Subword Parallelism

Subword parallelism

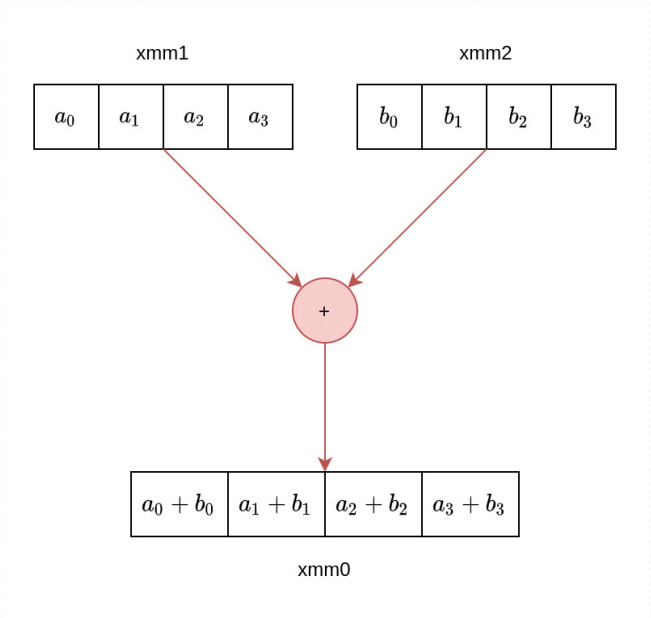
- **Subword parallelism**. Capacidad de las CPUs modernas de operar sobre datos empacados en un registro.
- También se le conoce como SWAR (SIMD within a register).
- En 1999, la extensión SSE (Streaming SIMD Extensions) agregó 70 instrucciones y 8 registros, `xmm0` a `xmm7`, de 128 bits al ISA x86.
- Luego se agregaron otros 8 registros, `xmm8` a `xmm15`.
- Cada registro puede guardar empacados 16 bytes, 8 palabras (words), 4 palabras dobles (double words), 2 palabras cuádruples (quad words).

Subword parallelism

Fuente: https://www.songho.ca/misc/sse/sse.html

- SSE proporciona instrucciones para operar sobre los registros de manera *independiente* y en *paralelo*.
- La instrucción `vaddps xmm0, xmm1, xmm2` interpreta como `float` las 4 palabras dobles empacadas en `xmm1` y `xmm2`, las suma en paralelo y las guarda en `xmm0`.

Subword parallelism

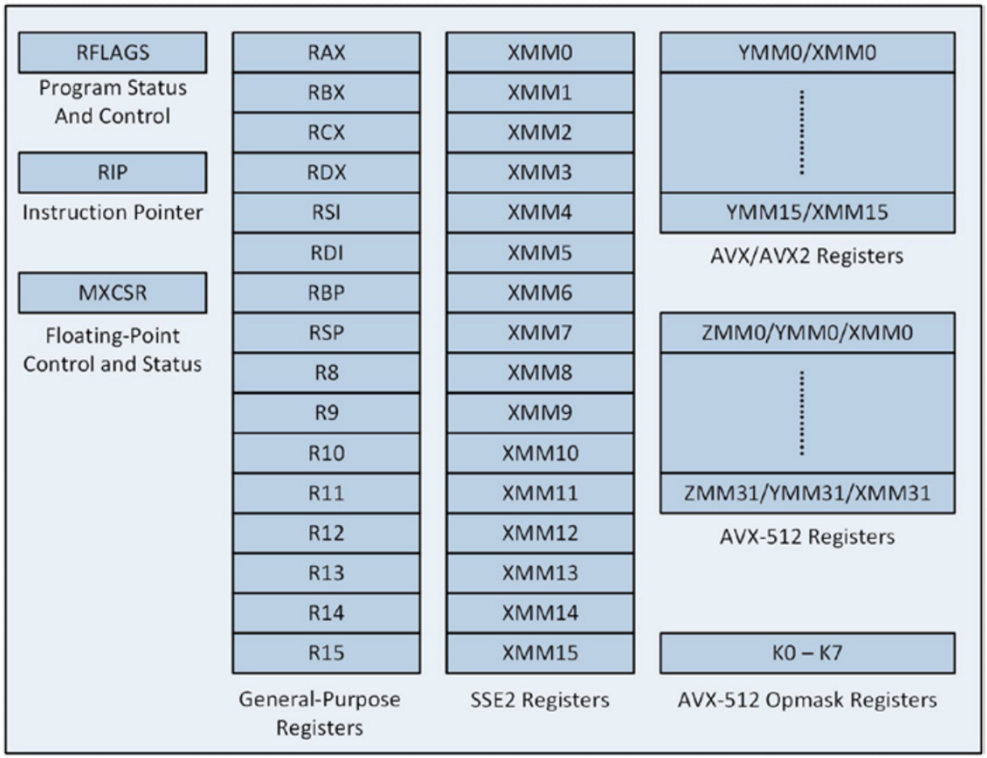


Fuente: https://www.researchgate.net/figure/Figure-S12-The-two-fundamental-SIMD-instructions-for-the-prefix-sum-Left-vaddps_fig5_355221823

Extensiones SIMD en x86

- 64 bits. MMX (1997).
- 128 bits. Streaming SIMD Extensions: SSE (1999), SSE2 (2000), SSE3 (2004), SSSE3 (2006), SSE4.1 (2008), SSE4.2 (2008).
- 256 bits. Advanced Vector Extensions: AVX (2011), AVX2 (2013).
- 512 bits. AVX-512 (2017).

Registros en x86



Fuente: Mx86ALP, p. 7

Ejemplo 1

- En graficación por computadora, la suma de vectores es muy utilizada.
- Para sumar dos vectores de cuatro componentes de precisión simple usando x86 se requieren cuatro instrucciones de suma de punto flotante.

```
v3.x = v1.x + v2.x;
v3.y = v1.y + v2.y;
v3.z = v1.z + v2.z;
v3.w = v1.w + v2.w;
```

- Esto corresponde a cuatro instrucciones `fadd` en x86.

Ejemplo 1

- Con SSE, una instrucción de suma empacada de 128 bits sustituye a las cuatro sumas.

```
movaps xmm0, [v1] ; xmm0 = v1.w | v1.z | v1.y | v1.x
addps xmm0, [v2]  ; xmm0 = v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x
movaps [v3], xmm0  ; v3[x|y|z|w] = xmm0
```

Ejemplo 2

- Usando instrucciones SIMD programar una función en ensamblador que sume dos arreglos de tipo `float`.

Suma de arreglos en C

```
void add(float *z, float *x, float *y, int n)
{
    int i;
    for (i = 0; i < n; i++) {
        z[i] = x[i] + y[i];
    }
}
```

Programa principal en C

```
int main()
{
   float x[] = {1, 2, 3, 4, 5};
   float y[] = {11, 12, 13, 14, 15};
   float z[5];
   int i, n = 5;
   add (&z[0], &x[0], &y[0], n);
   for (i = 0; i < n; i++) {
      printf("%f\n", z[i]);
   }
   return 0;
}
```

Suma de arreglos en ensamblador

```
; rcx points to z; rdx points to x; r8 points to y; r9 has n
NSE     equ 8                   ; num_simd_elements
        xor rax, rax            ; i = 0
Loop1:  mov r10, r9             ; r10 = n
        sub r10, rax            ; r10 = n – i
        cmp r10, NSE            ; is n – i < NSE?
        jb Loop2                ; jump if yes
```

Suma de arreglos en ensamblador

```
; Calculate z[i:i+7] = x[i:i+7] + y[i:i+7]
    vmovups ymm0,ymmword ptr [rdx+rax*4]   ; ymm0 = x[i:i+7]
    vmovups ymm1,ymmword ptr [r8+rax*4]    ; ymm1 = y[i:i+7]
    vaddps ymm2,ymm0,ymm1                  ; z[i:i+7] = x[i:i+7] + y[i:i+7]
    vmovups ymmword ptr [rcx+rax*4],ymm2   ; save z[i:i+7]

    add rax,NSE                    ; i += NSE
    jmp Loop1                      ; repeat Loop1 until done
```

Suma de arreglos en ensamblador

```
Loop2:  cmp rax, r9                          ; is i >= n?
        jae Done                             ; jump if yes
; Calculate z[i] = x[i] + y[i] for remaining elements
        vmovss xmm0,real4 ptr [rdx+rax*4]    ; xmm0 = x[i]
        vmovss xmm1,real4 ptr [r8+rax*4]     ; xmm1 = y[i]
        vaddss xmm2,xmm0,xmm1                ; z[i] = x[i] + y[i]
        vmovss real4 ptr [rcx+rax*4],xmm2    ; save z[i]
        inc rax                              ; i += 1
        jmp Loop2                            ; repeat Loop2 until done
Done:   vzeroupper                           ; clear upper bits of ymm regs
        ret                                  ; return to caller
```

Funciones intrínsecas

- Usando funciones intrínsecas se puede obtener el mismo resultado utilizando C o C++.
- **Función intrínseca**. Función que se puede llamar desde C/C++ y que el compilador la va a sustituir por una instrucción SIMD en ensamblador.
- Ejemplo: la función `_mm256_add_ps()` se sustituye por `vaddss` para sumar 8 elementos `float` empacados en un registro.

Requisitos

- Contar con una CPU con extensiones SIMD.
- Contar con un compilador que reconozca funciones intrinsecas: Visual Studio, gcc, Intel C++ Compiler.

Suma de arreglos con intrínsecas

```
void CalcZ_Iavx(float* z, const float* x, const float* y, size_t n)
{
    size_t i = 0;
    const size_t num_simd_elements = 8;
    for (; n - i >= num_simd_elements; i += num_simd_elements) {
        // Calculate z[i:i+7] = x[i:i+7] + y[i:i+7]
        __m256 x_vals = _mm256_loadu_ps(&x[i]);
        __m256 y_vals = _mm256_loadu_ps(&y[i]);
        __m256 z_vals = _mm256_add_ps(x_vals, y_vals);
        _mm256_storeu_ps(&z[i], z_vals);
    }
```

Suma de arreglos con intrínsecas

```
    // Calculate z[i] = x[i] + y[i] for any remaining elements
    for (; i < n; i += 1)
        z[i] = x[i] + y[i];
}
```

Bibliografía

Modern Parallel Programming with C++ and Assembly Language: X86 SIMD Development Using AVX, AVX2, and AVX-512

Daniel Kusswurm

Apress, 2022

Recursos

Intel Intrinsics Guide:
https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

Intel 64 and IA-32 Architectures Software Developer's Manuals:
https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html