OBLIGATORIO DE ARQUITECTURA DE COMPUTADORAS 2022

ArquiCalc

Índice

Descripción del problema

El objetivo es implementar una calculadora cuyo funcionamiento se basa en la notación polaca inversa.

La calculadora además de realizar las operaciones también puede detectar errores e indicarlos en un puerto diferente al que llamaremos *Bitácora*.

Los códigos de error posibles son:

- **16: CODIGO_EXITO:**
  - si la operación se pudo realizar con éxito
- **8: CODIGO_FALTAN_OPERANDOS**
  - Si la operación falló por falta de operandos en la pila. Tras ejecutarse este error la pila quedará vacía.
- **4: CODIGO_DESBORDAMIENTO_DE_PILA**
  - si la operación falló por desbordamiento de la pila (ya hay 31 operandos)
- **2: CODIGO_COMANDO_INVALIDO**
  - si no se reconoce el comando (comando inválido)

La calculadora tiene las siguientes funcionalidades:

- Colocar hasta 31 operandos enteros de 16 bits en la pila
- Evaluar las siguientes operaciones aritméticas:
  - Suma: Suma los 2 primeros elementos.
  - Resta: Resta al segundo elemento el primer elemento de la pila.
  - Producto: Realiza el producto de los 2 primeros elementos de la pila
  - División: Realiza la división del segundo elemento de la pila sobre el primero
  - Módulo: Obtiene el mod de la división del segundo elemento de la pila sobre el primero
- Evaluar las siguientes operaciones aritméticas binarias:
  - And: Realiza el and bit a bit entre los primeros 2 elementos de la pila
  - Or:Realiza el or bit a bit entre los primeros 2 elementos de la pila
  - Desplazamiento a la izquierda: Realiza op2 << op1
  - Desplazamiento a la derecha: Realiza op2 >> op1

Todas las operaciones descritas hasta ahora pueden provocar el error **CODIGO_FALTAN_OPERANDOS**

- Evaluar las siguientes operaciones unarias:
  - C_NEG
  - C_FACT
- Además tener los siguientes comandos:
  - C_TOP: muestra el tope de la pila
    - Puede provocar el error **CODIGO_FALTAN_OPERANDOS**

- C_DUMP: Realiza un volcado de la pila en el puerto de salida
- C_SWAP: Intercambia el tope de la pila con el elemento de debajo de él
    - Puede provocar el error **CODIGO_FALTAN_OPERANDOS**
- C_SUM: Suma todos los elementos de la pila
- C_DUP: Duplica el tope de la pila.
    - Puede provocar el error **CODIGO_FALTAN_OPERANDOS**
    - Puede provocar el error **CODIGO_DESBORDAMIENTO_DE_PILA**
- C_NUM: Inserta un nuevo número en la pila,
    - Recibe como parámetro el número desde el puerto de entrada ENTRADA
    - Puede provocar el error **CODIGO_DESBORDAMIENTO_DE_PILA**
- C_LOG: Establece el puerto de salida para la bitácora
    - Si no se define por el usuario se usará inicialmente el valor asignado en PUERTO_LOG_DEFECTO
- C_PORT: Establece el puerto de salida para los resultados
    - Si no se define por el usuario se usará inicialmente el valor asignado en PUERTO_SALIDA_DEFECTO
- C_CLS: Borra todo el contenido de la pila
- C_HALT: Detiene el procesamiento.

Además se cuenta con las operaciones binarias:

- C_SUMA: Realiza op2 + op1
- C_RESTA: Realiza op2 - op1
- C_MULT: Realiza op2 * op1
- C_DIV: Realiza op2 / op1
- C_MOD: Realiza op2 % op1
- C_AND: Realiza op2 & op1
- C_OR: Realiza op2 | op1
- C_SAL: Realiza op2 << op1
- C_SAR: Realiza op2 >> op1

Todas pueden provocar el error **CODIGO_FALTAN_OPERANDOS**

Descripción de la solución incluyendo detalle de las estructuras de datos y constantes utilizadas

Primero se realizó la implementación del código en C++.

Se adjunta el archivo con la solución propuesta.

El gran if

La misma consta en una entrada (que será un array) que se recorre con una iteración *while*. Dentro del while se cuenta con un if, else, donde en el if se controla si la entrada que llega es la definición del puerto de la bitácora y en la parte del else, se controla si esa entrada no corresponde a alguno de los otros comandos. El motivo por el que se separa el control de si la entrada corresponde al puerto de la bitácora es porque la bitácora imprime el log luego de haber cambiado el puerto, por lo que resultó necesario primero ver qué comando es para luego imprimir *0, C_LOG, parámetro*. Si la salida por la bitácora de ese cambio de puerto fuera antes de definir el puerto se podría haber colocarlo en el otro if con los otros comandos (de hecho así lo había definido y tuve que cambiarlo porque noté que en uno de los test el funcionamiento no se correspondía a como lo había pensado).

En cuanto a los otros comandos, antes que todos ellos se imprime el preprocesamiento, que corresponde a *0* y número de comando, y luego se fija con el gran if a qué comando corresponde. Si se necesita algún parámetro este se toma de la entrada y se imprime utilizando la función *log_parametro*(). Luego dentro de cada if, se llama a las funciones correspondientes para ejecutar la acción que corresponda en cada caso. Al final del gran if, se llama a *log_exitoso*() o a *log_comando_invalido*() en caso de que el comando ingresado no haya matcheado con ninguno de los que aparecen en los if.

Pueden ocurrir errores al manejar la pila tal cual se definió anteriormente, cuando esto ocurra será desde alguna de las funciones *apilar_ax*, *apilar_bx()*, que pueden disparar el error de desbordamiento de pila, así como las funciones *desapilar_hacia_ax()*, *desapilar_hacia_bx()* y *copiar_tope_a_ax()* pueden disparar el error de que faltan operandos.

Funciones

Para facilitar la lectura, comprensión y mantenimiento se usaron funciones ya desde el C++. Son algunas funciones específicas que enmascaran el manejo de la pila, de la entrada, y algunas otras operaciones como por ejemplo factorial(). Se contará con las mismas funciones al compilar manualmente al assembler.

Constantes

Para la solución se utilizaron constantes para todos los números, de tal forma de facilitar la lectura del código.

Las constantes definidas son las siguientes:

```
CODIGO_NUEVO_COMANDO equ 0
CODIGO_EXITO equ 16
CODIGO_FALTAN_OPERANDOS equ 8
CODIGO_DESBORDAMIENTO_DE_PILA equ 4
CODIGO_COMANDO_INVALIDO equ 2
```

Como se indicó más arriba, en las últimas 4 se definen los códigos para cada tipo de error. En el caso de CODIGO_NUEVO_COMANDO indica en la bitácora que se va a procesar otro comando de la entrada

```
C_NUM equ 1;
C_PORT equ 2;
C_LOG equ 3;
C_TOP equ 4;
C_DUMP equ 5;
C_DUP equ 6;
C_SWAP equ 7;
C_NEG equ 8;
C_FACT equ 9;
C_SUM equ 10;
C_CLS equ 254;
C_HALT equ 255;

C_SUMA equ 11;
C_RESTA equ 12;
C_MULT equ 13;
C_DIV equ 14;
C_MOD equ 15;
C_AND equ 16;
C_OR equ 17;
C_SAL equ 18;
C_SAR equ 19;
```

Son las funciones y comandos que puede realizar la calculadora. La explicación de cada una se encuentra en la página anterior.

Además también contamos con las siguientes constantes:

```
PILA_LLENA equ 60 ; hay que considerar al 0
COMIENZO_STACK equ -2
ADDRESS_STACK equ 0x02000
ENTRADA equ 10
PUERTO_SALIDA_DEFECTO equ 1
PUERTO_LOG_DEFECTO equ 2
```

Compilación

Todas las constantes anteriores deben ir en el apartado .data

Por otro lado en la sección .code se encuentra todo el código que se describe a continuación:

La compilación resultó lineal, se intentó pensar el código en C++ de tal forma que sea fácil de pensar su compilación. Se puede observar el loop general con el gran if principal, con un conjunto de if's else dentro pasados de manera casi lineal al assembler:

```
while (index < sizeof(ENTRADA)){
    if (ax == C_LOG){
        short comando = ax;
        index++;
        poner_siguiente_operando_en_ax(index);
        cout << "\nPuerto " << dx << ": 0 " << ...
    }else{
        log_preprocesamiento();
        if (ax == C_NUM){
            index++;
            poner_siguiente_operando_en_ax(index);
            log_parametro();
            apilar_ax();
        }else if(ax == C_PORT){
            index++;
            poner_siguiente_operando_en_ax(index);
            log_parametro();
            cx = ax;
        }
```

```
while_no_salir:
    cmp ax, C_LOG
    JNE procesar_operando_o_comando
    si_es_LOG:
        push ax
        call poner_siguiente_operando_comando_en_ax
        mov dx, ax ;
        pop ax
        call log_preprocesamiento
        mov ax, dx
        call log_parametro
    JMP fin_si

    procesar_operando_o_comando:
        call log_preprocesamiento
        si_es_NUM:
        cmp ax, C_NUM ;
        JNZ si_es_PORT;
            call poner_siguiente_operando_comando_en_ax
            call log_parametro
            call apilar_ax
        JMP fin_si;
        si_es_PORT:
        cmp ax, C_PORT
        JNZ si_es_TOP
            call poner_siguiente_operando_comando_en_ax
            call log_parametro
            mov cx, ax
        JMP fin_si
```

A la izquierda se puede ver el código en C++, que utiliza un while que se detiene cuando la entrada termina o cuando se recibe el comando C_HALT (debido al break que aparece en la correspondiente sección dentro del if anidado (el segundo)).
A la derecha su correspondiente compilación, como se puede ver se utilizó CMP para realizar las comparaciones.

- Podemos ver cómo se utiliza CMP para verificar si el comando actual es C_LOG, de ser así no se ejecutará el salto a procesar_operando_o_comando sino que se continuará hacia si_es_LOG, donde ya se empezarán a ejecutar las acciones correspondientes.
- Si CMP nos diera que ax != C_LOG entonces se ejecutará el salto procesar_operando_o_comando y en consecuencia, llamaremos a la función

log_preprocesamiento y luego se harán los análisis del if siguiente (en C++) o los posibles saltos en el assembler

- En el assembler podemos ver como también se utiliza CMP para verificar de qué comando se trata comparándolo por ejemplo con C_NUM, si coincide se ejecutarán las funciones poner_siguiente_operando_comando_en_ax, log_parametro y apilar_ax al igual que en el c++. Al finalizar se realiza el salto incondicinoal hacia la etiqueta *fin_si*. Si no se hubiera cumplido en el CMP que ax es C_NUM, entonces se pasa a la siguiente etiqueta si_es_PORT mediante el salto JNZ

Algunos lineamientos a seguir en el assembler:

Inicialmente se intentó seguir algunos lineamientos sobre el uso de los registros y memoria para mejor legibilidad y fácil manejo de los recursos al momento de programar ya que contamos con un CPU con registros con personalidad y varios de esos registros los necesitamos para funciones específicas.

- En ax se guardará el comando y también se usará como el operador 1
- En bx se usará el operador 2 (puede que en algún momento este sea el op1 y ax el op2)
- En dx se define la bitacora, inicialmente será PUERTO_LOG_DEFECTO
- En cx se define la salida de resultados,inicialmente PUERTO_SALIDA_DEFECTO
- Se definirá *si* para el 'índice' en la pila, empieza en -2 para que la primera inserción quede en el lugar 0 (ver cómo está hecho el método apilar). inicialmente valdrá COMIENZO_STACK.

Lo anterior no significa que no pueda cambiarse el uso de alguno de esos registros en determinados momentos. Para no perder el valor de los registros cuando se los necesita usar para otra cosa se usará la pila del sistema, por ejemplo de la siguiente manera:

```
push dx
call factorial_de_ax_dejarlo_en_bx
pop dx
```

Como en la función `factorial_de_ax_dejarlo_en_bx` se utiliza dx, para no perder el valor actual de dx (ya que que íbamos a dejar en él, el puerto de la bitácora), lo respaldamos con push y lo recuperamos luego con pop. Se puede ver más sobre esto en el teórico diapositiva 44.

¿Por qué elegimos dx para la bitácora y no para la salida de resultados?

Porque es notorio que pueden haber más usos de la bitácora que de la salida de resultados. Por lo que se evitarán más intercambios de valores entre registros si Se destina dx para la bitácora antes que para la salida de resultados.

Sobre el uso de funciones

Para simplificar las cosas, ha sido útil el uso de funciones para que el manejo de la pila y la entrada quede enmascarado en funciones tal que no cambien el estado de los registros luego de ser ejecutadas (o sea si los precisa, que los use, pero que al terminar deje los registros que use de manera auxiliar en el estado anterior al ejecutar la función) permitiendo dividir la dificultad en pequeñas partes aisladas y centralizar acciones.

Esto es útil para desentendernos en cuanto a que solo ciertos registros se pueden utilizar para ciertas funciones. Por ejemplo, íbamos a usar cx para definir el puerto de salida, pero cx no lo podemos usar con el comando out. Entonces, definimos una función para imprimir ax por el puerto de salida cx que queríamos:

```
imprimir_en_puerto proc
    push dx; respaldamos el dx actual para no perderlo
    mov dx, cx; copiamos a él el valor de cx que queremos
    out dx, ax; ahora estamos imprimiendo ax por el puerto que definimos en cx
    pop dx; recuperamos el valor de dx para dejar los registros como estaban al finalizar la función
    ret
imprimir_en_puerto endp
```

Lo anterior nos permite confiar en que el dx que asignamos en LOG, mantendrá el valor que corresponde a la salida del LOG.

En cuanto a el pasaje por parámetros, se utilizó siempre por registros ya que se cuenta con la fácil disponibilidad de los mismos (no resultó necesario pasar por memoria el valor de un operando ya que siempre lo necesitamos o lo tuvimos en ax o en bx). Por otro lado se mantuvo los nombres de las funciones definidos en el C++ al compilar.

Código Legible y fácil de modificar.

Se definieron operaciones concretas, como obtener datos de la entrada o apilarlos en la pila bajo funciones, para que la lectura del código sea amigable:

```
procesar_operando_o_comando:
    call log_preprocesamiento
    si_es_NUM:
    cmp ax, C_NUM ; si el comando ingresado es C_NUM...
    JNZ si_es_PORT; Si no es, paso a la siguiente etiqueta si_es_PORT
        call poner_siguiente_operando_comando_en_ax
        call log_parametro
        call apilar_ax
    JMP fin_si; Termina si_es_NUM tras haberse comprobado que ax == C_NUM y se salta a la etiqueta del fin
    si_es_PORT:
    cmp ax, C_PORT
    JNZ si_es_TOP
        call poner_siguiente_operando_comando_en_ax
        call log_parametro
        mov cx, ax ; setteo cx, que es la salida, con el valor tomado en bx
    JMP fin_si
```

```
    si_es_TOP:
    cmp ax, C_TOP
    JNZ si_es_DUP
        call copiar_tope_a_ax
        call imprimir_en_puerto
    JMP fin_si
    si_es_DUP:
```

Y si se desea ver la operación concreta se puede ir hasta la implementación de las funciones:

```
apilar_ax proc
    call check_si_habra_desbordamiento
    add si, 2
    mov word ptr [ADDRESS_STACK+si], ax
    ret
apilar_ax endp
```

donde además ya se controla que no haya desbordamiento al insertar (y por lo tanto esto afectará desde todos los puntos donde se intente apilar):

```
check_si_habra_desbordamiento proc
    cmp si, PILA_LLENA
    JL no_hay_desbordamiento
    call log_desbordamiento_de_pila
    no_hay_desbordamiento:
    ret
check_si_habra_desbordamiento endp
```

Y por último llama a:

```
log_desbordamiento_de_pila proc
    push ax
    mov ax, CODIGO_DESBORDAMIENTO_DE_PILA
    out dx, ax
    pop ax
    JMP fin_funcion_exception
    ret
log_desbordamiento_de_pila endp
```

Si se cumple que si == PILA_LLENA y entonces se salta a la etiqueta `fin_funcion_exception` que se encuentra al final del if, para saltarse cualquier operación siguiente:

```
fin_si:
    call log_exitoso
    JMP contiunar_if
fin_funcion_exception:
    mov sp, 0; setteo sp en 0 para vaciar la pila del sistema de calls anteriores
    JMP contiunar_if
```

```
    contiunar_if:
        call poner_siguiente_operando_comando_en_ax
    JMP while_no_salir
salir:
```

Notar que si mañana se agregan condiciones al apilar, estas pueden colocarse en esa misma función apilar_ax.

Entrada/Salida

La entrada se obtiene desde ENTRADA, definida en la sección .port y en particular la constante ENTRADA definida en la sección .data.

Las salidas definidas son puertos programables para la bitácora y para los resultados tal cual se describieron anteriormente.

Pila

En un principio se intentó usar la pila del sistema para manejar los operandos pero según la información obtenida de una clase de consulta, esto no era lo ideal, y lo comprobé, porque esto me generó complicaciones al usar funciones (ya que estas también utilizan la pila), por lo que hubo un cambio grande sobre esto y al final definí la pila usando un arreglo en memoria. ¿Por qué en memoria? Por ser global y por su tamaño.

Mejoras a futuro

Luego de haber terminado el obligatorio me dí cuenta que algunos datos que guardaba en registros, siguiendo el teórico, debería haberlos guardado en memoria por ser globales siguiendo estos lineamientos indicados en las diapositivas:

- Una variable con un alcance (scope) corto, podremos ubicarla únicamente en un registro.
- Una variable global, sí o sí debe tener un espacio en memoria principal

Pero al mismo tiempo no estoy del todo seguro si realmente sea necesario guardarlas en memoria, ya que las paso constantemente como parámetros entre funciones y no son más que los operandos, con los que justamente opero constantemente.
Quizás sí podría ser el caso para cx y dx que guardan el número de puerto de salida para la bitácora y los resultados.

Una de las conclusiones es que pude notar una estructura común a la hora de compilar los if's en alto nivel al assembler usando CMP y los saltos disponibles en la cartilla. Resultó similar también la compilación de el loop while a assembler también usando CMP y alguno de los saltos. En ambos casos agregué un salto condicional al final, en el caso del

loop, un salto que vuelve al comienzo del “while”, y en el caso del if, un salto a una etiqueta diferente que puede por ejemplo estar más adelante en el código.

En cuanto al resto, puedo destacar que me ha resultado muy cómodo el uso de funciones y quizás también haber hecho un código en C++ compilable (que probé con el ejemplo) para evitar pasar posibles errores al assembler.

Referencias

Manual de ArquiSim
https://eva.fing.edu.uy/pluginfile.php/144479/mod_resource/content/2/ManualDeUsuario.pdf

Foro de consultas de laboratorio
https://eva.fing.edu.uy/mod/forum/view.php?id=181527

Algunas operaciones de la cartilla 8086:
https://www.alpertron.com.ar/INST8088.HTM

Diff online para los Test
https://text-compare.com/es/

Conversor:
https://www.disfrutalasmatematicas.com/numeros/binario-decimal-hexadecimal-conversor.html

Notas del teórico del curso:
https://eva.fing.edu.uy/pluginfile.php/144878/mod_resource/content/2/07-intel-8086-2-BU.pdf

Instrucción Goto
https://learn.microsoft.com/es-es/cpp/cpp/goto-statement-cpp?view=msvc-170