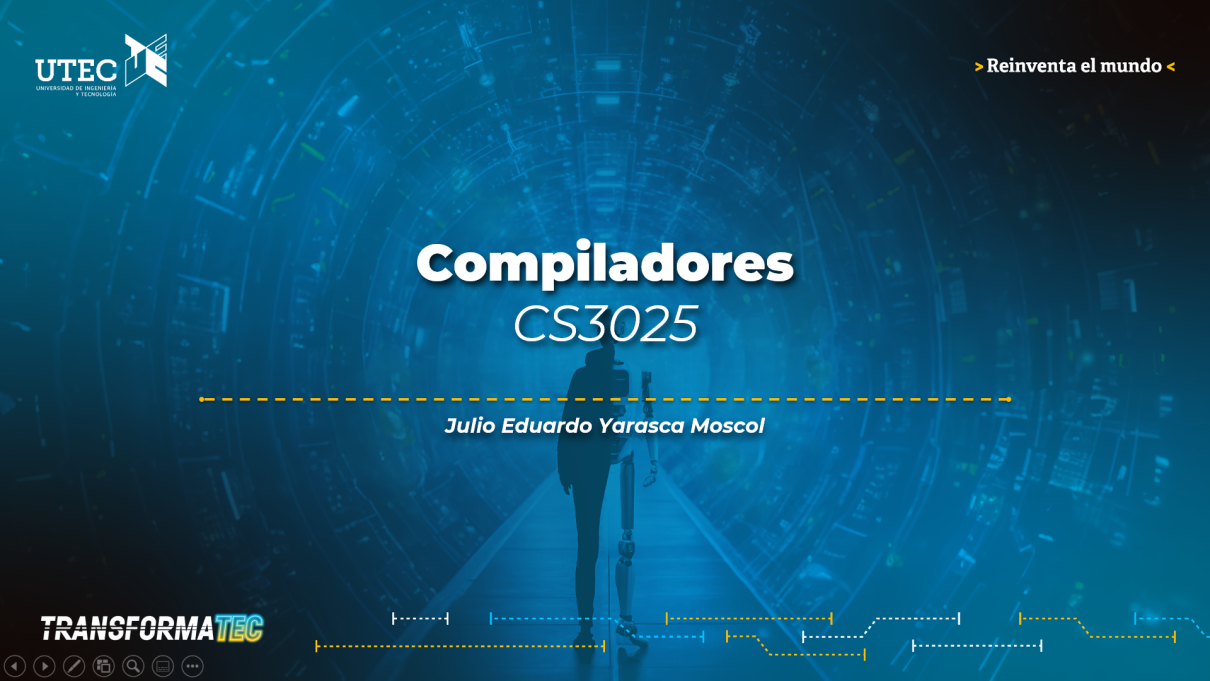
UTEC
UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

> Reinventa el mundo <

Compiladores

CS3025

Julio Eduardo Yarasca Moscol



TRANSFORMATEC

UTEC
UNIVERSIDAD DE INGENIERÍA TECNOLÓGICA

> Reinventa el mundo <

Indice:

- Unidad 5: Generación de código
  - 5.1. Llamadas a procedimientos y métodos en envío.
  - 5.2. Compilación separada; vinculación. Selección de instrucciones.
  - 5.3. Asignación de registros. Optimización por rendija (peephole)

TRANSFORMATEC

Logro de la sesión:

Identificar todos los pasos esenciales para convertir automáticamente código fuente en código ensamblador o otros lenguajes de bajo nivel. Generar código de bajo nivel para llamadas a funciones en lenguajes modernos.

Lenguaje ensamblador X86

Es un lenguaje de programación de bajo nivel diseñado para programar directamente procesadores de la familia **x86-64**, como los fabricados por Intel y AMD.

- Específico del hardware de arquitectura **x86-64**.
- Cercano al lenguaje máquina.
- Uso intensivo de registros de 64 bits.
- Permite control total del sistema y acceso a recursos de hardware.
- Arquitectura **CISC** (Complex Instruction Set Computing) con instrucciones complejas y variadas.

Arquitectura

- La arquitectura **x86-64** cuenta con **16 registros de propósito casi general**, desde `%rax` hasta `%r15`. Algunos mantienen funciones tradicionales, como `%rsp` para la pila y `%rbp` como base del marco de pila.
- Los ocho registros originales (`%rax`, `%rbx`, etc.) fueron diseñados con roles específicos (por ejemplo, `%rax` como acumulador), pero actualmente la mayoría pueden usarse de forma intercambiable, salvo en instrucciones particulares como las de manejo de cadenas (`%rsi`, `%rdi`).
- Cada registro de 64 bits tiene versiones de menor tamaño: `%eax` (32 bits), `%ax` (16 bits), y `%al`/ `%ah` (8 bits), lo que permite operar eficientemente con distintos tamaños de datos.

64-bit	32-bit	16-bit	8-bit	Nombre	Convención	Uso General
%rax	%eax	%ax	%al	Acumulador	Caller-saved	Valor de retorno de funciones. Acumulador para operaciones aritméticas y lógicas.
%rbx	%ebx	%bx	%bl	Base	Callee-saved	Registro general persistente. Usado frecuentemente para punteros o valores que deben conservarse entre llamadas.
%rcx	%ecx	%cx	%cl	Contador	Caller-saved	Cuarto argumento en llamadas de función. Contador de bucles y desplazamientos.
%rdx	%edx	%dx	%dl	Datos	Caller-saved	Tercer argumento. También para multiplicaciones/divisiones (resultado alto).
%rsi	%esi	%si	%sil	Source	Caller-saved	Segundo argumento. Fuente en operaciones de copia como `movs`.
%rdi	%edi	%di	%dil	Destination	Caller-saved	Primer argumento. Destino en operaciones de copia/comparación.
%rbp	%ebp	%bp	%bpl	Base Pointer	Callee-saved	Marco base de pila. Útil para variables locales y acceso a argumentos.
%rsp	%esp	%sp	%spl	Stack Pointer	—	Apunta al tope de la pila. Se modifica con `push`, `pop` y llamadas.

64-bit	32-bit	16-bit	8-bit	Nombre	Convención	Uso General
%r8	%r8d	%r8w	%r8b	—	Caller-saved	Quinto argumento de función. Registro general adicional.
%r9	%r9d	%r9w	%r9b	—	Caller-saved	Sexto argumento de función. Uso general.
%r10	%r10d	%r10w	%r10b	—	Caller-saved	Registro temporal. Se usa para cálculos o punteros intermedios.
%r11	%r11d	%r11w	%r11b	—	Caller-saved	Registro temporal. Ayuda en llamadas o direcciones indirectas.
%r12	%r12d	%r12w	%r12b	—	Callee-saved	Registro persistente entre llamadas. Útil en loops y estructuras.
%r13	%r13d	%r13w	%r13b	—	Callee-saved	Similar a %r12. Punteros, contadores, referencias.
%r14	%r14d	%r14w	%r14b	—	Callee-saved	General persistente. Frecuente en estructuras recursivas.
%r15	%r15d	%r15w	%r15b	—	Callee-saved	Uso general. Recurso auxiliar para estructuras grandes.

Dos sintaxis para el lenguaje ensamblador x86-64

- **Sintaxis AT & T (usada en GNU/Linux y herramientas como GAS)**
  Esta sintaxis tradicional utiliza registros con prefijo `%`. El orden de operandos es **fuente primero, destino después**, es decir, la instrucción indica que el contenido del primer operando se mueve o copia al segundo. Además, usa sufijos para indicar el tamaño de la operación, por ejemplo, `q` para 64 bits, `l` para 32 bits, `w` para 16 bits y `b` para 8 bits.
- **Sintaxis Intel (usada en Windows y documentación oficial de Intel)**
  Esta sintaxis, común en ensambladores como MASM, invierte el orden de los operandos: **destino primero, fuente después**. No usa el prefijo `%` en los registros y suele usar sufijos como `QWORD`, `DWORD`, `WORD`, y `BYTE` para especificar tamaños.

Registros

Cómo acceder:

Los registros son accesibles directamente por su nombre:

- `movq %rbx, %rax`
- `addq $5, %rcx`

Usos generales:

- Almacenar operandos para cálculos rápidos.
- Pasar argumentos a funciones según la convención ABI.
- Guardar direcciones de memoria (punteros).
- Almacenar valores temporales y resultados.

Memoria

Cómo acceder:

- Acceso mediante registros apuntando a memoria: `movq (%rbx),%rax`
- Almacenar valores con desplazamientos: `movq%rdx, 8(%rcx)`
- Uso de etiquetas: `movq var(%rip),%rax`

Usos generales:

- Guardar variables globales y estáticas.
- Almacenar estructuras o arreglos.
- Intercambio de datos con periféricos o memoria externa.

Pila

Cómo acceder:

- Uso de `pushq` y `popq`: `pushq %rax`, `popq %rbx`
- Acceso con desplazamientos relativos a `%rsp` o `%rbp`:
  - movq -8(%rbp), %rax
  - movq %rdx, 16(%rsp)

Usos generales:

- Almacenar variables locales de funciones.
- Guardar direcciones de retorno para llamadas a funciones.
- Pasar argumentos adicionales.
- Preservar registros durante llamadas (callee-saved).

Datos estáticos

- En ensamblador **x86-64**, la directiva `.data` se utiliza para definir regiones de datos estáticos, similares a variables globales. Estas regiones permanecen en memoria durante toda la ejecución del programa.
- Dentro de la sección `.data`, se pueden usar directivas como `.byte`, `.word`, `.long` y `.quad` para reservar 1, 2, 4 y 8 bytes respectivamente, asignándoles valores iniciales.
- Se pueden usar **etiquetas** (labels) para nombrar direcciones específicas de memoria, lo cual facilita su referencia posterior en las instrucciones del programa.

Datos estáticos

```
.data

var:
.byte 64    ; reserva 1 byte en memoria, lo llama var, y guarda el número 64

y:
.long 30000 ; reserva 4 byte en memoria, lo llama y, y guarda el número 30000

print_fmt:
.string "%ld\n"; reserva en memoria los bytes que representan la cadena de
                 formato "%ld\n", incluyendo al final un byte nulo (\0).
```

Memoria

- Los procesadores modernos compatibles con **x86-64** pueden direccionar memoria usando **direcciones de 64 bits**, lo que permite acceder a un espacio de direcciones mucho mayor que en arquitecturas anteriores.
- El modo de direccionamiento de **x86-64** permite calcular direcciones de la forma: `registro_base + registro_índice * escala + desplazamiento`, donde la **escala** puede ser 1, 2, 4 u 8.
- Estos modos pueden utilizarse en muchas instrucciones como `mov`, permitiendo copiar datos entre registros y memoria usando combinaciones válidas de registros de 64 bits (`%rax`, `%rsi`, `%rbx`, etc.) y desplazamientos constantes.

Memoria

- `mov (%rbx),%eax`
  Carga 4 bytes desde la dirección almacenada en `%rbx` hacia el registro `%eax` (registro de 32 bits).
- `mov %rbx, var`
  Mueve el contenido del registro de 64 bits `%rbx` a la dirección constante etiquetada como `var`.
- `mov -4(%rsi),%eax`
  Carga 4 bytes desde la dirección calculada como `%rsi` menos 4 y los guarda en `%eax`.
- `mov %cl, (%rsi,%rax,1)`
  Mueve el contenido de `%cl` al byte ubicado en la dirección calculada como `%rsi` más `%rax`.
- `mov (%rsi,%rbx,4),%edx`
  Carga 4 bytes desde la dirección calculada como `%rsi` más 4 veces `%rbx` hacia `%edx`.

Sufijos

- En x86-64, el **tamaño del dato** que se accede en memoria puede deducirse del **registro usado**; por ejemplo, usar `%rax` implica 8 bytes, `%eax` implica 4 bytes, `%ax` implica 2 bytes, y `%al` implica 1 byte.
- Cuando no hay un registro que indique el tamaño (como en `mov $2, (%rbx)`), el ensamblador no puede inferir automáticamente el tamaño, generando ambigüedad.
- Para resolverlo, se usan **prefijos de tamaño**: `b` (byte, 1 byte), `w` (word, 2 bytes), `l` (long, 4 bytes), y `q` (quad, 8 bytes), que indican explícitamente cuántos bytes se deben mover.

Sufijos

- `movb $2, (%rbx)`
  Mueve el valor 2 al byte ubicado en la dirección almacenada en `%rbx`.
- `movw $2, (%rbx)`
  Mueve el valor 2 como entero de 16 bits a los 2 bytes que empiezan en la dirección almacenada en `%rbx`.
- `movl $2, (%rbx)`
  Mueve el valor 2 como entero de 32 bits a los 4 bytes que empiezan en la dirección almacenada en `%rbx`.

Instrucciones de movimiento de datos

- **mov — Mover**
  Copia el dato del primer operando (registro, memoria o constante) al segundo (registro o memoria). No permite transferencias directas memoria a memoria; para eso se usa un registro intermedio.
- *Sintaxis:* mov <fuente>, <destino>
- *Ejemplos:*
  - `mov %rbx, %rax` — copia valor de RBX a RAX;
  - `movb $5, var` — escribe el valor 5 en la dirección de memoria etiquetada como var.

Instrucciones de movimiento de datos

- **push — Empujar en la pila**
  Coloca su operando en la parte superior de la pila, decrementando primero el puntero de pila `%rsp` y luego almacenando el valor. La pila crece hacia direcciones bajas.
- *Sintaxis:* push <reg> *o* push <mem> *o* push <constante>
- *Ejemplos:*
  - `push %rax` — empuja el valor de RAX en la pila;
  - `push var(,1)` — empuja 8 bytes en la dirección `var` en la pila.

Instrucciones de movimiento de datos

- **pop — Extraer de la pila**
  Extrae el elemento superior de la pila y lo coloca en el operando especificado. Luego incrementa el puntero de pila `%rsp`.
- *Sintaxis:* pop <reg> *o* pop <mem>
- *Ejemplos:*
  - `pop %rdi` — extrae el tope de pila en RDI;
  - `pop (%rbx)` — extrae el tope de pila en la memoria apuntada por RBX.

Instrucciones de movimiento de datos

- **lea — Cargar dirección efectiva**
  Calcula la dirección efectiva del operando fuente y la carga en el registro destino, sin acceder al contenido de memoria. Útil para cálculos de punteros o aritmética simple.
- *Sintaxis:* lea <mem>, <reg>
- *Ejemplos:*
  - `lea (%rbx,%rsi,8),%rdi` — carga en RDI la dirección RBX + 8 * RSI;
  - `lea val,%rax` — carga en RAX la dirección de `val`.

Instrucciones aritméticas y lógicas

- **add — Suma de enteros**
  Suma el primer operando al segundo y almacena el resultado en el segundo. Máximo un operando puede ser memoria.
- *Sintaxis:* add <fuente>, <destino>
- *Ejemplos:*
  - `add $10, %rax` — suma 10 a RAX;
  - `addb $10, (%rax)` — suma 10 al byte en la dirección almacenada en RAX.

Instrucciones aritméticas y lógicas

- **sub — Resta de enteros**
  Resta el primer operando del segundo y almacena el resultado en el segundo. Máximo un operando puede ser memoria.
- *Sintaxis:* sub <fuente>, <destino>
- *Ejemplos:*
  - `sub %ah, %al` — AL = AL - AH;
  - `sub $216, %rax` — resta 216 a RAX.

Instrucciones aritméticas y lógicas

- **inc, dec — Incremento y decremento**
  Incrementa o decrementa el operando en uno.
- *Sintaxis:* inc <operando> / dec <operando>
- *Ejemplos:*
  - `dec %rax` — resta uno a RAX;
  - `incq var` — incrementa en uno el entero de 64 bits almacenado en `var`.

Instrucciones aritméticas y lógicas

- **imul — Multiplicación de enteros**
  Multiplica operandos y almacena el resultado en el segundo o tercer operando (debe ser registro).
- *Sintaxis:*
  - `imul <reg>, <reg>`
  - `imul <mem>, <reg>`
  - `imul <con>, <reg>, <reg>`
  - `imul <con>, <mem>, <reg>`
- *Ejemplos:*
  - `imul (%rbx),%rax` — multiplica RAX por el valor 64 bits en memoria en RBX, resultado en RAX;
  - `imul $25,%rdi,%rsi` — RSI = RDI * 25.

Instrucciones aritméticas y lógicas

- **idiv — División de enteros**
  Divide el entero 128 bits RDX:RAX entre el operando; cociente en RAX, resto en RDX.
- *Sintaxis:* idiv <reg> / idiv <mem>
- *Ejemplos:*
  - `idiv %rbx` — divide RDX:RAX entre RBX;
  - `idivw (%rbx)` — divide RDX:RAX entre el valor en memoria en RBX.

Instrucciones aritméticas y lógicas

- **and, or, xor — Operaciones lógicas bit a bit**
  Realizan la operación lógica bit a bit entre operandos, resultado en el primer operando.
- *Sintaxis:*
  - `and <reg>, <reg>`
  - `and <mem>, <reg>`
  - `and <reg>, <mem>`
  - `and <con>, <reg>`
  - `and <con>, <mem>`
- *Ejemplos:*
  - `and $0x0f, %rax` — limpia todos los bits de RAX excepto los últimos 4;
  - `xor %rdx, %rdx` — pone RDX en cero.

Instrucciones aritméticas y lógicas

- **not — No lógico bit a bit**
  Niega todos los bits del operando (invierte bits).
- *Sintaxis:* not <reg> / not <mem>
- *Ejemplo:* `not %rax` — invierte todos los bits de RAX.

Instrucciones aritméticas y lógicas

- **neg — Negar (complemento a dos)**
  Calcula la negación en complemento a dos del operando.
- *Sintaxis:* neg <reg> / neg <mem>
- *Ejemplo:* `neg %rax` — RAX = -RAX.

Instrucciones aritméticas y lógicas

- **shl, shr — Desplazamientos a izquierda y derecha**
  Desplazan bits a la izquierda (shl) o derecha (shr) llenando con ceros. El desplazamiento puede ser constante de 8 bits o en el registro %cl.
- *Sintaxis:*
  - `shl <con8>, <reg>`
  - `shl <con8>, <mem>`
  - `shl %cl, <reg>`
  - `shl %cl, <mem>`
  - `shr <con8>, <reg>`
  - `shr <con8>, <mem>`
  - `shr %cl, <reg>`
  - `shr %cl, <mem>`
- *Ejemplos:*
  - `shl $1, %rax` — multiplica RAX por 2 (sin overflow);
  - `shr %cl, %rbx` — divide RBX por 2 elevado a CL.

Instrucciones de flujo de control

- **jmp — Salto incondicional**
  Transfiere el control a la instrucción etiquetada.
- *Sintaxis:* jmp <etiqueta>
- *Ejemplo:* `jmp begin` — salta a la instrucción con la etiqueta `begin`.

Instrucciones de flujo de control

- **j* — Saltos condicionales**
  Saltos basados en el resultado de la última operación aritmética o comparación (`cmp`). Las condiciones se reflejan en los flags del registro RFLAGS.
- *Sintaxis:*
  - `je <etiqueta>` — salto si es igual (zero flag = 1)
  - `jne <etiqueta>` — salto si no es igual
  - `jz <etiqueta>` — salto si el resultado fue cero (equiv. a `je`)
  - `jg <etiqueta>` — salto si mayor (signed)
  - `jge <etiqueta>` — salto si mayor o igual (signed)
  - `jl <etiqueta>` — salto si menor (signed)
  - `jle <etiqueta>` — salto si menor o igual (signed)
- *Ejemplo:*
  - `cmp %rbx, %rax`
  - `jle listo` — si RAX $\leq$ RBX, salta a `listo`.

Instrucciones de flujo de control

- **cmp — Comparación**
  Resta el primer operando al segundo y actualiza los flags, sin modificar los operandos. Es usada comúnmente antes de un salto condicional.
- *Sintaxis:*
  - `cmp <reg>, <reg>`
  - `cmp <mem>, <reg>`
  - `cmp <reg>, <mem>`
  - `cmp <con>, <reg>`
- *Ejemplo:*
  - `cmpb $10, (%rbx)` — compara el byte en memoria en RBX con 10;
  - `je bucle` — salta a `bucle` si son iguales.

Instrucciones de flujo de control

- **call, ret — Llamada y retorno de subrutina**
  `call` guarda la dirección de retorno en la pila y transfiere el control a la subrutina. `ret` restaura la dirección desde la pila y retorna el control.
- *Sintaxis:*
  - `call <etiqueta>`
  - `ret`
- *Ejemplo:*
  - `call imprimir_mensaje` — salta a la subrutina `imprimir_mensaje`, guardando el punto de retorno;
  - `ret` — vuelve a la instrucción después de la llamada.

Instrucciones auxiliares

- **.text — Sección de código**
  Indica el inicio de la sección de código ejecutable del programa, donde se colocan las instrucciones en ensamblador que serán ejecutadas.
- **.globl main — Símbolo global**
  Declara la etiqueta `main` como global, permitiendo que el enlazador la reconozca como punto de entrada del programa.
- **Modo de direccionamiento `(%rip)`**
  Se refiere a una dirección relativa al registro `RIP` (puntero de instrucción que apunta a la siguiente instrucción). Este modo, llamado *RIP-relative addressing*, permite acceder a datos (variables, etiquetas, constantes) mediante una dirección calculada sumando un desplazamiento al valor actual de `RIP`. Es muy útil para código *position-independent* (PIC), ya que el acceso es relativo al flujo de ejecución y no absoluto.

printf@PLT

- El primer argumento de `printf` es un puntero a la cadena de formato (`char *`), que debe colocarse en el registro `RDI`.
- Los argumentos siguientes, según los especificadores de formato en la cadena, se pasan en los registros `RSI`, `RDX`, `RCX`, etc.
- Antes de llamar a funciones variádicas como `printf`, es necesario poner `RAX = 0`, ya que este registro indica al ABI la cantidad de registros de punto flotante usados para argumentos.
- La llamada a `printf@PLT` se realiza a través de una entrada en la **Procedure Linkage Table (PLT)**, que funciona como un stub o trampolín.
- La primera vez que se invoca esta entrada, la PLT llama al **dynamic linker** para que localice la dirección real de la función en la librería compartida.
- Esa dirección se almacena para futuras llamadas, las cuales ya apuntarán directamente a la función real, evitando pasar nuevamente por el dynamic linker.

Instrucciones auxiliares

- `leave`
  Es una instrucción que se usa típicamente al final de una función. Equivale a:
  - `movq %rbp, %rsp`
  - `popq %rbp`

  Se utiliza para deshacer el "prólogo"de la función y preparar el retorno.
- `.section .note.GNU-stack,,@progbits`
  Es una directiva para el ensamblador que indica que esta unidad de compilación no necesita una pila ejecutable.

Ejemplo

```
.data
print_fmt: .string "%ld\n"
.text
.globl main
main:
pushq %rbp
movq %rsp, %rbp
sub  $16, %rsp
movq $10, %rax
movq %rax, -8(%rbp)
movq -8(%rbp), %rax
pushq %rax
movq $5, %rax
movq %rax, %rcx
```

```
popq %rax
cmpq %rcx, %rax
setl %al
movq -8(%rbp), %rax
pushq %rax
movq $5, %rax
movq %rax, %rcx
popq %rax
addq %rcx, %rax
movq %rax, %rsi
leaq print_fmt(%rip), %rdi
movl $0, %eax
call printf@PLT
jmp endif_0
else_0:
movq -8(%rbp), %rax
```

```
pushq %rax
movq $5, %rax
movq %rax, %rcx
popq %rax
subq %rcx, %rax
movq %rax, %rsi
leaq print_fmt(%rip), %rdi
movl $0, %eax
call printf@PLT
endif_0:
movl $0, %eax
leave
ret
.section .note.GNU-stack,"",@progbits
```

Ejemplo

```
.data
print_fmt: .string "%ld\n"
.text
.globl main
main:
pushq %rbp
movq %rsp, %rbp
sub $64, %rsp
movq $1, %rax
movq %rax, -32(%rbp)
movq $0, %rax
movq %rax, -8(%rbp)
movq $1, %rax
movq %rax, -16(%rbp)
while_0:
movq -32(%rbp), %rax
pushq %rax
movq $10, %rax
movq %rax, %rcx
popq %rax
cmpq %rcx, %rax
setl %al
movzbq %al, %rax
cmpq $0, %rax
je endwhile_1
movq -8(%rbp), %rax
pushq %rax
movq -16(%rbp), %rax
movq %rax, %rcx
popq %rax
addq %rcx, %rax
movq %rax, -24(%rbp)
movq -16(%rbp), %rax
movq %rax, -8(%rbp)
movq -24(%rbp), %rax
movq %rax, -16(%rbp)
movq -8(%rbp), %rax
movq %rax, %rsi
leaq print_fmt(%rip), %rdi
movl $0, %eax
call printf@PLT
movq -32(%rbp), %rax
pushq %rax
movq $1, %rax
movq %rax, %rcx
popq %rax
addq %rcx, %rax
movq %rax, -32(%rbp)
jmp while_0
endwhile_1:
movl $0, %eax
leave
ret
.section .note.GNU-stack,"",@progbits
```

Preguntas y cierre de la sesión

- ¿Cuál es el propósito del registro %rax en las llamadas a función?
- ¿Qué registro apunta al tope de la pila?
- ¿Qué hace el registro %rbp en el contexto de una función?
- ¿Qué registros se usan para pasar los argumentos de una función en Linux x86-64?
- ¿Qué diferencia hay entre %rsp y %rbp?

UTEC

UTEC
UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

> Reinventa el mundo <

GRACIAS

JULIO EDUARDO YARASCA MOSCOL

TRANSFORMATEC