Arquitectura del Computador
y
Sistemas Operativos

Sexta Clase

Lenguaje Assembler

Los procesadores x86-64

Antes de hablar del juego de instrucciones es inevitable mencionar que las mismas son un conjunto de *capas*, que se han ido agregando mientras se mantiene la compatibilidad con los procesadores anteriores de la misma línea.

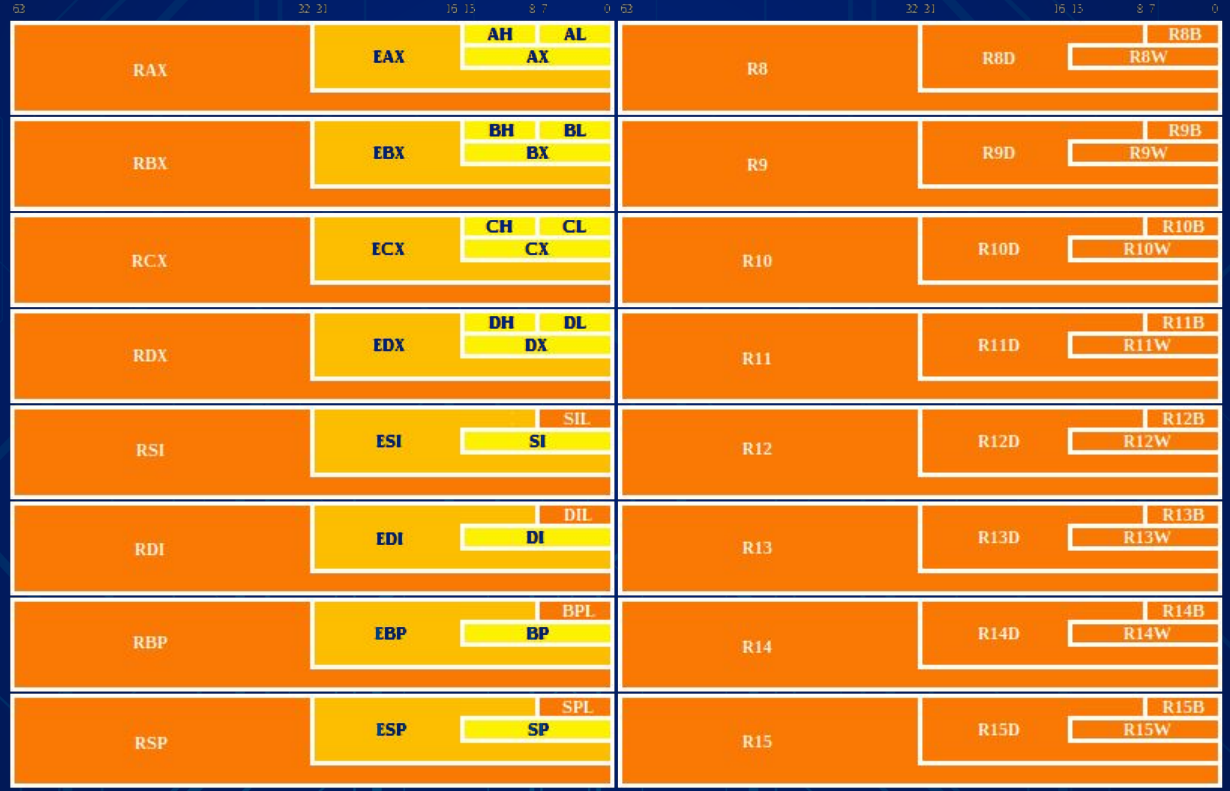
Hoy estos procesadores tienen cuatro modos:

Modo	Introducido	Descripción
Real	Intel 8086	Sin protección entre tareas
Protegido	Intel 80286	Con protección de tareas, 32 bits
Virtual x86	Intel 80386	Permite correr tareas sin protección en modo protegido
64 bit	AMD Opteron	Con protección de tareas, 64 bits

SO

Lenguaje Assembler

Los registros de uso general de la arquitectura x86-64:



Lenguaje Assembler

Convenciones: Hay dos formatos de código assembler:

1) AT&T / GNU Assembler Syntax:

Se la distingue porque los nombres de los registros están precedidos por el signo '%' y los valores inmediatos por '$'. Los parámetros van de izquierda a derecha:

```
MOV     %EAX, %EBX          # Equivale a EBX=EAX
SUB     $0x10,%RSP          # Equivale a RSP=RSP-0x10
```

2) INTEL:

Los parámetros van de derecha a izquierda:

```
MOV     EBX, EAX            # Equivale a EBX=EAX
SUB     RSP,0x10            # Equivale a RSP=RSP-0x10
```

En la materia vamos a usar la notación de INTEL

Lenguaje Assembler

Movimiento de datos entre registros:

```
MOV     RAX,0x200
MOV     RBX,RAX
MOV     RBP,0x12000
MOV     QWORD PTR [RBP],RBX
```

Operaciones matemáticas fundamentales con enteros:

```
ADD     RAX,0x12
MOV     RBP,0x11000
SUB     DWORD PTR [RBP],0x100
MOV     RAX,0x6
MOV     RCX,0x4
MUL     RCX             # RDX:RAX = 6 . 4
SIGNO
DIV     RCX             # RAX = (RDX:RAX DIV RCX)
                        # RDX = (RDX:RAX MOD RCX)
```

SIN

Lenguaje Assembler

Operaciones matemáticas fundamentales con enteros:

```
MOV     RAX,0x6
MOV     RCX,-0x4
IMUL    RCX                 # RDX:RAX = 6 . (-4)
IDIV    RCX                 # RAX = (RDX:RAX DIV RCX)
                            # RDX = (RDX:RAX MOD RCX)
```

CON SIGNO

Operaciones con bits:

```
AND     RAX,RBX             # RAX = RAX AND RBX
OR      RAX,0xFF            # RAX = RAX OR 0xFF
MOV     RBP,0x100
XOR     AX,WORD PTR [RBP]       # AX = AX XOR [0x100]
NOT     RAX                 # RAX = NOT RAX
```

Lenguaje Assembler

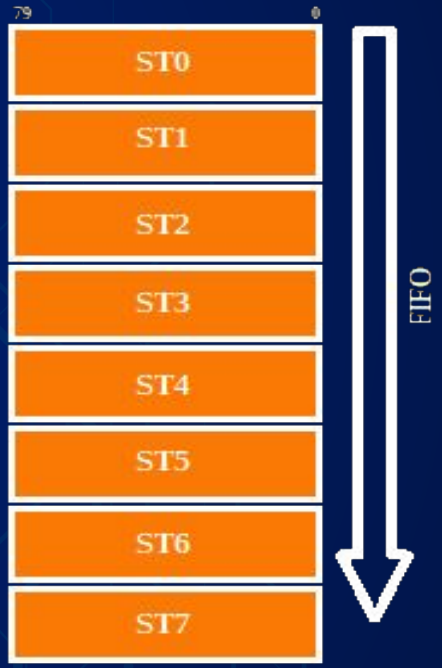
Rotaciones y desplazamiento de bits:

```
MOV     AX,0b10100101
MOV     BX,0b00010100
SHL     AX,1                # AX = 0b01001010, C=1
RCL     BX                  # BX = 0b00101001, C=0

SHR     EAX                 # Ingresa cero a la izquierda

SAR     EAX                 # Preserva signo

ROR     EAX                 # Rota sin considerar C
```

Lenguaje Assembler

Los registros del coprocesador matemático:

- El coprocesador (inicialmente 8087) se crea en 1980 como elemento opcional.
- En 1989 pasa a formar parte del procesador.
- No son de acceso aleatorio, es una pila de 8 registros de 80 bits cada uno.



Lenguaje Assembler

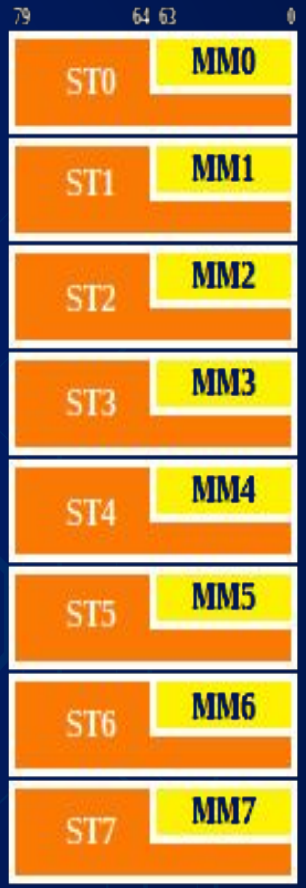
Operaciones matemáticas con los registros del 8087:

```
    FLD     TBYTE PTR [RBP+0]       # Cargo ST(0)
    FLD     TBYTE PTR [RBP+10]    # ST(0) pasa a ST(1), cargo ST(0)
    FDIVP   ST(1),ST(0)           # ST(1) = ST(1) / ST(0)

    FLD     TBYTE PTR [RBP+0]       # Cargo ST(0)
    FLD     TBYTE PTR [RBP+10]    # ST(0) pasa a ST(1), cargo ST(0)
    FMULP   ST(1),ST(0)           # ST(1) = ST(1) * ST(0), luego
                                  # Descarto ST(0), ST(n) = ST(n+1)

    FLD     QWORD PTR [RBP+0]       # Cargo ST(0)
    FLDD    QWORD PTR [RBP+8]       # ST(0) pasa a ST(1), cargo
ST(0)
    FADDP   ST(1),ST(0)           # ST(1) = ST(1) + ST(0), luego
                                  # Descarto ST(0), ST(n) = ST(n+1)

    FSQRT   ST(0)                 # ST(0) = raíz cuadrada de ST(0)
    FSTP    TBYTE PTR [RBP+0]       # Guardo ST(0) y lo
descarto.
```

Lenguaje Assembler

MMX, el primer intento de SIMD:

- Se comparte parte de cada registro con los ST(x) del coprocesador matemático.
- Ahora se los puede acceder en forma aleatoria, no como un stack.
- Se podía acceder a los registros como:
  1) 8 operaciones de 1 byte en paralelo
  2) 4 operaciones de 1 word en paralelo
  3) 2 operaciones de 1 doubleword en paralelo
- Define operaciones de suma, resta y algunas de multiplicación (solo 16 bits), lógicas, shift y para comparación.
- Dos problemas:
  1) Comparte registros, no se pueden usar a la vez.
  2) Sólo trabaja con enteros (con y sin signo)



Lenguaje Assembler

Uso de los registros MMX:

```
MOV     RBP,0x3400
MOVQ    MM0,QWORD PTR [RBP+0]     # Cargar MM0
MOVQ    MM1,QWORD PTR [RBP+8]     # Cargar MM1
PADDW   MM0,MM1                   # Sumar 4 bloques de 16 bits
MOVQ    QWORD PTR [RBP+16],MM0    # Guardarlos
```

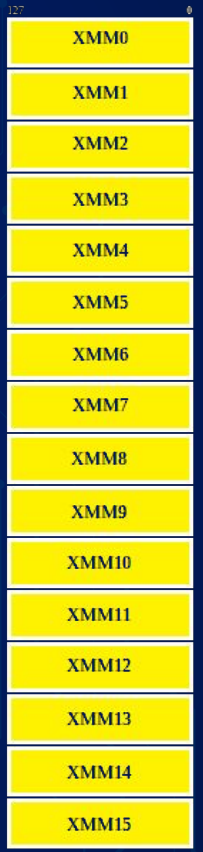
Para hacer los mismo sin MMX debería repetir 4 veces

```
MOV     RBP,0x3400
MOV     AX,WORD PTR [RBP+0]
MOV     BX,WORD PTR [RBP+8]
ADD     AX,BX
MOV     WORD PTR [RBP+16],AX

...
```

Lenguaje Assembler

SSE, SSE2, SSE3 y SSE4 vienen a resolver el problema:

- Se agregan 16 registros de 128 bits cada uno.
- Se definen operaciones de FP / INT.
- Se puede acceder a los registros como:
  1) 16 operaciones de INT8
  2) 8 operaciones de INT16
  3) 4 operaciones de FP32 (single) / INT32
  4) 2 operaciones de FP64 (double) / INT64



Lenguaje Assembler

Uso de los registros SSEx:

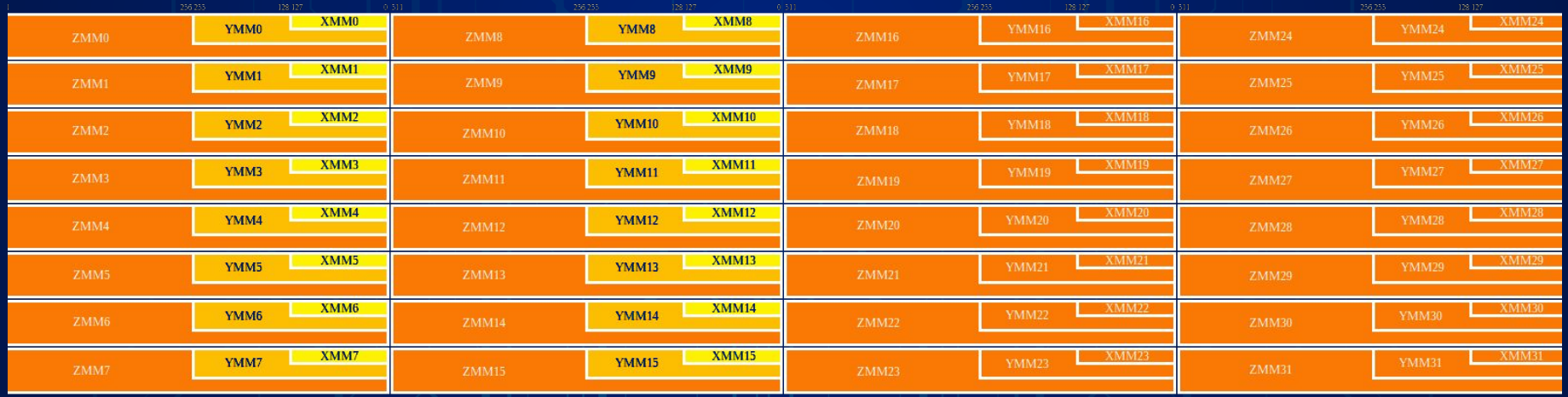
```
    MOV     RBP,0x3400
    MOVAPS  MM0,DWORD PTR [RBP]       # Levantar 4 FP32 a MM0
    MOVAPS  MM1,DWORD PTR [RBP+16]    # Levantar 4 FP32 a MM1
    ADDPS   MM0,MM1                   #  Sumar  los  8  FP32  de  a
pares
    MOVAPS  DWORD PTR [RBP],MM0       # Guardar las 4 sumas
```

Hacer ésto sin coprocesador llevaría cientos de instrucciones. Aún con el coprocesador original, para sumar 4 FP32 habría que repetir 4 veces:

```
    MOV     RBP,0x3400
    FLD     DWORD PTR [RBP]       # Cargo 1 FP32 en ST(0)
    FLDD    DWORD PTR [RBP+16]    # ST(0) pasa a ST(1), cargo ST(0)
    FADDP   ST(1),ST(0)           # ST(1) = ST(1) + ST(0), luego
                                  # Descarto ST(0), ST(n) = ST(n+1)
    FSTP    DWORD PTR [RBP]       # Guardo ST(0) y lo descarto.
```

Lenguaje Assembler

Las extensiones AVX y AVX2 extienden los registros:



Fin

¿Preguntas?