

Lenguaje de máquina: introducción

Organización del computador - FIUBA

2.^{do} cuatrimestre de 2023

Última modificación: Mon Oct 2 15:30:55 2023 -0300

Créditos

Para armar las presentaciones del curso utilizamos:



R. E. Bryant and D. R. O'Hallaron, *Computer systems: a programmer's perspective*, Third edition, Global edition. Boston Columbus Hoboken Indianapolis New York San Francisco Cape Town: Pearson, 2015.



D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, RISC-V edition. Cambridge, Massachusetts: Morgan Kaufmann Publishers, an imprint of Elsevier, 2017.



J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. 2017.

El contenido de los slides está basado en las presentaciones de Patricio Moreno y de Organización del Computador I - FCEN.

Tabla de contenidos

1. Arquitecturas

Instruction Set Architecture (ISA)

Historia (oficial) de los procesadores

2. Básicos de assembly: registros, operandos, instrucciones

Definiciones

Operandos

Addressing

Movimiento de datos

3. Operaciones aritméticas y lógicas

4. C, assembly, código de máquina

Tabla de contenidos

1. Arquitecturas

Instruction Set Architecture (ISA)

Historia (oficial) de los procesadores

2. Básicos de assembly: registros, operandos, instrucciones

Definiciones

Operandos

Addressing

Movimiento de datos

3. Operaciones aritméticas y lógicas

4. C, assembly, código de máquina

Tabla de contenidos

1. Arquitecturas

Instruction Set Architecture (ISA)

Historia (oficial) de los procesadores

2. Básicos de assembly: registros, operandos, instrucciones

Definiciones

Operandos

Addressing

Movimiento de datos

3. Operaciones aritméticas y lógicas

4. C, assembly, código de máquina

Hardware vs Software

- Arquitectura de Software o *Instruction Set Architecture (ISA)*
 - contiene todos los aspectos de diseño visibles para un desarrollador de software
 - también llamada **Arquitectura**
- Arquitectura de Hardware o *Microarquitectura*
 - refiere a una implementación específica de la ISA
 - cantidad de núcleos, frecuencia, instrucciones, etc.
 - las distintas arquitecturas de hardware para una determinada ISA se llaman *familia*

Hardware vs Software

- La separación entre **arquitectura** y **microarquitectura**
 - da garantías de compatibilidad (hacia atrás)
 - permite actualizar el *hardware* sin afectar el *software* (respetando la ISA)
- Por la retrocompatibilidad
 - *software* **viejo** puede correr en *hardware* **nuevo**
- Por la actualización del *hardware*
 - *software* **nuevo** puede no correr en *hardware* **viejo**

Partes de la arquitectura de software

La arquitectura cuenta de 4 partes

- **Set de Instrucciones**
 - conjunto de instrucciones disponibles en el procesador y las reglas para utilizarlas
- **Organización de registros**
 - cantidad, tamaño y reglas para su uso
- **Organización de la memoria y direccionamiento**
- **Modos de operación**
 - modos de operación del procesador (modo *user* y modo *system*)

Set de instrucciones

El set de instrucciones define

- la cantidad de instrucciones disponibles
- tipo de las instrucciones
 - RISC / CISC
- formatos
 - reglas de uso
- ancho del *datapath*
 - ancho del bus de datos
 - tamaño en bits de los registros
 - capacidad de memoria (memoria direccionable)

Set de instrucciones

Clasificación de las instrucciones

- aritméticas
 - operan con enteros
- lógicas
- relacionales
- control
 - pueden cambiar el flujo de ejecución
- punto flotante
 - operan con flotantes
- transferencia de datos
- desplazamientos
- manipulación de bits
- sistema

Clasificación de Arquitecturas (ISA)

Por características de las instrucciones

- *CISC – Complex Instruction Set Computers*
 - instrucciones complejas (que representan bien el código escrito)
 - *instrucciones de largo variable*
 - modos de direccionamiento variable
 - instrucciones muy específicas
 - tiende a ser lento
- *RISC – Reduced Instruction Set Computers*
 - instrucciones simples (y veloces)
 - *instrucciones de largo fijo*
 - modos de direccionamiento simples
 - registros de propósito general
 - tiende a ser veloz, ocupando más memoria

Clasificación de Arquitecturas (ISA)

Por características de las memorias

- *von Neumann*
 - la memoria de datos y programa está unificada
 - requiere una única memoria
 - permite modificar el código
 - código y datos comparten el mismo bus
 - uso más general
- *Harvard*
 - la memoria de datos y programa está separada
 - requiere memorias separadas físicamente
 - mayores velocidades
 - uso típico: DSPs y microcontroladores (*firmware* + datos)

Microprocesador vs. Microcontrolador

- **Microprocesador (μ P, uP)**
 - propósito general
 - rendimiento por velocidad, paralelismo, etc.
 - sin periféricos incluidos (*on-chip*)
 - la potencia “no” es un problema
- **Microcontrolador (μ C, uC)**
 - propósitos específicos
 - rendimiento “moderado”
 - (posiblemente) una gran cantidad de periféricos incluidos
 - eficiente en términos de potencia

Tabla de contenidos

1. Arquitecturas

Instruction Set Architecture (ISA)

Historia (oficial) de los procesadores

2. Básicos de assembly: registros, operandos, instrucciones

Definiciones

Operandos

Addressing

Movimiento de datos

3. Operaciones aritméticas y lógicas

4. C, assembly, código de máquina

Procesadores x86 de Intel

- Dominan los mercados de escritorio, notebooks y servidores
 - No dominan los sistemas embebidos (incluye telefonía)
- Diseño “evolucionado”
 - Es compatible hacia atrás (8086 @ 1978)
 - Se fue extendiendo y agregando *features*
 - Hoy: documentación \approx 5000 páginas (4 volúmenes)
- *Complex Instruction Set Computer (CISC)*
 - Muchas instrucciones diferentes con formatos distintos
 - Sólo un subconjunto de ellas se usa en GNU/Linux
 - Difícil alcanzar el rendimiento de las arquitecturas RISC
 - Intel lo hizo bastante bien en velocidad, en potencia no tanto

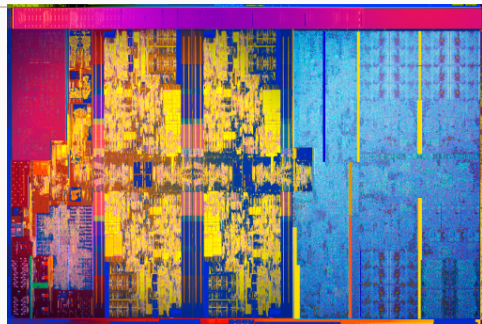
Evolución de Intel x86

| μ P | Fecha | #Transistores | f [MHz] | Litho [nm] |
|---|-------|---------------|-----------|------------|
| 8086 | 1978 | 29K | 5–10 | 3000 |
| <ul style="list-style-type: none"> • Primer procesador de Intel de 16 bits • 1 MB de memoria | | | | |
| 386 | 1985 | 275K | 16–33 | 1500 |
| <ul style="list-style-type: none"> • Primer procesador de Intel de 32 bits • Cambios en la memoria, puede correr Unix | | | | |
| Pentium 4E | 2004 | 125M | 2800–3800 | 90 |
| <ul style="list-style-type: none"> • Primer procesador de Intel de 64 bits de la familia x86 (llamado x86-64) | | | | |
| Core 2 | 2006 | 291M | 1060–3500 | 65 |
| <ul style="list-style-type: none"> • Primer procesador de Intel multi núcleo | | | | |
| Core i7 | 2008 | 731M | 1700–3900 | 45 |
| <ul style="list-style-type: none"> • Procesador de 4 núcleos | | | | |
| Core i9 | 2017 | — | 3300–4300 | 14 |
| <ul style="list-style-type: none"> • Procesador de 10 núcleos | | | | |

Evolución de Intel x86

Resumen

| | | |
|---------------|------|------|
| • 386 | 1985 | 0.3M |
| • Pentium | 1993 | 3.1M |
| • Pentium/MMX | 1997 | 4.5M |
| • PentiumPro | 1995 | 6.5M |
| • Pentium III | 1999 | 8.2M |
| • Pentium 4 | 2001 | 42M |
| • Core 2 Duo | 2006 | 291M |
| • Core i7 | 2008 | 731M |



Intel Core i7-8550U (Kaby Lake 2017)

Características agregadas

- Instrucciones multimedia
- Punto Flotante
- Branch Predictors
- Transición de 8, 16, 32 a 64 bits
- Más núcleos
- Más memoria caché
- Procesadores gráficos
- Periféricos

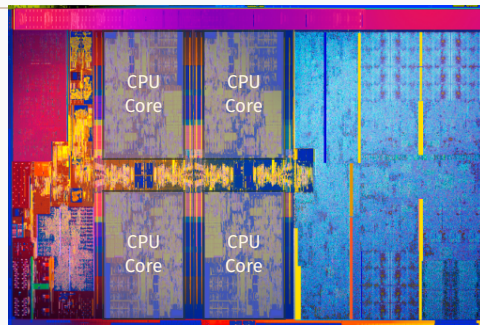
Evolución de Intel x86

Resumen

| | | |
|---------------|------|------|
| • 386 | 1985 | 0.3M |
| • Pentium | 1993 | 3.1M |
| • Pentium/MMX | 1997 | 4.5M |
| • PentiumPro | 1995 | 6.5M |
| • Pentium III | 1999 | 8.2M |
| • Pentium 4 | 2001 | 42M |
| • Core 2 Duo | 2006 | 291M |
| • Core i7 | 2008 | 731M |

Características agregadas

- Instrucciones multimedia
- Punto Flotante
- Branch Predictors
- Transición de 8, 16, 32 a 64 bits



Intel Core i7-8550U (Kaby Lake 2017)

- Más núcleos
- Más memoria caché
- Procesadores gráficos
- Periféricos

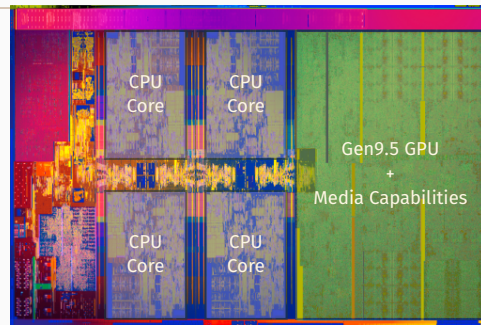
Evolución de Intel x86

Resumen

| | | |
|---------------|------|------|
| • 386 | 1985 | 0.3M |
| • Pentium | 1993 | 3.1M |
| • Pentium/MMX | 1997 | 4.5M |
| • PentiumPro | 1995 | 6.5M |
| • Pentium III | 1999 | 8.2M |
| • Pentium 4 | 2001 | 42M |
| • Core 2 Duo | 2006 | 291M |
| • Core i7 | 2008 | 731M |

Características agregadas

- Instrucciones multimedia
- Punto Flotante
- Branch Predictors
- Transición de 8, 16, 32 a 64 bits



Intel Core i7-8550U (Kaby Lake 2017)

- Más núcleos
- Más memoria caché
- Procesadores gráficos
- Periféricos

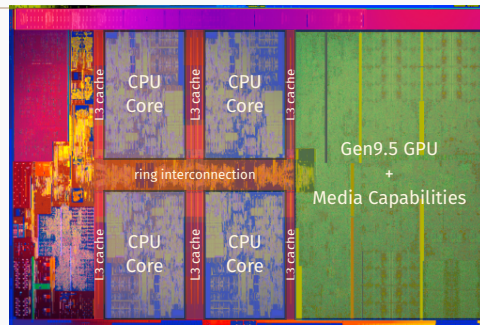
Evolución de Intel x86

Resumen

| | | |
|---------------|------|------|
| • 386 | 1985 | 0.3M |
| • Pentium | 1993 | 3.1M |
| • Pentium/MMX | 1997 | 4.5M |
| • PentiumPro | 1995 | 6.5M |
| • Pentium III | 1999 | 8.2M |
| • Pentium 4 | 2001 | 42M |
| • Core 2 Duo | 2006 | 291M |
| • Core i7 | 2008 | 731M |

Características agregadas

- Instrucciones multimedia
- Punto Flotante
- Branch Predictors
- Transición de 8, 16, 32 a 64 bits



Intel Core i7-8550U (Kaby Lake 2017)

- Más núcleos
- Más memoria caché
- Procesadores gráficos
- Periféricos

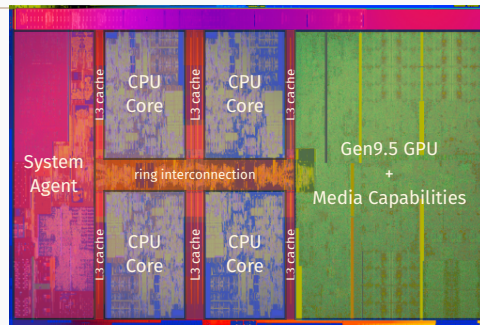
Evolución de Intel x86

Resumen

| | | |
|---------------|------|------|
| • 386 | 1985 | 0.3M |
| • Pentium | 1993 | 3.1M |
| • Pentium/MMX | 1997 | 4.5M |
| • PentiumPro | 1995 | 6.5M |
| • Pentium III | 1999 | 8.2M |
| • Pentium 4 | 2001 | 42M |
| • Core 2 Duo | 2006 | 291M |
| • Core i7 | 2008 | 731M |

Características agregadas

- Instrucciones multimedia
- Punto Flotante
- Branch Predictors
- Transición de 8, 16, 32 a 64 bits



Intel Core i7-8550U (Kaby Lake 2017)

- Más núcleos
- Más memoria caché
- Procesadores gráficos
- Periféricos

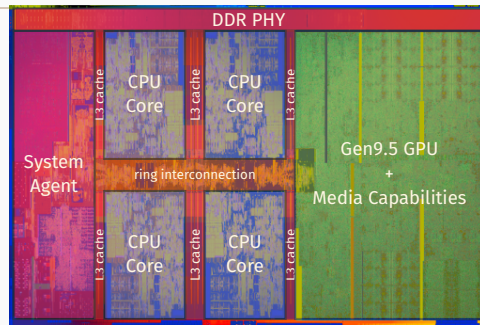
Evolución de Intel x86

Resumen

| | | |
|---------------|------|------|
| • 386 | 1985 | 0.3M |
| • Pentium | 1993 | 3.1M |
| • Pentium/MMX | 1997 | 4.5M |
| • PentiumPro | 1995 | 6.5M |
| • Pentium III | 1999 | 8.2M |
| • Pentium 4 | 2001 | 42M |
| • Core 2 Duo | 2006 | 291M |
| • Core i7 | 2008 | 731M |

Características agregadas

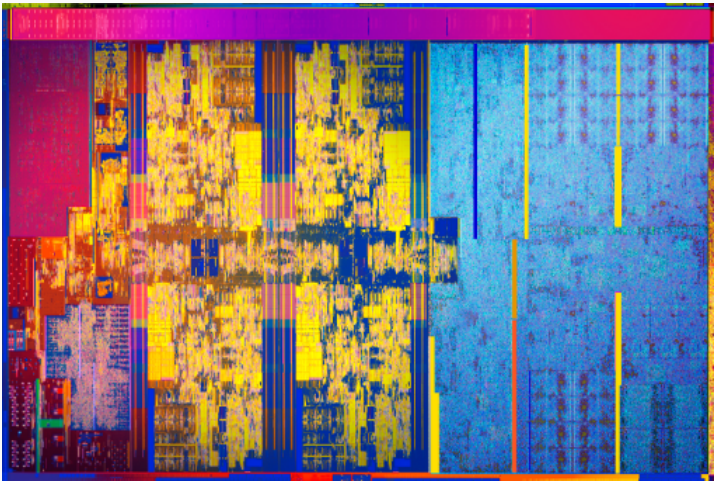
- Instrucciones multimedia
- Punto Flotante
- Branch Predictors
- Transición de 8, 16, 32 a 64 bits



Intel Core i7-8550U (Kaby Lake 2017)

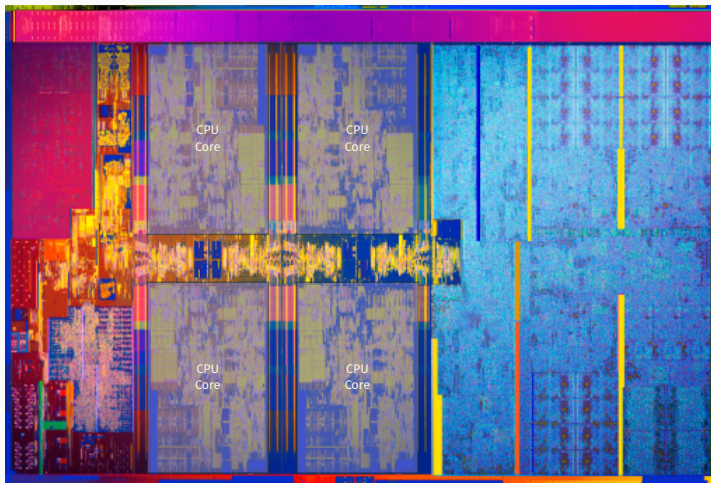
- Más núcleos
- Más memoria caché
- Procesadores gráficos
- Periféricos

Imagen: Kaby Lake Die



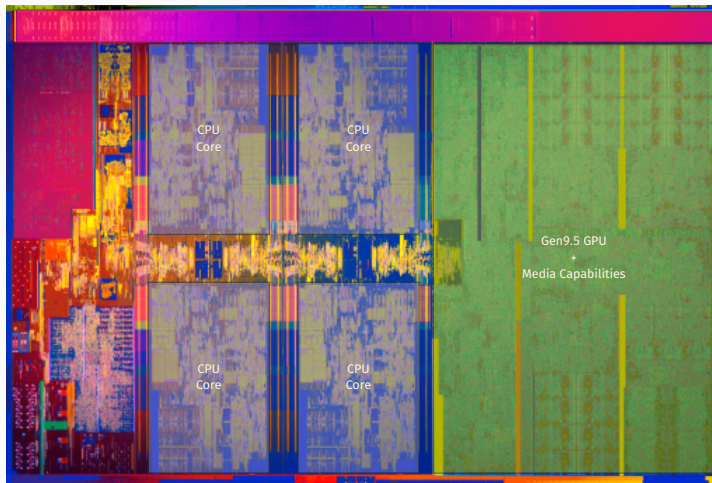
Área: 123 mm²

Imagen: Kaby Lake Die



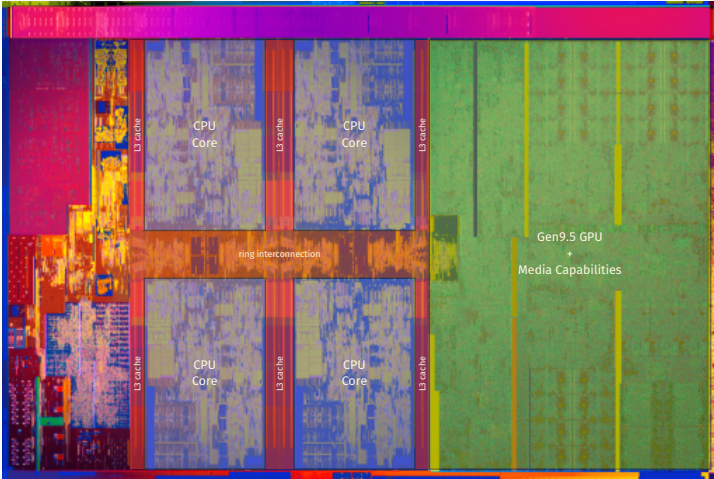
Área: 123 mm²

Imagen: Kaby Lake Die



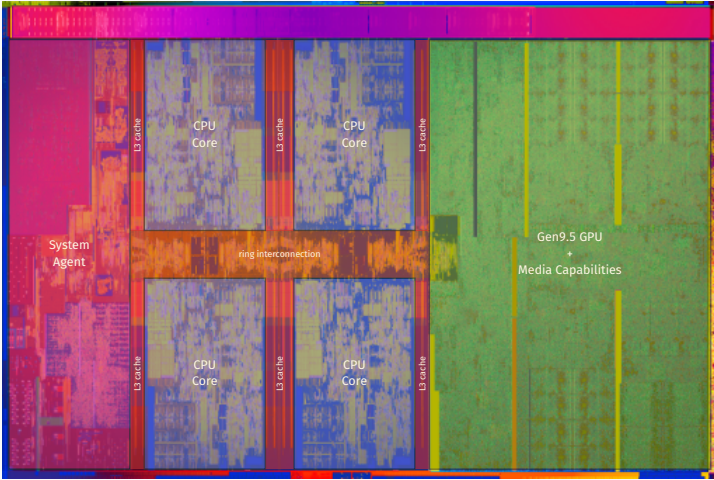
Área: 123 mm²

Imagen: Kaby Lake Die



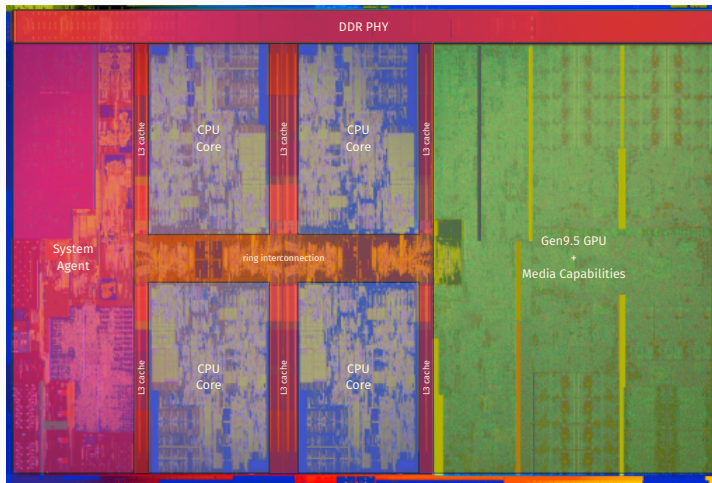
Área: 123 mm²

Imagen: Kaby Lake Die



Área: 123 mm²

Imagen: Kaby Lake Die



Área: 123 mm²

Clones de x86: Advanced Micro Devices (AMD)

- **Pre-2003**
 - AMD siempre está un paso atrás de Intel
 - siempre es un poquito más lento, pero mucho más barato
- **2003**
 - AMD contrata diseñadores de *Digital Equipment Corp. (DEC)* y otras empresas en desgracia
 - Construyen Opteron: compite fuerte contra Pentium 4
 - Lanzas AMD64 (x86-64) antes que Intel, como extensión de x86
- **Últimamente**
 - AMD e Intel se autoproclaman líderes
 - ARM domina en sistemas embebidos
 - Arquitecturas RISC-V empiezan a pesar

Los 64 bits de Intel

- **2001: experimentos frustrados**
 - Intel domina el mercado
 - Diseña una arquitectura completamente distinta a x86 (Itanium Processors—EPIC)
 - Resulta ser un fiasco (en el mercado)
- **2003: AMD extiende x86 a x86-64 (AMD64)**
 - Intel quiere continuar con IA64
 - pero termina admitiendo que AMD64 es mejor
- **2004: Intel anuncia EM64T**
 - Extensión de 64 bits para IA32
 - casi igual a x86-64
- **Hoy: (prácticamente) todos los procesadores soportan x86-64**
 - pero sigue habiendo mucho código que corre en modo de 32 bits.

Vamos a ver...

- **x86-64**
 - El estándar *de facto*
- **ARM**
 - queremos (a futuro)
- **RISC-V**
 - ídem ARM
- **Bibliografía**
 - el libro de Bryant & O'Hallaron: x86-64 (la versión vieja es en x86)
 - otros 3 libros en MIPS, RISC-V y ARM (no incluidos todavía)

Tabla de contenidos

1. Arquitecturas

Instruction Set Architecture (ISA)

Historia (oficial) de los procesadores

2. Básicos de assembly: registros, operandos, instrucciones

Definiciones

Operandos

Addressing

Movimiento de datos

3. Operaciones aritméticas y lógicas

4. C, assembly, código de máquina

Definiciones (*again*)

- **Arquitectura:** la parte del diseño del procesador que van a necesitar saber para escribir o entender assembly/código de máquina
- **Microarquitectura:** implementación de la arquitectura
- **Formas del código fuente:**
 - *Código de máquina:* los programas que el procesador ejecuta, a nivel de bytes, bits (binario)
 - *Código assembly:* una representación en formato texto del código de máquina
- **Ejemplos de ISAs:**
 - Intel: x86, IA32, Itanium, x86-64.
 - ARM: armv7, etc. Utilizada en la mayor parte de los teléfonos.
 - AMD64: K6, K6-2 (SIMD), K8 (Opteron), Bulldozer, Zen
 - RISC-V: nvidia (futuro), western digital (futuro)

Tabla de contenidos

1. Arquitecturas

Instruction Set Architecture (ISA)

Historia (oficial) de los procesadores

2. Básicos de assembly: registros, operandos, instrucciones

Definiciones

Operandos

Addressing

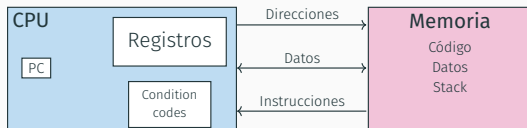
Movimiento de datos

3. Operaciones aritméticas y lógicas

4. C, assembly, código de máquina

Sobre la CPU: ¿qué vemos?

- **PC: Program Counter**
 - Guarda la dirección de la próxima instrucción
 - Nombre: **rip** en x86-64
- **Condition Codes**
 - Información sobre la última operación aritmética
 - Se usan para saltos condicionales
- **Memoria**
 - Es un arreglo direccionable
 - Guarda código y datos de usuario
 - Mantiene el stack (soporte para procedimientos)
- **Registros**
 - Almacenan datos que el programa está usando



Tipos de datos

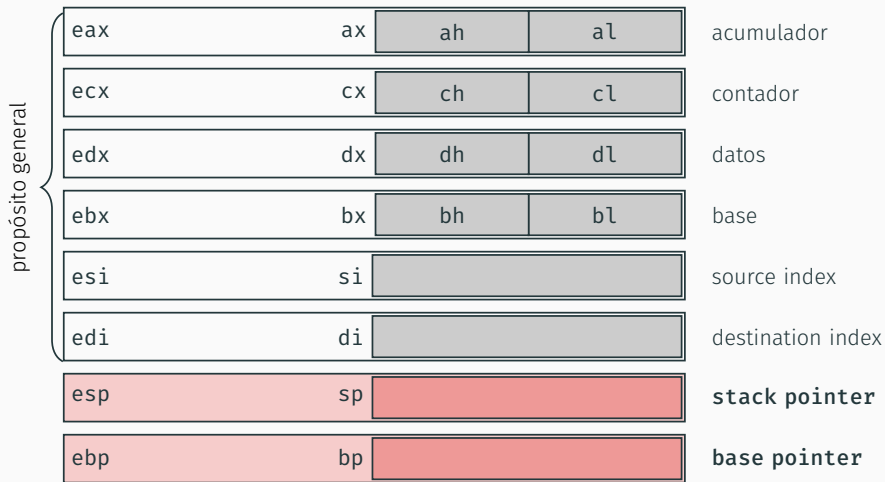
- Datos “enteros” de 1, 2, 4, u 8 bytes
 - Valores de datos en general
 - Direcciones (punteros *no tipados*)
- Datos en punto flotante de 4, 8, o 10 bytes
- Tipos de datos vectoriales (SIMD) de 8, 16, 32, o 64 bytes
- Código: secuencias de bytes que codifican una serie de instrucciones
- No hay tipos por agregación como arreglos o estructuras
 - Se consideran bytes contiguos en memoria

Registros x86-64 (enteros)

| | |
|-----|-----|
| rax | eax |
| rbx | ebx |
| rcx | ecx |
| rdx | edx |
| rsi | esi |
| rdi | edi |
| rsp | esp |
| rbp | ebp |

| | |
|-----|------|
| r8 | r8d |
| r9 | r9d |
| r10 | r10d |
| r11 | r11d |
| r12 | r12d |
| r13 | r13d |
| r14 | r14d |
| r15 | r15d |

Registros IA32



Operaciones

- Transferir datos entre la memoria y registros
 - Cargar datos de la memoria a un registro
 - Guardar datos de un registro en la memoria
- Realizar operaciones aritméticas con registros o datos de memoria
- Transferencia del control
 - Saltos incondicionales hacia o desde procedimientos
 - Saltos condicionales
 - Saltos indirectos

Tabla de contenidos

1. Arquitecturas

Instruction Set Architecture (ISA)

Historia (oficial) de los procesadores

2. Básicos de assembly: registros, operandos, instrucciones

Definiciones

Operandos

Addressing

Movimiento de datos

3. Operaciones aritméticas y lógicas

4. C, assembly, código de máquina

Operandos: ejemplo con mov

- Instrucción:
`mov dest, qword source`
- Operandos:
 - **Inmediato:** Constante entera
 - Ejemplo: `0x400`, `-533`
 - Como en C...
 - Codificado en 1, 2 o 4 bytes
 - **Registro:** uno de los registros
 - Ejemplo: `rax`, `r13`
 - `rsp` reservado
 - los demás pueden tener usos especiales
 - **Memoria:** 8 bytes consecutivos en una posición de memoria dada por un registro (el modificador es opcional cuando el tamaño del dato a mover se puede inferir de los operandos)
 - Ejemplo más simple: `(rax)`
 - Hay varios modos más de “direccionamiento”

rax

rbx

rcx

rdx

rsi

rdi

rsp

rbp

rN

Operandos: ejemplo con mov

- Instrucción:
mov dest, **qword** source
- Operandos:
 - **Inmediato:** Constante entera
 - Ejemplo: 0x400, -533
 - Como en C...
 - Codificado en 1, 2 o 4 bytes
 - **Registro:** uno de los registros
 - Ejemplo: rax, r13
 - **rsp** reservado
 - los demás pueden tener usos especiales
 - **Memoria:** **8** bytes consecutivos en una posición de memoria dada por un registro (el modificador es opcional cuando el tamaño del dato a mover se puede inferir de los operandos)
 - Ejemplo más simple: (rax)
 - Hay varios modos más de “direccionamiento”

| |
|-----|
| rax |
| rbx |
| rcx |
| rdx |
| rsi |
| rdi |
| rsp |
| rbp |

Combinación de operandos: mov

| | Origen | Destino | dst, src | C |
|-------|--------|---------|-----------------|----------------|
| mov { | Inm. | Reg. | mov rax, 0x4 | temp = 0x4; |
| | | Mem. | mov [rdx], -147 | *p = -147; |
| | Reg. | Reg. | mov rdx, rax | temp2 = temp1; |
| | | Mem. | mov [rdx], rax | *p = temp; |
| | Mem. | Reg. | mov rdx, [rax] | temp = *p; |

No se pueden hacer transferencias de memoria a memoria en una única instrucción

Modos de direccionamiento simple

- Normal $[R]$ $\text{Mem}[\text{Reg}[R]]$
 - El registro R especifica la dirección de memoria
 - ¡ R funciona como un puntero!
- Corrimiento $[R + D]$ $\text{Mem}[\text{Reg}[R] + D]$
 - El registro R especifica el comienzo de una región de memoria
 - El corrimiento D especifica el *offset*

Modos de direccionamiento simple

- Normal $[R]$ $\text{Mem}[\text{Reg}[R]]$
 - El registro R especifica la dirección de memoria
 - ¡ R funciona como un puntero!

`mov [rcx], rax`

- Corrimiento $[R + D]$ $\text{Mem}[\text{Reg}[R] + D]$
 - El registro R especifica el comienzo de una región de memoria
 - El corrimiento D especifica el *offset*

Modos de direccionamiento simple

- Normal $[R]$ $\text{Mem}[\text{Reg}[R]]$
 - El registro R especifica la dirección de memoria
 - ¡ R funciona como un puntero!

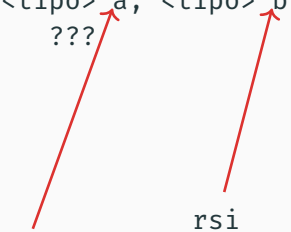
`mov [rcx], rax`

- Corrimiento $[R + D]$ $\text{Mem}[\text{Reg}[R] + D]$
 - El registro R especifica el comienzo de una región de memoria
 - El corrimiento D especifica el *offset*

`mov [rbp+8], rdx`

Ejemplo: direccionamiento simple

```
void  
misterio(<tipo> a, <tipo> b)  
    ???  
    rdi  
    rsi
```



```
misterio:  
    mov rax, [rdi]  
    mov rdx, [rsi]  
    mov [rdi], rdx  
    mov [rsi], rax  
    ret
```

Ejemplo: direccionamiento simple

```
1 void swap(long *xp, long *yp) {  
2     long t0 = *xp;  
3     long t1 = *yp;  
4  
5     *xp = t1;  
6     *yp = t0;  
7 }
```

swap:

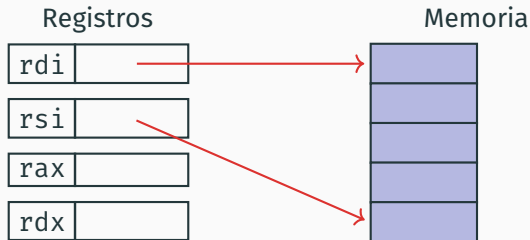
```
mov rax, [rdi]  
mov rdx, [rsi]  
mov [rdi], rdx  
mov [rsi], rax  
ret
```


Ejemplo: comprendiendo swap()

```

1 void swap(long *xp, long *yp) {
2     long t0 = *xp;
3     long t1 = *yp;
4
5     *xp = t1;
6     *yp = t0;
7 }

```



| Registro | Valor |
|----------|-------|
| rdi | xp |
| rsi | yp |
| rax | t0 |
| rdx | t1 |

```

1 swap:
2     mov     rax, [rdi]    # t0 = *xp
3     mov     rdx, [rsi]    # t1 = *yp
4     mov     [rdi], rdx    # *xp = t1
5     mov     [rsi], rax    # *yp = t0
6     ret

```

Ejemplo: comprendiendo swap()

| | |
|-----|-------|
| rdi | 0x120 |
| rsi | 0x100 |
| rax | |
| rdx | |

| | |
|-----|-------|
| 123 | 0x120 |
| | 0x118 |
| | 0x110 |
| | 0x108 |
| 456 | 0x100 |

swap:

```

mov rax, [rdi] # t0 = *xp
mov rdx, [rsi] # t1 = *yp
mov [rdi], rdx # *xp = t1
mov [rsi], rax # *yp = t0
ret

```

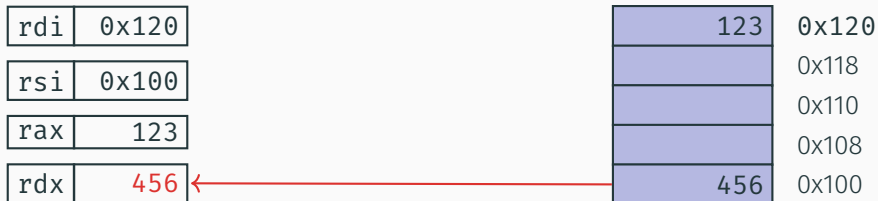
Ejemplo: comprendiendo swap()



swap:

```
mov rax, [rdi] # t0 = *xp
mov rdx, [rsi] # t1 = *yp
mov [rdi], rdx # *xp = t1
mov [rsi], rax # *yp = t0
ret
```

Ejemplo: comprendiendo swap()



swap:

```
mov rax, [rdi] # t0 = *xp
mov rdx, [rsi] # t1 = *yp
mov [rdi], rdx # *xp = t1
mov [rsi], rax # *yp = t0
ret
```

Ejemplo: comprendiendo swap()



swap:

```
mov rax, [rdi] # t0 = *xp
mov rdx, [rsi] # t1 = *yp
mov [rdi], rdx # *xp = t1
mov [rsi], rax # *yp = t0
ret
```

Ejemplo: comprendiendo swap()



swap:

```
mov rax, [rdi] # t0 = *xp
mov rdx, [rsi] # t1 = *yp
mov [rdi], rdx # *xp = t1
mov [rsi], rax # *yp = t0
ret
```

Tabla de contenidos

1. Arquitecturas

Instruction Set Architecture (ISA)

Historia (oficial) de los procesadores

2. Básicos de assembly: registros, operandos, instrucciones

Definiciones

Operandos

Addressing

Movimiento de datos

3. Operaciones aritméticas y lógicas

4. C, assembly, código de máquina

Modos de direccionamiento simple

- Normal $[R]$ $\text{Mem}[\text{Reg}[R]]$
 - El registro R especifica la dirección de memoria
 - ¡ R funciona como un puntero!

`mov [rcx], rax`

- Corrimiento $[R + D]$ $\text{Mem}[\text{Reg}[R] + D]$
 - El registro R especifica el comienzo de una región de memoria
 - El corrimiento D especifica el *offset*

`mov [rbp+8], rdx`

Modos de direccionamiento completos

- Forma general $[Rb + S * Ri + D]$ $Mem[Reg[Rb] + S * Reg[Ri] + D]$
 - D : corrimiento constante
 - Rb : registro base: cualquiera de los 16 registros enteros
 - Ri : registro índice: cualquiera de los registros, *excepto* rsp
 - S : escalado: 1, 2, 4 u 8
- Casos especiales

| | |
|-----------------|------------------------------|
| $[Rb + Ri]$ | $Mem[Reg[Rb] + Reg[Ri]]$ |
| $[Rb + Ri + D]$ | $Mem[Reg[Rb] + Reg[Ri] + D]$ |
| $[Rb + S * Ri]$ | $Mem[Reg[Rb] + S * Reg[Ri]]$ |

Ejemplo: cálculo de direcciones

| | |
|-----|--------|
| rdx | 0xf000 |
| rcx | 0x0100 |

$$[Rb + S * Ri + D]$$

$$Mem[Reg[Rb] + S * Reg[Ri] + D]$$

- **D**: corrimiento constante
- **Rb**: reg. base: cualquiera de los 16 registros enteros
- **Ri**: reg. índice: cualquiera, *excepto* **rsp**
- **S**: escalado: 1, 2, 4 u 8

| Expresión | Cálculo | Dirección |
|--------------------|---------|-----------|
| $[rdx + 0x8]$ | | |
| $[rdx + rcx]$ | | |
| $[rdx + 4 * rcx]$ | | |
| $[2 * rdx + 0x80]$ | | |

Ejemplo: cálculo de direcciones

| | |
|-----|--------|
| rdx | 0xf000 |
| rcx | 0x0100 |

 $[Rb + S * Ri + D]$
 $Mem[Reg[Rb] + S * Reg[Ri] + D]$

- **D**: corrimiento constante
- **Rb**: reg. base: cualquiera de los 16 registros enteros
- **Ri**: reg. índice: cualquiera, *excepto* **rsp**
- **S**: escalado: 1, 2, 4 u 8

| Expresión | Cálculo | Dirección |
|------------------|--------------------|-----------|
| $[rdx + 0x8]$ | $0xf000 + 0x8$ | $0xf008$ |
| $[rdx + rcx]$ | $0xf000 + 0x100$ | $0xf100$ |
| $[rdx + 4*rcx]$ | $0xf000 + 4*0x100$ | $0xf400$ |
| $[2*rdx + 0x80]$ | $2*0xf000 + 0x80$ | $0x1e080$ |

Ejercicio

Asumiendo que están guardados los siguientes valores en sus posiciones de memoria

| Dirección | Valor | Registro | Valor |
|-----------|-------|----------|-------|
| 0x100 | 0xFF | rax | 0x100 |
| 0x104 | 0xAB | rcx | 0x1 |
| 0x108 | 0x13 | rdx | 0x3 |
| 0x10C | 0x11 | | |

Completar la siguiente tabla:

| Operando | Valor |
|-------------------|-------|
| rax | |
| [0x104] | |
| 0x108 | |
| [rax] | |
| [rax+4] | |
| [rax + rdx + 9] | |
| [rcx + rdx + 260] | |
| [4*rcx + 0xFC] | |
| [rax + 4*rdx] | |

Ejercicio

Completar la siguiente tabla:

| Operando | Valor | Comentario |
|--------------------------------|--------------------|-------------------------------|
| <code>rax</code> | <code>0x100</code> | registro |
| <code>[0x104]</code> | <code>0xAB</code> | dirección absoluta |
| <code>0x108</code> | <code>0x108</code> | inmediata |
| <code>[rax]</code> | <code>0xFF</code> | dirección: <code>0x100</code> |
| <code>[rax+4]</code> | <code>0xAB</code> | dirección: <code>0x104</code> |
| <code>[rax + rdx + 9]</code> | <code>0x11</code> | dirección: <code>0x10C</code> |
| <code>[rcx + rdx + 260]</code> | <code>0x13</code> | dirección: <code>0x108</code> |
| <code>[4*rcx + 0xFC]</code> | <code>0xFF</code> | dirección: <code>0x100</code> |
| <code>[rax + 4*rdx]</code> | <code>0x11</code> | dirección: <code>0x10C</code> |

Tabla de contenidos

1. Arquitecturas

Instruction Set Architecture (ISA)

Historia (oficial) de los procesadores

2. Básicos de assembly: registros, operandos, instrucciones

Definiciones

Operandos

Addressing

Movimiento de datos

3. Operaciones aritméticas y lógicas

4. C, assembly, código de máquina

Algunas instrucciones de movimiento

Movimiento simple

| Formato | | Operación |
|-------------|----------|--|
| mov <byte> | src, dst | $\text{dst} \leftarrow \text{src} \text{ (byte)}$ |
| mov <word> | src, dst | $\text{dst} \leftarrow \text{src} \text{ (word)}$ |
| mov <dword> | src, dst | $\text{dst} \leftarrow \text{src} \text{ (double word)}$ |
| mov <qword> | src, dst | $\text{dst} \leftarrow \text{src} \text{ (quad word)}$ |

| | | |
|-----|-------------------------|--------------------------|
| mov | rax, 0x0011223344556677 | rax = 0x0011223344556677 |
| mov | al, -1 | rax = 0x00112233445566FF |
| mov | ax, -1 | rax = 0x001122334455FFFF |
| mov | eax, -1 | rax = 0x00000000FFFFFFFF |
| mov | rax, -1 | rax = 0xFFFFFFFFFFFFFFFF |

Tabla de contenidos

1. Arquitecturas

Instruction Set Architecture (ISA)

Historia (oficial) de los procesadores

2. Básicos de assembly: registros, operandos, instrucciones

Definiciones

Operandos

Addressing

Movimiento de datos

3. Operaciones aritméticas y lógicas

4. C, assembly, código de máquina

Instrucción de cálculo de direcciones

- `lea dst, src`

Load Effective Address

- *src* es una expresión de dirección
- *dst* es el destino de la dirección calculada

- Usos

- Cómputo de direcciones sin referencia a memoria
 - Por ejemplo, la traducción de `p = &x[i]`
- Cómputo de expresiones aritméticas de la forma $x + k*y$
 - $k = 1, 2, 4, \text{ u } 8$

Ejemplo

```

1 long m12(long x)
2 {
3     return x * 12;
4 }
```

Traducido a ASM

```

lea rax, [rdi + 2*rdi], # t = x + 2*x
sal rax,2                # return t << 2
```

Multiplica sin multiplicar

Algunas operaciones aritméticas

Instrucciones con 2 operandos

| Formato | | Operación |
|---------|----------|--|
| add | dst, src | $\text{dst} \leftarrow \text{dst} + \text{src}$ |
| sub | dst, src | $\text{dst} \leftarrow \text{dst} - \text{src}$ |
| imul | dst, src | $\text{dst} \leftarrow \text{dst} * \text{src}$ |
| sal | dst, src | $\text{dst} \leftarrow \text{dst} \ll \text{src}$ |
| sar | dst, src | $\text{dst} \leftarrow \text{dst} \gg \text{src}$ |
| shr | dst, src | $\text{dst} \leftarrow \text{dst} \gg \text{src}$ |
| xor | dst, src | $\text{dst} \leftarrow \text{dst} \wedge \text{src}$ |
| and | dst, src | $\text{dst} \leftarrow \text{dst} \& \text{src}$ |
| or | dst, src | $\text{dst} \leftarrow \text{dst} \mid \text{src}$ |

También llamada shl
Aritmética
Lógica

Más operaciones aritméticas

Instrucciones con 1 operando

| Formato | | Operación |
|---------|-----|--|
| inc | dst | $\text{dst} \leftarrow \text{dst} + 1$ |
| dec | dst | $\text{dst} \leftarrow \text{dst} - 1$ |
| neg | dst | $\text{dst} \leftarrow -\text{dst}$ |
| not | dst | $\text{dst} \leftarrow \sim\text{dst}$ |

Ejemplo: expresiones aritméticas

```

1 long
2 arith (long x, long y, long z)
3 {
4     long t1 = x + y;
5     long t2 = z + t1;
6     long t3 = x + 4;
7     long t4 = y * 48;
8     long t5 = t3 + t4;
9     long rval = t2 * t5;
10    return rval;
11 }

```

```

global arith
section .text
arith:
    lea rax, [rdi + rsi]      ; rax = x + y
    add rax, rdx              ; rax = x + y + z
    lea rcx, [rsi*2 + rsi]    ; rcx = 2*y + y
    sal rcx, 4                ; rcx = (3*y) << 4
    lea rcx, [rdi + rcx + 4]
                              ; rcx = x + y * 48 + 4
    imul rax, rcx
                              ; rax = (x + y + z) * (x + 48* + 4)
    ret

```

Instrucciones

- **lea**: cálculo de direcciones (?)
- **sal**: desplazamiento (*shift*)
- **imul**: multiplicación (1 vez)

Analizando el ejemplo

```

1 long
2 arith (long x, long y, long z)
3 {
4     long t1 = x + y;
5     long t2 = z + t1;
6     long t3 = x + 4;
7     long t4 = y * 48;
8     long t5 = t3 + t4;
9     long rval = t2 * t5;
10    return rval;
11 }

```

Optimizaciones

- reuso de registros
- sustitución
- *strength reduction*

arith:

```

    lea rax, [rdi + rsi]      ; rax = x + y
    add rax, rdx              ; rax = x + y + z
    lea rcx, [rsi*2 + rsi]    ; rcx = 2*y + y
    sal rcx, 4                ; rcx = (3*y) << 4
    lea rcx, [rdi + rcx + 4]  ; rcx = x + 48*y + 4
    imul rax, rcx
    ; rax = (x + y + z) * (x + 48*y + 4)
    ret

```

| Registro | Usos |
|----------|-------------------------------------|
| rdi | Argumento x |
| rsi | Argumento y |
| rdx | Argumento z , t4 |
| rax | t1 , t2 , rval |
| rcx | t5 |

Tabla de contenidos

1. Arquitecturas

Instruction Set Architecture (ISA)

Historia (oficial) de los procesadores

2. Básicos de assembly: registros, operandos, instrucciones

Definiciones

Operandos

Addressing

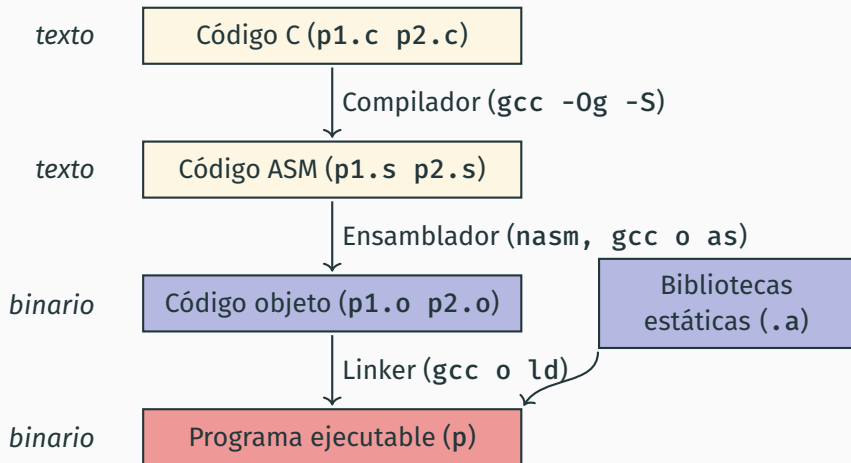
Movimiento de datos

3. Operaciones aritméticas y lógicas

4. C, assembly, código de máquina

Convirtiendo código C en código objeto

- Código en archivos `p1.c` y `p2.c`
- Compilar con el comando: `gcc -Og p1.c p2.c -o p`



Compilando a assembly

Código C (suma.c)

```
1 long plus(long x, long y);  
2  
3 void sumstore(long x, long y, long *dest) {  
4     long t = plus(x, y);  
5     *dest = t;  
6 }
```

Código assembly x86-64

```
sumstore:  
    push    rbx  
    mov     rbx, rdx  
    call    plus  
    mov     [rbx], rax  
    pop     rbx  
    ret
```

Se obtuvo con el comando

```
gcc -S -masm=intel -Og -std=c99 suma.c
```

que produjo el archivo suma.s.

Aviso: se pueden obtener diferentes resultados en otras computadoras debido a diferencias en las versiones de gcc, la configuración, la ISA, etc.

suma.s

```
.file    "suma.c"
.text
.glob    sumstore
.type    sumstore, @function
sumstore:
.LFB0:
.cfi_startproc
pushq    rbx
.cfi_def_cfa_offset 16
.cfi_offset 3, -16
mov      rbx, rdx
call     plus
mov      [rbx], rax
pop      rbx
.cfi_def_cfa_offset 8
ret
.cfi_endproc
.LFE0:
```

suma.s

```

.file    "suma.c"
.text
.glob    sumstore
.type    sumstore, @function
sumstore:
.LFB0:
.cfi_startproc
pushq    rbx
.cfi_def_cfa_offset 16
.cfi_offset 3, -16
mov      rbx, rdx
call     plus
mov      [rbx], rax
pop      rbx
.cfi_def_cfa_offset 8
ret
.cfi_endproc
.LFE0:

```

El texto que comienza con un punto (.) es una directiva.

cfi = call frame information

sumstore:

```

push rbx
mov  rbx, rdx
call plus
mov [rbx], rax
pop  rbx
ret

```

Hay que extraer el código assembly leyendo

Código objeto

0x004004d6:

0x53

0x48

0x89

0xd3

0xe8

0x05

0x00

0x00

0x00

Total: 14 bytes

0x48

Cada instrucción:

0x89

1, 3 o 5 bytes

0x03

0x5b

Comienza en la dirección:

0xc3

0x004004d6

- Ensamblador

- Traduce `.s` a `.o`
- Encoding binario de cada instrucción
- Casi lo mismo que el código ejecutable
- Falta que el *enlazador* llene algunos agujeros

- Enlazador

- Resuelve referencias entre archivos
- Combina las bibliotecas estáticas
- Las de enlazado dinámico se *linkean* al comenzar la ejecución del programa

Ejemplo de instrucción

Código C

```
1 *dest = t;
```

Assembly

```
1 movq [rbx], rax
```

Código Objeto

```
1 0x40059e: 48 89 03
```

- Almacenar el valor **t** donde apunte **dest**

- Mover el valor de 8 bytes a memoria

- Operandos:

| | | |
|---------------|----------|---------------|
| t: | Registro | rax |
| dest: | Registro | rbx |
| *dest: | Memoria | M[rbx] |

- Instrucción de 3 bytes
- Almacenada en la dirección **040059e**

Desensamblando código objeto

```
00000000004004d6 <sumstore>:
  4004d6: 53                push    %rbx
  4004d7: 48 89 d3          mov     %rdx,%rbx
  4004da: e8 05 00 00 00    callq  4004e4 <plus>
  4004df: 48 89 03          mov     %rax,(%rbx)
  4004e2: 5b                pop     %rbx
  4004e3: c3                retq
```

Desensamblador

`objdump -d <objfile>`

- Examina código objeto
- Analiza el patrón de bits de tiras de instrucciones
- Genera una versión aproximada de código assembly
- *Se puede ejecutar con ejecutables completos o compilaciones intermedias (.o)*

Desensamblando código objeto (otra forma)

Dump of assembler code for function sumstore:

```
0x00000000004004d6 <+0>:      push    rbx
0x00000000004004d7 <+1>:      mov     rbx, rdx
0x00000000004004da <+4>:      call   0x4004e4 <plus>
0x00000000004004df <+9>:      mov     [rbx], rax
0x00000000004004e2 <+12>:     pop     rbx
0x00000000004004e3 <+13>:     ret
```

Usando el debugger gdb

`gdb suma`

`disassemble sumstore`

- Es una herramienta más general
- Es más versátil
- *Puede desensamblar el pedacito que le pidamos*

- Imprimir 14 bytes comenzando en `sumstore`

`x/14xb sumstore`

¿Qué se puede desensamblar?

```
% objdump -d WINWORD.EXE
```

```
WINWORD.EXE: file format pei-i386
```

```
No symbols in "WINWORD.EXE".
```

```
Disassembly of section .text:
```

```
30001000 <.text>:
```

```
30001000: 55                push %ebp
30001001: 8b ec            mov %esp, %ebp
30001003: 6a ff ff ff ff   push $0xffffffff
30001005: 68 90 10 00 30    push $0x30001090
3000100a: 68 91 dc 4c 30    push $0x304cdc91
```

Licencia del estilo de beamer

Obtén el código de este estilo y la presentación demo en

`github.com/pamoreno/mtheme`

El estilo *en sí* está licenciado bajo la Creative Commons Attribution-ShareAlike 4.0 International License. El estilo es una modificación del creado por Matthias Vogelgesang, disponible en

`github.com/matze/mtheme`

