Universidad Nacional de Colombia

Ingeniería de Sistemas y Computación
2025966 Lenguajes de Programación (02)

Tarea 14
Diseño del computador

Integrantes:

Javier Andrés Tarazona Jiménez	jtarazonaj@unal.edu.co
Eder José Hernández Buelvas	ehernandezbu@unal.edu.co
Juan Sebastián Muñoz Lemus	jumunozle@unal.edu.co
David Felipe Marin Rosas	dmarinro@unal.edu.co

Mayo 19 de 2025

Tabla de Contenidos

1 Nombre Computador

El computador presentado se denomina **ORISC-I**, una fusión entre las palabras *Oráculo* y **RISC**. El término *Oráculo*, según la literatura, hace referencia a un lugar sagrado donde se acudía en busca de respuestas confiables. Esta metáfora representa nuestro objetivo: construir una máquina eficiente, precisa y confiable, capaz de ejecutar operaciones con mayor rapidez que un ser humano.

Por otro lado, **RISC** (Reduced Instruction Set Computer) alude a la arquitectura utilizada, caracterizada por un conjunto reducido de instrucciones simples pero suficientes, lo que permite optimizar la ejecución, facilitar la implementación y favorecer el rendimiento general del sistema.

2 Nombre de la empresa

La empresa se denomina **UN Sparkle**, donde *UN* hace referencia a la Universidad Nacional de Colombia, institución a la que pertenecen los autores de este proyecto, y *Sparkle* significa "destello" en inglés. Este nombre simboliza nuestra intención de proyectar nuestras ideas y desarrollos como *destellos de innovación*, capaces de generar un impacto positivo en el ámbito académico y tecnológico.

3 Diseño del computador

El computador propuesto está basado en la arquitectura de Von Neumann. En primer lugar, se cuentan con los buses que van a conectar con todos los componentes que se presentan posteriormente.

3.1 Buses

El computador cuenta con tres buses principales que permiten la comunicación entre la Unidad Central de Proceso (CPU), la Memoria Principal (RAM) y la Unidad de Entrada/Salida. Estos buses son esenciales para implementar el modelo de arquitectura Von Neumann, en el que todos los componentes comparten un medio común de transmisión de información.

Bus de Datos

Este bus se encarga de transportar las palabras de datos (de 64 bits) entre los distintos módulos del sistema. Utilizado para:

- Transferir instrucciones desde memoria hacia el registro de instrucciones (IR).
- Transferir datos entre registros, memoria, y dispositivos de E/S.
- Intercambiar operandos entre la CPU y la ALU.

El bus de datos es bidireccional, ya que tanto la CPU como la memoria y la E/S pueden escribir o leer a través de él.

Bus de Control

El bus de control transporta señales de sincronización y comandos que determinan el tipo de operación a realizar en cada ciclo del procesador. Estas señales incluyen:

- Lectura de memoria
- Escritura de memoria
- Lectura/Escritura de E/S
- Activación de ALU
- Ciclo de instrucción (fetch, decode, execute)

El bus de control también incluye señales de interrupción, reset y reloj, esenciales para el control del flujo de ejecución.

Bus de Direcciones

Este bus se utiliza para especificar la ubicación de memoria o dispositivo con el que se desea interactuar. En este computador, las direcciones son de 24 bits, lo cual permite direccionar hasta $2^{24} = 16{,}777{,}216$ posiciones (16 MiB) distintas. Este bus es unidireccional, va desde la CPU hacia la memoria o dispositivos de E/S, y es usado durante operaciones de lectura o escritura para indicar la dirección objetivo.

3.2 Memoria Principal (RAM)

- Se trata de una memoria direccionable de 24 bits, lo que permite un total de:

$$2^{24} = 16,777,216 \,\text{posiciones} = 16 \,\text{MiB de memoria direccionable}$$

- Cada posición puede almacenar una palabra (8 Bytes).
- Está conectada a la Unidad de Servicio mediante los tres buses (datos, direcciones y control).
- Toda palabra de memoria contiene 64 bits, tanto para datos como para instrucciones. En caso de datos más pequeños (enteros de 32 bits o 8 bits), se usa alineamiento y enmascaramiento de bits.
- Es así que las direcciones de memoria van de:

$$0x000000 \,\text{a}\, 0xFFFFFF$$

3.2.1 Espacio reservado para pila

El computador incluye un **modelo de pila** gestionado por el registro especial `SP` (Stack Pointer). La pila es una estructura **LIFO (Last In, First Out)** utilizada para almacenar temporalmente datos como:

- Direcciones de retorno en llamadas a subrutinas (`CALL`, `RET`)
- Registros temporales
- Variables locales

La pila **crece hacia direcciones de memoria menores**, es decir, cada operación `PUSH` decrementa el `SP`, y cada `POP` lo incrementa. Las instrucciones `PUSH Rn` y `POP Rn` permiten guardar o recuperar el contenido de un registro en la cima de la pila, respectivamente.

- Las direcciones van de la 2.031.616 a la 16.777.215 (14.745.599 direcciones)

$$0x1F0000 \,\text{a}\, 0xFFFFFF$$

3.2.2 Espacio de datos

- Espacio para datos estáticos o globales.
- Es la parte media de la memoria.
- Las direcciones van de la 131.072 a la 2.031.615 (1.900.543 direcciones).

$$0x020000 \,\text{a}\, 0x1EFFFF$$

3.2.3 Espacio reservado para E/S

- Ver más detalles en la Unidad de Entrada y salida.
- Se ubica desde la dirección 65.536 hasta la 131.071 (65.536 direcciones).

$$0x010000 \,\text{a}\, 0x01FFFF$$

3.2.4 Espacio reservado para código

- En este espacio se van a guardar las instrucciones.
- Este espacio es la parte inferior de la memoria.
- Se ubica en la parte baja de la memoria, con direcciones de 0 a 65535 (65.536 direcciones):

$$0x000000 \text{ a } 0x00FFFF$$

3.3 Unidad de Entrada y Salida (E/S)

- Administra la interacción con dispositivos periféricos.
- Opera tanto para entrada (recepción de datos) como para salida (envío de datos).
- Se comunica con la consola.
- El computador implementa **E/S mapeada en memoria (Memory-Mapped I/O)**, es decir, los dispositivos periféricos comparten el mismo espacio de direcciones que la memoria RAM.
- Se reserva un rango de direcciones para la comunicación con periféricos conectados a la Unidad de E/S. Esto permite acceder a los dispositivos usando instrucciones estándar, sin necesidad de instrucciones exclusivas de E/S.

3.4 Unidad Central de Proceso (CPU)

La CPU está compuesta por los dos módulos esenciales en un procesador clásico:

- **Unidad Aritmético-Lógica (ALU):**
  - Ejecuta operaciones matemáticas y lógicas.
  - Contiene los **registros de propósito general**, además de los registros PC, SP, IR, ESTADO. Ver detalle del campo `ESTADO` más adelante.
- **Unidad de Control (UC):**
  - Incluye el **decodificador de instrucciones**, el **program counter (PC)** y el **registro de instrucciones (IR)**.
  - Interpreta el `opcode` de cada palabra de instrucción y coordina el flujo de datos en el sistema.

Ambas unidades están en **comunicación constante** para avanzar en la ejecución del programa, decodificando y ejecutando instrucciones de 64 bits mediante los buses conectados de datos, direcciones y control.

3.4.1 Registros

- Codificados en 5 bits → **32 registros disponibles**.
- Se incluyen registros de propósito general (`R0` a `R31`), además de registros especiales:
  - `R0:` `PC`: Program Counter: Apunta a la dirección de la próxima instrucción.
  - `R1:` `SP`: Stack Pointer.
  - `R2:` `IR`: Instruction Register: Almacena la instrucción actual que está siendo ejecutada.
  - `R3:` `ESTADO`: Banderas que permiten obtener información sobre los resultados.

Bit	Nombre	Significado
0	`C`	Resultado igual a cero (`== 0`)
1	`P`	Resultado estrictamente positivo (`> 0`)
2	`N`	Resultado estrictamente negativo (`< 0`)
3	`D`	**Desbordamiento (Overflow)**: el resultado no cabe en 64 bits
4	—	**Reservado**
5	—	**Reservado**
6	—	**Reservado**
7	—	**Reservado**

Table 1: Banderas del registro ESTADO

- `R4` **a** `R31`: Propósito general.

Registro	Codificación
`R0: PC`	00000
`R1: SP`	00001
`R2: IR`	00010
`R3: ESTADO`	00011
`R4`	00100
⋮	⋮
`R31`	11111

Table 2: Tabla de registros

3.5 Ciclos del Computador

El ordenador va a pasar por los ciclos de `fetch-decode-execute`:

3.5.1 Fetch

- La dirección de la siguiente instrucción se encuentra en el **PC** (`R0`).
- Se accede a la memoria, se lee la instrucción (64 bits) y se guarda en el **IR** (`R2`).
- El `PC` se incrementa en 1.

3.5.2 Decode

- La Unidad de Control extrae el `opcode`, los registros y los operandos desde el **IR**.
- Determina el tipo de instrucción (I, R, J), identifica la instrucción y prepara el entorno.

3.5.3 Execute

- Según el tipo de instrucción:
  - **Tipo R**: la ALU ejecuta la operación entre registros.
  - **Tipo I**: se accede a memoria usando direccionamiento con desplazamiento o directo.
  - **Tipo J**: el `PC` se actualiza con la nueva dirección de salto.
- Si aplica, se actualiza el registro **ESTADO** (`R3`) con los indicadores del resultado.

3.6 Instrucciones

Las instrucciones del procesador están codificadas en palabras de 64 bits.
Se aplcian las siguientes convenciones:

- **R**: es para registros.
- **M**: es para dirección de memoria.
- **V***: es para valores inmediatos. Ocupa los 32 bits menos significativos de la instrucción. A no ser de que se especifique lo contrario.

3.6.1 Formatos de instrucción

Dado que la longitud del opcode es variable dependiendo de la instrucción, Se han diseñado los siguientes tipos:

De Código completo
Instrucciones que realizan acciones de control

	Opcode
Bits	63 a 0

Intrucciones tipo R:
Operaciones entre registros.

	Opcode	**R**
Bits	63 a 5	4 a 0

	Opcode	**R**	**R'**
Bits	63 a 10	9 a 5	4 a 0

Intrucciones tipo I:
Operaciones con valores inmediatos o memoria.

	Opcode	**R**	**M**
Bits	63 a 29	28 a 24	23 a 0

	Opcode	**R**	**V**
Bits	63 a 37	36 a 32	31 a 0

Intrucciones tipo J:
Operaciones que saltan a una dirección de memoria o la manejan.

	Opcode	**M**
Bits	63 a 24	23 a 0

3.6.2 Tablas de instrucciones

De Código completo

Código Ensamblador	**Instrucción**	**Descripción**

`PROCRASTINA`	No hace nada (instrucción vacía), sirve para no hacer nada por un ciclo de reloj, se usa para temas de sincronización, o simplemente para que el PC se parezca un poco más a sus diseñadores.	Si quieres ver un anime en vez de hacer la tarea o irte a jugar r6 en vez de adelantar un trabajo.
`VUELVE`	Vuelve desde la subrutina desde la que se encuentre, esto al hacer DESAPILA dos veces, la primera para sacar el registro ESTADO y la segunda para sacar el PC, la primera vez que DESSAPILA guarda el resultado en ESTADO y la segunda en PC.	DESAPILA R y luego: $ESTADO \leftarrow R$, después DESAPILA R y luego: $PC \leftarrow R$
`PARA`	Detiene la ejecución del programa	Se detiene el ciclo de instrucciones, ya sea deteniendo completamente la CPU o entrando en un estado de espera (IDLE)

Intrucciones tipo R:

Código Ensamblador	**Instrucción**	**Descripción**
`SUMA R, R'`	Suma dos registros R y R'	$R \leftarrow R + R'$
`RESTA R, R'`	Resta dos registros	$R \leftarrow R - R'$
`MULT R, R'`	Multiplica dos registros	$R \leftarrow R \times R'$
`DIVI R, R'`	Realiza la división entera entre dos registros	$\lfloor R \leftarrow R/R' \rfloor$
`AND R, R'`	Conjunción bit a bit	$R \leftarrow R \wedge R'$
`OR R, R'`	Disyunción bit a bit	$R \leftarrow R \vee R'$
`XOR R, R'`	XOR bit a bit	$R \leftarrow R \oplus R'$
`COMP R, R'`	Compara dos registros	Actualiza el registro ESTADO
`COPIA R, R'`	Copia el valor de un registro a otro	R ← R'
`NOT R`	Negación bit a bit	R ← ∼R (inversión de bits)
`LIMP R`	Limpia (pone en cero) el contenido de un registro	R ← 0
`INCRE R`	Incrementa un registro en 1	$R \leftarrow R + 1$
`DECRE R`	Decrementa un registro en 1	$R \leftarrow R - 1$
`APILA R`	Decrementa (la pila crece hacia abajo en la memoria) el registro SP (R1) y hace push del registro R en la pila (lo guarda)	$R1(SP) \leftarrow R1 - 1$ y guarda en la dirección que contiene SP a R: $*SP \leftarrow R$

`DESAPILA R`	Guarda el contenido de la dirección que está en SP en el registro R, y luego incrementa SP (R1). (la pila crece hacia abajo, esto es un pop)	$R \leftarrow *SP(R1)$, luego incrementa SP: $R1(SP) \leftarrow R1 + 1$

Intrucciones tipo I:

Código Ensamblador	Instrucción	Descripción
`CARGA R, M`	Carga desde el espacio de memoria M al registro R	$R \leftarrow M$
`GUARD R, M`	Guarda el contenido del registro R en el espacio de memoria M	$M \leftarrow R$
`SIREGCERO R M`	Salta a M si el registro R es igual a 0.	Verifica si R == 0 y al confirmarlo hace un SALTA M.
`SIREGNCERO R M`	Salta a M si un registro R no es igual a cero.	Verifica si R $\neq$ 0 y al confirmarlo hace un SALTA M.
`ICARGA R, V`	Carga un valor inmediato V a un registro R.	$R \leftarrow V$
`ISUMA R, V`	Suma inmediata de un valor V a un registro R	$R \leftarrow R + V$
`IRESTA R, V`	Resta inmediata de un valor V a un registro R	$R \leftarrow R - V$
`IMULT R, V`	Multiplica un registro por un inmediato	$R \leftarrow R \times V$
`IDIVI R, V`	Realiza la división entera de un registro R entre un valor inmediato V	$R \Leftarrow \lfloor R/V \rfloor$
`IAND R, V`	Conjunción AND entre un registro R y un valor inmediato V	$R \leftarrow R \wedge R'$
`IOR R, V`	Disyunción OR entre un registro R y un valor inmediato V	$R \leftarrow R \vee R'$
`IXOR R, V`	XOR bit a bit entre un registro R y un valor inmediato V	$R \leftarrow R \oplus R'$
`ICOMP R, V`	Compara un registro R con un valor inmediato V y guarda el resultado en R3 (Registro de estado)	Se realiza la resta $R - V$ y se indica si el resultado es positivo ($R > V$), negativo ($R < V$) o cero ($R = V$).

Intrucciones tipo J:

Código Ensamblador	Instrucción	Descripción

`SALTA M`	Salta incondicionalmente a la dirección M, es decir, guarda la dirección M en el R0 (PC) program counter para que en el siguiente ciclo continúe desde ahí, después de saltar deja el registro ESTADO en ceros. (No guarda contexto, no guarda el registro ESTADO, lo limpia después de saltar)	$PC(R0) \leftarrow M$
`LLAMA M`	Llama a la subrutina en la dirección M, primero guarda el PC(R0) en la pila al hacer un APILA, después hace lo mismo con el registro ESTADO, finalmente, pone en el PC (R0) el valor de M, donde empieza la subrutina (SALTA M).	APILA PC(R0), APILA ESTADO(R3) y luego: $PC \leftarrow M$
`SICERO M`	Salta si el resultado de la última comparación fue cero (C=1), al verificar el bit C del R3 (estado).	Verifica si C=1 y al confirmarlo hace un SALTA M
`SINCERO M`	Salta a M si el resultado de la última comparación no fue cero (C=0)	Verifica si C=0 en el R3 (estado) y al confirmarlo hace un SALTA M
`SIPOS M`	Salta si el resultado de la última comparación fue positivo (P=1)	Verifica si P=1 y al confirmarlo hace un SALTA M
`SINEG M`	Salta si el resultado de la última comparación fue negativo (N=1)	Verifica si N=1 en el R3 (estado) y al confirmarlo hace un SALTA M
`SIOVERFL M`	Salta si el resultado de la última comparación u operación tuvo overflow (D=1)	Verifica si D=1 en R3 (estado) y al confirmarlo hace un SALTA M.
`SIMAYOR M`	Salta si el resultado de la última operación es de mayor que, es decir, al comparar, el resultado fue positivo y no fue igual a cero. ($P = 1 \wedge C = 0$)	Verifica si los flags P y C cumplen las condiciones y al confirmarlo hace un SALTA M.
`SIMENOR M`	Salta si el resultado de la última operación es de menor que, es decir, al comparar, el resultado fue negativo. (N=1)	Verifica si el flag N en R3 (estado) está activado, al confirmarlo hace un SALTA M.

`INTERRUP M`	Dispara una interrupción por software, guardando el estado actual del program counter en la pila junto con los flags o registros relevantes, luego salta a la rutina de interrupción M.	Se guardan los estados del sistema, se hace un APILA PC y se SALTA M.

3.7 Detalles

3.7.1 Variables

Las variables **globales** se almacenan en el segmento de **datos estáticos**, ubicado entre las direcciones `0x020000` y `0x1EFFFF`. Estas variables existen durante toda la ejecución del programa.

Las variables **locales**, por el contrario, se almacenan dinámicamente en la **pila**, gestionada por el registro `SP`. Se reservan y liberan mediante instrucciones que modifican el puntero de pila, y solo existen mientras la subrutina correspondiente está en ejecución.

3.7.2 Tipos de datos

Representación de enteros

En el computador **ORISC-I**, los números enteros se representan en formato de *complemento a dos* sobre **64 bits**.

Esto permite representar eficientemente tanto enteros positivos como negativos, sin necesidad de un bit de signo separado ni lógica adicional para la ALU.

El *bit más significativo* (MSB, por sus siglas en inglés) indica el signo del número:

- Si el MSB es **0**, el número es **positivo**.
- Si el MSB es **1**, el número es **negativo** (en complemento a dos).

Representación de números en punto flotante

Para representar números en punto flotante, el computador **ORISC-I** adopta el estándar **IEEE 754 de doble precisión** (*64 bits*).

Cada palabra puede codificar un número real con la siguiente estructura:

- **1 bit** de signo.
- **11 bits** para el exponente, con un *sesgo de 1023*.
- **52 bits** para la mantisa (fracción).

En versiones futuras, se planea implementar instrucciones especializadas para operaciones aritméticas en coma flotante, tales como:

`FADD`, `FSUB`, `FMUL`, `FDIV`.

Manejar valores inmediatos grandes

Las instrucciones de tipo **I** permiten cargar *inmediatos* de hasta **24 bits**. Para valores negativos, se realiza automáticamente una *extensión de signo* a 64 bits utilizando el formato de *complemento a dos*.

Si se desea cargar valores inmediatos mayores a 24 bits, se propone una estrategia extendida basada en dos instrucciones secuenciales:

- Una instrucción tipo `LUI` (*Load Upper Immediate*) carga los bits superiores.
- Una instrucción tipo `ORI` (*OR Immediate*) combina los bits inferiores.

Pero en un diseño posterior o si se requiere para extender funcionalidades.

4 Módulo Enlazador–Cargador

El Módulo Enlazador–Cargador se encarga de preparar y montar en la memoria RAM un programa en código máquina generado por el ensamblador. A continuación se detallan las responsabilidades específicas de cada parte:

4.1 Enlazador

El módulo Enlazador se encarga de combinar varios módulos objeto y bibliotecas para generar un único programa ejecutable. Sus tareas principales incluyen:

1. **Análisis de dependencias:**
   Examina las referencias a símbolos externos en cada módulo objeto y determina la secuencia de enlace adecuada.

2. **Construcción de la tabla de símbolos global:**
   Fusiona las tablas de símbolos de todos los módulos, asignando direcciones provisionales a los símbolos definidos y marcando los externos no resueltos.

3. **Resolución de referencias cruzadas:**
   Para cada símbolo externo referenciado, busca su definición en la tabla global y actualiza las referencias en el código objeto con direcciones relativas o absolutas según convenga.

4. **Gestión de bibliotecas:**
   Incluye en el ejecutable sólo los módulos necesarios de las bibliotecas estáticas, minimizando el tamaño final.

5. **Generación del fichero ejecutable:**
   Ensambla los segmentos relocados en un único fichero de salida con formato ejecutable, incluyendo una cabecera que especifica:

   - Dirección de carga predeterminada.
   - Punto de entrada (símbolo `main`).
   - Tamaños y ubicaciones de segmentos de código, datos y pila.

6. **Producción de información de depuración (opcional):**
   Incorpora en el ejecutable tablas de símbolos y mapas de segmentos para facilitar el debugging y análisis de rendimiento.

4.2 Cargador

1. **Lectura de cabecera del programa:**
   El cargador extrae del fichero de objeto la tabla de segmentos (código, datos, pila) y la tabla de símbolos relocables, incluyendo:

   - Dirección de carga propuesta por el usuario.
   - Tamaños de cada segmento.
   - Entradas relocables (instrucciones o datos con direcciones relativas).

2. **Relocalización de direcciones:**
   Para cada entrada relocable, modifica el campo de dirección sumándole la *dirección de carga base*: dirección_absoluta = dirección_relativa + base_carga De esta forma convierte todas las referencias relativas a direcciones absolutas en el espacio de memoria.

3. **Resolución de símbolos externos:**
   Si el programa referenció símbolos definidos en otros módulos, busca en la tabla de símbolos del enlazador las direcciones finales y parchea las instrucciones correspondientes con sus direcciones absolutas.

4. **Escritura en memoria RAM:**

   - Copia el segmento de código (instrucciones de 64 bits) en las celdas de memoria a partir de la dirección de carga base.
   - Copia el segmento de datos inicializados en las celdas posteriores.
   - Ajusta el puntero de pila (`SP`) al espacio reservado para la pila, calculado en función de la dirección de carga.

5. **Inicialización de registros de control:**

   - Carga el registro `PC` (contador de programa) con la dirección de entrada especificada en la cabecera.
   - Borra o inicializa el registro `ESTADO` (banderas) a cero.

6. **Verificación y chequeo de errores:**
   Comprueba que ninguna dirección de carga exceda los límites de la memoria disponible y que no haya solapamiento entre segmentos. En caso de error, devuelve un código de fallo.

7. **Transferencia de control:**
   Una vez completada la carga, transfiere el control de la ejecución al `PC`, iniciando el ciclo de fetch–decode–execute de la CPU.

5 Algoritmos de Prueba

5.1 Suma de N enteros consecutivos

Un algoritmo de tipo iterativo que suma los primeros N enteros consecutivos usando un ciclo `for` y una variable acumuladora:

```
int suma(int n) {
  int i, suma = 0;
  for (i = 1; i <= n; i++) {
    suma += i;
  }
  return suma;
}
```

Ahora, lo escribiremos en terminos del lenguaje ensamblador para nuestro computador ORISC-I:

```
// Suma de N enteros consecutivos
// ORISC-I
// Entradas: R4 = N >= 1
// Salidas: R5 = Suma
// Variables: R6 = i, R5 = Suma

// Inicializar variables en 0
LIMP R5
LIMP R6
ICARGA R6, 0
COPIA R5, R6
// Inicio del loop
LOOP_START:
  COMP R4, R6
  SICERO LOOP_END
```

```
    // Incrementar i
    INCRE R6
    // Sumar i al resultado (suma)
    SUMA R5, R6
    // Volver al inicio del loop
    SALTA LOOP_START
  LOOP_END:
  // Fin del loop
  // El resultado está en R5
```

5.2 Conteo de ceros en un arreglo

Un algoritmo iterativo que recorre un arreglo y cuenta cuántas celdas contienen el valor cero:

```
// Conteo de ceros en un arreglo
// ORISC-I
// Entradas: R3 = dirección base del arreglo; R4 = longitud N >= 1
// Salidas: R5 = Cantidad de ceros
// Variables: R6 = índice i, R7 = valor actual, R8 = registro auxiliar, R9 = cero

// Inicializar contadores y registros
LIMP R5          // R5 = 0 (contador de ceros)
LIMP R6          // R6 = 0 (índice)
ICARGA R6, 0     // Asegura R6 = 0
COPIA R5, R6     // R5 = 0
LIMP R9          // R9 = 0 (valor cero para comparación)

// Inicio del loop
LOOP_START:
  COMP R6, R4
  SICERO LOOP_END      // Si i == N, terminar
  // Calcular dirección del elemento: R8 = R3 + R6
  COPIA R8, R3
  SUMA R8, R6
  // Cargar elemento en R7
  CARGA R7, R8
  // Comparar elemento con 0
  COMP R7, R9
  SICERO INCREMENT_COUNT
  SALTA CHECK_END
  INCREMENT_COUNT:
    INCRE R5            // count++
  CHECK_END:
    INCRE R6            // i++
    SALTA LOOP_START
LOOP_END:
// Resultado: R5 contiene el número de ceros
```

5.3 Factorial de un número

```
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
```

```
}
```

Ahora, lo escribiremos en terminos del lenguaje ensamblador para nuestro computador ORISC-I:

```
; factorial(n) en ensamblador
; Entrada: R4 = n
; Salida:  R5 = factorial(n)

; factorial(n) en ensamblador
; Entrada: R4 = n
; Salida:  R5 = factorial(n)

; factorial(n)
; Entrada: R4 = n
; Salida: R5 = factorial(n)

FACTORIAL:
    APILAR R4              ; Guardar n en la pila
    ICARGA R6, #0         ; Cargar 0 en R6
    COMP   R4, R6         ; Comparar n con 0 (actualiza R3)
    SICERO   BASE_CASE      ; Si n == 0, ir a caso base
    ICARGA R7, #1         ; Cargar 1 en R7
    RESTA   R4, R7         ; R4 = R4 - 1
    LLAMA  FACTORIAL      ; Llamar recursivamente factorial(n-1)
    DESAPILAR   R6             ; Recuperar n original en R6
    COPIA   R7, R5         ; Guardar factorial(n-1) en R7
    COPIA   R5, R6         ; Poner n en R5
    MULT   R5, R7         ; R5 = n * factorial(n-1)
    VUELVE                ; Retornar con resultado en R5
BASE_CASE:
    ICARGA R5, #1         ; factorial(0) = 1
    DESAPILAR   R4             ; Restaurar n antes de retornar
    VUELVE
```

6 Referencias

[1] M. Bichler, S. Merting, and A. Uzunoglu, "Assigning Course Schedules: About Preference Elicitation, Fairness, and Truthfulness," arXiv preprint arXiv:1812.02630, 2018. [En línea]. Disponible en: https://arxiv.org/abs/1812.02630