

Modularización y Reuso: Rutinas en Q3

Organización de Computadoras 2019

Universidad Nacional de Quilmes

1. Modularización y reuso

A menudo un problema complejo se transforma en algo mas abordable si se lo descompone en problemas más pequeños. Esta descomposición recibe el nombre de **modularización**.

Una rutina o subrutina (pues en general se la utiliza como parte de un programa más grande y específico) es un programa que se espera usar más de una vez. Muchas veces tenemos un problema complejo y necesitamos partirlo en problemas más simples. Para ello utilizamos subrutinas, las cuales nos permiten descomponer un programa grande en sub partes reusando la idea de solución de las mismas.

El uso de rutinas requiere llevar a cabo dos tareas, el **encapsulamiento** y la **invocación**. Para lo primero es indispensable determinar el alcance de la rutina (teniendo en cuenta que debe ser acotada). Una vez hecho esto se crea el código que cumpla con la función de la rutina y se delimita su comienzo y su fin, para lo cual se utiliza una **etiqueta** que la identifique inequívocamente, ubicada en la primera instrucción para que la rutina pueda ser invocada por su nombre, y una instrucción especial **RET** al final para que al finalizar su ejecución vuelva al programa que la invocó. Cabe destacar que no necesariamente la instrucción **RET** estará sólo al final de la rutina, también puede encontrarse en el cuerpo de la misma.

Considerar como ejemplo la siguiente rutina **agregar**, que incrementa en uno el valor de R0.

```
agregar: ADD R0, 0x0001
        RET
```

La rutina esta delimitada en su inicio por la etiqueta **agregar** por lo cual para su invocación desde el *programa principal o programa cliente* se utilizá la instrucción especial **CALL** haciendo referencia a la misma.

Siguiendo el ejemplo anterior, la siguiente es una invocación a la rutina **agregar**.

```
CALL agregar
```

La etiqueta **agregar** se traducirá en un **inmediato** cuyo valor será la dirección de memoria a partir de la cual está ensamblada la rutina, por lo cual la primer instrucción de la misma será la siguiente a ejecutar.

Si la rutina **agregar** se encuentra ensamblada a partir de la celda de memoria **0x568A** la invocación anterior equivale a la siguiente:

```
CALL 0x568A
```

Por otro lado, si se utilizara esta rutina para incrementar en 3 en vez de en uno el valor de R0, es posible utilizarla varias veces, provocando el **reuso de la subrutina**. En ese caso se modifica el programa principal que la invoca. Por ejemplo:

```
CALL agregar
CALL agregar
CALL agregar
```

1.1. Haciendo rutinas flexibles: pasaje de parámetros

Considerar la siguiente rutina `avg` (*average* = promedio), que calcula el promedio entre los valores 20 y 30 (0x0014 y 0x001E en hexadecimal respectivamente):

```
avg: MOV R0, 0x0014
      ADD R0, 0x001E
      DIV R0, 0x0002
      RET
```

El programa cliente que utilice esta rutina podría ser:

```
CALL avg
```

En este ejemplo se ve claramente una posible mejora para hacer: que la rutina pueda calcular el promedio entre *cualquier par de valores*. Para lograr esto, la rutina debe ser **parametrizable** ya que no conocemos los valores involucrados. Haciendo una analogía con las matemáticas, lo que se intenta hacer es pasar de una expresión

$$avg = \frac{20 + 30}{2}$$

a esta otra:

$$avg = \frac{a + b}{2}$$

siendo a y b dos variables.

A diferencia de otros lenguajes de programación, Q3 no cuenta con una sintaxis específica para el *pasaje de parámetros a las rutinas*. Pero es posible que la rutina se alimente de esos valores de entrada para el cálculo del promedio a través del uso de variables, que en esta arquitectura serían registros o celdas de la memoria principal.

Con esta idea, la rutina puede mejorarse de la siguiente manera:

```
avg: MOV R0, R1
      ADD R0, R2
      DIV R0, 0x0002
      RET
```

De esta manera, la rutina `avg` ahora calcula el promedio entre dos valores cualesquiera que estén almacenados en R1 y R2. Es importante notar que por esta razón el programa que la usa también debe modificarse para colocar los valores respectivos para calcular el promedio de R1 y R2:

```
MOV R1, 0x0014
MOV R2, 0x001E
CALL avg
```

1.2. Documentación de las rutinas

Para usar apropiadamente cualquier rutina es importante contar con información sobre qué hace, cómo pasarle los datos que necesita (parámetros), qué modifica del estado (registros, celdas de memoria, etc) y de qué manera dispone el o los resultados. Esa información se conoce con el nombre de **documentación de la rutina** y tiene tres piezas de información:

- **Requiere:** se utiliza para describir los parámetros de entrada. Qué registro o celda contiene cada uno y qué naturaleza tienen.

- **Modifica:** se utiliza para describir qué variables se modifican previniendo posibles efectos colaterales de la ejecución de la rutina.
- **Retorna:** se utiliza para indicar en qué variable (registro o celda) se retorna el/los resultado/s y en qué consiste el mismo.

Por ejemplo, la documentación que acompañe la rutina `avg` mencionada arriba podría ser:

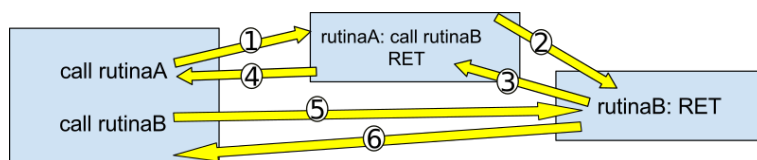
```
# ----- Rutina avg -----
# Requiere: dos valores a promediar en los registros R1 y R2
# y que la suma de ambos se pueda representar en 16 bits
#
# Modifica: R0
#
# Retorna: en R0 el promedio (entero) entre R1 y R2
```

La documentación de una rutina es un **contrato** entre el autor de la rutina y los usuarios de la misma ¿Qué quiere decir esto? Que la persona que programa escribe la rutina y su documentación comprometiéndose a que si las especificaciones escritas en la documentación son respetadas a la hora de invocar la rutina (respetar el requiere”), ésta funcionará de manera correcta con los resultados esperados (respetando el ”modificaz el retorna”).

Esto, lo que nos permite es que cualquier programador/a pueda utilizar una rutina sólo conociendo su etiqueta (nombre) y su documentación sin necesidad de conocer su implementación (encapsulamiento).

2. Cómo funcionan CALL y RET

El objetivo de la programación mediante rutinas es el de abstraer al programador/a de la ubicación de dichas rutinas en memoria principal a través del encapsulamiento. Esta idea se describe gráficamente en el ejemplo de la siguiente figura:



En ese ejemplo se ve que al finalizar las ejecuciones de la rutina `rutinaB` se debe retornar a dos lugares distintos y dependerán de la ubicación de la instrucción `CALL` que lo llamó: en una oportunidad la rutina `rutinaB` es invocada desde la rutina `rutinaA` y en la otra desde el programa principal por lo que vemos que el punto de retorno varía.

Es importante destacar que el ensamblador cuando carga el código máquina del programa y las rutinas en memoria no necesariamente lo hace en ubicaciones continuas de memoria. Por ejemplo:

	...
(programa ppal)0x9999	<call>
0x999A	<rutinaA>
0x999B	<call>
0x999C	<rutinaB>
	:
(rutina A)AAAA	<call>
0xAAAB	<rutinaB>
0xAAAC	<ret>
	:
(rutina B)0xBAA0	<ret>
	...

Para llevar control de cuál es la instrucción que se debe ejecutar, se utiliza el registro *PC*. Este registro se incrementa al finalizar la etapa de *búsqueda de instrucción* para saber en qué celda de memoria está la siguiente instrucción a ejecutar.

A diferencia de las instrucciones aritméticas en las que el PC no se modifica hasta que comience un ciclo nuevo, la instrucción **CALL** requiere alterar el valor del PC para poder causar el efecto desviación de flujo necesario. Veamos cómo debería ser la ejecución del programa del ejemplo:

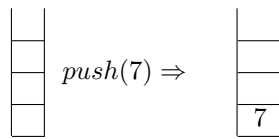
1. PC=9999
2. Se busca la instrucción **CALL rutinaA**. Por lo cual el nuevo valor de PC es 0x999B, ya debe quedar preparado para ejecutar la siguiente instrucción del programa principal.
3. Se realiza la ejecución de **CALL rutinaA**, que principalmente preparar PC para que se desvíe el flujo a la rutina **rutinaA**, por lo cual su nuevo valor es 0xAAAA. Pero además se debe **recordar el valor anterior de PC** (0x999B) para restituir el flujo al retornar de la rutina.
4. Se busca la instrucción **CALL rutinaB** (ensamblada en la dirección de memoria 0xAAAA). Entonces el valor de pc es ahora 0xAAAC, pues debe quedar preparado para la siguiente instrucción.
5. Se realiza la ejecución de **CALL rutinaB**, entonces PC=BAA0. Nuevamente se debe **recordar el valor de PC** (AAAC).
6. Se busca la instrucción **RET** (ensamblada en BAA0). Entonces PC=BAA1, pues debe quedar preparado para la siguiente instrucción.
7. Se realiza la ejecución de **RET**, es decir **se restituye el último valor de PC: PC=AAAC**
8. Se busca la instrucción **RET** (ensamblada en AAAC). Entonces PC=AAAD, pues debe quedar preparado para la siguiente instrucción.
9. Se realiza la ejecución de **RET**, es decir **se restituye el anterior valor de PC: PC=999B**
10. Se busca la instrucción **CALL rutinaB** (ensamblada en 999B). Entonces PC=999D, pues debe quedar preparado para la siguiente instrucción.
11. Se realiza la ejecución de **CALL rutinaB**, entonces PC=BAA0
12. Se busca la instrucción **RET** (ensamblada en BAA0). Entonces PC=BAA1, pues debe quedar preparado para la siguiente instrucción.
13. Se realiza la ejecución de **RET**, es decir **se restituye el último valor de PC: PC=999D**

El desafío entonces es darle a la UC (Unidad de Control) una herramienta que permita llevar cuenta de los valores de PC acumulados para ir restituyendolos en orden inverso. La solución no puede ser tener un registro de respaldo del PC porque como vemos en los pasos 3 y 5, se necesitan mantener dos valores (y podrían ser aún más).

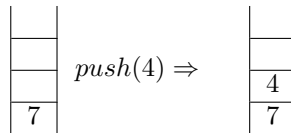
2.1. La estructura de pila

Los cambios en el PC producidos por la ejecución de CALL dan lugar a la necesidad de contar con una estructura de datos que permita registrar los diferentes valores de retorno y en el orden necesario para ser consumidos de manera inversa a la que fueron registrados. Esta estructura se denomina **pila** y está caracterizada por dos operaciones: apilado (*push*) y desapilado (*pop*).

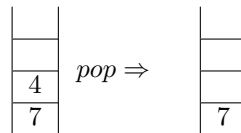
Suponer una estructura de pila vacía, donde se apila el valor 7:



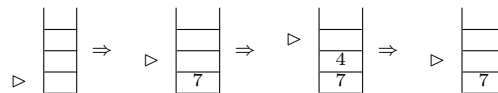
Si luego se apila el valor 4:



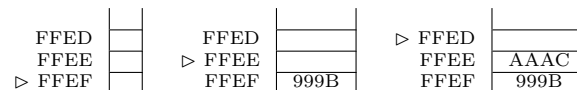
Si a continuación se desapila:



Este concepto abstracto de pila se puede implementar mediante la memoria principal en combinación con un registro que indique el **tope de pila** (indicado en el dibujo con un ▷)



El mencionado registro se denomina **SP** (*Stack pointer*, puntero de pila), y su valor cuando la pila está vacía es FFEF. Con esta idea, se debe revisar el funcionamiento esperado para el CALL y el RET. Para facilitarlos, veamos los cambios esperados en la pila según el ejemplo de ejecución de la sección anterior:



Es decir que con cada **CALL** se debe **apilar** y con cada **RET** se debe **desapilar**.

CALL Se apila el PC anterior y luego se lo reemplaza por la dirección de inicio de la rutina:

1. Se copia PC a la celda indicada por **SP**
2. Se decrementa (hay un elemento más en la pila) **SP**

3. Se actualiza PC

RET Se desapila el último PC anterior:

1. Se incrementa (hay un elemento menos en la pila) **SP**
2. Se copia a PC el valor de celda indicada por **SP**

3. Simulación de un programa

Durante la ejecución de un programa además de los cambios en los registros y en las celdas de memoria se realizan cambios en los registros especiales PC, SP y en la pila que dependerán de las instrucciones que lo conformen. Para seguir paso a paso estos cambios se completará la siguiente tabla simulando ejecución instrucción por instrucción:

Búsq. de inst.			Decod. de inst.			Operandos		Ejecución			
PC	IR	PC2	operación	modo destino	modo origen	Destino	Origen	Efecto	Pila	SP	PC3

Donde:

- **PC2** es el valor de PC luego de la *busqueda de instrucción*
- **PC3** es el valor de PC luego de la *ejecución*.
- **SP** es el valor del registro SP luego de la *ejecución*
- **Pila** es la secuencia de valores apilados (en orden invertido con respecto a su inserción en la pila)

Por ejemplo, suponer la ejecución del siguiente programa llamado **principal**:

```
rutinaB: RET
rutinaA: CALL rutinaB
        RET
principal: CALL rutinaA
          CALL rutinaB
```

Y se tienen los siguientes datos:

- El programa **principal** esta ensamblado a partir de la celda 0x0008
- La rutina **rutinaA** esta ensamblada a partir de la celda 0123
- La rutina **rutinaB** esta ensamblada a partir de la celda 0678
- La pila está vacía.

La primer entrada de la tabla se completa como sigue:

PC es el valor inicial de PC, en este caso: 0008

Búsq. de inst.			Decod. de inst.			Operandos		Ejecución			
PC	IR	PC2	operación	modo destino	modo origen	Destino	Origen	Efecto	Pila	SP	PC3
0008											

IR es el código máquina de la instrucción leída. En este caso: B0000123

Búsq. de inst.			Decod. de inst.			Operandos		Ejecución			
PC	IR	PC2	operación	modo destino	modo origen	Destino	Origen	Efecto	Pila	SP	PC3
0008	B0000123										

PC2 Inmediatamente después de la búsqueda de instrucción, el PC contiene la dirección de memoria donde comienza la siguiente instrucción a ejecutar. Siendo la instrucción **CALL rutinaA** y sabiendo que ocupa dos celdas, sumamos esa cantidad a 0008. Entonces el valor del registro PC luego de la búsqueda de instrucción es 000A.

Búsq. de inst.			Decod. de inst.			Operandos		Ejecución			
PC	IR	PC2	operación	modo destino	modo origen	Destino	Origen	Efecto	Pila	SP	PC3
0008	B0000123	000A									

Decodificación Se identifica la operación a realizar y sobre qué operandos. En este caso es un **CALL**, que se trata de una instrucción de un solo operando origen, dado que el operando destino es el registro PC y por lo tanto está implícito (la instrucción no debe indicarlo).

Búsq. de inst.			Decod. de inst.			Operandos		Ejecución			
PC	IR	PC2	operación	modo destino	modo origen	Destino	Origen	Efecto	Pila	SP	PC3
0008	B0000123	000A	CALL	--	inmediato						

Operandos El operando origen de este ejemplo es el nuevo valor que tendrá pc: 0123

Búsq. de inst.			Decod. de inst.			Operandos		Ejecución			
PC	IR	PC2	operación	modo destino	modo origen	Destino	Origen	Efecto	Pila	SP	PC3
0008	B0000123	000A	CALL	--	inmediato	--	0123				

Ejecución Como indica el **efecto de la instrucción CALL**, es decir: $[SP] \leftarrow PC$; $SP \leftarrow SP - 1$; $PC \leftarrow \text{Origen}$, se llevan a cabo 3 pasos:

1. Apilar PC: La pila debe almacenar el valor de retorno para PC
2. Decrementar SP: El valor del registro SP se actualiza para que se apunte a un lugar libre, es decir que toma el valor FFEE.
3. Actualizar PC: Finalmente, valor de PC luego de la ejecución se carga con el valor **Origen**, y como se trata de una etiqueta (valor inmediato), dicho valor se leyó con el código máquina de la instrucción.

Búsq. de inst.			Decod. de inst.			Operandos		Ejecución			
PC	IR	PC2	operación	modo destino	modo origen	Destino	Origen	Efecto	Pila	SP	PC3
0008	B0000123	000A	CALL	--	inmediato	--	0123	$PC \leftarrow 0123$	{000A}	FFEE	0123

Es importante notar que las columnas **PC-Ex**, **SP** y **pila** se modifican sólo en los casos de instrucciones como **CALL** y **RET** que son, hasta ahora, las que modifican PC y la pila.

Para continuar la simulación se repite el proceso descrito con las siguientes instrucciones, en el orden del **flujo de ejecución** incluyendo el llamado a subrutinas (y las instrucciones de las mismas) hasta la última instrucción del programa principal inclusive. La tabla terminada con la simulación de ejecución del programa anterior sería la siguiente:

Búsq. de inst.			Decod. de inst.			Operandos		Ejecución			
PC	IR	PC2	operación	modo destino	modo origen	Destino	Origen	Efecto	Pila	SP	PC3
0008	B0000123	000A	CALL	--	inmediato	--	0123	$PC \leftarrow 0123$	{000A}	FFEE	0123
0123	B0000678	0125	CALL	--	inmediato	--	0678	$PC \leftarrow 0678$	{0125,000A}	FFED	0678
0678	C000	0679	RET	--	---	--	---	$PC \leftarrow 0125$	{000A}	FFEE	0125
0125	C000	0126	RET	--	---	--	---	$PC \leftarrow 000A$	{}	FFEF	000A
000A	B0000678	000C	CALL	--	inmediato	--	0678	$PC \leftarrow 0678$	{000C}	FFEE	0678
0678	C000	0679	RET	--	---	--	---	$PC \leftarrow 000C$	{}	FFEF	000C