# Programación a Nivel-Máquina I: Conceptos Básicos y Aritmética

Estructura de Computadores
Semana 3

## Bibliografía:

[BRY16] Cap.3 Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016
Signatura ESIIT/C.1 BRY com

Transparencias del libro CS:APP, Cap.3
Introduction to Computer Systems: a Programmer's Perspective
**Autores:** Randal E. Bryant y David R. O'Hallaron
http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html

# Guía de trabajo autónomo (4h/s)

- **Lectura:** del Cap.3 CS:APP (Bryant/O'Hallaron)
  - Historical perspective, Program Encodings
    - § 3.1 – 3.2 pp.199-213
  - Data Formats, Accessing Info.
    - § 3.3 – 3.4 pp.213-227
  - Arithmetic and Logical Operations
    - § 3.5 pp.227-236

- **Ejercicios:** del Cap.3 CS:APP (Bryant/O'Hallaron)
  - Probl. 3.1 – 3.5 § 3.4, pp.218, 221, 222, 223, 225
  - Probl. 3.6 – 3.12 § 3.5, pp.228, 229, 230, 231, 232, 233, 236

## Bibliografía:

[BRY16] Cap.3 Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016
Signatura ESIIT/C.1 BRY com

# Programación Máquina I: Conceptos Básicos

- **Historia de los procesadores y arquitecturas de Intel**
- Lenguaje C, ensamblador, código máquina
- Conceptos básicos asm: Registros, operandos, move
- Operaciones aritméticas y lógicas

# Procesadores Intel x86

- **Dominan el mercado portátil/sobremesa/servidor**
- **Diseño evolutivo**
  - Compatible ascendentemente hasta el 8086, introducido en 1978
  - Va añadiendo características conforme pasa el tiempo
- **Computador con repertorio instrucciones complejo (CISC)**
  - Muchas instrucciones diferentes, con muchos formatos distintos
    - Según como se cuenten, entre 2.034-3.683 instrucciones
    - Pero sólo un pequeño subconjunto aparece en programas Linux
- **(RISC) Computador con repertorio instrucciones reducido**
  - RISC: ***muy pocas*** instrucciones, con ***muy pocos*** modos de direccionamiento
  - En principio funcionarían más rápido (reloj más rápido, segmentación…)
    - aún así Intel gana en velocidad (no tanto en bajo consumo)
    - Renacimiento RISC actual (p.ej. ARM, RISC-V), sobre todo bajo consumo

# Evolución Intel x86: Hitos significativos

| *Nombre* | *Fecha* | *Transistores* | *MHz* |
|---|---|---|---|
| **8086** | **1978** | **29K** | **5-10** |
| **386** | **1985** | **275K** | **16-33** |
| **Pentium 4E** | **2004** | **125M** | **2800-3800** |
| **Core 2** | **2006** | **291M** | **1060-3500** |
| **Core i7** | **2008** | **731M** | **1700-3900** |

- Primer procesador Intel 16-bit. Base para el IBM PC & MS-DOS
- Espacio direccionamiento 1MB
- Primer procesador Intel 32-bit de la familia (x86 luego llamada) IA32
- Añadió "direccionamiento plano"†, capaz de arrancar Unix
- 1er proc. Intel 64-bit de la familia (x86, llamada x86-64, EM64t) Intel 64
- Primer procesador Intel multi-core
- Cuatro cores, hyperthreading (2 vías)

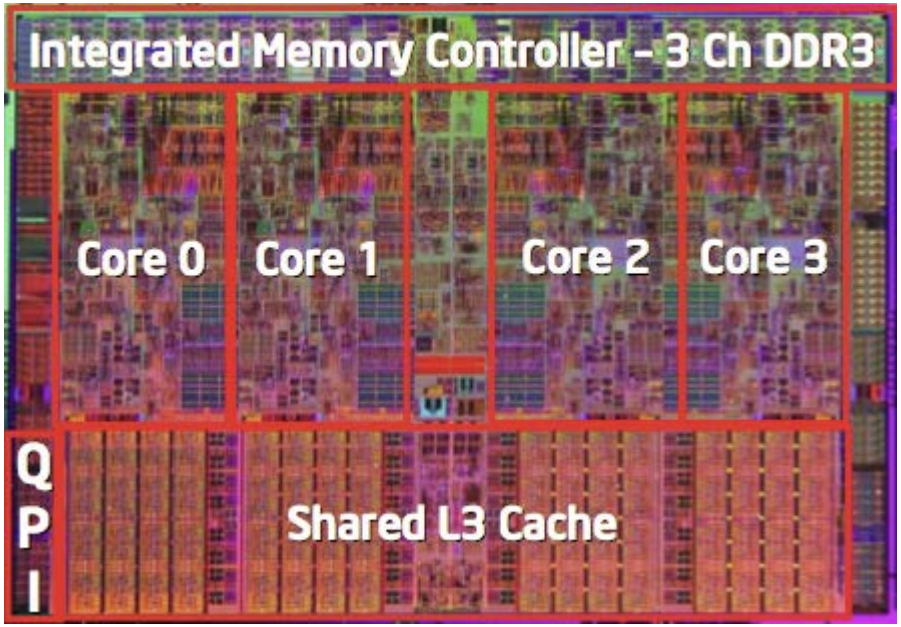


† *"flat addressing"*

# Procesadores Intel x86: Visión general

## Evolución de las máquinas

| | | |
|---|---|---|
| 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 |
| Core i7 Skylake | 2015 | 1.9B |




*Microfotografía de un dado Core i7*

## Características añadidas

- Instrucciones de soporte para operación multimedia (ops. en paralelo)
- Instrucciones para posibilitar operaciones condicionales más eficientes
- Transición de 32 bits a 64 bits
- Más núcleos (cores)

# Procesadores Intel x86

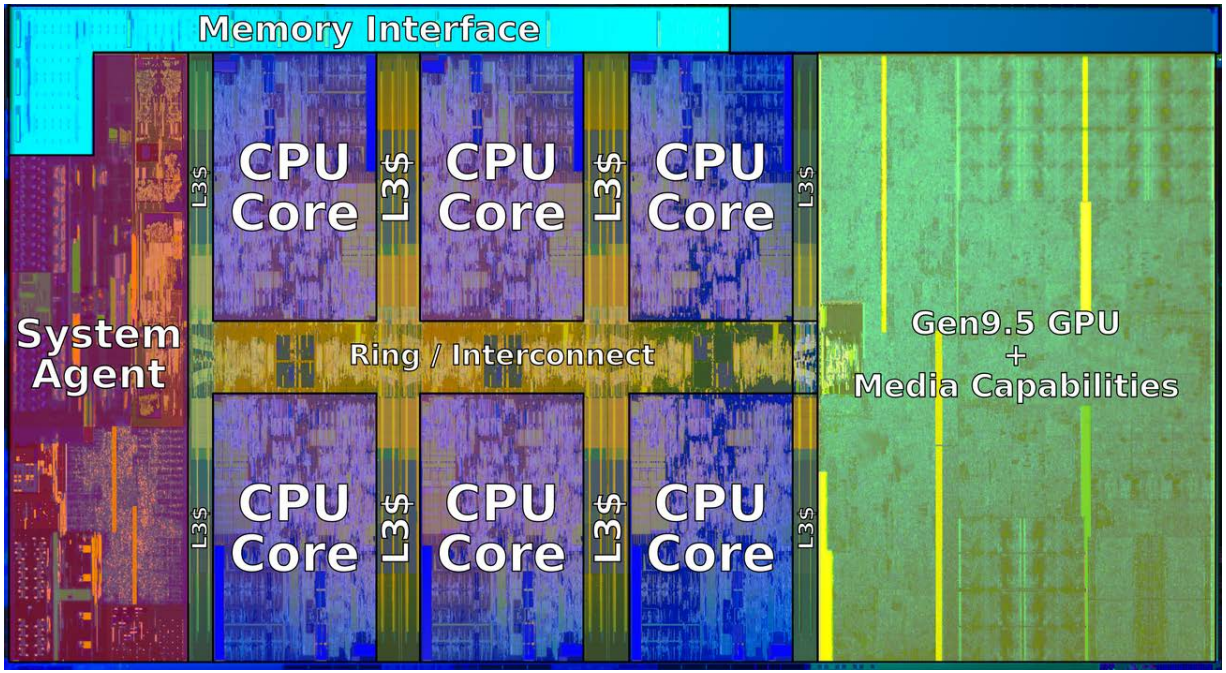
- **Generaciones pasadas** **Tecnología del proceso**
  - 1er Pentium Pro 1995 600 nm
  - 1er Pentium III 1999 250 nm
  - 1er Pentium 4 2000 180 nm
  - 1er Core 2 Duo 2006 65 nm

**Tamaño tecnología proceso = anchura minima trazo (10 nm ≈ 100 átomos ancho)**

- **Generaciones recientes y venideras**

| | | | | | |
|---|---|---|---|---|---|
| 1. | Nehalem | 2008 | 45 nm | | |
| 2. | Sandy Bridge | 2011 | 32 nm | | |
| 3. | Ivy Bridge | 2012 | 22 nm | | |
| 4. | Haswell | 2013 | 22 nm | | |
| 5. | Broadwell | 2014 | 14 nm | | |
| 6. | Skylake | 2015 | 14 nm | | |
| 7. | Kaby Lake | 2016 | 14 nm | Core i3, i5, i7, | 7xxx |
| 8. | Coffee Lake | 2017 | 14 nm | Core i3, i5, i7, i9, | 8xxx , 9xxx |
| 9. | Cannon Lake | 2018 | 10 nm | Core i3 | 8121 |
| 10. | Ice Lake | 2019 | 10 nm | Core i3, i5, i7, | 10xx[NG]x |
| 11. | Comet Lake? | 2020? | 10 nm | Core i3, i5, i7, i9 | 10xxx[KFT] |

# Lo último† en 2018: Coffee Lake




- **Modelo portátil: Core i7**
  - 2.2-3.2 GHz
  - 45 W
- **Sobremesa: Core i7**
  - Gráficos integrados
  - 2.4-4.0 GHz
  - 35-95 W
- **Servidor: Xeon E**
  - Gráficos integrados
  - Habilitado para multi-zócalo‡
  - 3.3-3.8 GHz
  - 80-95 W

† *"state of the art"*
‡ *"multi-socket"*

# Clones x86: Advanced Micro Devices (AMD)

- **Históricamente**
  - AMD ha ido siguiendo a Intel en todo
  - CPUs un poco más lentas, mucho más baratas
- **Y entonces**
  - Reclutaron los mejores diseñadores de circuitos de Digital Equipment Corp. y otras compañías con tendencia descendente
  - Construyeron el Opteron: duro competidor para el Pentium 4
  - Desarrollaron x86-64, su propia extensión a 64 bits
- **En años recientes**
  - Intel se reorganizó para ser más efectiva
    - 1995-2011: líder mundial semiconductores, 2019: 2º detrás de Samsung
  - AMD se ha quedado rezagada
    - Externalizo su “fab” semiconductores (spin-off GlobalFoundries)
    - Recientemente nuevas CPUs competitivas (p.ej. Ryzen)

# La historia de los 64-bit de Intel

- **2001: Intel intenta un cambio radical de IA32 a IA64**
  - Arquitectura totalmente diferente (Itanium)
  - Ejecuta código IA32 sólo como herencia†
  - Prestaciones decepcionantes
- **2003: AMD interviene con una solución evolutiva**
  - x86-64 (ahora llamado “AMD64”)
- **Intel se sintió obligada a concentrarse en IA64**
  - Difícil admitir error, o admitir que AMD es mejor
- **2004: Intel anuncia extensión EM64T‡ de la IA32** (ahora llamada Intel64)
  - Extended Memory 64-bit Technology
  - ¡Casi idéntica a x86-64!
- **Todos los procesadores x86 salvo gama baja soportan x86-64**
  - Pero gran cantidad de código se ejecuta aún en modo 32-bits

*† “legacy” = herencia de características*
*‡ Intel usa ahora "IA32" e "Intel64" para distinguir IA32 de EM64T y evitar confusión con IA64*

# Nosotros cubrimos:

- ~~IA32~~
  - ~~El x86 tradicional~~
  - ~~Para EC: RIP, verano 2018~~
- **x86-64** / Intel64
  - El estándar
  - `ubuntu_18> gcc hello.c`
  - `ubuntu_18> gcc –m64 hello.c`
- **Presentación**
  - El libro cubre x86-64. Transparencias, prácticas, ejercicios… todo en x86-64.
  - En el libro hay un "añadido Web"† sobre IA32
  - En SWAD puede quedar material (tests/exámenes/…) sobre IA32
  - Sólo algunos detalles querremos recordar de IA32 (alineamiento, pila…)



† *"Web aside"*

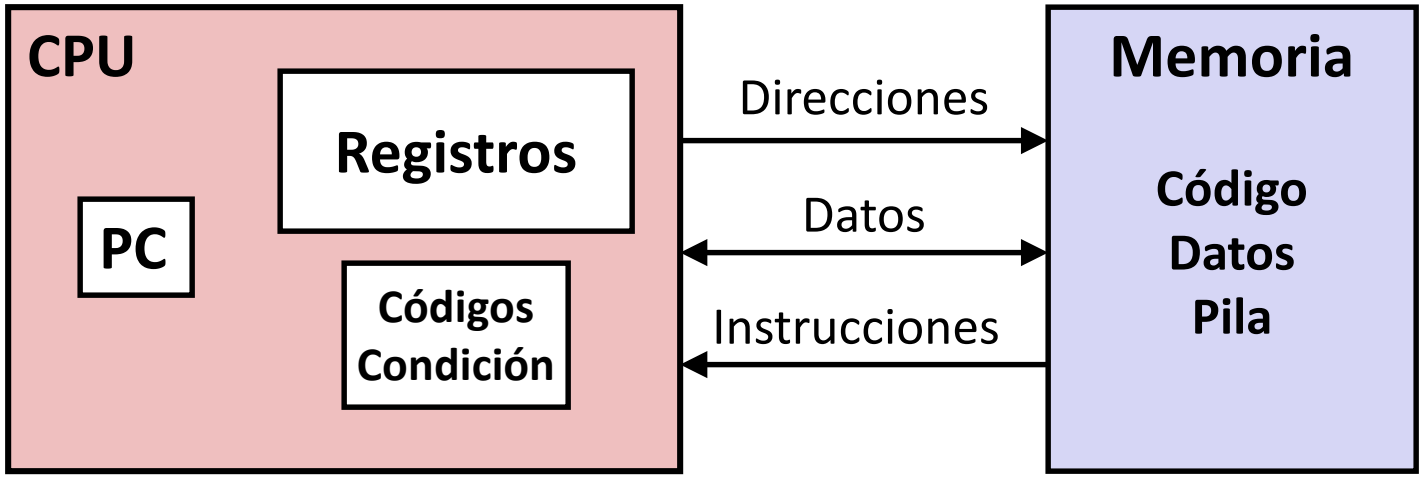
# Programación Máquina I: Conceptos Básicos

- Historia de los procesadores y arquitecturas de Intel
- **Lenguaje C, ensamblador, código máquina**
- Conceptos básicos asm: Registros, operandos, move
- Operaciones aritméticas y lógicas

# Definiciones

- **Arquitectura: (también arquitectura del repertorio de instrucciones: ISA) Las partes del diseño de un procesador que se necesitan entender para escribir código ensamblador.**
  - Ejemplos: especificación del repertorio de instrucciones, registros.
- **Formas del código:**
  - Código máquina: Programas (codops, bytes) que ejecuta el procesador
  - Código ensamblador: Representación textual del código máquina
- **Microarquitectura: Implementación de la arquitectura.**
  - Ejemplos: tamaño de las caches y frecuencia de los cores.
- **Ejemplos de ISAs:**
  - Intel: (x86 = ) IA32, Itanium ( = IA64 = IPF), x86-64 ( = Intel 64 = EM64t)
  - ARM: Usado en casi todos los teléfonos móviles
  - RISC-V: Nueva ISA open-source

# Perspectiva Código Ensamblador/Máquina



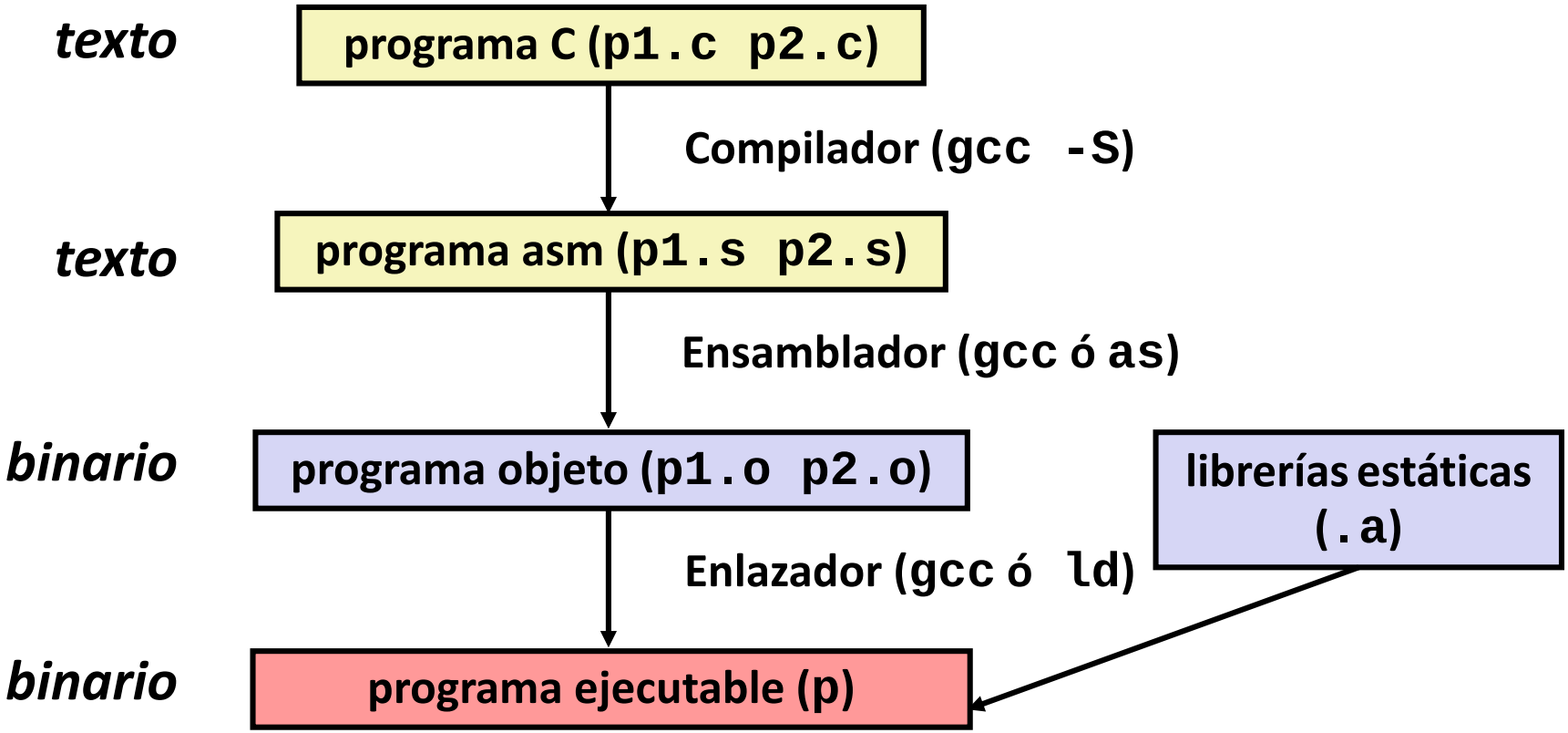

## Estado visible al programador

- **PC: Contador de programa**
  - Dirección de la próxima instrucción
  - Llamado "RIP" (x86-64)
- **Archivo de registros**
  - Datos del programa muy utilizados
- **Códigos de condición / flags de estado**
  - Almacenan información estado sobre la operación aritmética/lógica más reciente
  - Usados para bifurcación condicional

- **Memoria**
  - Array direccionable por bytes
  - Código y datos usuario
  - Pila soporte a procedimientos

# Convertir C en Código Objeto

- Código en ficheros **`p1.c p2.c`**
- Compilar con el comando: **`gcc –Og p1.c p2.c -o p`**
  - Usar optimizaciones básicas (**`-Og`**) [versiones recientes de GCC†]
  - Poner binario resultante en fichero **`p`**






† *GCC >= 4.8. Como referencia, Ubuntu 18.04 LTS viene con GCC 7.3*

# Compilar a ensamblador

## Código C (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

## Ensamblador x86-64 generado†

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

Obtenerlo con el comando

```
gcc –Og –S sum.c
```

Produce el fichero `sum.s`

***Aviso:*** **Se obtendrán resultados diferentes en cuanto se usen diferentes versiones de gcc y diferentes ajustes del compilador†**

† *Añadir –fno-asynchronous-unwind-tables en GCC 7.3 Ubuntu 18.04*

# El aspecto que tiene realmente

```
        .globl  sumstore
        .type   sumstore, @function
sumstore:
.LFB35:
        .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
.LFE35:
        .size   sumstore, .-sumstore
```

**Las cosas raras que van precedidas de un "." suelen ser directivas**

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

*† Añadir –fno-asynchronous-unwind-tables en GCC 7.3 Ubuntu 18.04*

# Representación Datos C, IA32, x86-64

- **Tamaño de Objetos C (en Bytes)**

| *Tipo de Datos C* | *Normal 32-bit* | *Intel IA32* | *x86-64* |
|---|---|---|---|
| unsigned | 4 | 4 | 4 |
| int | 4 | 4 | 4 |
| long int | 4 | 4 | 8 |
| char | 1 | 1 | 1 |
| short | 2 | 2 | 2 |
| float | 4 | 4 | 4 |
| double | 8 | 8 | 8 |
| long double | 8 | 10/12 | 16 |
| char * | 4 | 4 | 8 |

  - *o cualquier otro puntero*

# Características Ensamblador: Tipos de Datos

- **Datos "enteros" de 1, 2, 4 u 8 bytes**
  - Valores de datos
  - Direcciones (punteros sin tipo)
- **Datos en punto flotante de 4, 8 ó 10 bytes**
- **Código: secuencias de bytes codificando serie de instrucciones**
- **No hay tipos compuestos como arrays o estructuras**
  - Tan sólo bytes ubicados contiguamente (uno tras otro) en memoria

# Características Ensamblador: Instrucciones

- **Realizan función aritmética sobre datos en registros o memoria**
  - "Operaciones" = Instrucciones aritmético/lógicas
- **Transfieren datos entre memoria y registros**
  - Cargar datos de memoria a un registro
  - Almacenar datos de un registro en memoria
  - "Instrucciones de transferencia"
- **Transferencia de control**
  - Incondicionales: saltos, llamadas a procedimientos, retornos desde procs.
  - Saltos condicionales
  - "Instrucciones de control"

# Código Objeto

## Código de `sumstore`

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

- 14 bytes total
- Cada instrucción 1, 3, ó 5 bytes
- Empieza en direcc. `0x0400595`

- **Ensamblador**
  - Traduce `.s` pasándolo a `.o`
  - Instrucciones codificadas en binario
  - Imagen casi completa del código ejecutable
  - Le faltan enlaces entre código de ficheros diferentes
- **Enlazador**
  - Resuelve referencias entre ficheros
  - Combina con libs. de tiempo ejec. estáticas†
    - P.ej., código para **`malloc`**, **`printf`**
  - Algunas libs. son *dinámicamente enlazadas*‡
    - El enlace ocurre cuando el programa empieza a ejecutarse

† *"static run-time libraries" = bibliotecas estáticas para soporte en tiempo de ejecución*

‡ *"dynamically linked libraries", o también "shared libs"*

# Ejemplo de Instrucción Máquina

```
*dest = t;
```

- **Código C**
  - Almacenar valor **t** adonde indica (apunta) **dest**

```
movq %rax, (%rbx)
```

- **Ensamblador**
  - Mover un valor de 8-byte a memoria
    - "Palabra Quad"† en jerga x86-64
  - Operandos:
    - **t:** Registro **%rax**
    - **dest:** Registro **%rbx**
    - ***dest:** Memoria **M[%rbx]**

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

- **Código Objeto**
  - Instrucción de 3-byte
  - Almacenada en dir. **0x40059e**



† *"Quad word"*

# Desensamblar Código Objeto

## Desensamblado

```
0000000000400595 <sumstore>:
  400595:  53                   push   %rbx
  400596:  48 89 d3             mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff       callq  400590 <plus>
  40059e:  48 89 03             mov    %rax,(%rbx)
  4005a1:  5b                   pop    %rbx
  4005a2:  c3                   retq
```

- **Desensamblador**

  **`objdump -d sum`**

  - Herramienta útil para examinar código objeto
  - Analiza el patrón de bits de series de instrucciones
  - Produce versión aproximada del código ensamblador (correspondiente)
  - Puede ejecutarse sobre el fich. `a.out` (ejecutable completo) ó el `.o`

# Desensamblado Alternativo

**Objeto**

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

**Desensamblado**

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- **Desde el Depurador gdb**

  **`gdb sum`**

  **`disassemble sumstore`**

  - Desensamblar procedimiento

  **`x/14xb sumstore`**

  - Examinar 14 bytes a partir de `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:
30001001:
30001003:
30001005:
3000100a:
```

**Ingeniería inversa prohibida por licencia Microsoft End User License Agreement (EULA)**

- Cualquier cosa que se pueda interpretar como código ejecutable
- El desensamblador examina bytes y reconstruye el fuente asm.

# Programación Máquina I: Conceptos Básicos

- Historia de los procesadores y arquitecturas de Intel
- Lenguaje C, ensamblador, código máquina
- **Conceptos básicos asm: Registros, operandos, move**
- Operaciones aritméticas y lógicas

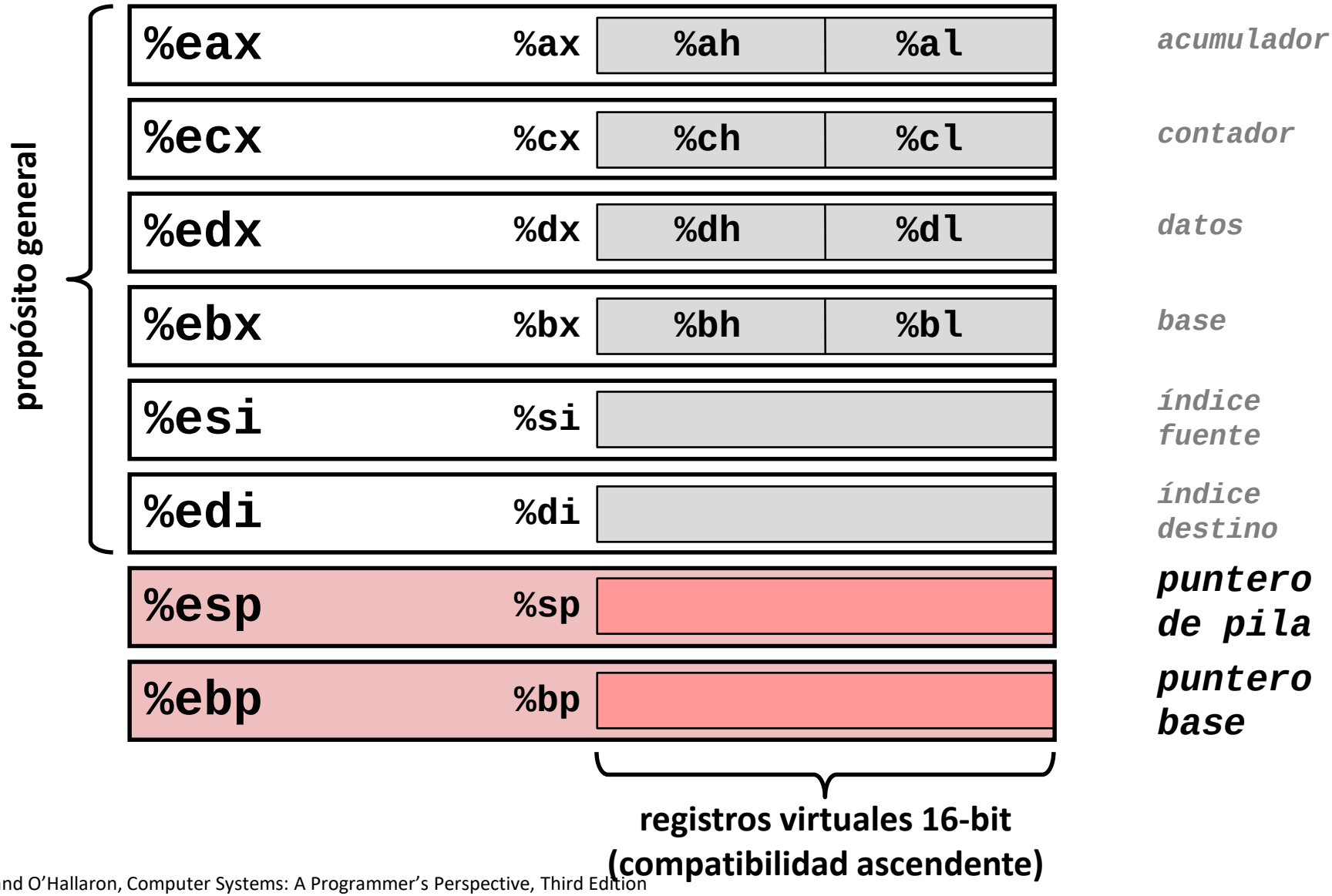
# Registros enteros x86-64


- Pueden referenciarse los 4 bytes de menor peso† (los 4 LSBs)
  - (también los 2 LSB y el 1 LSB)



† *"low-order 4 bytes"*

# Un poco de historia: registros IA32



# Mover Datos

- **Mover Datos**

  `movq` ***Source*, *Dest***[†]

- **Tipo de Operandos**
  - ***Inmediato:*** Datos enteros constantes
    - Ejemplo: **`$0x400`, `$-533`**
    - Como constante C, pero con prefijo **`'$'`**
    - Codificado mediante 1, 2, ó 4 bytes[‡]
  - ***Registro:*** Alguno de los 16 registros enteros
    - Ejemplo: **`%rax`, `%r13`**
    - Pero **`%rsp`** reservado para uso especial
    - Otros tienen usos especiales con instrucciones particulares
  - ***Memoria:*** 8 bytes consecutivos mem. en dirección dada por un registro
    - Ejemplo más sencillo: **`(%rax)`**
    - Hay otros diversos "modos de direccionamiento"

| |
|---|
| `%rax` |
| `%rcx` |
| `%rdx` |
| `%rbx` |
| `%rsi` |
| `%rdi` |
| `%rsp` |
| `%rbp` |
| `%rN` |

‡ *luego veremos movabsq para literales 8B*

† *"source/destination" = fuente/destino*

# Combinaciones de Operandos `movq`

| | Source | Dest | Src,Dest | Análogo C |
|---|---|---|---|---|
| `movq` | *Imm*† | *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;` |

***Ver resto instrucciones transferencia (incluyendo pila) en el libro***

***No se puede transferir Mem-Mem con sólo una instrucción***



† *"Immediate" = Inmediato*

# Modos Direccionamiento a memoria sencillos

- **Normal† (R) Mem[Reg[R]]**
  - El registro R indica la dirección de memoria
  - ¡Exacto! Como seguir (*desreferenciar‡*) un puntero en C

```
movq (%rcx),%rax
```

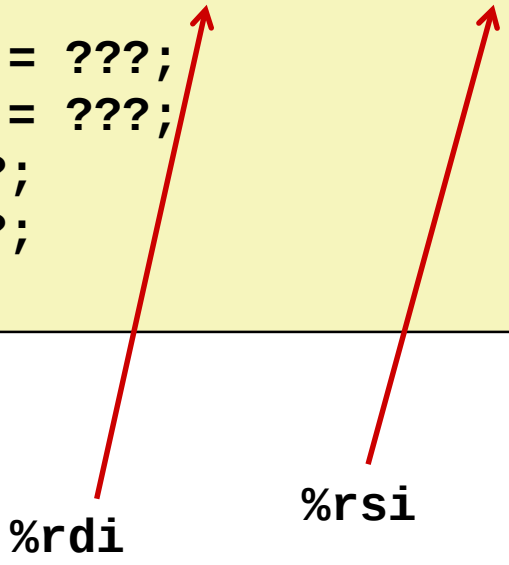
- **Desplazamiento D(R) Mem[Reg[R]+D]**
  - El registro R indica el inicio de una región de memoria
  - La constante de desplazamiento D indica el *offset‡*

```
movq 8(%rbp),%rdx
```

‡ *"offset"=compensación, para nosotros "desplazamiento"*
† *"indirecto a través de registro" según otros autores*
‡ *"dereferencing" en el original*

# Ejemplo Modos Direccionamiento sencillos

```
void adiv(<tipo> a, <tipo> b)
{
  ???? ?? = ???;
  ???? ?? = ???;
  ??? = ??;
  ??? = ??;
}
```




```
adiv:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

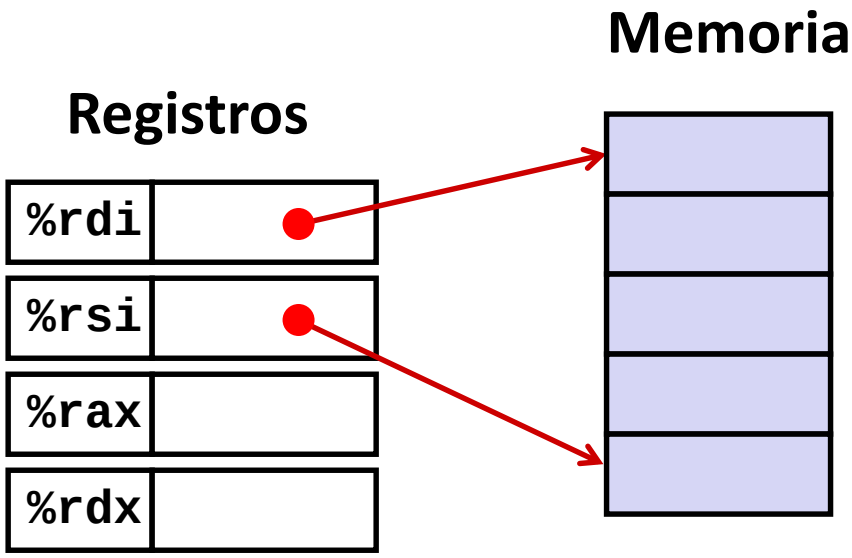
# Ejemplo Modos Direccionamiento sencillos

```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

# Comprendiendo swap()†

```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

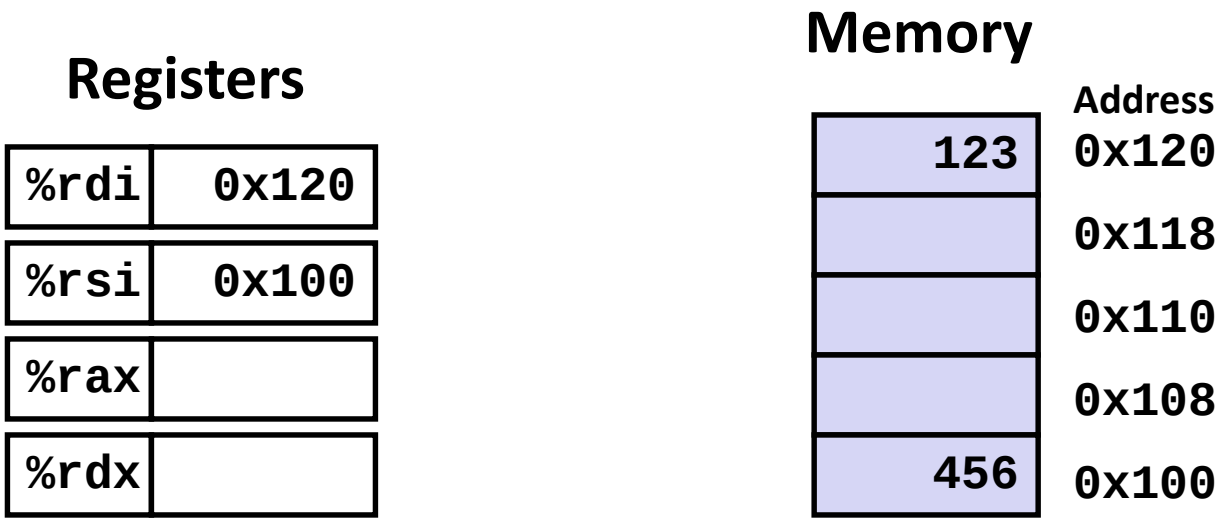



| Registro | Valor |
|---|---|
| %rdi | xp |
| %rsi | yp |
| %rax | t0 |
| %rdx | t1 |

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

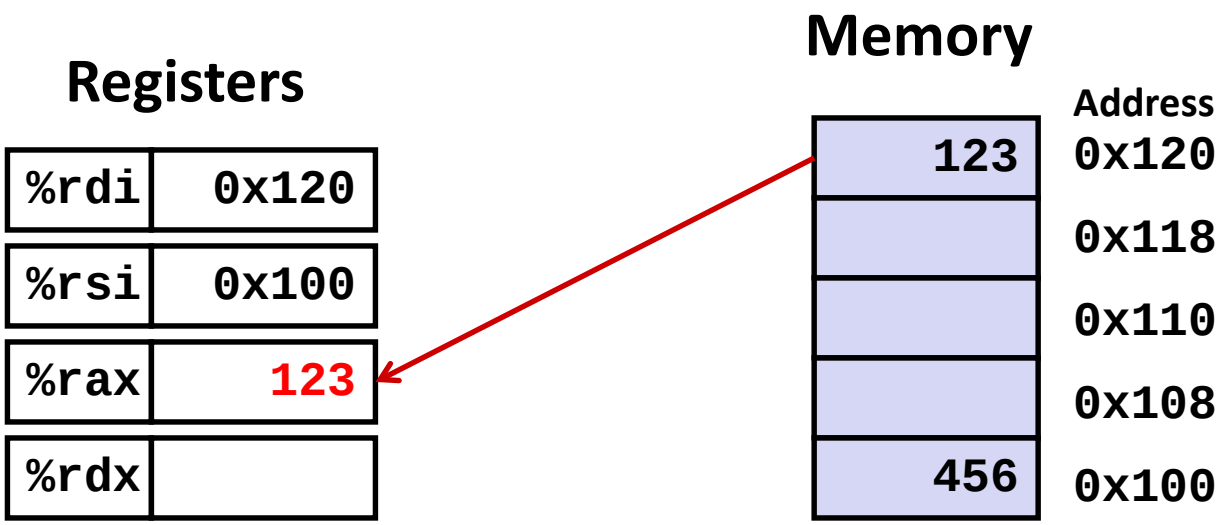
† *"swap" = intercambio*

# Comprendiendo swap( )



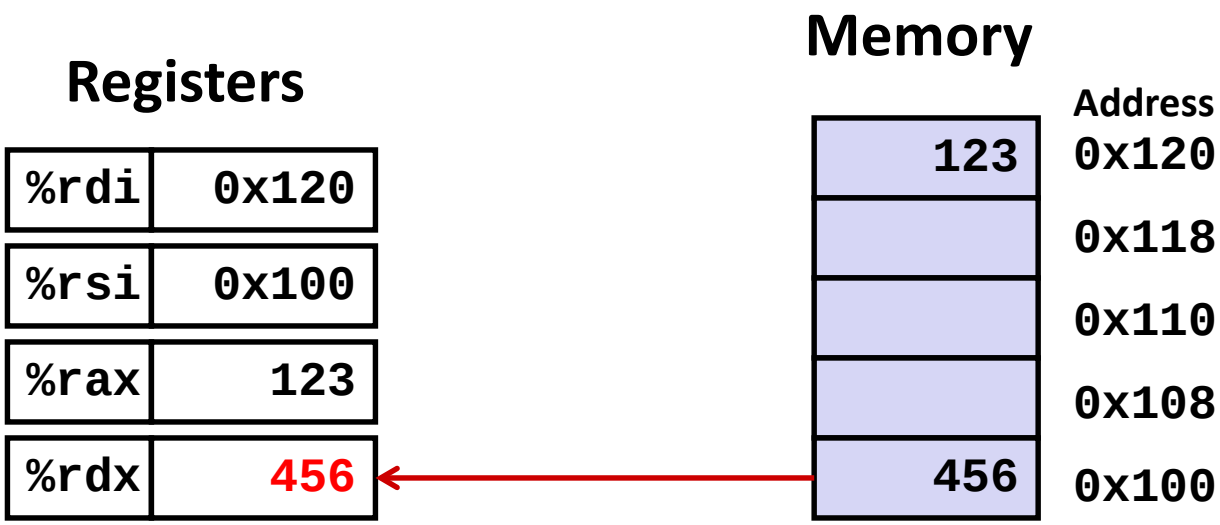

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

# Comprendiendo swap( )



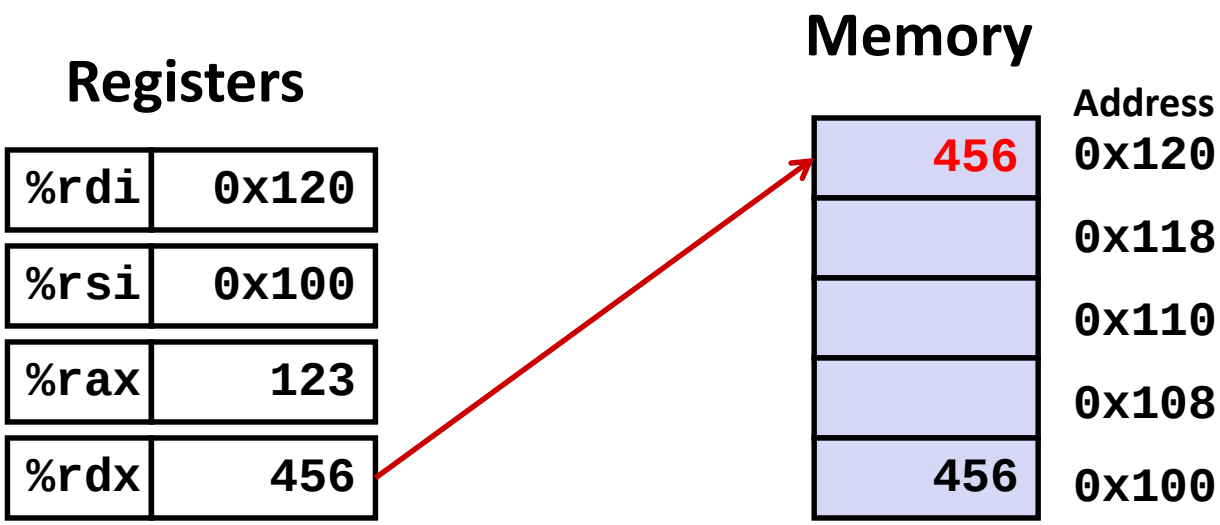

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

# Comprendiendo swap( )



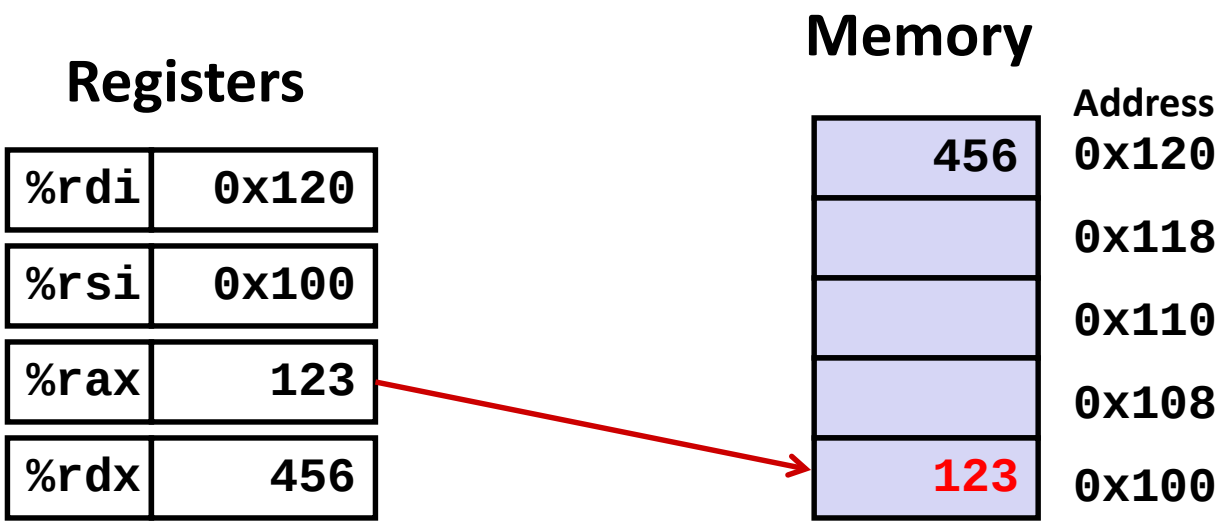

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

# Comprendiendo swap( )




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

# Comprendiendo swap( )




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

# Modos Direccionamiento a memoria sencillos

- **Normal[‡] (R) Mem[Reg[R]]**
  - El registro R indica la dirección de memoria
  - ¡Exacto! Como seguir (*desreferenciar*[†]) un puntero en C

```
movq (%rcx),%rax
```

- **Desplazamiento D(R) Mem[Reg[R]+D]**
  - El registro R indica el inicio de una región de memoria
  - La constante de desplazamiento D indica el *offset*[*]

```
movq 8(%rbp),%rdx
```

# Modos Direccionamiento a memoria completos

- **Forma más general**

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

  - D: "Desplazamiento" constante 1, 2, ó 4 bytes
  - Rb: Registro base: Cualquiera de los 16 registros enteros
  - Ri: Registro índice: Cualquiera, excepto **`%rsp`**
  - S: Factor de escala: 1, 2, 4, ú 8 (*¿por qué esos números?*)

- **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]]**

# Ejemplos de Cálculo de Direcciones

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

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

- D: "Desplazamiento" constante 1, 2, ó 4 bytes
- Rb: Registro base: Cualquiera de los 16 registros enteros
- Ri: Registro índice: Cualquiera, excepto `%rsp`
- S: Factor de escala: 1, 2, 4, ú 8 (*¿por qué esos números?*)

| Expresión | Cálculo de Dirección | Dirección |
|---|---|---|
| 0x8(%rdx) | 0xf000 + 0x8 | 0xf008 |
| (%rdx,%rcx) | 0xf000 + 0x100 | 0xf100 |
| (%rdx,%rcx,4) | 0xf000 + 4*0x100 | 0xf400 |
| 0x80(,%rdx,2) | 2*0xf000 + 0x80 | 0x1e080 |

# Programación Máquina I: Conceptos Básicos

- Historia de los procesadores y arquitecturas de Intel
- Lenguaje C, ensamblador, código máquina
- Conceptos básicos asm: Registros, operandos, move
- **Operaciones aritméticas y lógicas**

# Instrucción para el Cálculo de Direcciones

- **leaq *Src*, *Dest*†**
  - *Src* es cualquier expresión de modo direccionamiento (a memoria)
  - Ajusta *Dest* a la dirección indicada por la expresión

- **Usos**
  - Calcular direcciones sin hacer referencias a memoria
    - P.ej., traducción de p = &x[i];
  - Calcular expresiones aritméticas de la forma x + k*y
    - k = 1, 2, 4 ú 8

- **Ejemplo**

```
long m12(long x)
{
  return x*12;
}
```

**Traducción a ASM por el compilador:**

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

† *"source/destination" = fuente/destino*

# Algunas Operaciones Aritméticas

- **Instrucciones de Dos Operandos:**

| *Formato* | | *Operación*† | |
|---|---|---|---|
| `addq` | *Src,Dest* | Dest = Dest + Src | |
| `subq` | *Src,Dest* | Dest = Dest – Src | |
| `imulq` | *Src,Dest* | Dest = Dest * Src | |
| `salq` | *Src,Dest* | Dest = Dest << Src | *También llamada shlq* |
| `sarq` | *Src,Dest* | Dest = Dest >> Src | *Aritméticas* |
| `shrq` | *Src,Dest* | Dest = Dest >> Src | *Lógicas* |
| `xorq` | *Src,Dest* | Dest = Dest ^ Src | |
| `andq` | *Src,Dest* | Dest = Dest & Src | |
| `orq` | *Src,Dest* | Dest = Dest \| Src | |

- **¡Cuidado con el orden de los argumentos!** (Intel vs. AT&T)
- **No se distingue entre enteros con/sin signo (*¿por qué?*)**



† *"source/destination" = fuente/destino*

# ¡Hora de test!

Consultar:

https://canvas.cmu.edu/courses/13182

# Algunas Operaciones Aritméticas

- **Instrucciones de Un Operando:**

| *Formato* | | *Operación* |
|---|---|---|
| `incq` | *Dest* | *Dest = Dest + 1* |
| `decq` | *Dest* | *Dest = Dest – 1* |
| `negq` | *Dest* | *Dest = – Dest* |
| `notq` | *Dest* | *Dest = ~Dest* |

- **Consultar más instrucciones en el libro** (sólo 10-24, no 2034-3683)
  - Aritméticas: [i]mulq Src, [i]divq Src, cqto
  - Transferencia: movX (bwlq), movabsq, movzXX (bw,bl,bq,wl,wq),† movsXX (bw,bl,bq,wl,wq,lq), cltq, pushq, popq

† *luego veremos que movzlq %eax, %rbx sería lo mismo que mov %eax, %ebx*

# Ejemplo de Expresiones Aritméticas

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

## Instrucciones interesantes

- **`leaq`**: cálculo de direcciones
- **`salq`**: desplazamiento aritmético
- **`imulq`**: multiplicación
  - pero sólo se usa una vez

# Comprendiendo arith()

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax    # t1
   addq    %rdx, %rax           # t2
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx             # t4
   leaq    4(%rdi,%rdx), %rcx   # t5
   imulq   %rcx, %rax           # rval
   ret
```

| Registro | Uso(s) |
|---|---|
| %rdi | Argumento x |
| %rsi | Argumento y |
| %rdx | Argumento z |
| %rax | t1, t2, rval |
| %rdx | t4 |
| %rcx | t5 |

# Programación a Nivel-Máquina I: Resumen

- **Historia de los procesadores y arquitecturas de Intel**
  - Diseño evolutivo lleva a demasiados artefactos y peculiaridades
- **Lenguaje C, ensamblador, código máquina**
  - Nuevas formas de estado visible†: contador de programa, registros, ...
  - El compilador debe transformar sentencias, expresiones, procedimientos, en secuencias de instrucciones a bajo nivel
- **Conceptos básicos asm: Registros, operandos, move**
  - Las instrucciones x86-64 **`mov`** cubren un amplio rango de variedades de movimientos de datos (transferencia)
- **Operaciones aritméticas y lógicas**
  - El compilador C saldrá con diversas combinaciones de instrucciones para realizar los cálculos



† *En el sentido de "observabilidad" (máquina de estados)*

# Guía de trabajo autónomo (4h/s)

- **Estudio:** del Cap.3 CS:APP (Bryant/O'Hallaron)
  - Historical perspective, Program Encodings
    - § 3.1 – 3.2 pp.199-213
  - Data Formats, Accessing Info.
    - § 3.3 – 3.4 pp.213-227
  - Arithmetic and Logical Operations
    - § 3.5 pp.227-236

- **Ejercicios:** del Cap.3 CS:APP (Bryant/O'Hallaron)
  - Probl. 3.1 – 3.5 § 3.4, pp.218, 221, 222, 223, 225
  - Probl. 3.6 – 3.12 § 3.5, pp.228, 229, 230, 231, 232, 233, 236

## Bibliografía:

[BRY16] Cap.3 Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016
Signatura ESIIT/C.1 BRY com