e)
```
LD R0, p
LD R1, x
ST 0(R0), R1
```

f)
```
LD   R0, x
LD   R1, y
SUB  R0, R0, R1
BLTZ *R3, R0
```

8.3 Direcciones en el código destino

En esta sección le mostraremos cómo pueden convertirse los nombres en la representación intermedia en direcciones en el código destino, analizando la generación de código para las llamadas simples a procedimientos y sus retornos, mediante el uso de la asignación estática y de pila. En la sección 7.1 describimos la forma en que cada programa se ejecuta en su propio espacio de direcciones lógicas, que se particionaba en cuatro áreas de código y cuatro de datos:

1. Un área de *Código* determinada en forma estática, la cual contiene el código destino ejecutable. El tamaño del código destino se puede determinar en tiempo de compilación.
2. Un área de datos *Estática* determinada en forma estática, para contener constantes globales y demás datos que genera el compilador. El tamaño de las constantes globales y de los datos del compilador también puede determinarse en tiempo de compilación.
3. Un área *Montículo* administrada en forma dinámica, para contener objetos de datos que se asignan y se liberan durante la ejecución del programa. El tamaño del *Montículo* no puede determinarse en tiempo de compilación.
4. Un área *Pila* administrada en forma dinámica, para contener los registros de activación a medida que se crean y se destruyen, durante las llamadas a los procedimientos y sus retornos. Al igual que el *Montículo*, el tamaño de la *Pila* no puede determinarse en tiempo de compilación.

8.3.1 Asignación estática

Para ilustrar la generación de código para las llamadas y retornos simplificados de procedimientos, vamos a enfocarnos en las siguientes instrucciones de tres direcciones:

- `call` *receptor*
- `return`
- `halt`
- `action`, que es un receptáculo para otras instrucciones de tres direcciones.

El tamaño y la distribución de los registros de activación se determinan mediante el generador

la llamada a un procedimiento, y cómo devolverle el control después de la llamada. Por conveniencia, asumiremos que la primera ubicación en la activación contiene la dirección de retorno.

Primero vamos a considerar el código necesario para implementar el caso más simple, la asignación estática. Aquí, una instrucción `call` *receptor* en el código intermedio puede implementarse mediante una secuencia de dos instrucciones de máquina destino:

```
ST  receptor.areaEstatica, #aqui + 20
BR  receptor.areaCodigo
```

La instrucción ST guarda la dirección de retorno al principio del registro de activación para *receptor*, y la instrucción BR transfiere el control al código de destino para el procedimiento *receptor* al que se llamó. El atributo antes de *receptor.areaEstatica* es una constante que proporciona la dirección del inicio del registro de activación para *receptor*, y el atributo *receptor.areaCodigo* es una constante que se refiere a la dirección de la primera instrucción del procedimiento *receptor* al que se llamó en el área *Código* de la memoria en tiempo de ejecución.

El operando *#aqui* + 20 en la instrucción ST es la dirección de retorno literal; es la dirección de la instrucción que sigue después de la instrucción BR. Suponemos que *#aqui* es la dirección de la instrucción actual, y que las tres constantes más las dos instrucciones en la secuencia de llamada tienen una longitud de 5 palabras, o 20 bytes.

El código para un procedimiento termina con un retorno al procedimiento que hizo la llamada, sólo que el primer procedimiento no tiene emisor, por lo que su instrucción final es HALT, la cual devuelve el control al sistema operativo. Una instrucción `return` *receptor* puede implementarse mediante una instrucción simple de salto:

```
BR  *receptor.areaEstatica
```

la cual transfiere el control a la dirección guardada al principio del registro de activación para *receptor*.

Ejemplo 8.3: Suponga que tenemos el siguiente código de tres direcciones:

```
                // código para c
action1
call p
action2
halt
                // código para p
action3
return
```

La figura 8.4 muestra el programa destino para este código de tres direcciones. Usamos la seudoinstrucción ACTION para representar la secuencia de instrucciones de máquina que van a ejecutar la instrucción `action`, que representa el código de tres direcciones que no es relevante para esta discusión. Empezamos de manera arbitraria el código para el procedimiento c en la dirección 100, y para el procedimiento p en la dirección 200. Suponemos que cada instrucción ACTION requiere 20 bytes. Además, suponemos que los registros de activación para estos procedimientos se asignación de manera estática en las ubicaciones 300 y 364, respectivamente.

L es que empiez ción 100 imp s siguientes in

`action`$_1$`; call p; action`$_2$`; halt`

del primer procedimiento c. Por lo tanto, la ejecución empieza con la instrucción ACTION$_1$ en la dirección 100. La instrucción ST en la dirección 120 guarda la dirección de retorno 140 en el campo de estado de la máquina, que es la primera palabra en el registro de activación de p. La instrucción BR en la dirección 132 transfiere el control a la primera instrucción en el código de destino del procedimiento p que se llamó.

```
                        // código para c
100:  ACTION1           // código para action1
120:  ST 364, #140      // guarda la dirección de retorno 140 en la ubicación 364
132:  BR 200            // llama a p
140:  ACTION2
160:  HALT              // regresa al sistema operativo
      ...
                        // código para p
200:  ACTION3
220:  BR *364           // regresa a la dirección guardada en la ubicación 364
      ...
                        // 300-363 contienen el registro de activación para c
300:                    // dirección de retorno
304:                    // datos locales para c
      ...
                        // 364-451 contienen el registro de activación para p
364:                    // dirección de retorno
368:                    // datos locales para p
```

Figura 8.4: Código destino para la asignación estática

Después de ejecutar ACTION$_3$, se ejecuta la instrucción de salto en la ubicación 220. Como la ubicación 140 se guardó en la dirección 364 mediante la secuencia de llamada anterior, *364 representa 140 cuando se ejecuta la instrucción BR en la dirección 220. Por lo tanto, cuando termina el procedimiento p, el control regresa a la dirección 140 y se reanuda la ejecución del procedimiento c. □

8.3.2 Asignación de pila

La asignación estática se puede convertir en asignación de pila mediante el uso de direcciones relativas para el almacenamiento en los registros de activación. Sin embargo, en la asignación de pila la posición de un registro de activación para un procedimiento no se conoce sino hasta el tiempo de ejecución. Por lo general, esta posición se almacena en un registro, por lo que se puede acceder a las palabras en el registro de activación como desplazamientos a partir del valor en este registro. El modo de direccionamiento indexado para nuestra máquina destino es conveniente para este fin.

Las direcciones relativas en un registro de activación pueden tomarse como desplazamientos a partir de (ición conocid o de activaci os en el capítu

Por conveniencia, vamos a usar desplazamientos positivos, manteniendo en un registro SP un apuntador al inicio del registro de activación en la parte superior de la pila. Cuando ocurre una llamada a un procedimiento, el procedimiento que llama incrementa a SP y transfiere el control al procedimiento al que llamó. Una vez que el control regresa al emisor, decrementamos SP, con lo cual se desasigna el registro de activación del procedimiento al que se llamó.

El código para el primer procedimiento inicializa la pila, estableciendo SP al inicio del área de la pila en la memoria:

```
LD    SP, #inicioPila                    // inicializa la pila
código para el primer procedimiento
HALT                                     // termina la ejecución
```

La secuencia de la llamada a un procedimiento incrementa a SP, guarda la dirección de retorno y transfiere el control al procedimiento al que se llamó:

```
ADD   SP, SP, #emisor.tamRegistro        // incrementa el apuntador de la pila
ST    *SP, #aqui + 16                    // guarda la dirección de retorno
BR    receptor.areaCodigo                // regresa al emisor
```

El operando #*emisor.tamRegistro* representa el tamaño de un registro de activación, por lo que la instrucción ADD hace que SP apunte al siguiente registro de activación. El operando #*aqui* + 16 en la instrucción ST es la dirección de la instrucción que va después de BR; se guarda en la dirección a la que apunta SP.

La secuencia de retorno consiste en dos partes. El procedimiento al que se llamó transfiere el control a la dirección de retorno, usando lo siguiente:

```
BR   *0(SP)                              // regresa al emisor
```

La razón de usar *0(SP) en la instrucción BR es que necesitamos dos niveles de indirección: 0(SP) es la dirección de la primera palabra en el registro de activación, y *0(SP) es la dirección de retorno que está guardada ahí.

La segunda parte de la secuencia de retorno está en el emisor, el cual decrementa a SP, con lo cual SP se restaura a su valor anterior. Es decir, después de la resta SP apunta al inicio del registro de activación del emisor:

```
SUB   SP, SP, #emisor.tamRegistro        // decrementa el apuntador de la pila
```

El capítulo 7 contiene una explicación más amplia sobre las secuencias de las llamadas y las concesiones que se deben hacer en la división de la labor entre el procedimiento que llama y el procedimiento al que se llamó.

Ejemplo 8.4: El programa en la figura 8.5 es una abstracción del programa quicksort del capítulo anterior. El procedimiento q es recursivo, por lo que puede haber más de una activación de q viva al mismo tiempo.

S… …s tamaños de … de activación … …dimientos m, deter… *mtam*, *ntam* …ctivamente. I… …labra en cada

```
                    // código para m
action1
call q
action2
halt
                    // código para p
action3
return
                    // código para q
action4
call p
action5
call q
action6
call q
return
```

Figura 8.5: Código para el ejemplo 8.4

activación guardará una dirección de retorno. Suponemos en forma arbitraria que el código para estos procedimientos empieza en las direcciones 100, 200 y 300, respectivamente, y que la pila empieza en la dirección 600. El programa destino se muestra en la figura 8.6.

Vamos a suponer que $\texttt{ACTION}_4$ contiene un salto condicional a la dirección 456 de la secuencia de retorno de q; de no ser así, el procedimiento recursivo q se condena a llamarse a sí mismo por siempre.

Si *mtam*, *ptam* y *qtam* son 20, 40 y 60, respectivamente, la primera instrucción en la dirección 100 inicializa SP a 600, la dirección inicial de la pila. SP contiene 620 justo antes que el control se transfiera de m a q, ya que *mtam* es 20. Más adelante, cuando q llama a p, la instrucción en la dirección 320 incrementa SP a 680, en donde empieza el registro de activación para p; SP se regresa a 620 después de que el control regresa a q. Si las dos siguientes llamadas de q regresan de inmediato, el valor máximo de SP durante esta ejecución es de 680. Sin embargo, observe que la última ubicación de la pila que se utiliza es 739, ya que el registro de activación de q que empieza en la ubicación 680 se extiende 60 bytes. □

8.3.3 Direcciones para los nombres en tiempo de ejecución

La estrategia de asignación de almacenamiento y la distribución de los datos locales en un registro de activación para un procedimiento son los que determinan cómo se accede al almacenamiento para los nombres. En el capítulo 6, asumimos que un nombre en una instrucción de tres direcciones es en realidad un apuntador a una entrada en la tabla de símbolos para ese nombre. Este método tiene una ventaja considerable; hace que el compilador sea más portable, ya que el front-end no tiene que modificarse, ni siquiera cuando el compilador se mueve a una máquina distinta, en donde se requiere una organización distinta en tiempo de ejecución. Por otro lado, la acción de generar la secuencia específica de pasos de acceso mientras se genera el código inter representar considerable ador optimiz

```
                          // código para m
100:  LD SP, #600         // inicializa la pila
108:  ACTION1             // código para action1
128:  ADD SP, SP, #mtam   // empieza la secuencia de llamadas
136:  ST *SP, #152        // mete la dirección de retorno
144:  BR 300              // llama a q
152:  SUB SP, SP, #mtam   // restaura SP
160:  ACTION2
180:  HALT
      ...
                          // código para p
200:  ACTION3
220:  BR *0(SP)           // regresa
      ...
                          // código para q
300:  ACTION4             // contiene un salto condicional a 456
320:  ADD SP, SP, #qtam
328:  ST *SP, #344        // mete la dirección de retorno
336:  BR 200              // llama a p
344:  SUB SP, SP, #qtam
352:  ACTION5
372:  ADD SP, SP #qtam
380:  BR *SP, #396        // mete la dirección de retorno
388:  BR 300              // llama a q
396:  SUB SP, SP, #qtam
404:  ACTION6
424:  ADD SP, SP, #qtam
432:  ST *SP, #440        // mete la dirección de retorno
440:  BR 300              // llama a q
448:  SUB SP, SP, #qtam
456:  BR *0(SP)           // regresa
      ...
600:                      // la pila empieza aquí
```

Figura 8.6: Código destino para la asignación de pila

ya que permite al optimizador aprovechar los detalles que no vería en la instrucción simple de tres direcciones.

En cualquier caso, los nombres deben sustituirse en algún momento dado por código para acceder a las ubicaciones de almacenamiento. Por ende, consideramos ciertas elaboraciones de la instrucción de copia simple de tres direcciones `x = 0`. Después de procesar las declaraciones en un procedimiento, suponga que la entrada en la tabla de símbolos para `x` contiene una dirección relativa 12 para `x`. Consideremos el caso en el que `x` se encuentra en un área asignación en forma estática, que empieza en la dirección *estatica*. Entonces, la verdadera dirección en tiempo de ejecución de `x` es *estatica* + 12. Aunque el compilador puede determinar en un momento dado el valor de *estatica* + 12 en tiempo de compilación, tal vez no se conozca la posición del área estática al generar el código intermedio para acceder al nombre. En ese caso, tiene sentido generar un código de tres direcciones para "calcular" *estatica* + 12, con el entendimiento de que este cálculo se llevará a cabo durante la fase de generación de código, o quizá por medio del cargador, antes de que se ejecute el programa. Así, la asignación `x = 0` se traduce en:

```
estatica[12] = 0
```

Si el área estática empieza en la dirección 100, el código de destino para esta instrucción es:

```
LD 112, #0
```

8.3.4 Ejercicios para la sección 8.3

Ejercicio 8.3.1: Genere el código para las siguientes instrucciones de tres direcciones, suponiendo la asignación de pila en donde el registro `SP` apunta a la parte superior de la pila.

```
call p
call q
return
call r
return
return
```

Ejercicio 8.3.2: Genere el código para las siguientes instrucciones de tres direcciones, asumiendo la asignación de pila en donde el registro `SP` apunta a la parte superior de la pila.

a) `x = 1`

b) `x = a`

c) `x = a + 1`

d) `x = a + b`

e) Las siguientes dos instrucciones:

```
x = b * c
;
```