Universidad Nacional Autónoma de México

Facultad de Ciencias

Organización y Arquitectura de Computadoras
2025-2

Práctica 08

Docentes:
José Galaviz Ricardo Pérez Ximena Lezama

Autores:
Fernanda Ramírez Juárez Ianluck Rojo Peña

Fecha de entrega: Domingo 20 de abril de 2025

Ejercicios.

Ejercicio 1.

¿Qué hace la rutina?

El programa realiza la operación: b^a. Utiliza lo valores `a` que representa el exponente, y `b`, que representa la base.

Como agregado, la explicación al flujo del programa es el siguiente:

1. En `main` cargmos los valores de `a` y `b` en los registros `$a0` y `$a1` e invocamos a la subrutina `mist_1`, que se encarga de realizar el cálculo de elevar `b` a la potencia `a`.
2. Esta subrutina usa `loop_1` que se repite `a` veces y en cada iteración llama a `mist_0` con el que llevamos la cuenta de la multiplicación.
3. El valor inicial del acumulador es 1 y en cada iteración se multiplica por la base `b`, logrando así la operación `b * b * ... * b` (repetido `a` veces).
4. Cargamos el resultado final a `$v0` y lo imprimimos junto con el mensaje `resultado` en consola.

Teniendo en cuenta que los valores ya están establecidos con $a = 5$ y $b = 4$, la salida al correr el programa es:

```
El resultado es: 1024
```

Ejercicio 2.

```cpp
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

//Funcion que implementa el coeficiente binomial n en k utilizando la identidad de Pascal
int coeficiente_binomial(int n, int k) {
  //Si k es mayor a n simplemente devolvemos 0
  if (k > n)
  return 0;
  //Caso base
  if (k == 0 || k == n)
  return 1;

  //Paso recursivo utilizando la identidad de Pascal
  return coeficiente_binomial(n - 1, k - 1) + coeficiente_binomial(n - 1, k);
}

int main() {
  int n, k;
  cout<< "Ingrese los valores para n, k: " << endl;
  cin >> n >> k;

  //Verificamos que n y k sean valores validos para calcular el coeficiente binomial
  if (n <= -1 || k <= 0) {
    cout << "[ERROR]: Los valores para n, k deben ser positivos.\n" << endl;
    return 0;
  }

  int r = coeficiente_binomial(n, k); //El resultado a de realizar el coeficiente
  cout << "El coeficiente binomial (" << n << " en " << k << ") es: " << r << endl;
}
```

Código 1: Código en C++ Coeficiente binomial n en k utilizando la identidad de Pascal

Preguntas.

1. ¿Qué es la Arquitectura Harvard Modificada (Modified Harvard Arquitecture)? ¿Cuáles son sus componentes?

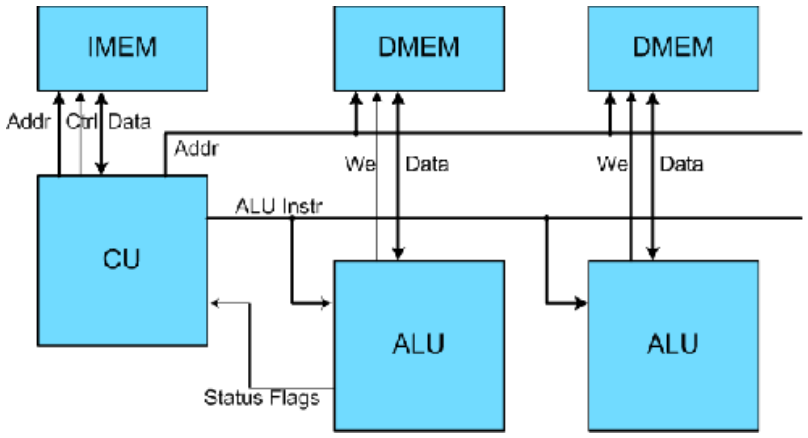
   La **Arquitectura Harvard Modificada** *(Modified Harvard Architecture)* es una variante de la arquitectura Harvard clásica que combina características de esta última con elementos de la arquitectura von Neumann para mejorar la flexibilidad y eficiencia en sistemas modernos, sin sacrificar la ventaja principal de Harvard: el acceso simultáneo a ambas memorias a través de buses separados.

   Diferencias:

   - **Harvard Clásica:** Separación total entre las memorias de instrucciones y de datos, con buses independientes y espacios de direcciones distintos.
   - **Von Neumann:** Memoria unificada para instrucciones y datos, con un solo bus, lo que puede generar cuellos de botella.
   - **Harvard Modificada:** Mantiene la separación física de memorias y buses, pero permite cierto grado de intercambio entre instrucciones y datos, especialmente para acceder a constantes almacenadas en la memoria de programa.

   Componentes principales de la arquitectura Harvard modificada.

   - **Memoria de Instrucciones *(Program Memory)*:**
     Almacena el código ejecutable (programa). Suele ser de solo lectura (ROM/Flash) pero puede permitir acceso para lectura de datos.
   - **Memoria de Datos *(Data Memory)*:**
     Almacena variables y datos temporales. Normalmente es de lectura/escritura (RAM).
   - **Buses separados:**
     Un bus para instrucciones (fetch) y otro para datos (load/store). Permite operaciones simultáneas (mayor velocidad que von Neumann).
   - **Unidad de Intercambio *(Data/Instruction Bridge)*:**
     Permite transferir datos entre memorias (como cargar constantes desde Flash a RAM). Es la principal mejora frente a la Harvard pura.
   - **CPU *(Unidad de Procesamiento Central)*:**
     Incluye ALU, registros y unidad de control. Puede acceder a ambas memorias en paralelo.
   - **Periféricos y Puertos de Entrada/Salida:**
     Módulos adicionales para comunicación *(UART, SPI, ADC, etc.)*.



- *Ventajas:*
  1) Mayor velocidad al leer instrucciones y datos en paralelo.
  2) Flexibilidad para manejar constantes almacenadas en la memoria de programa.
  3) Ideal para sistemas en tiempo real (DSP, microcontroladores).
- *Desventajas:*
  1) Mayor complejidad en el diseño.
  2) Coste más alto que arquitecturas simples como von Neumann.

Nota:

Esta arquitectura es popular en entornos embebidos donde el rendimiento y la optimización del acceso a memoria son esenciales, sin renunciar a cierta flexibilidad en el manejo de datos e instrucciones.

En muchos sistemas modernos, se utiliza una variante conocida como arquitectura split-cache, donde hay una única memoria física pero con cachés separadas para instrucciones y datos, simulando una Harvard modificada en los niveles inferiores del procesador. También existen variantes que permiten ejecutar instrucciones desde memoria de datos o acceder a memoria de programa como si fuera de datos, aumentando la flexibilidad del sistema.

2. En el contexto de las subrutinas, ¿Qué es el Wheeler Jump? Describe que pasos y para qué sirve el Wheeler Jump.

   El **Wheeler Jump** es una técnica utilizada en programación de bajo nivel, especialmente en arquitecturas antiguas como la EDSAC (una de las primeras computadoras), para implementar subrutinas (funciones o procedimientos) de manera eficiente cuando no existían instrucciones nativas para manejar llamadas a subrutinas (como CALL y RET en CPUs modernas). Fue propuesto por *David J. Wheeler*, uno de los pioneros de la computación.

   En sistemas primitivos sin pila de hardware ni instrucciones dedicadas para subrutinas, el Wheeler Jump permitía:

   a) Llamar a una subrutina guardando automáticamente la dirección de retorno.

   b) Retornar al punto de llamada después de ejecutar la subrutina.

   c) Era una solución ingeniosa para reutilizar código en entornos con recursos limitados.

Pasos del Wheeler Jump:

1) Almacenamiento de la dirección de retorno:
   - Antes de saltar a la subrutina, el programa modificaba la propia instrucción de salto dentro de la subrutina para que apuntase de vuelta a la dirección siguiente a la llamada.
   - Esto se hacía copiando el contador de programa (PC) actual en el operando del salto.
2) Ejecución de la subrutina:
   - La subrutina se ejecutaba normalmente.
   - Al final, incluía una instrucción de salto modificada dinámicamente para retornar.
3) Retorno automático:
   - Al llegar al salto modificado, la ejecución volvía a la instrucción siguiente a la llamada original.

Sin embargo, el **Wheeler Jump** contaba un ciertas limitaciones, comoq que dependía de auto-modificación de código, lo que hoy se considera mala práctica pues esto deriv en problemas de seguridad y mantenimiento. No permitía recursión, ya que solo se podía almacenar una dirección de retorno por subrutina, además que acceder y escribir en memoria era lento comparado con registros o pilas.

Fue posteriormente reemplazado por instrucciones como CALL/RET y el uso de pilas de hardware en arquitecturas siguietenes como x86 o ARM.

No obstante, es un precursor clave de los mecanismos modernos de manejo de subrutinas. Demostró cómo resolver problemas complejos con recursos limitados, un principio aún relevante en sistemas embebidos.

Hoy, el concepto persiste en formas más avanzadas, como los frames de pila en lenguajes de alto nivel, pero el Wheeler Jump sigue siendo un hito en la historia de la computación.

3. ¿Por qué se causa la vulnerabilidad del Buffer Overflow (tambien llamado String Overflow)? ¿En C, cuál es el método que causaba este error?

   Buffer Overflow *(o String Overflow)* ocurre cuando un programa escribe datos en un buffer (espacio de memoria reservado) más allá de su capacidad asignada, sobrescribiendo áreas adyacentes de la memoria. Esto puede corromper datos, alterar el flujo de ejecución del programa o incluso permitir la ejecución de código malicioso.

   Causas del Buffer Overflow:

   - **Falta de verificación de límites.**
     En lenguajes como C y C++, funciones que manejan buffers *(como strcpy, gets, scanf)* no verifican automáticamente el tamaño del dato copiado. Si el dato es más grande que el buffer, se produce un overflow.

- **Organización de la memoria.**

  En la pila (stack), los buffers locales comparten espacio con: variables locales, la dirección de retorno de funciones, datos de control del programa. Un desbordamiento puede sobrescribir la dirección de retorno, redirigiendo la ejecución del programa.

- **Arquitectura de pila (Stack).**

  En C, las variables locales y la dirección de retorno de funciones se almacenan en la pila (stack). Si un buffer local se desborda, puede sobrescribir la dirección de retorno, permitiendo a un atacante redirigir la ejecución a código malicioso.

Funciones inseguras en C (evitables):

Función	**Riesgo principal**
`strcpy`	Copia una cadena sin verificar límites. Riesgo de sobrescritura.
`gets`	Lee entrada del usuario sin límite de tamaño. Vulnerable a overflow. (Eliminada en C11)
`strcat`	Concatena sin verificar espacio restante en el buffer. Puede causar overflow.
`scanf` con `%s`	Sin especificar tamaño (`%64s`) puede desbordar el buffer.
`sprintf`	No verifica tamaño del buffer destino. Mejor usar `snprintf`.