Arquitecturas Intel 64 e IA-32

Organización del Computador II

28 de agosto de 2022

Actividad Individual

La presente es una guía de ejercicios preparatorios para el parcial individual. Hay ejercicios de diferente grado de dificultad que buscan practicar la programación en Assembler y C, y repasar los conceptos teóricos relevantes para la parte práctica de la materia.

Indicamos con estrellas el nivel de dificultad relativo. No es necesario que completen toda la guía pero si que traten de elegir los ejercicios en función de lo que más comprenden o no. Es decir, si se sienten seguros pueden hacer sólo los difíciles, pero si hay temas que no les cierran pueden elegir los más básicos y luego pasar a los intermedios o difíciles.

1. Guía de evaluación

Esta es una breve enumeración de algunos temas que vimos en la práctica y nos interesa que hayan comprendido. No son los únicos temas que se van a evaluar pero les pedimos que en su preparación para el examen se aseguren manejar los siguientes.

- Armado de stackframe
- Alineación de la pila en 32 y 64 bits
- Convención C en 32 y 64 bits:
  - Registros a preservar
  - Pasaje de parámetros a función en 32 y 64 bits
- Llamadas a función con CALL y retorno con RET impacto en la pila
- Saber dibujar todo lo que se guarda en la pila en llamadas anidadas y no anidadas a función en C y Assembler
- Conversión de enteros a float y double para operar matemáticamente
- Manejo de datos:
  - Lectura/escritura/almacenamiento de distintos tamaños de datos (8, 16. 32 y 64 bits).
  - Manejo de caracteres, strings, enteros, floats y doubles.
  - Tamaño de los registros y lecturas a memoria con tamaño adecuado al dato.
- Punteros en C

2. Ejercicios

2.1. Assembler en 64 bits

Ejercicio 1 ⋆
Escriba un programa únicamente en Assembler que tome un vector de 16 enteros de 32 bits y los sume.

Tip 1 Utilice una etiqueta en la sección `data` para crear el vector.

Tip 2 Revise los defines vistos en la primer clase práctica (DB, DW, DQ).
¿Cuál es el apropiado para un vector de enteros de 32 bits?

Ejercicio 2 ⋆
Sobre el ejercicio anterior,

a) ¿Cómo realizó las lecturas de los enteros desde memoria? Si utilizó un registro indique cuantos bits tiene.

b) Ahora, escriba un programa en Assembler que lea los enteros en un registro de 32 bits y los sume en 64 bits

c) ¿Qué ventaja encuentra en sumarlos en 64 bits?

d) Asuma que los registros de 64bits están completos con unos. ¿Que pasaría cuando muevan los enteros de 32 bits y sume en 64 bits? Escriba un código para solucionar el problema.

Ejercicio 3 ⋆

a) Escriba un ejercicio en Assembler que lea los enteros desde memoria y guárdelos en registros distintos antes de sumarlos.

b) ¿Debería armar el stackframe? ¿Qué registros tuvo que preservar? ¿Alcanzaron los registros? Si no: ¿Cómo lo resolvió?

Tip Explore la clase práctica número 3 sobre armado de stackframe y preservación de registros en 64 bits.

Ejercicio 4 ⋆

a) Repita una vez más el ejercicio pero guardando todos los enteros en la pila, sacarlos de la pila e irlos sumando. Volver a guardar el resultado en la pila. Trabaje los números en 32 bits.

b) ¿Debería armar el stackframe?

c) ¿Cuál es el tamaño en bytes del ancho de la pila? ¿Cómo quedaría organizada? Pruebe pusheando con registros de 32 y 64 bits.

Ejercicio 5 ⋆⋆

a) Escriba un programa que tenga dos funciones: `llamadora` e `invocada`. Desde `llamadora` debe llamar a la función `invocada`.

`invocada` recibe un entero entre 1 y 10, y debe retorna la dirección de memoria a ejecutar cuando resuma la `llamadora` más ese entero.

b) Utilice GDB para debuguear y verificar que la direccion de memoria retornada es correcta.

c) ¿Cómo puede encontrar las direcciones que corresponden a la pila en cada función?

d) Realice un seguimiento del uso de la pila, esto significa, haga un gráfico en lapiz y papel de la pila, indique direcciones, y contenido. Puede explorarla usando GDB.

2.2. C y Assembler

En la siguiente, sección deben escribir código en C y Assembler desde cero. Recuerden hacer la signatura de las funciones en C y desde C llamar a Assembler.

Observen que las funciones en C tienen la palabra `extern` y en Assembler, `global`. Esto permite exponer funciones de C a Assembler y viceversa.

Ejercicio 6 ⋆⋆
Dada una cadena de texto s definiremos su *codificación césar simplificada* respecto de un entero x de la siguiente manera:

La función recibe únicamente caracteres en mayúsculas (no lo necesitamos verificar, viene dado). Para explicarlo vamos a usar las funciones ord y chr de c.

```
Por cada letra c de la cadena:
    emitir chr(ord(c) + x)
```

`ord(c)` es el número de c en ASCII.

`chr(n)` es la letra correspondiente al número n en ASCII.
La función "vuelve a empezar desde el princpio" cuando se queda sin letras.

Ejemplo:

- `cesar("CASA", 3) = "FDVD"`
- `cesar("CALABAZA", 7) = "JHSHIHGH"`

Más información: https://es.wikipedia.org/wiki/Cifrado_César

Puede asumir que $0 \leq x < 26$. Opcionalmente resuelva el problema general.

Recuerden que todos los bytes que representan las letras estan dadas por la codificación ASCII. El 1 no corresponde precisamente al caracter Áḿayúscula. Fijense cual sería el valor binario que tomaría cada letra mayúscula en la codificación ASCII.

a) Escribir una función en C que dada una cadena de caracteres y un entero, devuelva su codificación césar simplificada.

b) Realice otra implementación en assembler respetando el ABI visto en la cátedra. Cree un pequeño programa en C que utilice esta implementación. Para la versión Assembler, no tiene que usar ord ni chr de C.

Ejercicio 7 ⋆
Escribir un programa en C que dados dos strings determine la longitud de su prefijo común más largo.

Ejemplos:

- `prefijo_de("Astronomia", "Astrologia") = 5` ("Astro")
- `prefijo_de("Pinchado", "Pincel") = 3` ("Pin")
- `prefijo_de("Boca", "River") = 0` ("")
- `prefijo_de("ABCD", "ABCD") = 4` ("ABCD")

Opcional Escribir un programa que dados dos strings devuelva "el segundo sin el prefijo común con el primero".

Para construir un nuevo string deberán hacer uso de memoria dinámica (`malloc`/`free`).

Ejemplos:

- `quitar_prefijo("Astro", "Astrologia") = "logia"`
- `quitar_prefijo("Pinchado", "Pincel") = "cel"`
- `quitar_prefijo("Boca", "River") = "River"`
- `quitar_prefijo("ABCD", "ABCD") = ""`

Ejercicio 8 ⋆⋆
Teniendo en cuenta estructuras en C:

```
#define NAME_LEN    21

typedef struct cliente_str {
    char nombre[NAME_LEN];
    char apellido[NAME_LEN];
    uint64_t compra;
    uint32_t dni;
} cliente_t;

typedef struct  __attribute__((__packed__)) packed_cliente_str {
    char nombre[NAME_LEN];
```

```
    char apellido[NAME_LEN];
    uint64_t compra;
    uint32_t dni;
} __attribute__((packed)) packed_cliente_t;
```

a) Dibuje un diagrama de organización en memoria de las estructuras `cliente_t` y `packed_cliente_t` marcando el espacio usado para alineación y padding.

b) Escriba un programa en Assembler que dado un arreglo `cliente_t[]` elija uno de sus elementos al azar.

c) Replique la lógica para un arreglo `packed_cliente_t[]`.

Tip Para la generación de números aleatorios puede utilizar la función `rand` provista por la biblioteca estándar de C.

Ejercicio 9 ⋆⋆
Una lista enlazada es una estructura de datos que nos permite almacenar datos como una secuencia de nodos. Cada nodo consiste de su valor y un puntero al nodo que lo sucede. Lo único que se necesita para tener acceso a toda la lista es un puntero al nodo que la encabeza (llamado `head`).

Considere la siguiente estructura y sus operaciones:

```
typedef struct node_str {
    struct node_t *siguiente;
    int32_t valor;
} node_t;

node_t* agregarAdelante(int32_t valor, node_t* siguiente);
node_t* agregarAdelante_asm(int32_t valor, node_t* siguiente);
int32_t valorEn(uint32_t indice, node_t* cabeza);
void destruirLista(node_t* cabeza);
```

a) La lista utiliza memoria dinámica. ¿Qué funciones de C deberá utilizar para administrar la memoria? ¿En que situaciones usaría cada una? Detalle la importancia de usarlas correctamente.

b) Implemente `agregarAdelante(int32_t, node_t*)` de forma que la expresión `head = agregarAdelante(123, head);` nos permita agregar el valor 123 al principio de la lista llamada `head`.

c) Implemente `agregarAdelante_asm(int32_t, node_t*)` en lenguaje ensamblador.

Opcional ⋆⋆⋆ Implemente `valorEn(uint32_t, node_t*)` en Assembler de forma que nos permita obtener el iésimo elemento de la lista.

Opcional ⋆⋆⋆ Implemente `destrurLista(node_t*)` en C o Assembler que dada la cabeza de la lista elimina toda la lista . Corrobore su correcto funcionamiento utilizando `valgrind`.

2.3. Ejercicios C y Assembler con código

Se incluye el código para completar los siguientes ejercicios que permiten un mejor entedimiento del funcionamiento de la pila.

Ejercicio 10 Tamaño del stack ⋆⋆⋆⋆

a) Observe y determine qué hace la función de `test_medir_stack_C` definida en `medir-stack/main.c`.

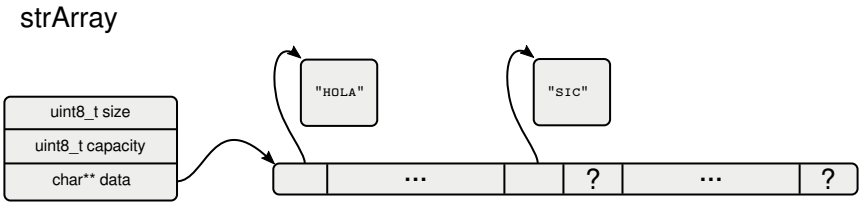
b) Suponga que se ejecuta `test_medir_stack_C(3)` desde `main_en_c`. Dibuje cómo quedaría compuesta la pila al final de la tercer llamada recursiva.

c) Implemente la función `medir_stack_asm` que cuenta las apariciones del RBP en la pila hasta que encuentra el marcador 0.

2.4. Ejercicios de Parcial

Ejercicio 11 mplementa un array dinámico de capacidad limitada. El mismo puede contener como máximo la cantidad de strings indicada en `capacity`. Los datos serán todos del tipo *String*

La estructura `str_array_t` contiene un puntero al arreglo identificado como *data* y la cantidad de elementos ocupados como *size*.




```
typedef struct str_array {
uint8_t size;
uint8_t capacity;
char** data;
} str_array_t;
```

Implementar las siguientes funciones en asm:

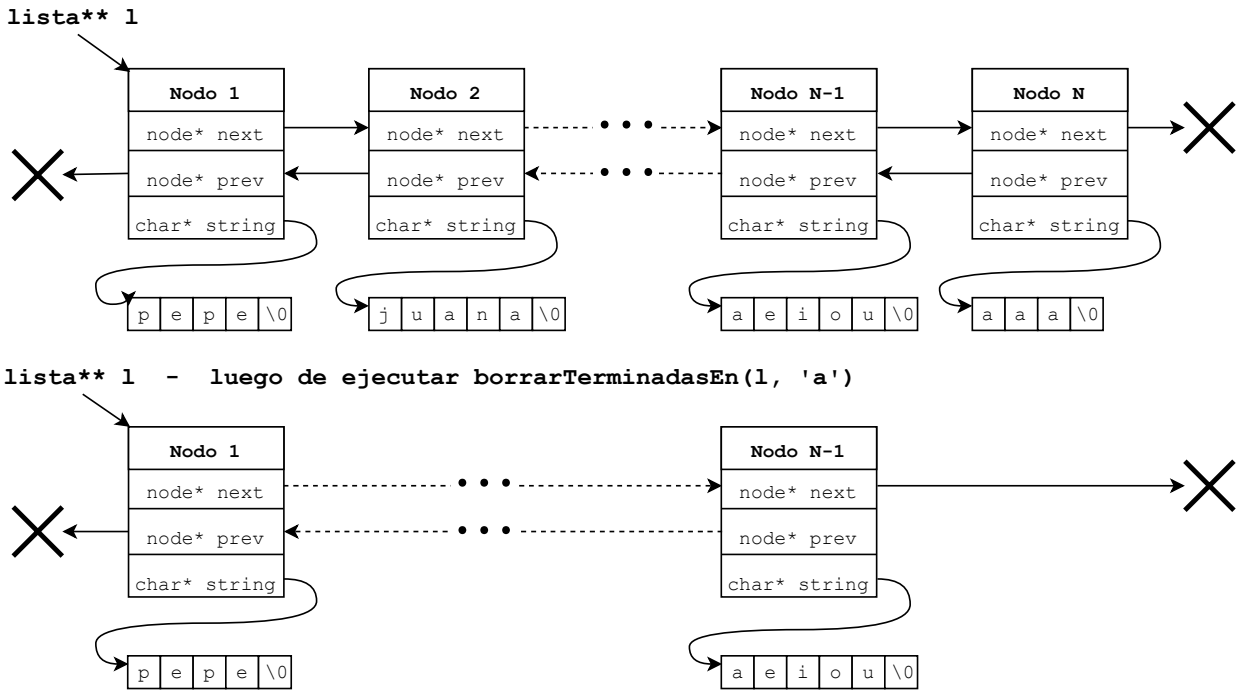
- `str_array_t* strArrayNew(uint8_t capacity)`
  Crea un array de strings nuevo con capacidad indicada por `capacity`.
- `uint8_t strArrayGetSize(str_array_t* a)`
  Obtiene la cantidad de elementos ocupados del arreglo.
- `char* strArrayGet(str_array_t* a, uint8_t i)`
  Obtiene el i-esimo elemento del arreglo, si `i` se encuentra fuera de rango, retorna 0.
- `char* strArrayRemove(str_array_t* a, uint8_t i)`
  Quita el i-esimo elemento del arreglo, si `i` se encuentra fuera de rango, retorna 0. El arreglo es reacomodado de forma que ese elemento indicado sea quitado y retornado.
- `void strArrayDelete(str_array_t* a)`
  Borra el arreglo, para esto borra todos los strings que el arreglo contenga, junto con las estructuras propias del tipo arreglo.

Ejercicio 12 Considerar una estructura de lista doblemente enlazada en donde cada nodo almacena un string de C, es decir, un arreglo de caracteres finalizado en el caracter nulo (`'\0'`).

```
typedef struct node_t {
    struct node_t *next;
    struct node_t *prev;
    char *string;
} node;
```

a. Escribir en ASM una función que reciba como parámetros `l`: *doble puntero a nodo* y `c`: *un caracter*, y borre todos los nodos, junto con la string, para los que el último caracter sea igual a `c`. Utilizar la función `free` para borrar tanto los nodos como las strings. Su aridad es: `void borrarTerminadasEn(node** l, char c)`. En el caso de borrar el primer elemento de la lista se debe actualizar el puntero recibido.

Resultado de aplicar borrarTerminadasEn(l, 'a')



b. Escribir en ASM la función `char* superConcatenar(node* n)`, que toma un puntero a un nodo y retorna la concatenación de todas las strings almacenadas en la lista.
Se cuenta con la función `uint32_t strlen(char* s)` que dada una string, retorna la cantidad de caracteres válidos de la misma.

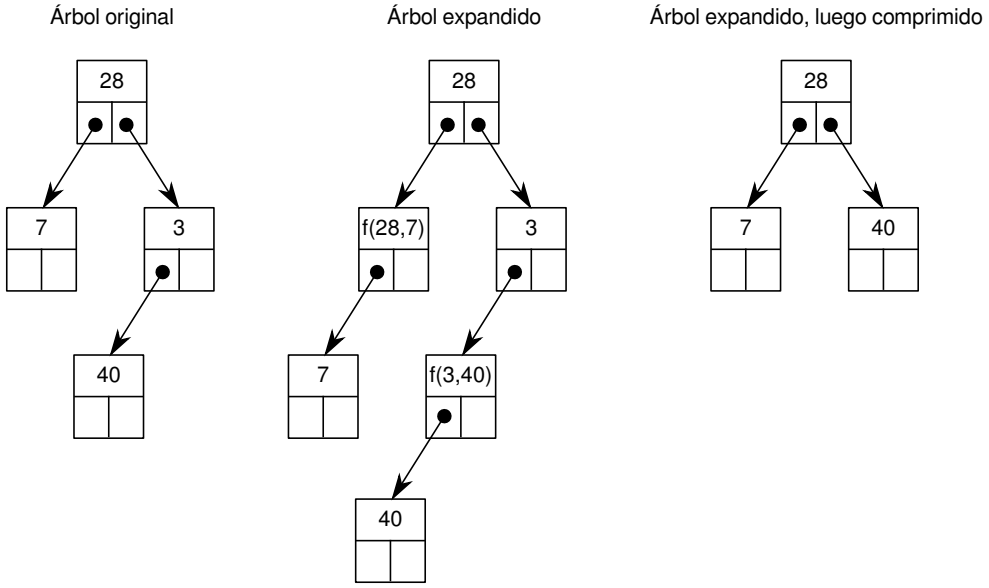
Ejercicio 13 Considerar un árbol binario de búsqueda con la siguiente estructura.

```
typedef struct node_t {
   node *izquierda;
   int value;
   node *derecha;
} node;
```

y un conjunto de funciones con la siguiente aridad: `typedef int func(node *a, node *b)`

a) (15p) Construir en ASM una función `void expandir(node* arbol, func f)` que recorra todo el árbol evaluando `f` para cada par (padre, hijo izquierdo) del árbol, agregando un nodo entre ambos con el resultado.

b) (25p) Construir en ASM una función `void comprimir(node *nodo)` para recorra todo el árbol realizando la siguiente acción: si el hijo de un nodo tiene exactamente un hijo, entonces será eliminado (conectando al nodo con su nieto).
Recomendación: utilizar una función auxiliar `void comprimir_aux(node **ppHijo)`.

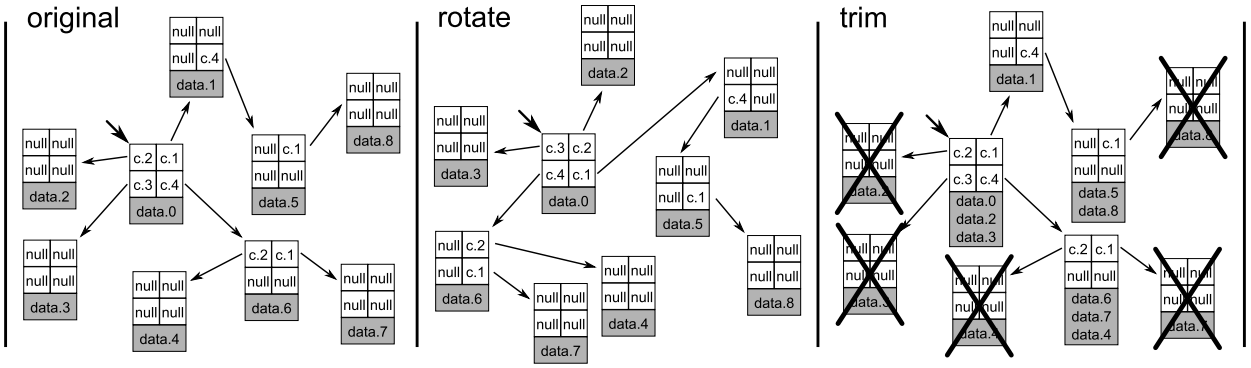


Ejercicio 14 (40 puntos)

Un sistema de control para satélites utiliza una curiosa estructura de representación para las posiciones de los mismos. La misma se almacena por cuadrantes de forma recursiva.

```
struct pos {
  pos* cuad1;
  pos* cuad2;
  pos* cuad3;
  pos* cuad4;
  info* data;
}
```

La estructura contiene un puntero a cada uno de los cuadrantes, que a su vez contienen un nuevo conjunto de cuadrantes de forma recursiva. Estos además contienen un puntero a `data`, utilizado para almacenar la información sobre los satélites.



(a) Implementar en ASM la función rotación (`void rotate(pos* cuad)`), que realiza un movimiento de los punteros a los cuadrantes de la siguiente manera: `cuad1` → `cuad2`, `cuad2` → `cuad3`, `cuad3` → `cuad4` y `cuad4` → `cuad1`. Esta rotación debe realizarse para el cuadrante **cuad** pasado por parámetro y para todos sus cuadrantes sucesivamente de forma recursiva.

(b) Implementar en ASM la función borrar hojas (`void trim(pos* cuad, void* f_unir)`), que se encarga de buscar todos los cuadrantes hoja (es decir aquellos en los cuales los punteros a cuadrantes son nulos) y unir sus datos con los de su padre por medio de la función `f_unir` pasada por parámetro. La aridad de la función es la siguiente: `info* f_unir(info* padre, info* hoja)`.

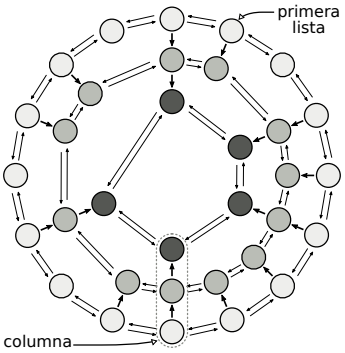
Nota: se sugiere implementar la función auxiliar `boolean esHoja(pos* cuad)`.

Ejercicio 15 Sea la siguiente estructura de listas doblemente enlazadas encadenadas entre si.

```
struct supernode {
supernode* abajo,
supernode* derecha,
supernode* izquierda,
int dato }
```

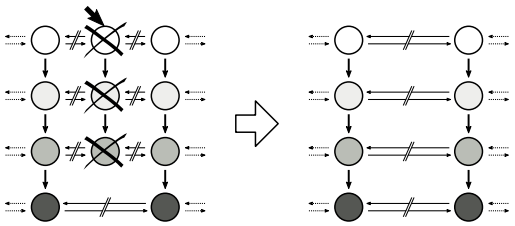
- todos los nodos pertenecen a una lista doblemente enlazada
- todos los nodos son referenciados desde algun otro nodo en otra lista (excepto en la primera)
- todas las listas respetan el orden de los nodos que las apuntan

ejemplo



Implementar en ASM las siguientes funciones.

a. `void borrar_columna(supernode** sn)`: Dada un doble puntero a nodo dentro de la primera lista, borra una columna de nodos. Modifica el doble puntero dejando un nodo valido de la primer lista.



b. `void agregar_abajo(supernode** sn, int d)` Agrega un nuevo nodo a la lista inmediata inferior del nodo apuntado. Considerar que el nodo donde agregar puede no tener vecinos inmediatos en la lista inferior.