PARTE 1 MULT.ASM

Descripción del Código de Multiplicación en Ensamblador

Este código realiza la multiplicación de dos valores almacenados en los registros R0 y R1 y guarda el resultado en el registro R2. La multiplicación se lleva a cabo mediante sumas repetitivas. A continuación, se explica el funcionamiento del código paso a paso:

1. **Inicialización del Resultado (R2)**: El valor del registro R2 se inicializa en 0, ya que se utilizará para almacenar el resultado acumulado de la multiplicación.
2. **Inicio del Bucle de Multiplicación**:
    - Se marca la etiqueta (LOOP_START) para indicar el comienzo del bucle.
    - Se carga el valor de R1 y se verifica si es igual a cero. Si R1 es 0, se salta al final del programa, ya que no hay más sumas que realizar.
3. **Suma Repetitiva para Multiplicar**:
    - El valor de R0 se suma al valor de R2. Esta suma se repite tantas veces como el valor en R1. Este proceso simula la multiplicación, ya que R0 se suma R1 veces.
    - Luego, el valor actualizado se almacena nuevamente en R2.
4. **Decremento de R1**:
    - Cada vez que se realiza la suma, R1 se decrementa en 1 para llevar la cuenta de cuántas veces se ha realizado la suma.
    - Este proceso se repite hasta que R1 llegue a 0.
5. **Repetición del Bucle**:
    - El bucle regresa a la etiqueta (LOOP_START) mientras R1 no sea igual a cero, repitiendo el proceso de sumar R0 a R2 y disminuir R1.
6. **Finalización del Programa**:
    - Cuando R1 llega a 0, el flujo del programa salta a la etiqueta (END), donde el programa se detiene.

En Resumen: El código implementa una multiplicación utilizando sumas iterativas. R2 empieza en 0 y se incrementa con el valor de R0 tantas veces como indique el valor de R1. Esta técnica se conoce como multiplicación por sumas repetidas, donde R0 se agrega a sí mismo R1 veces. Al finalizar, R2 contiene el producto de R0 y R1.

```
1
2     // Inicializar R2 a 0
3     @R2
4     M=0
5     // Comenzar el bucle
6     (LOOP_START)
7     @R1
8     D=M
9     @END         // Si R1 es 0, saltar a END
10    D;JEQ
11
12    // Sumar R0 a R2
13    @R0
14    D=D+M        // D = R2 + R0
15    @R2          // Dirección de R2
16    M=D          // Guardar nuevo valor en R2
17
18    // Decrementar R1
19    @R1          // Cargar R1
20    M=M-1        // R1 = R1 - 1
21
22    @LOOP_START  // Volver al inicio del bucle
23    0;JMP
24
25    (END)        // Fin del programa
26    @END         // Detener el programa
27    0;JMP
```

Hack Assembler

Symbolic

```
// Multiplicación: R0 * R1 -> R2

// Inicializar R2 a 0
@R2
M=0          // R2 = 0

// Comenzar el bucle
(LOOP_START)
@R1          // Cargar R1
D=M          // D = R1
@END         // Si R1 es 0, saltar a END
D;JEQ

// Sumar R0 a R2
@R0          // Cargar R0
D=D+M        // D = R2 + R0
@R2          // Dirección de R2
M=D          // Guardar nuevo valor en R2

// Decrementar R1
@R1          // Cargar R1
M=M-1        // R1 = R1 - 1
```

Binary

```
0000000000000010
1110101010001000
0000000000000001
1111110000010000
0000000000001110
1110001100000010
0000000000000000
1111000010010000
0000000000000010
1110001100001000
0000000000000001
1111110010001000
0000000000000010
1110101010000111
0000000000001110
1110101010000111
```

El resultado es una secuencia de instrucciones de bajo nivel en el lenguaje de máquina que utiliza la arquitectura Hack del proyecto Nand2Tetris. Cada línea corresponde a una instrucción en formato binario, que la máquina Hack puede interpretar directamente.

Interpretación de las Instrucciones

- La arquitectura Hack tiene dos tipos de instrucciones:
    a. **Instrucción A** (Address): Utilizada para definir una dirección en la memoria.
    b. **Instrucción C** (Compute): Utilizada para realizar operaciones aritméticas y lógicas, así como almacenamiento.

Cada instrucción está representada por 16 bits (es decir, 16 dígitos binarios).

Explicación General

- Las instrucciones que comienzan con `000` son **instrucciones A**, las cuales apuntan a una dirección de memoria específica.
- Las instrucciones que comienzan con `111` son **instrucciones C**, que realizan operaciones y cálculos.
- Este conjunto de instrucciones refleja exactamente la lógica del programa ensamblador que has escrito: inicializa `R2`, luego suma repetidamente `R0` a `R2` tantas veces como indique `R1` y finalmente se detiene.

El resultado son las instrucciones traducidas en lenguaje de máquina, listas para ser ejecutadas por la CPU Hack. Esta representación es equivalente al programa ensamblador original, pero ahora en una forma más baja, que la máquina puede entender y ejecutar directamente.

PARTE 2 .ASM

Explicación del Código

7. **(START)**: Esta es la etiqueta de inicio del programa, que se usará para el bucle principal.
8. **Verificar el Estado del Teclado**:
    - `@KBD` apunta a la dirección del teclado.
    - `D=M` carga el valor actual del teclado.
    - `@FILL` indica la etiqueta de llenado de la pantalla.

- `D;JNE` hace un salto a `FILL` si el valor de `D` es diferente de cero, lo que significa que una tecla está siendo presionada.

9. **Vaciar la Pantalla** (Si ninguna tecla está presionada):
    - `@SCREEN` apunta a la dirección de inicio de la pantalla (ubicada en `16384`).
    - `D=A` guarda esa dirección en `D` para poder iterar.
    - `(CLEAR_LOOP)` es el bucle que se repite hasta que todas las direcciones de la pantalla están en `0`.
    - `A=D` selecciona la dirección actual.
    - `M=0` limpia esa dirección.
    - `D=D+1` incrementa la dirección.
    - `@CLEAR_LOOP` y `D-A;JLT` aseguran que el bucle continúe hasta que se llegue al final de la pantalla (`ENDSCREEN`).
10. **Llenar la Pantalla** (Si una tecla está presionada):
    - `(FILL)` es la etiqueta de llenado.
    - El proceso es similar al bucle de limpieza, pero esta vez llenamos cada dirección de la pantalla con `-1` (`M=-1`), que representa todos los bits en `1`.
11. **Etiqueta (ENDSCREEN)**:
    - `@24576` apunta a la dirección justo después del área de la pantalla (`16384 + 8192`), utilizada como límite para los bucles de llenado y limpieza.
12. **Bucle Infinito**:
    - Al final de cada bucle (`CLEAR_LOOP` o `FILL_LOOP`), el programa regresa a la etiqueta `(START)` para verificar el estado del teclado continuamente.

Este código asegura que, cuando una tecla está presionada, la pantalla se llene completamente, y cuando no hay teclas presionadas, la pantalla se vacíe. El bucle principal verifica constantemente el estado del teclado para actualizar la pantalla en consecuencia.

Hack Assembler

Symbolic

```
// Fill.asm
// Programa que llena o vacía la pantalla dependiendo de si el botón está presionado

(START)
    @KBD       // Dirección del teclado
    D=M        // D = valor del teclado (0 si no se presiona, distinto de 0 si se
presiona)
    @FILL      // Si el valor del teclado no es 0, saltar a FILL
    D;JNE

    // Vaciar la pantalla
    @SCREEN    // Dirección de la pantalla
    D=A        // D = Dirección de la pantalla (empezar desde el inicio)
    @ENDSCREEN // Dirección de fin de pantalla
(CLEAR_LOOP)
    A=D
    M=0        // Poner 0 en cada dirección (vaciar la pantalla)
    D=D+1      // Incrementar dirección
    @CLEAR_LOOP
    D-A;JLT    // Repetir hasta que se llegue al final de la pantalla

    @START     // Vuelve al inicio para comprobar el estado del teclado
    0;JMP

(FILL)
    // Llenar la pantalla
```

Binary

```
0110000000000000
1111110000010000
0000000000001110
1110001100000101
0100000000000000
1110110000010000
0000000000011000
1110001100100000
1110101010001000
1110011111010000
0000000000000111
1110010011000100
0000000000000000
1110101010000111
0100000000000000
1110110000010000
0000000000011000
1110001100100000
1110111010001000
1110011111010000
0000000000010001
1110010011000100
0000000000000000
1110101010000111
0110000000000000
```

```
// Fill.asm
// Programa que llena o vacía la pantalla dependiendo de si el botón está presionado

(START)
    @KBD       // Dirección del teclado
    D=M        // D = valor del teclado (0 si no se presiona, distinto de 0 si se presiona)
    @FILL      // Si el valor del teclado no es 0, saltar a FILL
    D;JNE

    // Vaciar la pantalla
    @SCREEN    // Dirección de la pantalla
    D=A        // D = Dirección de la pantalla (empezar desde el inicio)
    @ENDSCREEN // Dirección de fin de pantalla
(CLEAR_LOOP)
    A=D
    M=0        // Poner 0 en cada dirección (vaciar la pantalla)
    D=D+1      // Incrementar dirección
    @CLEAR_LOOP
    D-A;JLT    // Repetir hasta que se llegue al final de la pantalla

    @START     // Vuelve al inicio para comprobar el estado del teclado
    0;JMP

(FILL)
    // Llenar la pantalla
    @SCREEN    // Dirección de la pantalla
    D=A        // D = Dirección de la pantalla (empezar desde el inicio)
    @ENDSCREEN // Dirección de fin de pantalla
(FILL_LOOP)
    A=D
    M=-1       // Poner 1 en cada dirección (llenar la pantalla)
    D=D+1      // Incrementar dirección
    @FILL_LOOP
    D-A;JLT    // Repetir hasta que se llegue al final de la pantalla

    @START     // Vuelve al inicio para comprobar el estado del teclado
    0;JMP

(ENDSCREEN)
    @24576     // Dirección justo después del final de la pantalla (SCREEN + 8192, ya que SCREEN = 16384)
```

Explicación General

Este código parece estar dividido en secciones que manipulan la pantalla y la entrada del teclado:

Inicializa algunas direcciones de pantalla y teclado.

Utiliza instrucciones condicionales para verificar valores en la memoria y tomar decisiones.

Realiza bucles basados en las comparaciones para repetir acciones (como llenar o vaciar la pantalla).

Al final, vuelve al inicio para mantener el programa en ejecución constante (lo cual es común en los programas Hack).

Este flujo se parece al funcionamiento del programa `Fill`, que básicamente verifica si una tecla se presiona o no para decidir si debe llenar o vaciar la pantalla. El programa está diseñado para ser cíclico, y este ciclo se ejecuta continuamente para mantener actualizada la pantalla dependiendo del estado del teclado.