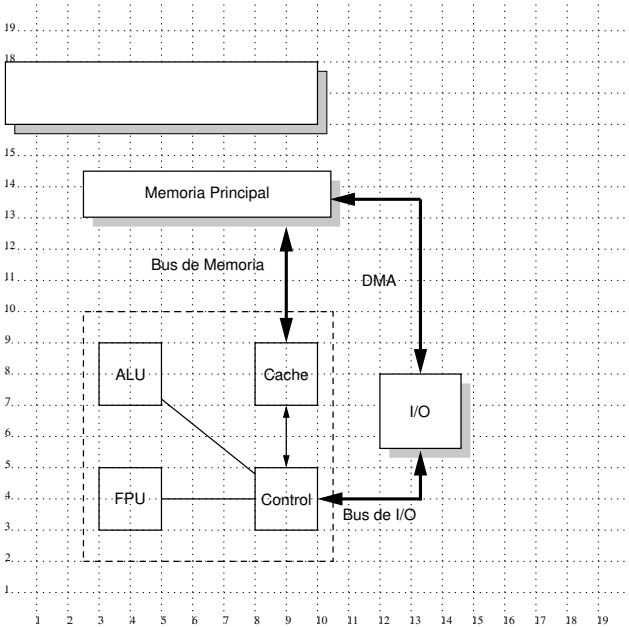
Modern Processor and SIMT

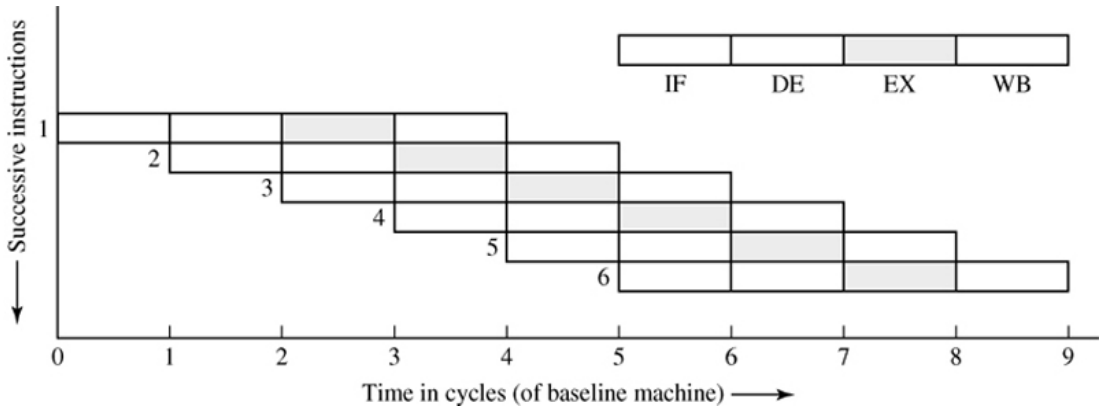
Fernando R. Rannou
Departamento de Ingeniería Informática
Universidad de Santiago de Chile

August 30, 2023

Procesador escalar

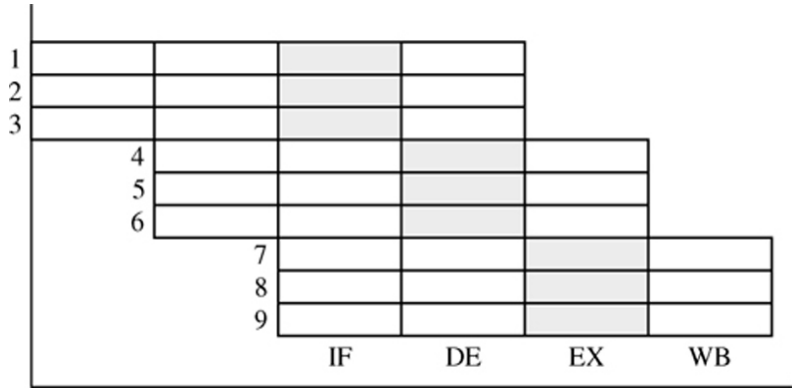
- Un *procesador escalar base* consiste de un procesador que "emite" una instrucción por ciclo de reloj
- También existe una o más unidades de I/O
- Los elementos se comunican a través de buses del sistema
- En general, los procesadores escalares están basados en pipelining
- El 99.2% de los computadores top 500 (2007) tienen procesadores escalares





Procesador super escalar

- Un *procesador super escalar* emite múltiples instrucciones por ciclo de reloj
- Por ejemplo, si el procesador tiene m pipelines, puede ejecutar m instrucciones en paralelo (si éstas son independientes)
- El número real de instrucciones ejecutadas por ciclo depende de la independencia entre las instrucciones.
- Uso de optimización en fase de compilación




```
for i=0 to N {
    a[i] = sin(30*i/PI)
}
```

```
a[0] = 0.5
for i=1 to N {
  a[i] = a[i-1]*sin(30*i/PI)
}
```

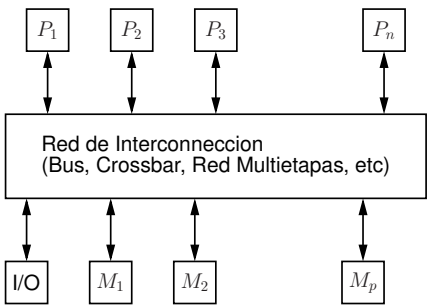
Procesador vectorial

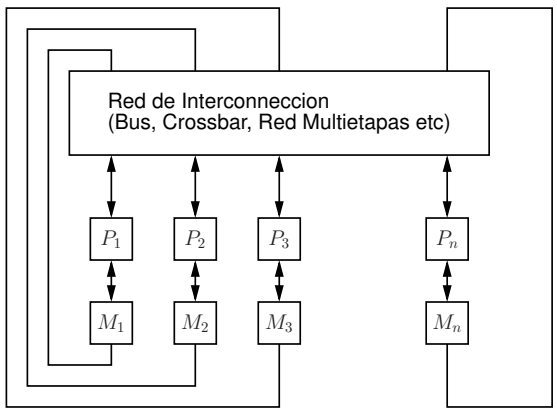
- En un procesador vectorial paralelo (PVP) una instrucción opera simultáneamente sobre varios elementos de vectores o arreglos de datos
- Generalmente son muy caros y limitados en aplicaciones
- No hay mucho desarrollo en la actualidad (really...???)
- Apoyo del compilador para vectorizar loops (Fortran 90)
- Ejemplos: Cray SV1, Cray SV2, NEC Earth simulator
- El 0.8 % de los computadores top 500 (2007) tienen procesadores vectoriales

```
z(1:100) = x(1:100) + y(1:100)
```

Multiprocesadores de memoria compartida

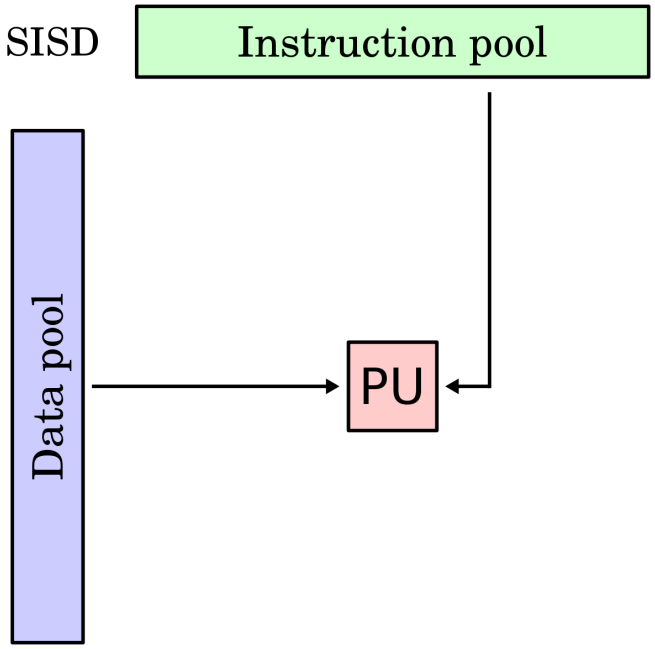
- Un multiprocesador de memoria compartidad contiene dos o más procesadores que comparten el espacio de memoria. Dos modelos:
  1. UMA (Uniform memory-access): la memoria física es compartida por todos los procesadores; por lo tanto el tiempo de acceso es el mismo para todos.
  2. NUMA (Non-uniform memory-access): la memoria física esta distribuida (local a cada procesador) y por lo tanto el tiempo de acceso a ella no es uniforme.
- La colección de todas las memorias locales forman el espacio global de direcciones
- Programación multihebra usa eficientemente esta aquitectura



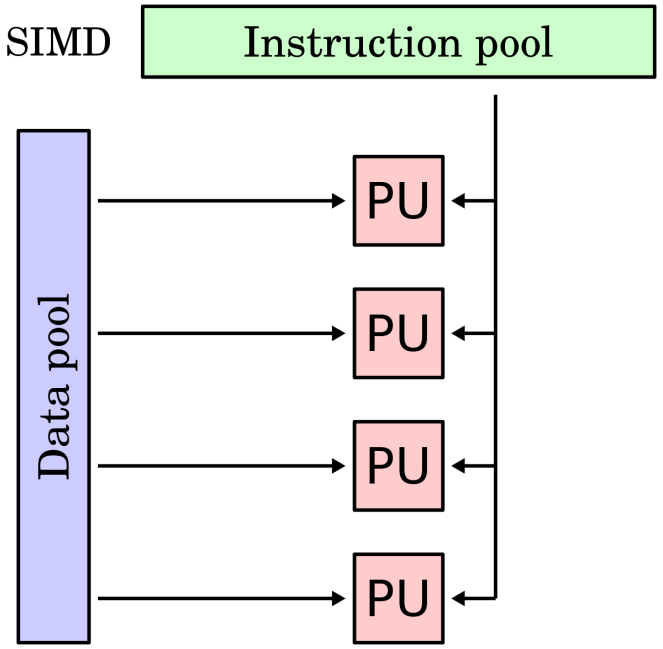


Taxonomía de Flynn

SISD
Single-Instruction, Single-Data

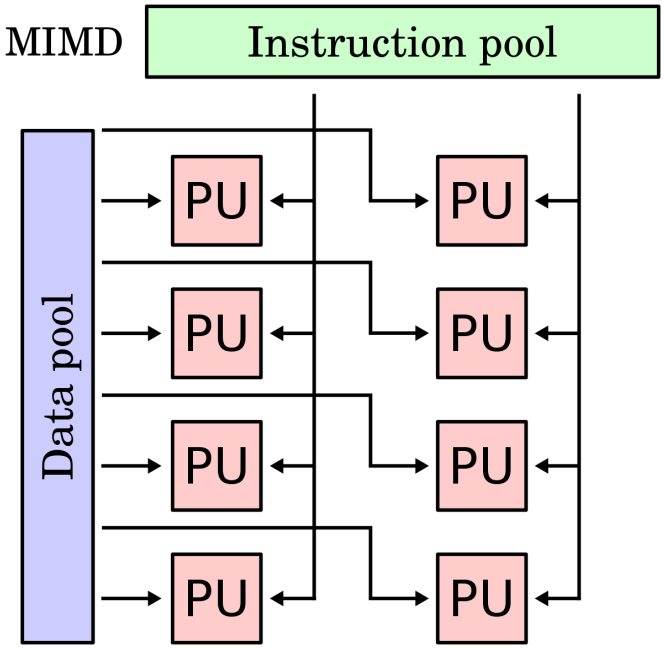


SIMD
Single-Instruction, Multiple-Data

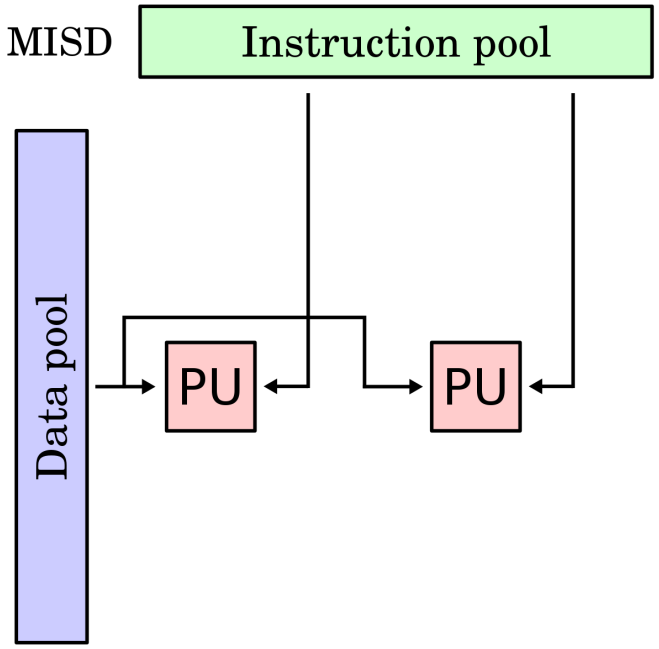


Taxonomía de Flynn

MIMD
Multiple-Instruction, Multiple-Data



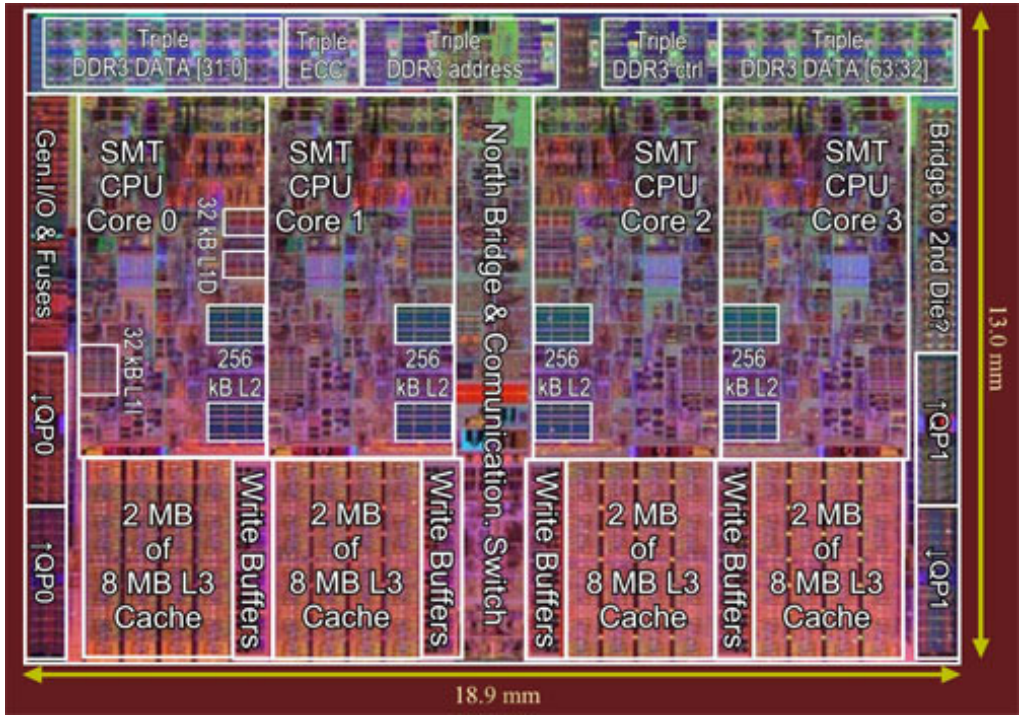
SIMD
Multiple-Instruction, Single-Data



Nehalem de Intel (2009)

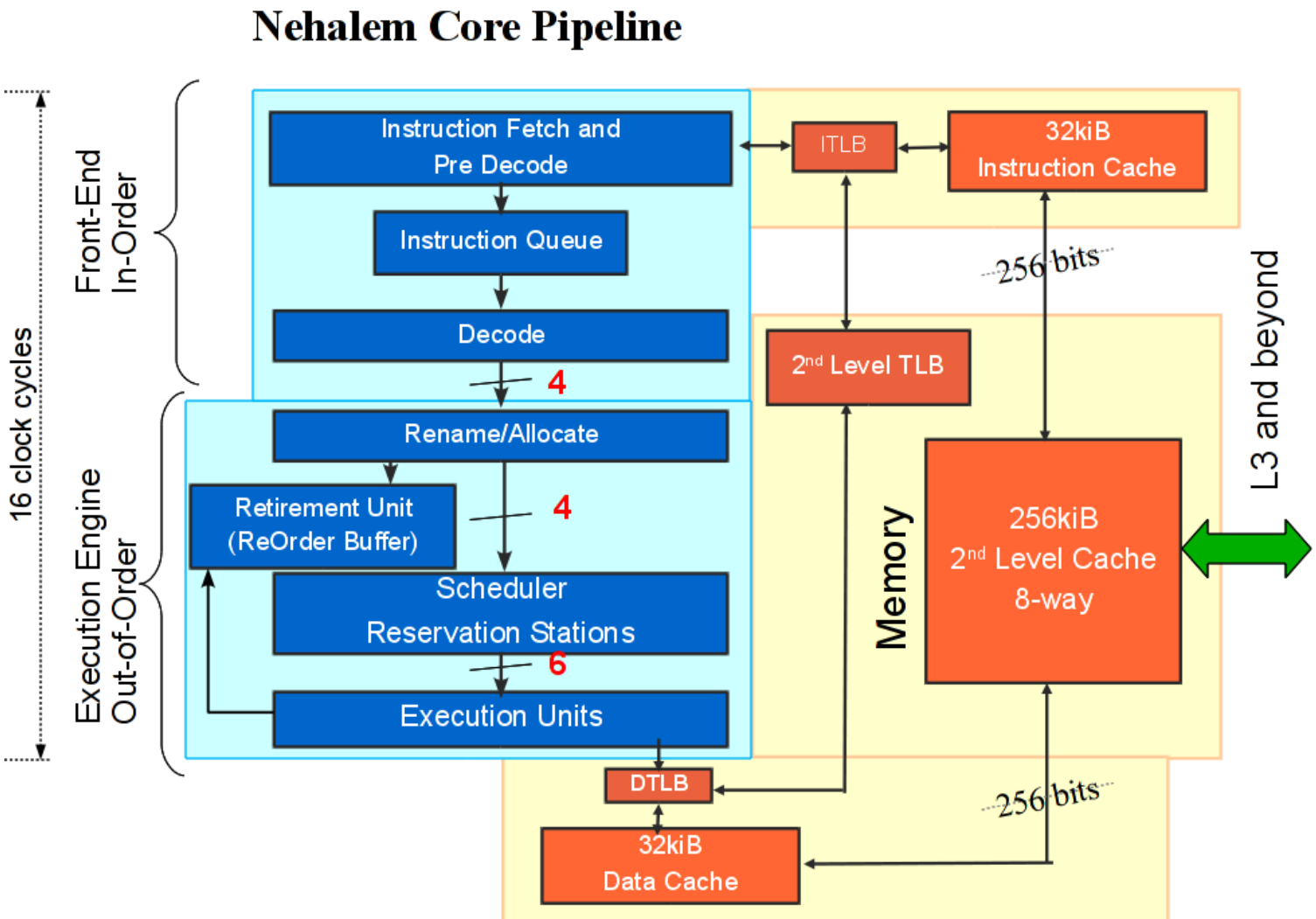
- **Intel64** derivada de IA-32
- Introduce mejoras sobre architecturas Core anteriores
- Base de los procesadores Core i7
- Architectura orientada a altas tasas de computación para aplicaciones científicas
- Super escalar, multi-core (2, 4, 8), SIMD
- Cada Core soporta 2-way multi-threading (hyperthreading)
- Lo suceden Westmere y Sandy Bridge

Especificaciones básicas



- 2 niveles de TLB
- Split Cache L1 de 64KB/core
- L2 unificada de 256 KB/core
- L3 compartida de 4-12 MB
- Pipeline de largo 14
- Pipeline de ancho 4
- Prefetching
- SIMD mejorada

Pipeline de Nehalem

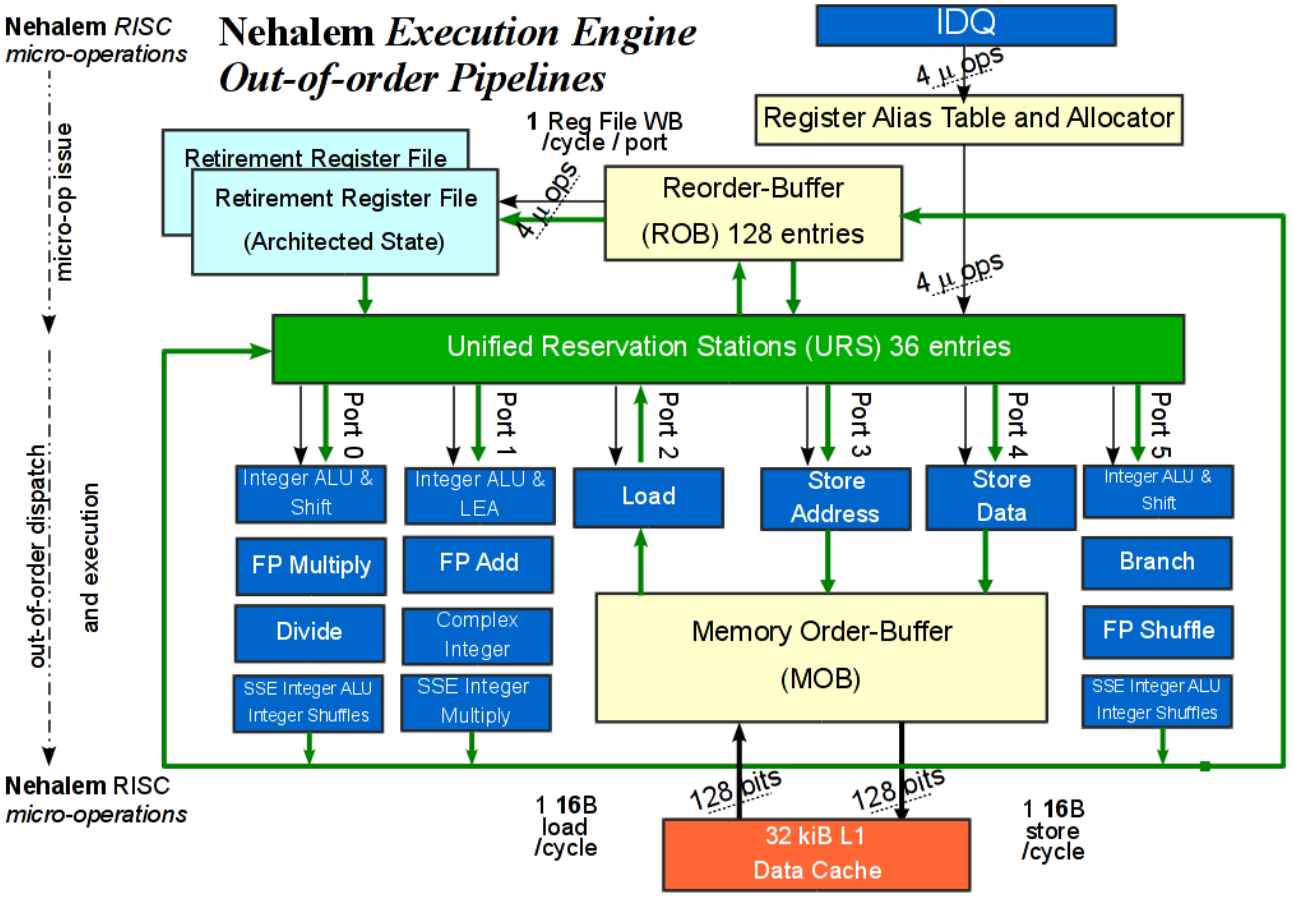


1

[1] M. E. Thomadakis, Super Computing Facility, Texas A&M

Unidades de Ejecución (por Core)

- Cada puerto representa un cluster de ejecución
- Cada cluster se compone de varias Execution Units (microOps)



- 3 uops de memoria
- 3 uops aritmética/lógicas
- 6 uops/cycle en el mejor de los casos!

Cluster de cómputo

Port 0

- Int. ALU & Shift
- Int. SIMD ALU, Int. SIMD Shuffle
- SP , DP VMUL

Port 1

- Int. ALU, Int. MUL
- Int. SIMD ALU, Int. SIMD Shift
- FP VADD

Port 5

- Int. ALU & Shift, JMP
- Int. SIMD ALU, Int. SIMD Shuffle
- FP/SIMD/SSE2 Move and Logic

Cluster de operaciones de memoria

Port 2: LOAD

Port 3: STO Address

Port 4: STO Data

Maximum rate: 1 load más 1 store de 128 bits c/u por ciclo

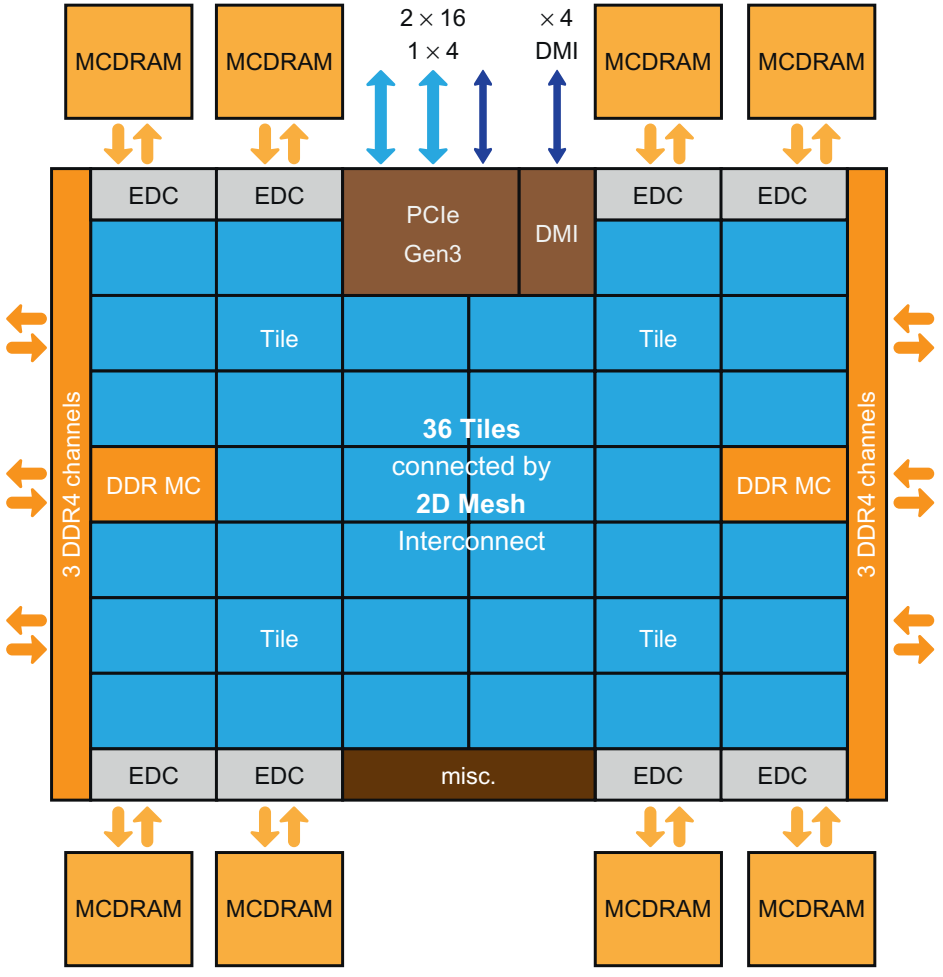
Knights Landing: Procesador Xeon Phi de segunda generación

KNL: Procesador Intel Many-Core para High Performance Computing

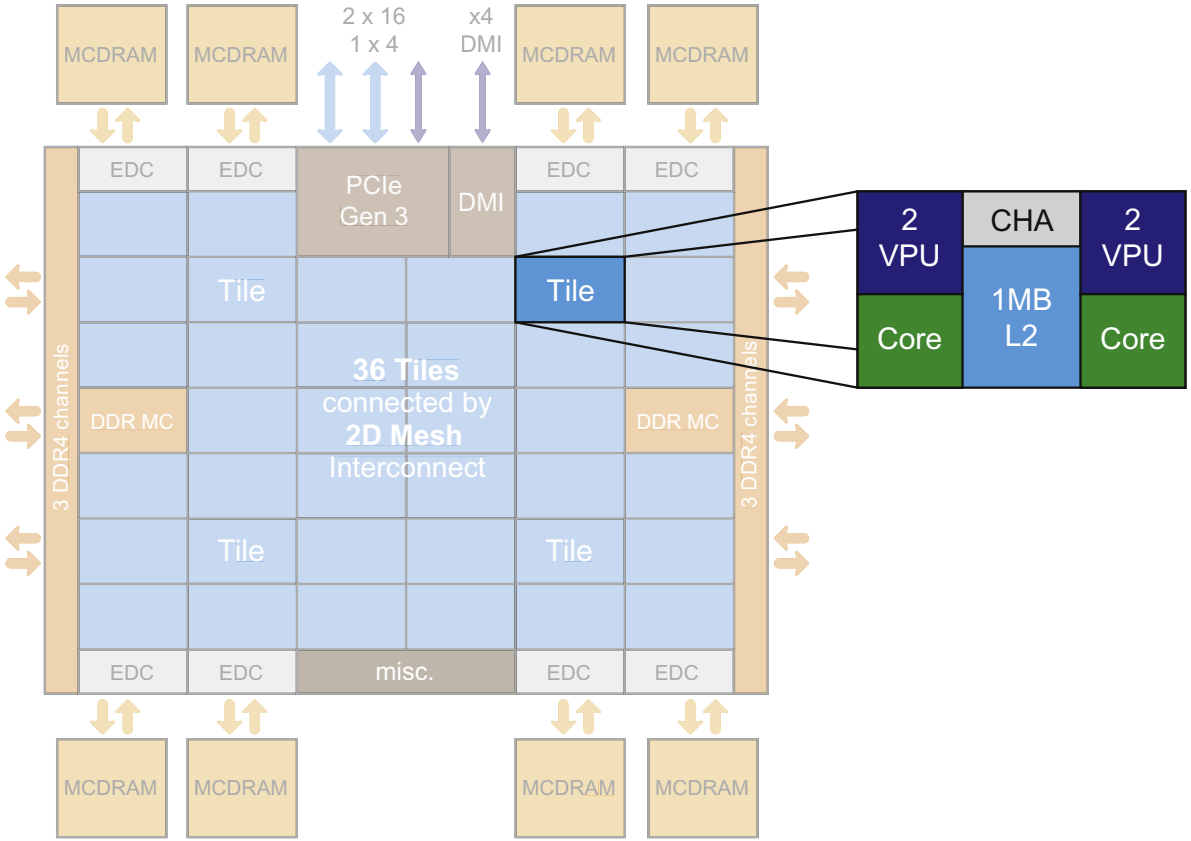
- Procesadores de auto-booteo y co-procesador
- Compatible con código binario con procesadores IA
- Memoria on-chip
- Many-core
- 3 TFLOPS precisión doble, 6 TFLOPS precisión simple
- (Kinghts Corner (KNC): co-procesador conectado a través de PCI)

Arquitectura general

- 38 tiles físicos (36 activos)
- Malla 2D de interconección
- Dos tipos de memoria: Multi-Channel DRAM (MCDRAM), y Double Data-rate (DDR)
- MCDRAM: 16 Gbytes on-package (450 GBps)
- DDR 384 Gbytes max (90 Gbps)
- Dos tipo de modelos de memoria



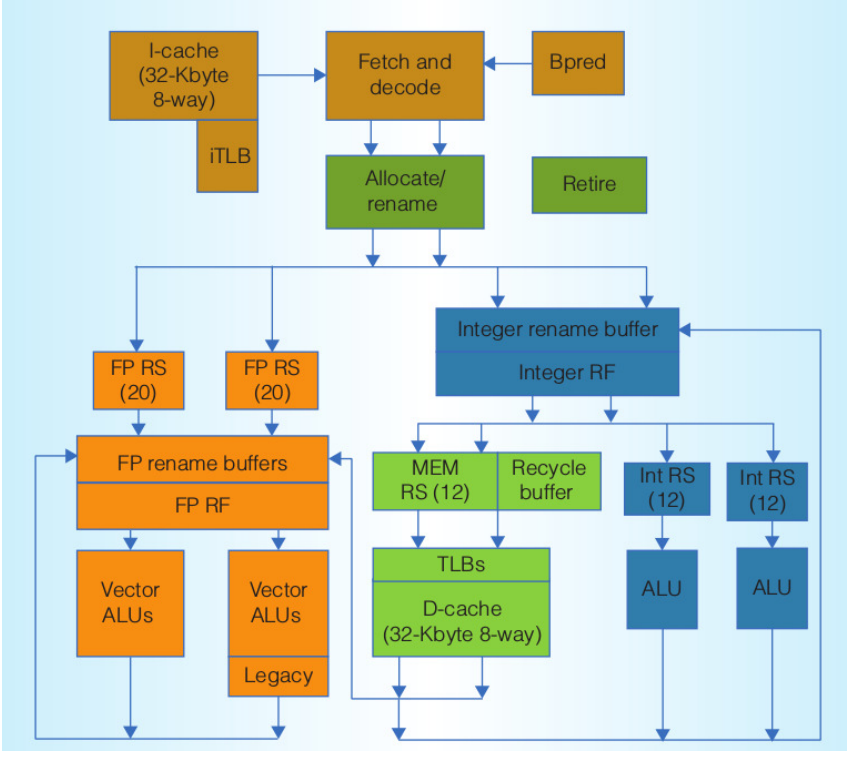
Los tiles



- Dos cores por tile
- Cada core posee 2x AVX512
- Cache L2 de 1 Mbyte, compartida
- Coherencia L2 en toda la malla

El core de KNL

- **Front-end Unit** (FEU)
- **Allocation unit**
- **Integer exec unit** (IEU)
- **Memory exec unit** (MEU)
- **Vector processing unit** (VPU)

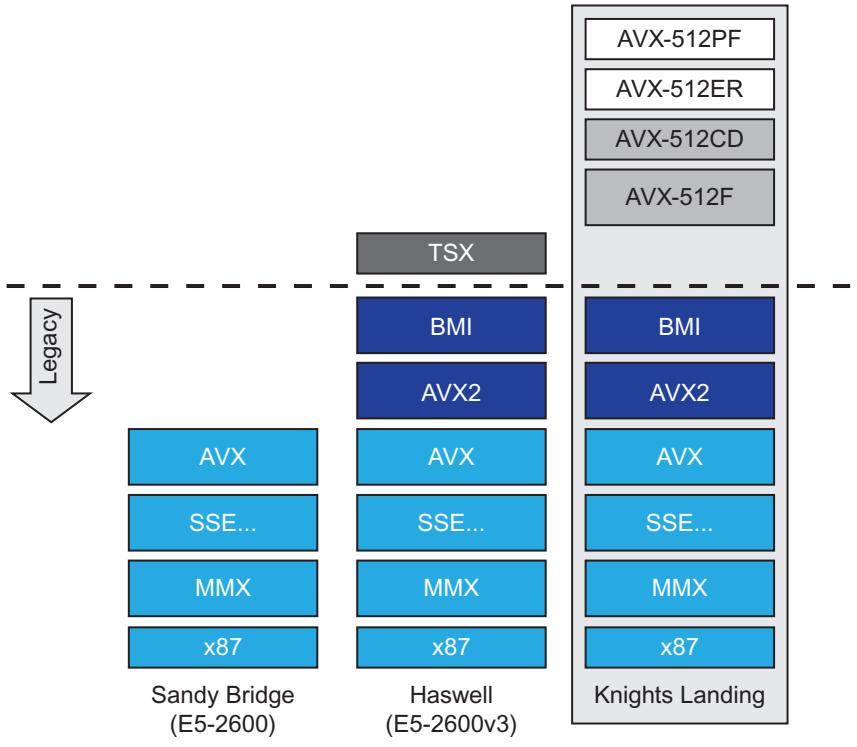


Threads en KNL

- Cada core soporta 4 hebras de hardware (o contextos)
- Seleccionadores de thread dinámicamente asignan, particionan los recursos del core entre las hebras
- Tres modos:
  1. Single-thread : 1 hebra activa, 3 inactivas
  2. Dual-thread: 2 hebras activas, 2 inactivas
  3. Quad-thread: 3 hebras activas, 1 inactiva
- Recursos compartidos: Reorder buffer, rename buffers, reservation stations, data buffers, instruction queue
- Recursos replicados (no compartidos): Registros de control (IP, IFLAGS, etc)

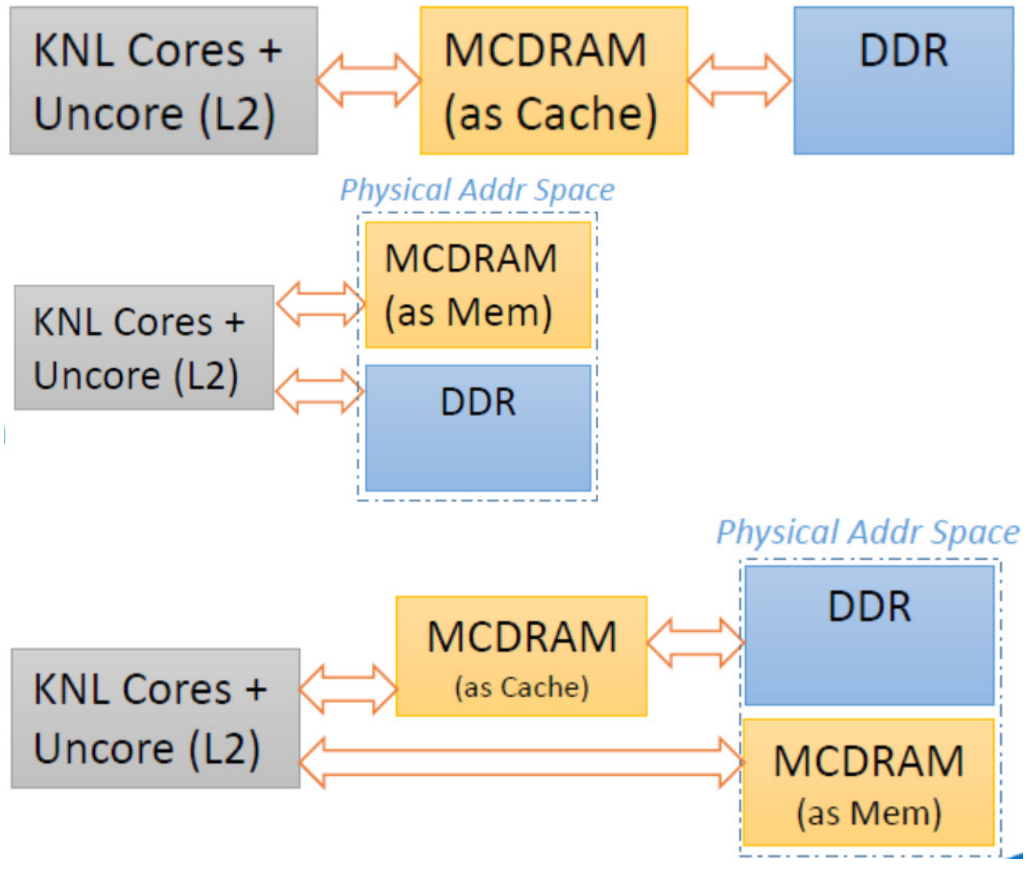
Conjunto de instrucciones

- SSE 128 bits, AVX 256 bits, AVX2
- AVX-512 soporta registros de 512 bits.
- 1 instrucción AVX-512 por ciclo $\Rightarrow$
- 8 instrucciones mult-add precisión doble por VPU por ciclo
- 16 instrucciones mult-add precisión simple por VPU por ciclo



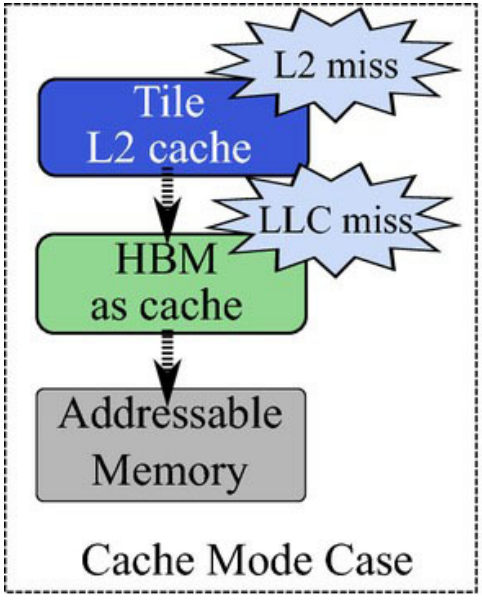
Modelos de memoria

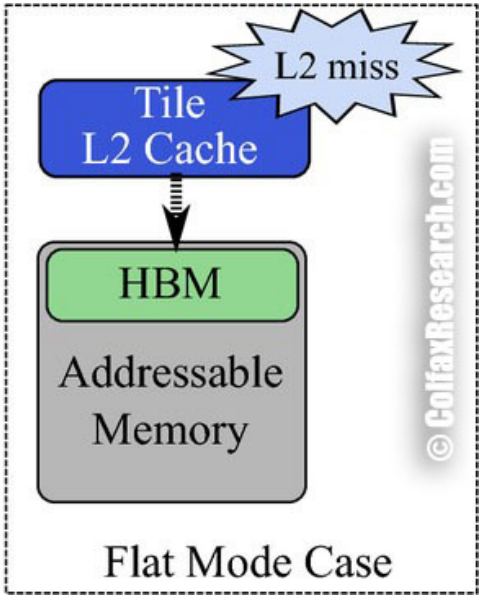
- Dos tipos de memoria: MCDRAM y DDR
- Tres modos de memoria:
  1. Cache: MCDRAM usada como memoria cache de la DDR
  2. Flat: MCDRAM y DDR como un sólo espacio físico de direccionamiento
  3. Híbrido: 4 o 8 GB de MCDRAM usado como cache y el resto como plano
- Selección de modo durante booteo
- Modo cache es transparente
- Modos Flat e híbrido manejados por SW



Modelos de memoria

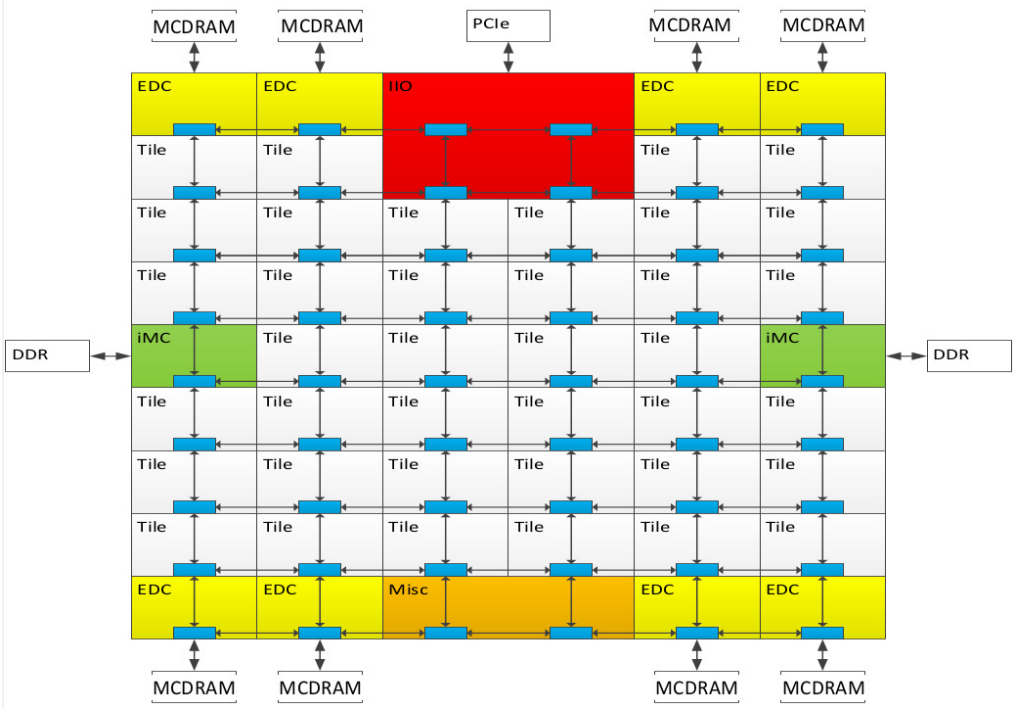
1. Modo Cache
   - Transparante al programador
   - Mejora en bandwidth
   - En caso de miss, aumenta latencia
   - Memoria global disponible es menor
2. Modo Flat
   - Más memoria
   - Baja latencia (en caso de miss)
   - Máximo bandwidth
   - No es transparante al programador





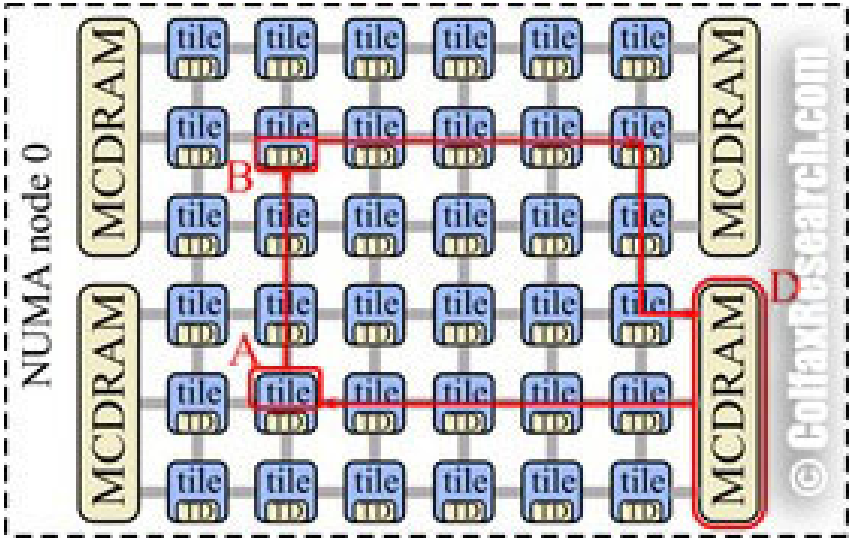
Arquitectura de la malla 2D

- Cada fila y columna es un camino bidireccional
- Mensajes viajan en modo YX
- Coherencia entre todas las cache
- Directorio Distribuido de Tags (DTD)
- Tres modos de clustering:
  - All-to-all
  - Quadrant/hemisphere
  - Sub-NUMA
- Modo clustering seteado durante booteo



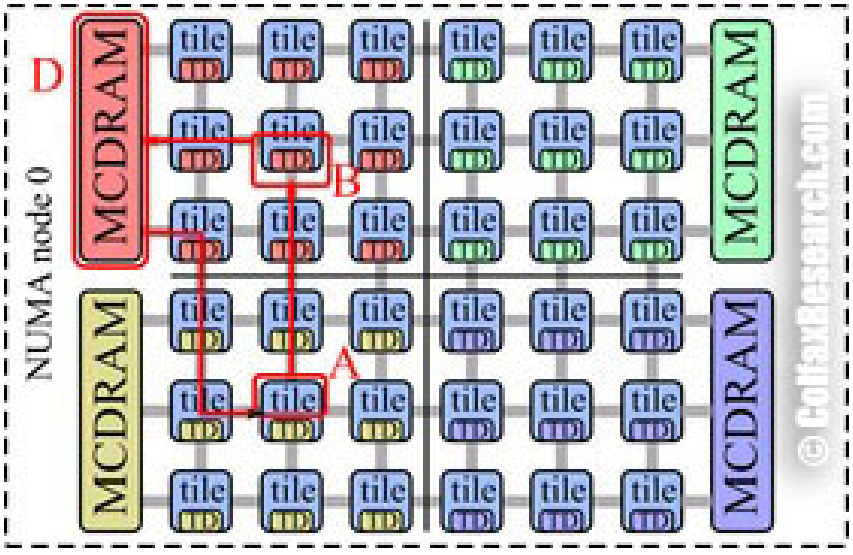
Clustering All-to-all

- Direcciones son distribuidas uniformemente entre directorios de todos los tiles
- Sin afinidad entre tile, Directorio y Memoria
- Latencia alta en el peor caso
- Peor rendimiento entre los modos



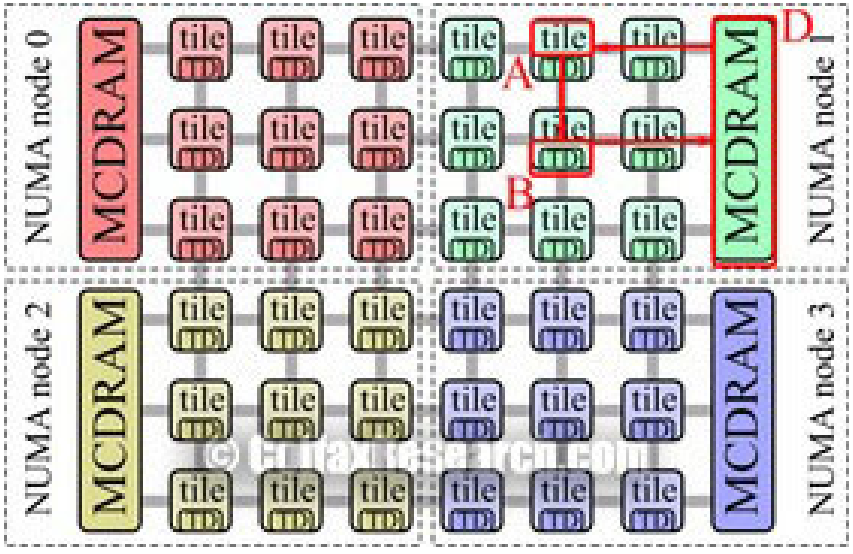
Clustering Quadrant

- La malla es dividida en 4 quadrantes
- Cada grupo de directorio es afin a un subconjunto de memoria
- No hay afinidad entre tile y directorio, y tile y memoria
- Directorio solo accede a memoria de su quadrante
- Mejor rendimiento que all-to-all



Clustering Sub-NUMA

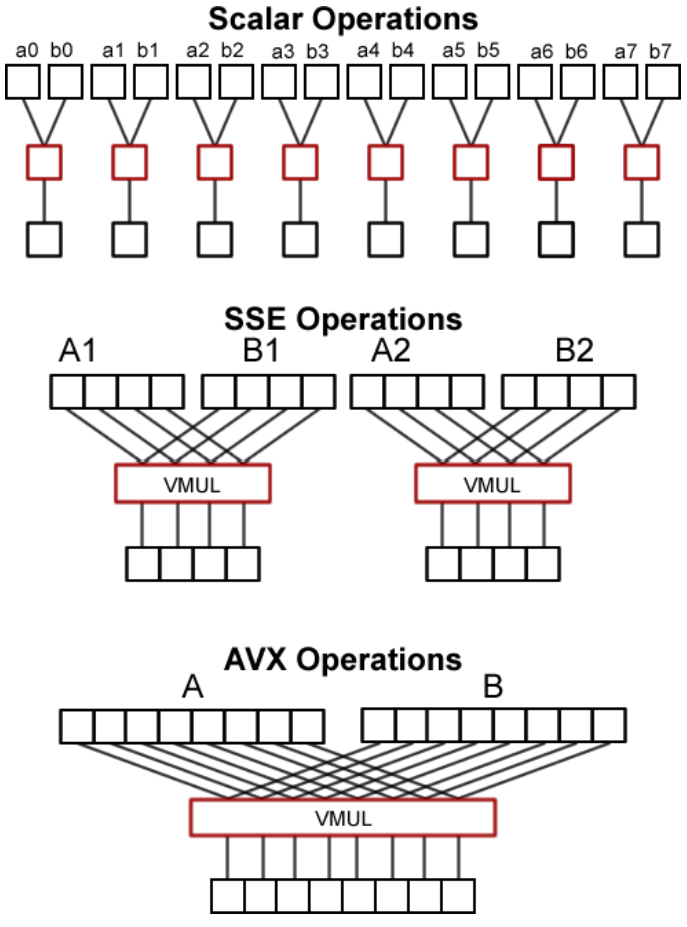
- Similar a modo quadrante
- Afinidad entre tile y directorio y memoria
- El chip aparece como 4 dominios NUMA al SO
- Hebras "pinned" a quadrantes
- No transparente al programador
- Mejor rendimiento entre los modos



SIMD

- La misma uop se ejecuta sobre un conjunto de datos del mismo tipo
- Nehalem usa registros de 128 bits, es decir 4×32 (4-way) o 2×64 (2-way) etc.
- Instrucciones AVX operan sobre 256 bits, en Sandy Bridge

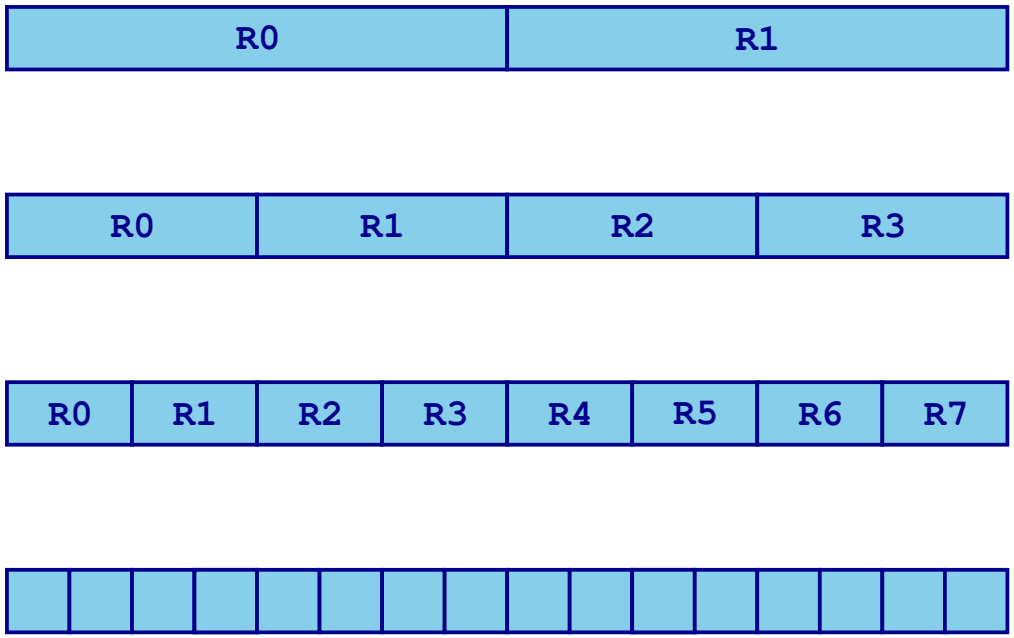
```
void vectormul(float *a, float *b, float *c,
               int size) {
    for (int i=0; i < size; i++)
        c[i] = a[i]*b[i];
}
```



Tipos de datos SSE

Un registro MMX es un registro de 128 bits, que puede alojar distintos tipos de datos, dependiendo de las siguientes declaraciones

```
__m128d R;  // 2 x 64 bits doubles
 __m128 R;  // 4 x 32-bits floats,
__m128i R;  // 4 x 32-bits ints
__m128i R;  // 8 x 16-bits ints
__m128i R;  // 16 x 8-bits ints
__m128i R;  // 2 x 64-bits ints
__m128i R;  // 128 x 1-bits
```



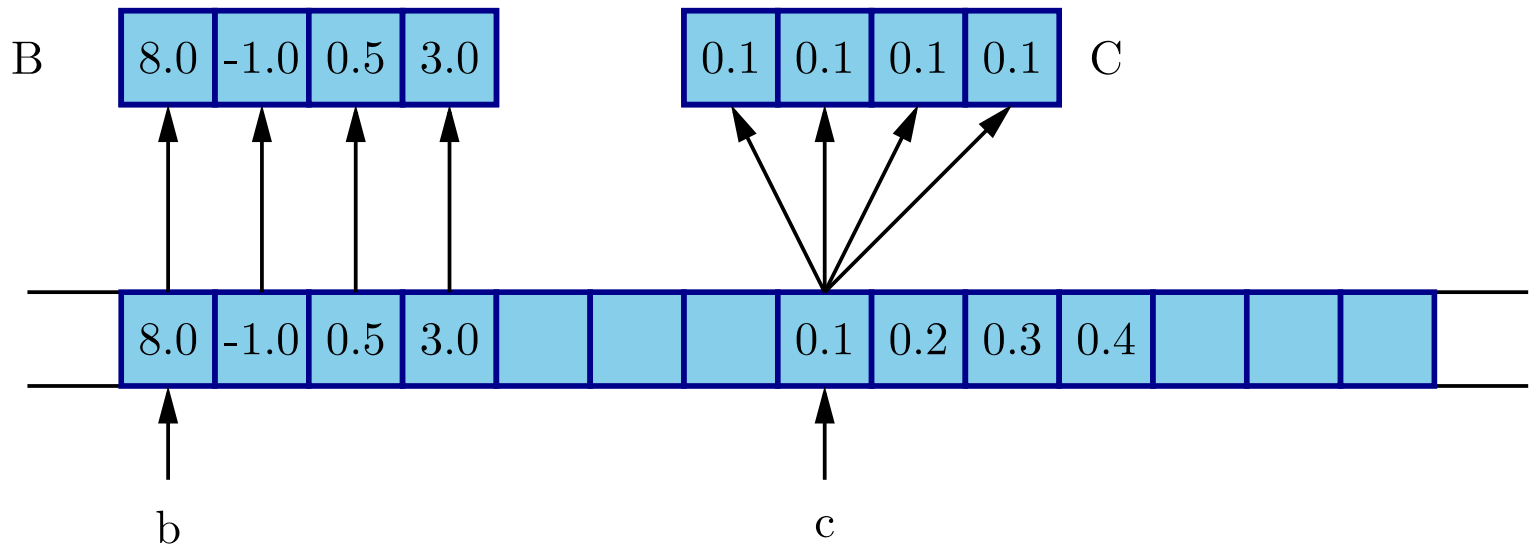
Carga y almacenamiento

```
__m128 B = _mm_load_ps(b)
```

Carga 4 floats en el registro MMX B desde la dirección de memoria b

```
__m128 C = _mm_load1_ps(c)
```

Carga el mismo float apuntado por la dirección c, en las cuatro palabras del registro MMX C




```
float b[4] __attribute__((aligned(16)))   = { 8.0, -1.0, 0.5, 3.0 };
float c[4] __attribute__((aligned(16)))   = { 0.1, 0.2, 0.3, 0.4 };
```

Operaciones load y store

Intrinsic	R0	R1	R2	R3	Comentario
`_mm_load_ss(float *p)`	p[0]	0.0	0.0	0.0	Carga 1 FPS
`_mm_load1_ps(float *p)`	p[0]	p[0]	p[0]	p[0]	Carga 1 FPS
`_mm_load_ps(float *p)`	p[0]	p[1]	p[2]	p[3]	Carga 4 FPS
`_mm_loadu_ps(float *p)`	p[0]	p[1]	p[2]	p[3]	Puede no estar alineado 16 bytes

Intrinsic	p[0]	p[1]	p[2]	p[3]	Comentario
`void _mm_store_ss(float *p, __m128 a);`	a0				
`void _mm_store1_ps(float *p, __m128 a);`	a0	a0	a0	a0	
`void _mm_store_ps(float *p, __m128 a);`	a0	a1	a2	a3	
`void _mm_storeu_ps(float *p, __m128 a);`	a0	a1	a2	a3	Puede no estar alineado 16 bytes

Operaciones de asignación

Intrinsic	R0	R1	R2	R3
`_mm_set_ss(float p)`	p	0.0	0.0	0.0
`_mm_set1_ps(float p)`	p	p	p	p
`_mm_set_ps(float z, float y, float x, float w)`	w	x	y	z
`_mm_setr_ps(float z, float y, float x, float w)`	z	y	x	w
`_mm_setzero_ps(void)`	0.0	0.0	0.0	0.0

Note que `p, x, y, z, w` no son punteros. Puede ser un literal.

```
_m128 one = _mm_set_ss(1.0);
_m128 mymax = _mm_set1_ps(1000.0);
_m128 mymin = _mm_set1_ps(-1000.0);
_m128 zero = _mm_set_zero();
```

Otra forma de inicializar un registro con valores literales:

```
#define PI 3.14159
const _m128 pi = {PI, PI, PI, PI};
```

Operaciones artiméticas Intrinsics SSE

Intrinsic	R0	R1	R2	R3
`_mm_add_ss(__m128 a, __m128 b);`	a0+b0	a1	a2	a3
`_mm_sub_ss(__m128 a, __m128 b);`	a0-b0	a1	a2	a3
`_mm_add_ps(__m128 a, __m128 b);`	a0+b0	a1+b1	a2+b2	a3+b3
`_mm_sub_ps(__m128 a, __m128 b);`	a0-b0	a1-b1	a2-b2	a3-b3
`_mm_mul_ss(__m128 a, __m128 b);`	a0*b0	a1	a2	a3
`_mm_mul_ps(__m128 a, __m128 b);`	a0*b0	a1*b1	a2*b2	a3*b3
`_mm_min_ps(__m128 a, __m128 b);`	min(a0,b0)	min(a1,b1)	min(a2,b2)	min(a3,b3)
`_mm_max_ps(__m128 a, __m128 b);`	max(a0,b0)	max(a1,b1)	max(a2,b2)	max(a3,b3)

Ejemplo simple

- Convertir temperaturas en celcius a Farenheit
- Chequear si SSE1 está disponible en el procesador
- Funciones utilizadas:
  - `_mm_set1_ps()`
  - `_mm_load_ps()`
  - `_mm_mul_ps()`
  - `_mm_add_ps()`
  - `_mm_store_ps()`
- Conversión

$$F = \frac{9 \times C}{5} + 32$$

$$F = 1.8 \times C + 32$$

- Dibujar esquema

Includes y main

```
#include <stdio.h>
#include <stdlib.h>    /* exit  */
#include <string.h>    /* memcpy */
#include <cpuid.h>     /* __get_cpuid_max, __get_cpuid */
#include <mmintrin.h>  /* MMX instrinsics  __m64 integer type  */
#include <xmmintrin.h> /* SSE  __m128  float */

int main (int argc, char * argv[])
{
  unsigned int eax, ebx, ecx, edx;
  unsigned int extensions, sig;
  int result, sse1_available;

  result = __get_cpuid (FUNC_FEATURES, &eax, &ebx, &ecx, &edx);

  sse1_available =  (bit_SSE & edx);
  if (0 == sse1_available) {
     fprintf(stderr, "Error: SSE1 features not available\n");
     exit(-1);
  } else
    fprintf(stderr, "SSE1 features ARE available\n");

  simple_sse1();

}
```

simple_sse1()

```
void test_sse1(void)
{
  __m128 celsiusvector, fahrenheitvector, partialvector, coeffvector;
  const __m128 thirtytwovector = {32.0, 32.0, 32.0, 32.0};
  float celsiusarray[]  __attribute__ ((aligned (16))) = {-100.0, -80.0, -40.0, 0.0};
  float fahrenheitarray[4] __attribute__ ((aligned (16)));
  int i;

  coeffvector      = _mm_set1_ps(9.0 / 5.0);
  celsiusvector    = _mm_load_ps(celsiusarray);
  partialvector    = _mm_mul_ps(celsiusvector, coeffvector);
  fahrenheitvector = _mm_add_ps(partialvector, thirtytwovector);

  _mm_store_ps(fahrenheitarray, fahrenheitvector);

  for (i = 0; i < 4; i++)
      printf("%f celsius es %f fahrenheit\n", celsiusarray[i], fahrenheitarray[i]);
}
-100.000000 celsius es -148.000000 fahrenheit
-80.000000 celsius es -112.000000 fahrenheit
-40.000000 celsius es -40.000000 fahrenheit
0.000000 celsius es 32.000000 fahrenheit
```

Intrinsics, multiplicación de matrices 2×2

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \times \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{pmatrix}$$

$$\begin{array}{|c|c|} \hline a_{00} & a_{00} \\ \hline a_{10} & a_{10} \\ \hline \end{array} \times \begin{array}{|c|c|} \hline b_{00} & b_{01} \\ \hline \end{array} = \begin{array}{|c|c|} \hline a_{00}b_{00} & a_{00}b_{01} \\ \hline a_{10}b_{00} & a_{10}b_{01} \\ \hline \end{array}$$

Matmul 2×2 SSE

```
#include<stdio.h>
#include<emmintrin.h>

int main(void){
    double A[4] __attribute__((aligned(16))) = { 2.0, 1.0, 1.0, 3.0};
    double B[4] __attribute__((aligned(16))) = { 3.4, 2.8, 1.2, 0.5};
    double C[4] __attribute__((aligned(16))) = { 0.0, 0.0, 0.0, 0.0};
    int lda=2, i=0;

    __m128d a1, a2, b, c1, c2;
    c1 = _mm_load_pd(C+0*lda);
    c2 = _mm_load_pd(C+1*lda);

    for (i=0; i< 2; i++) {
        a1 = _mm_load1_pd(A + i + 0*lda);
        a2 = _mm_load1_pd(A + i + 1*lda);
        b  = _mm_load_pd(B + i*lda);
        c1 = _mm_add_pd(c1, _mm_mul_pd(a1, b));
        c2 = _mm_add_pd(c2, _mm_mul_pd(a2, b));
    }
    _mm_store_pd(C+0*lda, c1);
    _mm_store_pd(C+1*lda, c2);
    printf("%g\t%g\t%g\t%g\n", C[0], C[1], C[2], C[3]);
}
```

Operaciones lógicas

Intrinsic	Operación
`_mm_and_ps(__m128 a, __m128 b);`	AND bitwise
`_mm_andnot_ps(__m128 a, __m128 b);`	ANDNOT bitwise
`_mm_or_ps(__m128 a, __m128 b);`	OR bitwise
`_mm_xor_ps(__m128 a, __m128 b);`	XOR bitwise

Operaciones artiméticas Intrinsics SSE2

1. Cada vector se compone 128 bits
2. Precisión doble

`__m128d R;`

R0	R1

Intrinsic	R0	R1
`_mm_add_sd(__m128 a, __m128 b);`	a0+b0	a1
`_mm_add_pd(__m128 a, __m128 b);`	a0+b0	a1+b1
`_mm_sub_sd(__m128 a, __m128 b);`	a0-b0	a1
`_mm_sub_pd(__m128 a, __m128 b);`	a0-b0	a1-b1
`_mm_mul_pd(__m128 a, __m128 b);`	a0*b0	a1*b1
`_mm_div_pd(__m128 a, __m128 b);`	a0/b0	a1/b1
`_mm_sqrt_pd(__m128 a, __m128 b);`	sqrt(a2)	sqrt(a1)
`_mm_min_pd(__m128 a, __m128 b);`	min(a0,b0)	min(a1,b1)
`_mm_max_pd(__m128 a, __m128 b);`	max(a0,b0)	max(a1,b1)

Operaciones load y store SSE2

- Por defecto, direcciones fuentes deben estar alineadas en 16 bytes

Intrinsic	R0	R1	Comentario
`_mm_load_sd(double *p)`	p[0]	0.0	Carga 1 FPD
`_mm_load1_pd(double *p)`	p[0]	p[0]	Carga 1 FPD
`_mm_load_pd(double *p)`	p[0]	p[1]	Carga 2 FPS
`_mm_loadu_pd(double *p)`	p[0]	p[1]	Puede no estar alineado 16 bytes

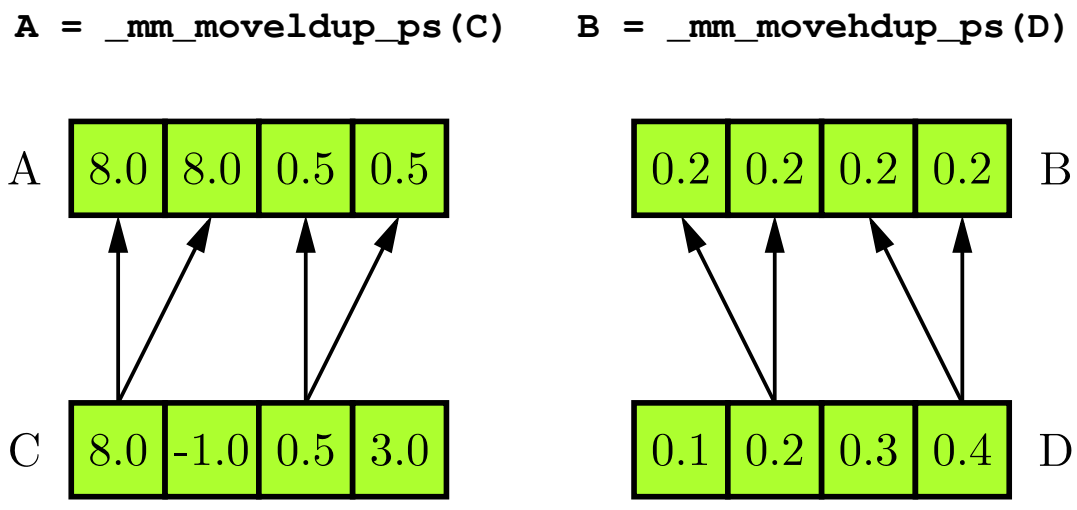
Intrinsic	p[0]	p[1]	Comentario
`void _mm_store_sd(double *p, __m128 a);`	a0		
`void _mm_store1_pd(double *p, __m128 a);`	a0	a0	
`void _mm_store_pd(double *p, __m128 a);`	a0	a1	
`void _mm_storeu_pd(double *p, __m128 a);`	a0	a1	Puede no estar alineado 16 bytes

Movimiento SSE3

- `__m128 _mm_moveldup_ps(__m128 source);`

  Duplica el primer elemento de `source` y los posiciona en el primer y segundo elemento de destino. Duplica el tercer elemento de `source` y los posiciona en el tercer y cuarto elemento de destino
- `__m128 _mm_movehdup_ps(__m128 source);`

  Duplica el segundo elemento de `source` y los posiciona en el primer y segundo elemento de destino. Duplica el cuarto elemento de `source` y los posiciona en el tercer y cuarto elemento de destino



Suma-resta SSE3

Intrinsic	R0	R1	R2	R3
`__m128 _mm_addsubp_ps(__m128 a, __m128 b);`	a0-b0	a1+b1	a2-b2	a3+b3
`__m128 _mm_addsubp_pd(__m128d a, __m128d b);`	a0-b0	a1+b1	a2-b2	a3+b3

Ejemplo simple SSE3

```
#include <stdio.h>
#include <pmmintrin.h>

int main( )
{
    __m128 u, v, w;
    float a[4] __attribute__((aligned(16)))   = { 0.1, 0.2, 0.3, 0.4 };
    float b[4] __attribute__((aligned(16)))   = { 0.0001, 0.002, 0.003, 0.004 }

    printf("Loading %5.3f %5.3f %5.3f %5.3f into XMM register.\n",
             a[0], a[1], a[2], a[3] );
    u = _mm_load_ps(a);
    printf("Loading %5.3f %5.3f %5.3f %5.3f into XMM register.\n",
             b[0], b[1], b[2], b[3] );
    v = _mm_load_ps(b);
    w = _mm_addsub_ps ( u , v);
    _mm_store_ps(b, w);
    printf("Result: %5.3f %5.3f %5.3f %5.3f\n", w[0], w[1], w[2], w[3] );
}
$ gcc -msse3 -o addsub addsub.c
$ ./addsub
Loading 0.100 0.200 0.300 0.400 into XMM register.
Loading 0.000 0.002 0.003 0.004 into XMM register.
Result: 0.100 0.202 0.297 0.404
```

Shuffle SSE y SSE2

La operación shuffle permite componer un registro con ciertas permutaciones desde uno o dos otros registros.

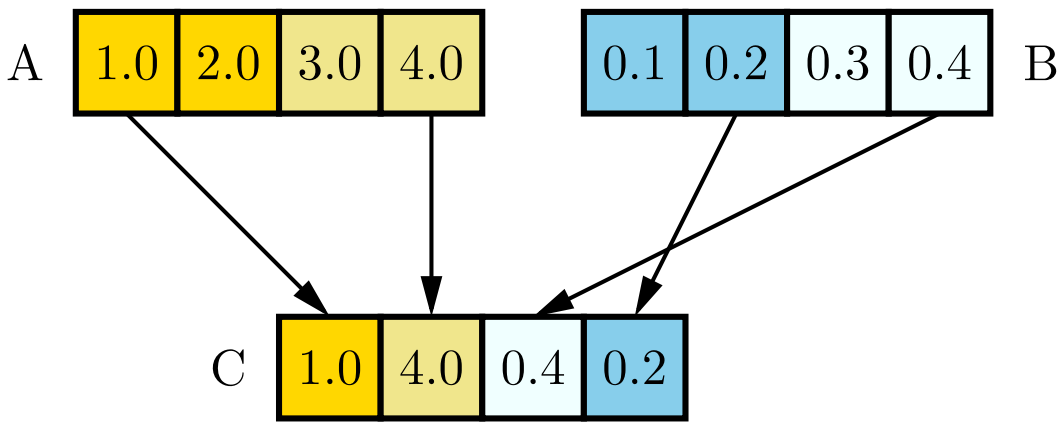
```
__m128  C = _mm_shuffle_ps(A, B, _MM_SHUFFLE(c3,c2,c1,c0))
__m128d C = _mm_shuffle_pd(A, B, _MM_SHUFFLE(c1,c0))
```

- La macro `_MM_SHUFFLE`(c_3, c_2, c_1, c_0) especifica las permutaciones
- c_0 indica la posición del registro `A` que debe copiarse a la posición 0 del registro destino
- c_1 indica la posición del registro `A` que debe copiarse a la posición 1 del registro destino
- c_2 indica la posición del registro `B` que debe copiarse a la posición 2 del registro destino
- c_3 indica la posición del registro `B` que debe copiarse a la posición 3 del registro destino

Ejemplo de shuffle (1)

Shuffle con dos registros fuentes

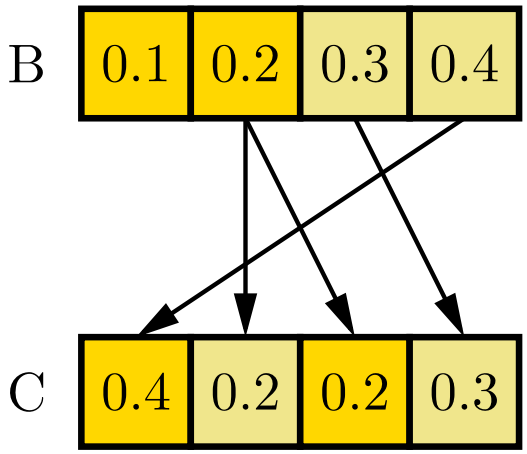
```
C = _mm_shuffle_ps(A, B, _MM_SHUFFLE(1,3,3,0))
```



Ejemplo de shuffle (2)

Shuffle con un registro fuente

```
C = _mm_shuffle_ps(B, B, _MM_SHUFFLE(2,1,1,3))
```



Ejemplo simple SSE3

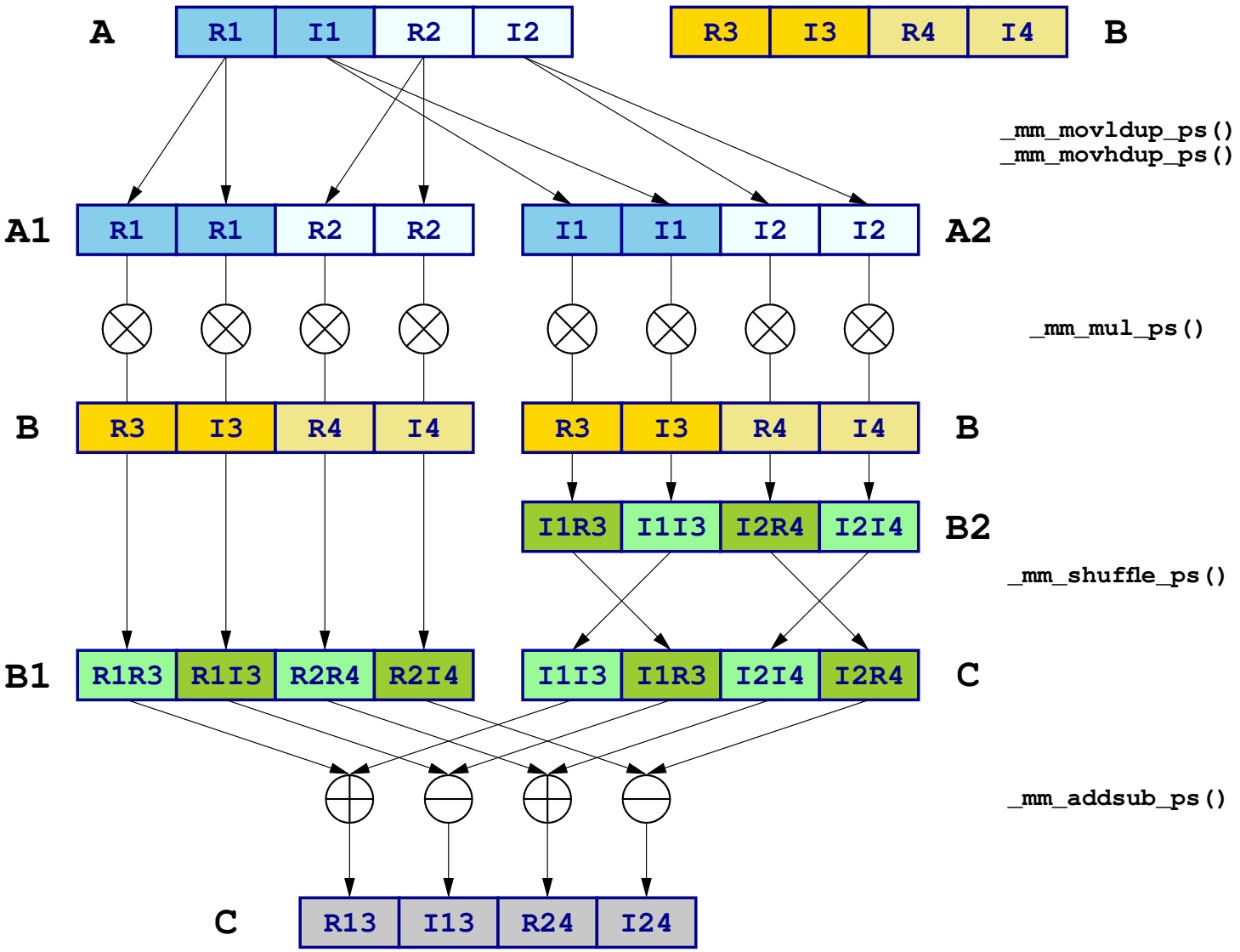
```
#include <stdio.h>
#include <stdlib.h>
#include <pmmintrin.h>

int main( )
{
     __m128 A1, A2, A, B, C, B1, B2, D;
    float a[4] __attribute__((aligned(16)))   = { 1.0, 2.0, 3.0, 4.0 };
    float b[4] __attribute__((aligned(16)))   = { 0.1, 0.2, 0.3, 0.4 };

    A = _mm_load_ps(a);                 //C3 C2 C1 C0
    B = _mm_load_ps(b);                 //B1 B3 A3 A0
    C = _mm_shuffle_ps(A, B, _MM_SHUFFLE(1, 3, 3, 0));
    _mm_store_ps(a, C);

    printf("%f  %f  %f  %f\n", a[0], a[1], a[2], a[3]);

    D = _mm_shuffle_ps(B, B, _MM_SHUFFLE(2, 1, 1, 3));
    _mm_store_ps(a, D);

    printf("%f  %f  %f  %f\n", a[0], a[1], a[2], a[3]);
}

$ ./shuffle
1.000000  4.000000  0.400000  0.200000
0.400000  0.200000  0.200000  0.300000
```

Ejemplo: multiplicación compleja

$$
\begin{aligned}
(c_{1r} + jc_{1i}) &= (a_{1r} + ja_{1i}) \times (b_{1r} + jb_{1i}) \\
c_{1r} &= a_{1r}b_{1r} - a_{1i}b_{1i} \\
c_{1i} &= a_{1r}b_{1i} + a_{1i}b_{1r}
\end{aligned}
$$

- Suponga que los registros `A` y `B` almacenan dos números complejos cada uno.
- `A =` `A1r A1i A2r A2i`, `B =` `B1r B1i B2r B2i`
- Utilizando los intrinsic SS3 recién vistos, escriba un programa que calcule la multiplicación compleja de estos números.
- Es decir `A1*B1` y `A2*B2`

Solución (1)

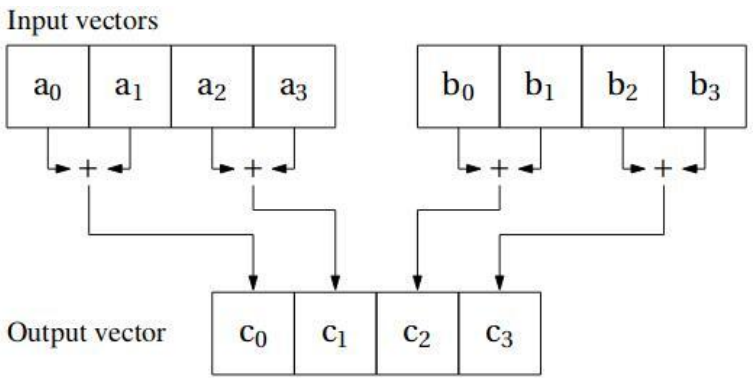

A
R1
I1
R2
I2
R3
I3
R4
I4
B
_mm_movldup_ps()
_mm_movhdup_ps()
A1
R1
R1
R2
R2
I1
I1
I2
I2
A2
_mm_mul_ps()
B
R3
I3
R4
I4
R3
I3
R4
I4
B
I1R3
I1I3
I2R4
I2I4
B2
_mm_shuffle_ps()
B1
R1R3
R1I3
R2R4
R2I4
I1I3
I1R3
I2I4
I2R4
C
_mm_addsub_ps()
C
R13
I13
R24
I24

Solución (2)

```
#include <stdio.h>
#include <pmmintrin.h>

int main( )
{
    __m128 A1, A2, A, B, C, B1, B2, D;
    float a[4] __attribute__((aligned(16)))   = { 1.0, 2.0, 3.0, 4.0 };
    float b[4] __attribute__((aligned(16)))   = { 0.1, 0.2, 0.3, 0.4 };

    A  = _mm_load_ps(a);
    B  = _mm_load_ps(b);
    A1 = _mm_moveldup_ps(A);
    A2 = _mm_movehdup_ps(A);
    B1 = _mm_mul_ps(A1, B);
    B2 = _mm_mul_ps(A2, B);

    C  = _mm_shuffle_ps(B2, B2, _MM_SHUFFLE(2, 3, 0, 1));
    D  = _mm_addsub_ps(B1, C);
    _mm_store_ps(a, D);

    printf("(%f, %f)  (%f, %f)\n", a[0], a[1], a[2], a[3]);
}
```

Suma horizontal

```
__m128  c = _mm_hadd_ps(a, b)
```



Loop unrolling y SIMD

1. Loop unrolling es una técnica de transformación de la estructura de un loop (for-loop)
2. Intenta optimizar el cómputo
3. Generalmente aumenta el tamaño del código (muy poco)
4. Puede que el códifo resultante sea menos claro
5. Técnica básica para implementar SIMD
6. Compiladores hacen loop unrolling muy bien!

Ejemplo loop unrolling

- El siguiente código suma los elementos de un vector

```
float sum(int *a, float n) {
    float  sum = 0;

    for( int i = 0; i < n; i++ )
        sum += a[i];
    return sum;
}
```

- Asumiendo que n es múltiplo de 4:

```
float suma_unrolled(int *a, float n) {
    float sum = 0;

    for (int i = 0; i < n/4; i += 4 ) {
        sum += a[i+0];
        sum += a[i+1];
        sum += a[i+2];
        sum += a[i+3];
    }

    return sum;
}
```

Suma con SIMD

```
float suma_simd(float *a, int n) {
    float sum;

    __m128 vsum = _mmset1_ps(0.0);

    for (int i=0; i < n; i += 4) {
        __m128 v = _mm_load_ps(&a[i]);
        vsum = _mm_add_ps(vsum, v);
    }

    vsum = _mm_hadd_ps(vsum, vsum);
    vsum = _mm_hadd_ps(vsum, vsum);
    _mm_store_ss(&sum, vsum);

    return(sum);
}
```

Cómo tratar largos no múltiplos de 4

```
int suma_unrolled(int *a, int n) {
    int sum = 0;

    for (int i = 0; i < n/4*4; i += 4 ) {
        sum += a[i+0];
        sum += a[i+1];
        sum += a[i+2];
        sum += a[i+3];
    }

    for (int i = n/4*4; i < n; i++ )
        sum += a[i];

    return sum;
}
```

¿Qué intrinsic soporta mi procesador?

```
$ lshw -class processor
  *-cpu
       product: Intel(R) Core(TM) i5 CPU         661  @ 3.33GHz
       vendor: Intel Corp.
       physical id: 1
       bus info: cpu@0
       size: 1197MHz
       capacity: 1197MHz
       width: 64 bits
       capabilities: fpu fpu_exception wp vme de pse tsc msr pae mce cx8 apic
       sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht
       syscall nx rdtscp x86-64 constant_tsc arch_perfmon pebs bts rep_good nop
       xtopology nonstop_tsc aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx
       est tm2 ssse3 cx16 xtpr pdcm sse4_1 sse4_2 popcnt aes lahf_lm ida arat
       dtherm tpr_shadow vnmi flexpriority ept vpid cpufreq
```

Alignment

- En C, con inicialización inmediata:

```
float a[4] __attribute__((aligned(16))) = { 1.0, 2.0, 3.0, 4.0 };
float b[4] __attribute__((aligned(16))) = { 0.1, 0.2, 0.3, 0.4 };
```

- En C, memoria diámica:

```
float *a, *b
posix_memalign((void **) &a, 16, 64);
posix_memalign((void **) &b, 16, 64);
```

- En C++11, memoria dinámica:

```
float *a = (float *) aligned_alloc(16, 64);
float *b = (float *) aligned_alloc(16, 64);
```

Más información

https://software.intel.com/en-us/node/524261