

Lenguaje de máquina: introducción

95.57/75.03 Organización del computador

Docentes: Patricio Moreno y Adeodato Simó

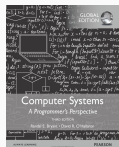
2.^{do} cuatrimestre de 2020

Última modificación: Tue Aug 4 14:36:17 2020 -0300

Facultad de Ingeniería (UBA)

Créditos

Para armar las presentaciones del curso nos basamos en:



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.

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

μP	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 				

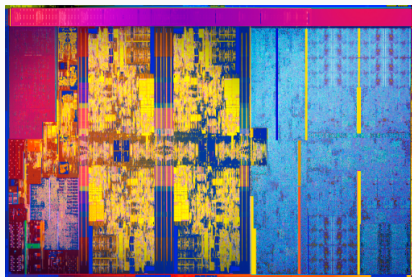
Evolución de Intel x86

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

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

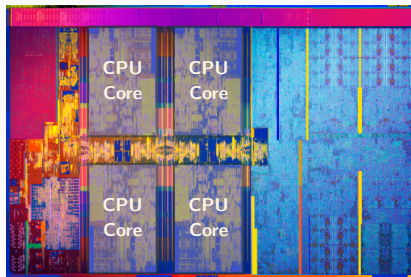
Evolución de Intel x86

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

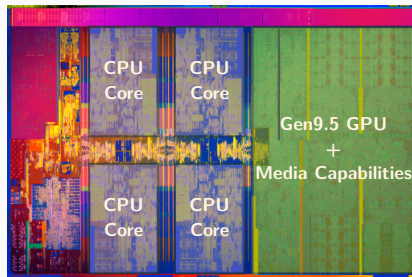
Evolución de Intel x86

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

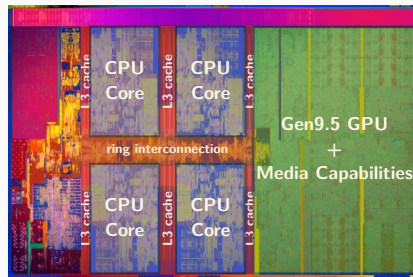
Evolución de Intel x86

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

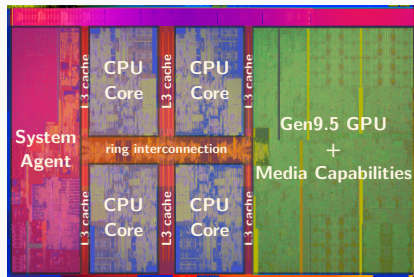
Evolución de Intel x86

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

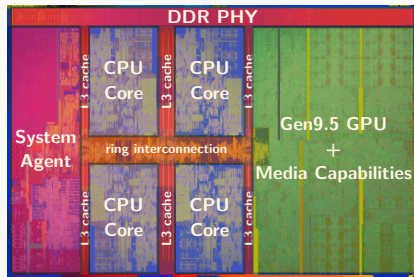
Evolución de Intel x86

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

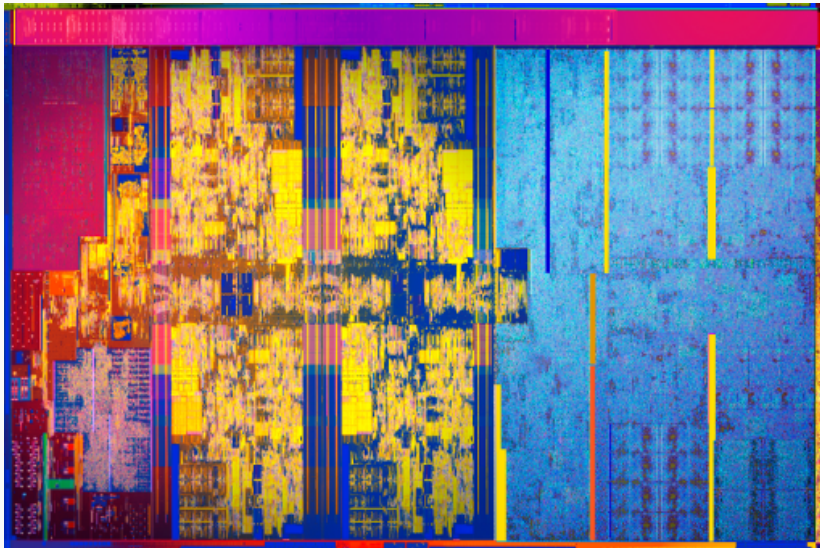
- Instrucciones multimedia
- Punto Flotante
- Branch Predictors
- Transición de 8, 16, 32 a 64 bits



Intel Core i7-8550U (Kaby Lake 2017)

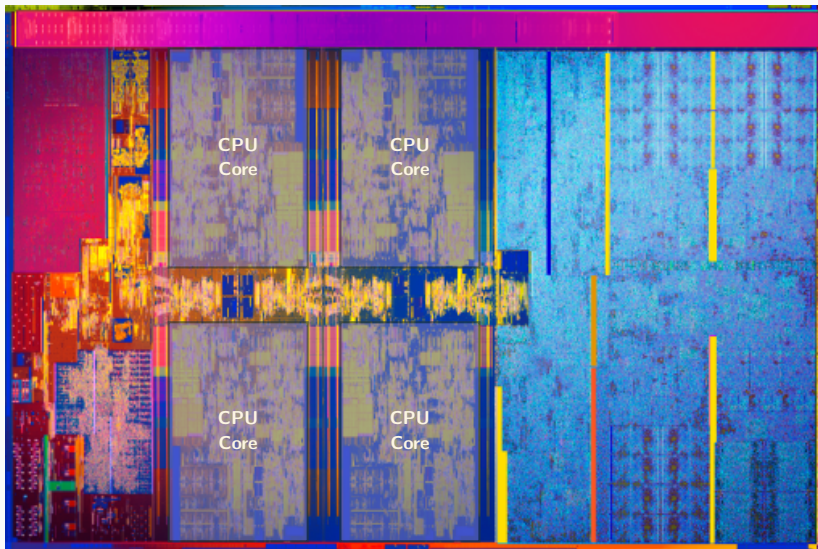
- Más núcleos
- Más memoria caché
- Procesadores gráficos
- Periféricos

Imagen: Kaby Lake Die



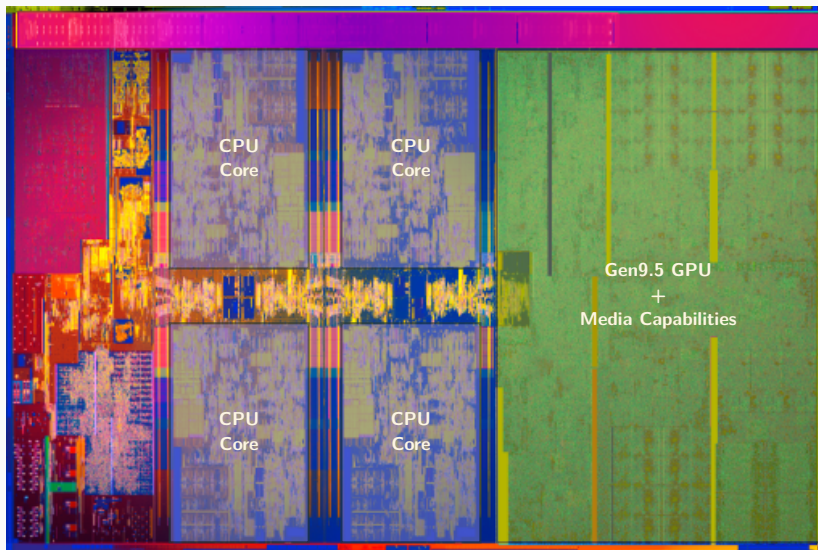
Área: 123 mm²

Imagen: Kaby Lake Die



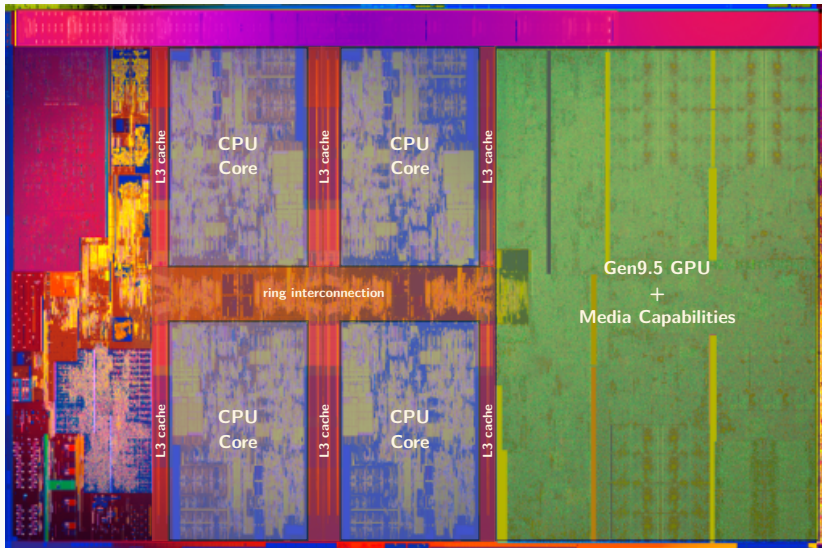
Área: 123 mm²

Imagen: Kaby Lake Die



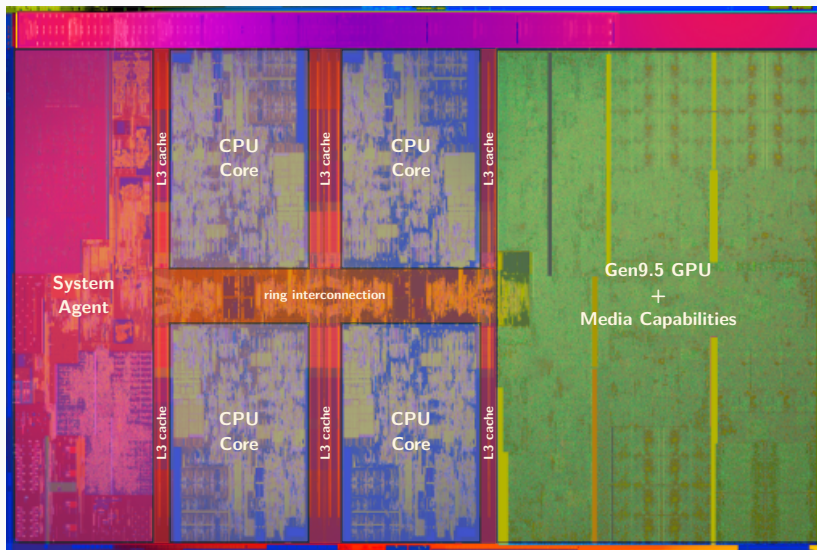
Área: 123 mm²

Imagen: Kaby Lake Die



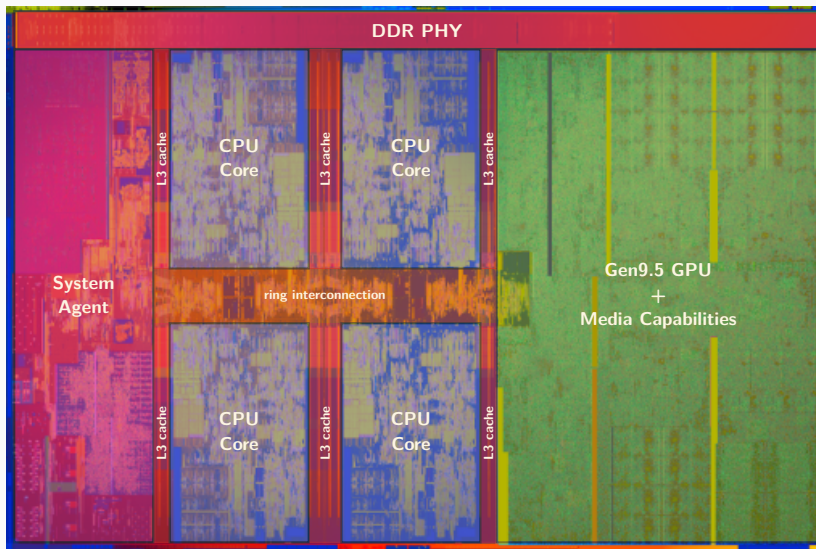
Área: 123 mm²

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
- Lanzan AMD64 (x86-64) antes que Intel, como extensión de x86

■ Últimamente

- Intel volvió a liderar
- AMD lo sigue de atrás
- 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 (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

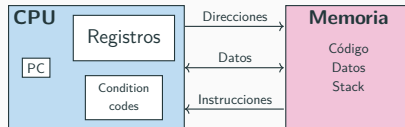
3. Operaciones aritméticas y lógicas

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

propósito general	%eax	%ax	%ah	%al	acumulador
	%ecx	%cx	%ch	%cl	contador
	%edx	%dx	%dh	%dl	datos
	%ebx	%bx	%bh	%bl	base
	%esi	%si			source index
	%edi	%di			destination index
	%esp	%sp			stack pointer
	%ebp	%bp			base pointer

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 movq

■ Instrucción:

`movq source, dest`

■ Operandos:

- **Inmediato:** Constante entera
 - Ejemplo: \$0x400, \$-533
 - Como en C, pero con '\$'
 - 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
 - Ejemplo más simple: (%rax)
 - Hay varios modos más de "direccionamiento"

%rax

%rbx

%rcx

%rdx

%rsi

%rdi

%rsp

%rbp

%rN

Operandos: ejemplo con movq

■ Instrucción:

movq 8 source, dest

■ Operandos:

- **Inmediato:** Constante entera
 - Ejemplo: \$0x400, \$-533
 - Como en C, pero con '\$'
 - 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
 - Ejemplo más simple: (%rax)
 - Hay varios modos más de "direccionamiento"

%rax

%rbx

%rcx

%rdx

%rsi

%rdi

%rsp

%rbp

%rN

Combinación de operandos: movq

	Origen	Destino	src, dst	C
movq	Inm.	Reg.	movq \$0x4, %rax	temp = 0x4;
		Mem.	movq \$-147, (%rax)	*p = -147;
	Reg.	Reg.	movq %rax, %rdx	temp2 = temp1;
		Mem.	movq %rax, (%rdx)	*p = temp;
	Mem.	Reg.	movq (%rax), %rdx	temp = *p;

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

Modos de direccionamiento simple

- **Normal** (R) $Mem[Reg[R]]$
 - El registro R especifica la dirección de memoria
 - ¡ R funciona como un puntero!
- **Corrimiento** $D(R)$ $Mem[Reg[R]+D]$
 - 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!

`movq (%rcx), %rax`

- **Corrimiento** $D(R)$ $\text{Mem}[\text{Reg}[R]+D]$
 - 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!

`movq (%rcx), %rax`

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

`movq 8(%rbp), %rdx`

Ejemplo: direccionamiento simple

```
1 void
2 misterio(<tipo> a, <tipo> b)
3         ???
4
5
6
7
8
9
```

`%rdi` `%rsi`

```
1 misterio:
2     movq (%rdi), %rax
3     movq (%rsi), %rdx
4     movq %rdx, (%rdi)
5     movq %rax, (%rsi)
6     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 }
```

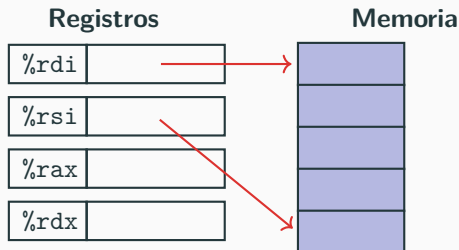
```
1 swap:  
2     movq (%rdi), %rax  
3     movq (%rsi), %rdx  
4     movq %rdx, (%rdi)  
5     movq %rax, (%rsi)  
6     ret
```


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     movq    (%rdi), %rax    # t0 = *xp
3     movq    (%rsi), %rdx    # t1 = *yp
4     movq    %rdx, (%rdi)    # *xp = t1
5     movq    %rax, (%rsi)    # *yp = t0
6     ret

```

Ejemplo: comprendiendo swap()

%rdi	0x120
%rsi	0x100
%rax	
%rdx	

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

swap:

```
movq (%rdi),%rax # t0 = *xp
movq (%rsi),%rdx # t1 = *yp
movq %rdx, (%rdi) # *xp = t1
movq %rax, (%rsi) # *yp = t0
ret
```

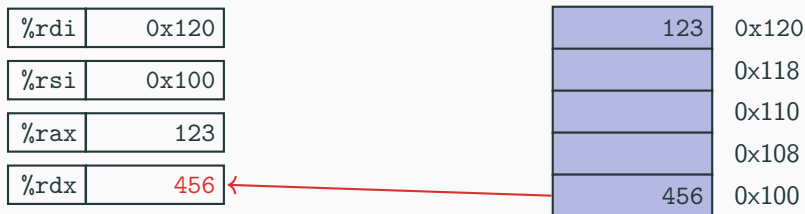
Ejemplo: comprendiendo swap()



swap:

```
movq (%rdi), %rax # t0 = *xp
movq (%rsi), %rdx # t1 = *yp
movq %rdx, (%rdi) # *xp = t1
movq %rax, (%rsi) # *yp = t0
ret
```

Ejemplo: comprendiendo swap()



swap:

```
movq (%rdi), %rax # t0 = *xp
movq (%rsi), %rdx # t1 = *yp
movq %rdx, (%rdi) # *xp = t1
movq %rax, (%rsi) # *yp = t0
ret
```

Ejemplo: comprendiendo swap()



swap:

```
movq (%rdi), %rax # t0 = *xp
movq (%rsi), %rdx # t1 = *yp
movq %rdx, (%rdi) # *xp = t1
movq %rax, (%rsi) # *yp = t0
ret
```

Ejemplo: comprendiendo swap()



swap:

```
movq (%rdi), %rax # t0 = *xp
movq (%rsi), %rdx # t1 = *yp
movq %rdx, (%rdi) # *xp = t1
movq %rax, (%rsi) # *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!

`movq (%rcx), %rax`

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

`movq 8(%rbp), %rdx`

Modos de direccionamiento completos

- **Forma general** $D(Rb, Ri, S)$ $Mem[Reg[Rb] + S * Reg[Ri] + D]$
 - 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)	$Mem[Reg[Rb] + Reg[Ri]]$
$D(Rb, Ri)$	$Mem[Reg[Rb] + Reg[Ri] + D]$
(Rb, Ri, S)	$Mem[Reg[Rb] + S * Reg[Ri]]$

Ejemplo: cálculo de direcciones

<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

$D(Rb, Ri, S)$ $Mem[Reg[Rb] + S * Reg[Ri] + D]$

- **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
<code>0x8(%rdx)</code>		
<code>(%rdx, %rcx)</code>		
<code>(%rdx, %rcx, 4)</code>		
<code>0x80(, %rdx, 2)</code>		

Ejemplo: cálculo de direcciones

<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

$D(Rb, Ri, S) \quad Mem[Reg[Rb] + S * Reg[Ri] + D]$

- **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
<code>0x8(%rdx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%rdx,%rcx)</code>	<code>0xf000 + 0x100</code>	<code>0xf100</code>
<code>(%rdx,%rcx,4)</code>	<code>0xf000 + 4*0x100</code>	<code>0xf400</code>
<code>0x80(,%rdx,2)</code>	<code>2*0xf000 + 0x80</code>	<code>0x1e080</code>

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)	
4(%rax)	
9(%rax,%rdx)	
260(%rcx,%rdx)	
0xFC(,%rcx,4)	
(%rax,%rdx,4)	

Ejercicio

Completar la siguiente tabla:

Operando	Valor	Comentario
%rax	0x100	registro
0x104	0xAB	dirección absoluta
\$0x108	0x108	inmediata
(%rax)	0xFF	dirección: 0x100
4(%rax)	0xAB	dirección: 0x104
9(%rax,%rdx)	0x11	dirección: 0x10C
260(%rcx,%rdx)	0x13	dirección: 0x108
0xFC(,%rcx,4)	0xFF	dirección: 0x100
(%rax,%rdx,4)	0x11	dirección: 0x10C

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
<code>movb <i>src, dst</i></code>	$\text{dst} \leftarrow \text{src}$ (byte)
<code>movw <i>src, dst</i></code>	$\text{dst} \leftarrow \text{src}$ (word)
<code>movl <i>src, dst</i></code>	$\text{dst} \leftarrow \text{src}$ (double word)
<code>movq <i>src, dst</i></code>	$\text{dst} \leftarrow \text{src}$ (quad word)
<code>movabsq <i>src, dst</i></code>	$\text{dst} \leftarrow \text{src}$ (abs. quad word)

1	<code>movabsq \$0x0011223344556677, %rax</code>	<i>%rax = 0x0011223344556677</i>
2	<code>movb \$-1, %al</code>	<i>%rax = 0x00112233445566FF</i>
3	<code>movw \$-1, %ax</code>	<i>%rax = 0x001122334455FFFF</i>
4	<code>movl \$-1, %eax</code>	<i>%rax = 0x00000000FFFFFFFF</i>
5	<code>movq \$-1, %rax</code>	<i>%rax = 0xFFFFFFFFFFFFFFFF</i>

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

- `leaq src, dst` # 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

```

1 leaq (%rdi,%rdi,2), %rax # t = x + 2*x
2 salq $2, %rax           # return t << 2

```

Multiplica sin multiplicar

Algunas operaciones aritméticas

Instrucciones con 2 operandos

Formato		Operación
addq	<i>src, dst</i>	$\text{dst} \leftarrow \text{dst} + \text{src}$
subq	<i>src, dst</i>	$\text{dst} \leftarrow \text{dst} - \text{src}$
imulq	<i>src, dst</i>	$\text{dst} \leftarrow \text{dst} * \text{src}$
salq	<i>src, dst</i>	$\text{dst} \leftarrow \text{dst} \ll \text{src}$
sarq	<i>src, dst</i>	$\text{dst} \leftarrow \text{dst} \gg \text{src}$
shrq	<i>src, dst</i>	$\text{dst} \leftarrow \text{dst} \gg \text{src}$
xorq	<i>src, dst</i>	$\text{dst} \leftarrow \text{dst} \wedge \text{src}$
andq	<i>src, dst</i>	$\text{dst} \leftarrow \text{dst} \& \text{src}$
orq	<i>src, dst</i>	$\text{dst} \leftarrow \text{dst} \text{src}$

*También llamada **shlq**
Aritmética
Lógica*

Más operaciones aritméticas

Instrucciones con 1 operando

Formato		Operación
incq	<i>dst</i>	$\text{dst} \leftarrow \text{dst} + 1$
decq	<i>dst</i>	$\text{dst} \leftarrow \text{dst} - 1$
negq	<i>dst</i>	$\text{dst} \leftarrow -\text{dst}$
notq	<i>dst</i>	$\text{dst} \leftarrow \sim\text{dst}$

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 }
```

```
1  arith:
2      leaq    (%rdi,%rsi), %rax
3      addq    %rdx, %rax
4      leaq    (%rsi,%rsi,2), %rcx
5      salq    $4, %rcx
6      leaq    4(%rdi,%rcx), %rcx
7      imulq   %rcx, %rax
8      ret
```

Instrucciones

- `leaq`: cálculo de direcciones (?)
- `salq`: desplazamiento (*shift*)
- `imulq`: 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 }

```

```

1  arith:
2      leaq    (%rdi,%rsi), %rax    # t1
3      addq    %rdx, %rax          # t2
4      leaq    (%rsi,%rsi,2), %rcx
5      salq    $4, %rcx            # t4
6      leaq    4(%rdi,%rcx), %rcx  # t5
7      imulq   %rcx, %rax          # rval
8      ret

```

Optimizaciones

- reuso de registros
- sustitución
- *strength reduction*

Registro	Uso(s)
%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

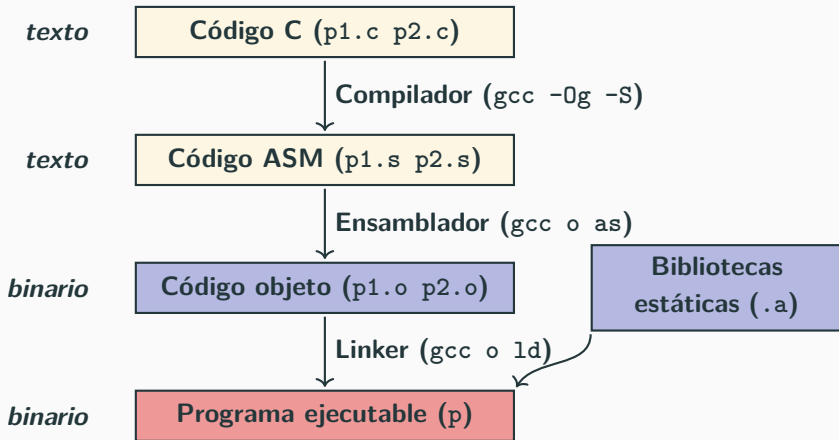
Movimiento de datos

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

```
1  sumstore:
2      pushq   %rbx
3      movq    %rdx, %rbx
4      call    plus
5      movq    %rax, (%rbx)
6      popq    %rbx
7      ret
```

Se obtuvo con el comando

```
gcc -S -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
    movq     %rdx, %rbx
    call     plus
    movq     %rax, (%rbx)
    popq     %rbx
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc
.LFE0:
    .size    sumstore, .-sumstore
```

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
movq     %rdx, %rbx
call     plus
movq     %rax, (%rbx)
popq     %rbx
.cfi_def_cfa_offset 8
ret
.cfi_endproc
.LFE0:
.size    sumstore, .-sumstore
```

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

cfi = call frame information

sumstore:

```
pushq %rbx
movq %rdx, %rbx
call plus
movq %rax, (%rbx)
popq %rbx
ret
```

Hay que extraer el código assembly
leyendo

Código objeto

```

1  4004d6:
2  0x53
3  0x48
4  0x89
5  0xd3
6  0xe8
7  0x05
8  0x00
9  0x00
10 0x00      Total: 14 bytes
11 0x48      Cada instrucción:
12 0x89      1, 3 o 5 bytes
13 0x03
14 0x5b Comienza en la dirección:
15 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 %rax, (%rbx)
```

Código Objeto

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

- Almacenar el valor `t` donde apunte `dest`
- Mover el valor de 8 bytes a memoria
- Operandos:

<code>t:</code>	Registro	<code>%rax</code>
<code>dest:</code>	Registro	<code>%rbx</code>
<code>*dest:</code>	Memoria	<code>M[%rbx]</code>
- 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     %rdx,%rbx
0x00000000004004da <+4>:      callq  0x4004e4 <plus>
0x00000000004004df <+9>:      mov     %rax,(%rbx)
0x00000000004004e2 <+12>:     pop     %rbx
0x00000000004004e3 <+13>:     retq
```

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`

