ESTRUCTURA Y TECNOLOGÍA DE

COMPUTADORES

Ingeniería Técnica de Informática de Gestión

CURSO 2005-2006

Práctica 4:

Repaso de programación en ensamblador MIPS

David Miraut Andrés

Escuela Superior de Ciencias Experimentales y Tecnología

Universidad
Rey Juan Carlos

Did you know...?
If you have a digital cable set-top box, chances are it's MIPS-based.
If you have a video game console, it's probably MIPS-based.
Your email very likely travels through a MIPS-based Cisco router.
Your company's laser printers are probably powered by 64-bit MIPS-based processors.
De la Web de MIPS Technologies

Objetivo de la práctica

El objetivo de esta práctica consiste en profundizar en los conceptos de programación a bajo nivel explicados en clase, una vez ya nos hemos familiarizado con programas sencillos como los vistos en la práctica anterior[1]. De nuevo haremos hincapié en el efecto de las instrucciones sobre la memoria y los registros de la CPU, tratando de comprender mejor el porqué de lo que sucede. En concreto, trataremos los siguientes temas:

- Repaso del entorno del simulador y el juego de instrucciones del procesador.
- Programación de subrutinas con el ensamblador.
- Programación recursiva en MIPS.
- Manejo de cadenas.

Para hacer esta práctica utilizaremos nuevamente la aplicación MipsIt Studio 2000 (`MipsIt.exe`) y el simulador MipsSim (`Mips.exe`), en los que podemos escribir los programas en ensamblador y realizar su simulación[2].

IMPORTANTE:
• En esta práctica habrá que entregar una memoria escrita en la que se respondan claramente a las preguntas y cuestiones propuestas en cada apartado.
• Será absolutamente **obligatorio** comentar los resultados obtenidos.

Índice

[1] La tercera del primer cuatrimestre: Introducción a la programación en el ensamblador MIPS
[2] Si no recuerdas de dónde bajarlo ó cómo instalarlo, te invitamos a que consultes el Apéndice B de la práctica anterior.

1. Repaso del entorno de Simulación

Para calentar motores vamos a ejecutar un sencillo programa y con él repasaremos las distintas partes del simulador, los mecanismos de control de flujo, la utilización de cadenas y buena parte de las instrucciones vistas en clase.

El primer programa que vamos a estudiar (cuyo código se encuentra dentro de dos páginas) se llama `palindromo.s` , tal como su nombre indica es un programa que nos permite comprobar si una determinada cadena forma un palíndromo.

Los palíndromos son aquellas palabras, frases o números que se pueden leer igual del derecho que del revés (en el caso de los números también se les llama capicúas). Un par de ejemplos bien conocidos son: "*Dábale arroz a la zorra el abad*" ó "*La ruta natural*".

Nuestro programa de partida examinará sistemáticamente la cadena mirando y comparando los caracteres a ambos extremos, y avanzando hacia el centro en ambos lados a cada paso.

NOTA: Si en el ordenador en el que os encontráis no está instalado el ensamblador / simulador de MIPS, podéis bajarlo de:
http://dac.escet.urjc.es/docencia/ETC-Gestion/practicas-cuat2-06/
Basta descomprimirlo en un directorio para poder utilizarlo. Tened cuidado con el *path* del directorio, porque no es capaz de manejar directorios con nombres largos ni caracteres especiales.
En la misma página web se encuentra este enunciado y el código fuente de los programas de esta práctica → Repaso de programación en Ensamblador MIPS (7 de Marzo).

Es necesario iniciar tanto el ensamblador/editor (`MipsIt.exe`) como el simulador (`Mips.exe`). En el primero abriremos el proyecto `palindromo`, que se encontrará en los directorios dónde hayamos descomprimido el código de los programas de la práctica.
Léelo y trata de comprenderlo, en especial el funcionamiento de los bucles.
Una vez te hayas hecho una idea de su funcionamiento compílalo (F7) y lánzalo al simulador (F5) para ejecutarlo y ver el flujo de control paso a paso.

El listado del código comienza con una serie de comentarios (los comentarios vienen precedidos por el carácter almohadilla (*splash*) '#' en este ensamblador). Los comentarios tienen una gran importancia en los programas, y esta se vuelve crítica en el caso de los escritos en lenguaje ensamblador debido a la complejidad de comprensión que tienen los proyectos de tamaño medio. En este listado se ha omitido la explicación de algunas ideas que se pedirán en los siguientes ejercicios.

En la parte inicial de la práctica anterior (con el programa *Hola Mundo*) se explicó que las cadenas se suelen encontrar en el segmento de datos, que es un segmento distinto al del código ejecutable[3], a pesar de que el nombre de este último se preste a confusión al ser el segmento de texto.

[3] Si bien es posible meter éste tipo de datos en la zona ejecutable, es una práctica poco recomendada.

Mediante directivas podemos especificar dónde queremos guardar cada elemento (código, datos, partes del núcleo de sistema…). Por defecto, el ensamblador prepara el código para que el simulador inicie la ejecución en el segmento de texto/código y si no lo indicamos expresamente lo tomará todo como tal. En nuestro caso, el segmento de datos será todo aquello que se encuentre entre la directiva `.data` y `.text` y por tanto se colocará en éste.

Como puede leerse en el código, se reserva e inicializa el espacio para ellas con las directivas `.asciiz` (para las cadenas terminadas en cero, típicas de rutinas de lenguaje C) y `.ascii` (para cadenas con otra terminación). También existen otras directivas con las que se pueden almacenar datos en este segmento como `.byte` o `.word` pero no las utilizaremos por ahora.

Así pues, por simplicidad, nuestro programa tiene escrito "en duro" la palabra que vamos a analizar en la cadena `posible_palindromo`. Puedes cambiarla por otras expresiones[4] palindromas o por otras que no lo sean para comprobar su funcionamiento.

Una vez comienza el segmento de texto se pueden distinguir tres partes funcionales en el programa:

- Cálculo de las direcciones de memoria de los extremos de la cadena para los punteros
- Recorrido de estos punteros hacia el centro de la cadena, comparando el contenido de sus posiciones de memoria para comprobar si se trata de un palíndromo o no de acuerdo a su simetría
- Salida por pantalla del resultado de la comparación

Cuando hayáis ejecutado unas cuantas veces el programa, leedlo más detenidamente (con ayuda de la chuleta de instrucciones anexa al enunciado de esta práctica).
Ejecutad paso a paso el programa tratando de comprender cada uno de sus detalles: cambios en los registros de la CPU, correspondencia de los saltos y condiciones con las estructuras de control y bucles típicos, flujo de ejecución del programa (podéis ayudaros de la opción *PC tracking*)… tal y como se explicó con el entorno del simulador en la práctica anterior.

1.1 Ejercicios de la primera parte de la memoria:

1. Comprobar dónde se guarda la cadena de texto en el resultado ensamblado. Explicar porqué.
2. Tras las instrucciones de salto suele aparecer una instrucción NOP ¿a qué se debe?
3. Cita al menos 3 pseudoinstrucciones y su correspondiente código ensamblador real.
4. Explícalos algoritmos y dibuja el diagrama de flujo del programa, con especial cuidado en la correspondencia de saltos y condiciones con las estructuras típicas de control.
5. Los verdaderos palíndromos no tienen en cuenta los espacios, mayúsculas y signos de puntuación. Modifica el programa para que soporte espacios (supondremos que no escribimos con tildes y todo esta en minúsculas). Ayúdate de una tabla ASCII. Incluye el código de este nuevo programa junto con la memoria con el nombre: `palidromo2.s`

[4] En Español tenemos una gran cantidad de expresiones palindrómicas, se han contabilizado más de 36.000, puedes encontrar numerosos ejemplos en*: http://www.carbajo.net/varios/pal.html*

```
###############################################################################
# Fichero:          palidromo.s
# Descripción:      Verifica si una expresión forma un palidromo
# Autor:            David Miraut
# Asignatura:       ETC-II (ITIG) - URJC
# Fecha:            Febrero 2005
# Lenguaje:         MIPS-assembler, GNU/AT&T-syntax
# Web:              http://dac.escet.urjc.es/docencia/ETC-Gestion/
###############################################################################
## Uso de los registros:
## t1 - Dirección del extremo inicial de la cadena
## t2 - Dirección del extremo final de la cadena
## t3 - Carácter en el extremo inicial de la cadena
## t4 - Carácter en el extremo final de la cadena
                        # Incluimos los alias de los registros
#include <iregdef.h>
```

Segmento de datos

```
### Segmento de datos
.data
.globl posible_palindromo
posible_palindromo:
.asciz "reconocer"      # Hay miles de palindromos, puedes probar unos cuantos
                        # cambiando el valor de la cadena para comprobar el
                        # funcionamiento del programa (como: rapar, orejero...)
msg_positivo:
    .asciiz "La cadena <<%s>> es un palindromo.\n"
msg_negativo:
    .asciiz "La cadena <<%s>> no es un palindromo (%c) != (%c).\n"
```

Ejemplos de directivas

Cadenas terminadas en carácter nulo

Segmento de texto (código)

```
### Segmento de texto (programa)
.text                   # Comienzo de seccion de codigo de usuario
.align 2                # Alineamiento en formato palabra (multiplo de 4)
.globl main             # La etiqueta "main" se hace conocida a nivel global
.ent   main             # La etiqueta "main" marca un punto de entrada
main:
    la t1, posible_palindromo  # Colocarnos en el extremo inicial es fácil
## Lo ideal sería tener una función a parte que contara la longitud de la
## cadena para luego posicionar un puntero (t2) al final de ésta
## En esta versión inicial del programa lo calculamos dentro de main
    la t2, posible_palindromo  # Lo ponemos al principio
                               # y lo movemos hacia el final
bucle_longitud:
lb   t3, (t2)              # Carga en t3 el byte al que apunta t2
beqz t3, fin_bucle_long    # si t3 es igual a 0, hemos llegado
                           #   al final de la cadena
addu t2, t2, 1             # sino nos movemos un caracter en la cadena
    b bucle_longitud           # y repetimos el bucle.
fin_bucle_long:
    subu t2, t2, 1             # La cadena además de terminar en el caracter
                               # nulo tiene dos caracteres que no nos interesan
                               # De modo que nos movemos "2 bytes" hacia atrás
```

Ejemplos de directivas

Hace el cálculo para poner uno de los punteros en el extremo final de la cadena

```
bucle_palindromo:
    bge  t1, t2, es_palin      # Condición de final de bucle (explicar)
    lb   t3, (t1)              # Carga el byte en el extremo inicial (desplazado)
    lb   t4, (t2)              # Carga el byte en el extremo final   (desplazado)
    bne  t3, t4, no_es_palin   # Condición de final de bucle (explicar)
    # Si no se sale del bucle (explicar)
    addu t1, t1, 1             # Desplazamos el extremo inicial
    subu t2, t2, 1             # Desplazamos el extremo final
    j bucle_palindromo       # Y repetimos el bucle
```

Comprueba que se trata de una expresión palindrómica

```
es_palin:                  # Imprime el mensaje
    la  a0, msg_positivo
    la  a1, posible_palindromo
    jal printf
    j   fin                # Saltamos a la rutina de salida para terminar

no_es_palin:               # Imprime el mensaje
    la  a0, msg_negativo
    la  a1, posible_palindromo
    move a2, t3            # caráteres "responsables"
    move a3, t4            # de la falta de simetría
    jal printf
    j   fin                # Saltamos a la rutina de salida para terminar
```

Imprime el mensaje con el resultado

```
fin:                       # finaliza el programa (explicar su importancia)
jal     _exit
.end    main               # Final de la seccion "main"
```

2. Programación de subrutinas en el ensamblador MIPS

A estas alturas los lectores de esta práctica ya habrán comprobado que la programación en ensamblador se vuelve tediosa y caótica si no se tiene orden y cuidado, aún más si lo que buscamos es un error en el código (*debugging*) de otra persona, o de nosotros mismos pasado un tiempo.

Los lenguajes de alto nivel –como el lenguaje C- dan una serie de facilidades en forma de abstracciones que simplifican mucho la labor del programador. La noción de función supuso una revolución[5] y un gran avance en los años 70, ya que nos permite dividir el problema en partes más fáciles de abordar (*divide y vencerás*).

Estas ideas pueden ser aprovechadas igualmente en lenguajes de bajo nivel (como el ensamblador de procesadores MIPS) siendo metódicos y atendiendo a una serie de convenciones entre desarrolladores, y de esta forma crear un mecanismo similar, que nos permita:

- Hacer una correspondencia entre los parámetros (registros) que tienen la información que queremos utilizar y los parámetros formales que por convención utilicemos en este mecanismo.
- Reserva e inicialización de almacenamiento temporal. Esto es particularmente interesante si queremos que nuestros programas puedan hacer uso de recursión (como veremos en la sección siguiente): cada llamada a la función debe tener su propia copia de las variables locales, para evitar que se pisen entre ellas.

La información que describe el estado de una función durante su ejecución (los parámetros que tenga en ese momento –incluido la dirección del programa desde la que fue llamada-, los valores de sus variables locales, el puntero a la instrucción que se ejecuta en ese momento…) conforman lo que llamaremos el *entorno de la función*. En un programa de ensamblador MIPS el entorno de una función viene dado por el valor de todos los registros a los que se hace referencia en la función.

De forma genérica, justo antes de que una función A llame a una función B, ésta vuelva todo su entorno en la pila y luego ya salta a la función B. Cuando la función B termina y devuelve el control a la función A, ésta última recupera su entorno extrayéndolo de la pila.

Si bien las funciones de los procesadores MIPS pueden ejecutarse con cualquier registro, los programadores y desarrolladores han llegado a una especie de convenio implícito para el uso de los registros, de modo que el código que escriban sea compatible y respetuoso con el de los demás en entornos de programación funcional con o sin sistema operativo. En el tema 12 se explicaron esta serie de conceptos en una transparencia que reproducimos en la tabla.

[5] Además, en aquella época se programaba casi todo en lenguajes de muy bajo nivel (ensabladores) dependientes de la maquina y su arquitectura. De modo que la la falta de lenguajes protables hacía que fuera necesario reescribir todo el código cada vez que se cambiaba / actualizaba el hardware.

Registro	Número	Utilización	¿Es necesario respetarlo en las llamadas a funciones?
$0	0	Constante con valor 0	No aplicable
at	1	Reservada para el ensamblador	No
v0 – v1	2 – 3	**Valores devueltos** en las funciones	No
a0 – a3	4 – 7	**Argumentos** para funciones	No
t0 – t7	8 – 15	Temporales	No
s0 – s7	16 – 23	Temporales guardados	Sí
t8 – t9	24 – 25	Temporales	No
k0 – k1	26 – 27	Reservados para el núcleo	No aplicable (en nuestro caso)
gp	28	Puntero global	Sí
sp	29	Puntero de pila	Sí
fp	30	Puntero de Marco	Sí
ra	31	Dirección de retorno a la llamada de la función	Sí

Siendo conservadores en la arquitectura de los procesadores MIPS esto puede realizarse atendiendo a estas reglas:

- La función que llama:
  - Pone los parámetros en los registros `a0`, `a1`, `a2` y `a3`. Si hubiera más de tres parámetros, éstos se colocarían en la pila.
  - Guardar cualquiera de los registros temporales `t0` a `t7` que pudiera ser utilizado por la función que se va a llamar.
  - Ejecutar `jal` (o `jalr`) para llamar a la función (esta función se encarga de guardar la posición del contador de programa en el registro 31 `-ra-`)
- La función[6] llamada, debe antes de ejecutar su código:
  - Crear un marco de pila[7], restando el tamaño del marco de la posición del puntero a pila (`sp`). Date cuenta que el tamaño mínimo de marco en la arquitectura MIPS es de 32 bytes, por tanto aún cuando el espacio de memoria que vayamos a utilizar sea menor, deberíamos utilizar este tamaño.
  - Guardar en la pila cualquiera de los registros que se deben preservar según la tabla de arriba (`s0 – s7`, `fp` y `ra`), para evitar efectos colaterales con la función (o serie de funciones) que ha/n dado lugar a la llamada. Preservar `ra` es especialmente importante en funciones que se llaman a sí mismas (recursivas).
  - Colocar el puntero de marco en la posición del puntero de pila más el tamaño del mismo.

[6] En el enunciado de esta práctica no se distingue entre el concepto de función y procedimiento, aunque la distinción implice ciertas diferencias.

[7] Comprobaremos más adelante, cómo el compilador de C que tenemos integrado en MipsIt.exe utiliza de forma sistemática éste mecanismo, aunque pueda ser "aligerado" en casos concretos como nuestros pequeños programas.

- La función llamada ejecuta su código propiamente dicho.
- Antes de devolver el control y volver a la función que la llamo, la función invocada debe:
  - Poner lo valores de los resultados en los registros `v0` (y `v1`)
  - Recuperar los registros que se habían guardado en la pila (indicados en la tabla)
  - Devolver el control a la función que la llamo saltando a `ra` con `jr` (muy importante no utilizar de nuevo `jal`)
- Una vez recupera el control la función que había llamado, debe:
  - Recuperar los registros que se habían guardado en la pila
  - Si se habían pasado argumentos en la pila (en vez o además de `a0`, `a1`, `a2` y `a3`) se ha de recolocar su posición.
  - Extraer el valor del resultado devuelto en `v0` (y `v1`)

2.1 Ejercicios de la segunda parte de la memoria

1. ¿Por qué no se debe utilizar `jal` a la hora de regresar a la función desde la que hemos sido llamados?
2. ¿Por qué es importante preservar el registro `ra` cuando tratamos con funciones recursivas?
3. Atendiendo a las reglas enumeradas en esta sección, reescribir el código del programa de verificación de palíndromos de forma funcional, con al menos dos o tres funciones / subrutinas:
   - Una función "main" desde la que se pasa la cadena que deseamos comprobar
   - Una función que realiza dicha tarea y que a su vez llama a distintas subrutinas que:
     - Calculan la longitud de la cadena.
     - Verifican que dos caracteres sean iguales o no.
     - Imprimen el resultado de la operación.

   Incluye el código de este nuevo programa junto con la memoria con el nombre: `palidromo3.s`

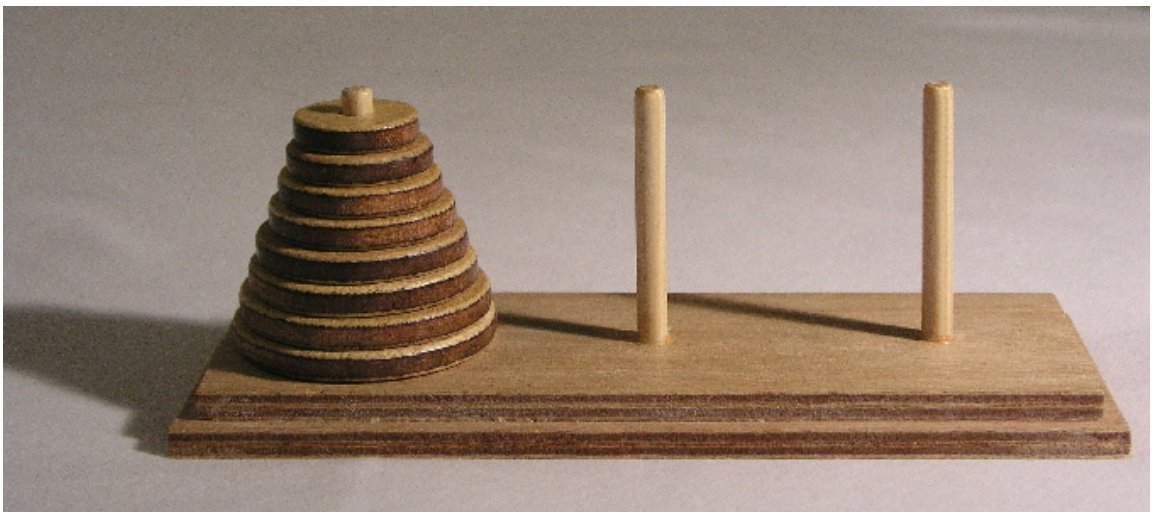
3. Programación recursiva con MIPS: Caso de las Torres de Hanoi

3.1 La leyenda

Cuenta una leyenda milenaria, que en el gran templo de Varanasi, bajo la cúpula que señala el centro del mundo, se venera una bandeja de cobre sobre la cual están colocadas tres agujas de diamante. La historia dice que en el principio de los tiempos un dios hindú, en una mañana lluviosa, colocó en una de estas agujas sesenta y cuatro discos de oro puro, ordenados por tamaños; desde el mayor, que reposa en la bandeja, hasta el más pequeño, en lo alto de la aguja. Y lo llamó la torre de Brahma.

Incansablemente, día tras día, los sacerdotes del templo mueven los discos haciéndolos pasar de una aguja a otra, de acuerdo a las indicaciones de Brahma:

- El sacerdote encargado no podrá mover más de un disco simultáneamente,
- Ni lo podrá situar encima de un disco de menor tamaño.



Si hacemos el cálculo, en este caso, el número mínimo de movimientos que necesitan es:

$2^{64} - 1$ (2 a la 64 menos 1), o sea 18,446,744,073,709,551,615 movimientos.

Suponiendo que los sacerdotes realicen un movimiento por segundo y trabajen las 24 horas del día, durante los 365 días del año, tomando en cuenta los años bisiestos tardarían 58,454,204,609 siglos más 6 años en concluir la obra[8], siempre que no se equivoquen, pues un pequeño descuido podría echar por tierra todo lo hecho.

[8] Dado que nuestro planeta tiene unos 5 mil millones de años (y la edad del Universo se estima entre 15 y 20 mil millones de años) no parece que tengamos que preocuparnos demasiado del fin del mundo, por lo menos os dará tiempo a la entrega de esta práctica.

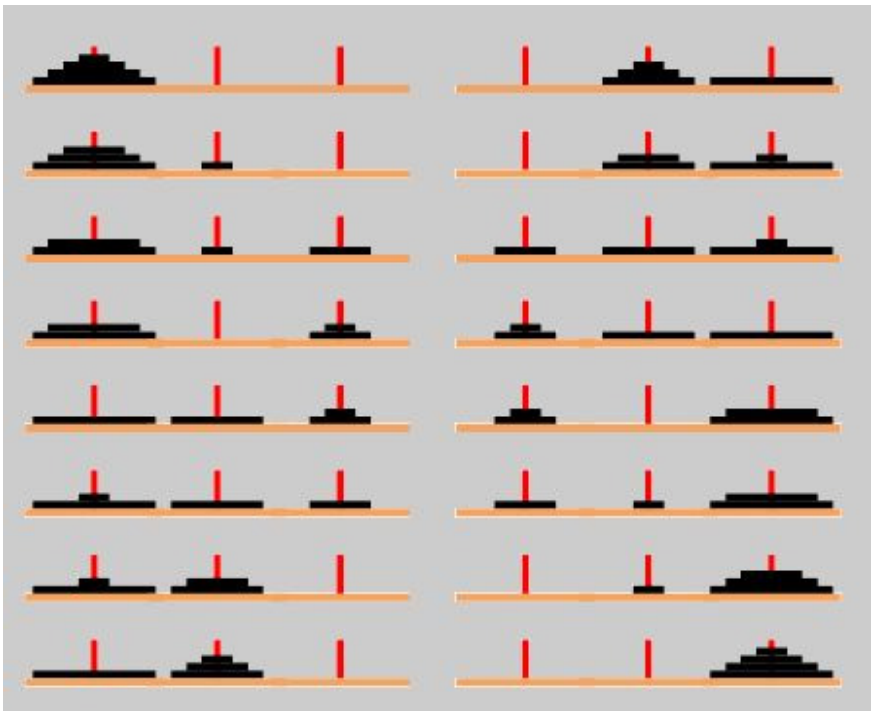
3.2 Recursividad en las Torres de Hanoi

Hoy día el templo ya no existe, pero la leyenda ha perdurado en forma de juego, y es un excelente ejemplo para explicar y practicar algoritmos recursivos.

La forma de resolver recursivamente este puzzle es muy sencilla. Si numeramos los discos desde 1 hasta n, y llamamos ORIGEN a la primera pila de discos, DESTINO a la tercera y AUXILIAR a la intermedia, el algoritmo a seguir sería el siguiente:

- Traslada la torre `1...n-1` (movemos n-1 discos) usando este mismo algoritmo, de ORIGEN a DESTINO, usando como ayuda el AUXILIAR. Esto deja sólo el enésimo disco en ORIGEN
- Lleva el disco `n` de ORIGEN a AUXILIAR.
- Traslada la torre `1...n-1` (movemos n-1 discos) usando este mismo algoritmo, de DESTINO a AUXILIAR, usando como ayuda el. ORIGEN, de modo que quedan sobre ese disco enésimo.

Una imagen vale más que mil palabras, en esta figura tenemos los pasos para resolver el caso con cuatro discos:



Probablemente viendo y ejecutando el código en C correspondiente se entienda incluso mejor, en el proyecto `hanoic` se encuentra el código de la siguiente página. Abre el proyecto desde `MipsIt.exe` (no olvides lanzar también el simulador `Mips.exe`), compílalo (CTRL+F7) y ejecútalo (F5) para distintos valores de `n`. En la consola del simulador se irán indicando los movimientos necesarios para resolver el puzzle con el número de discos indicado en la variable `n` de la función `main`.

Código C de las Torres de Hanoi

```
#include <stdio.h>

static char varilla1[] = "fuente";
static char varilla2[] = "aux";
static char varilla3[] = "destino";
int num = 0;

void Hanoi (char * origen, char * auxiliar, char * destino, int n)
{
  if (n > 0)
  {
    Hanoi (origen, destino, auxiliar, n-1);
    num++;
    printf("(Mov. %d) Mueve el disco de %s a %s \n", num, origen, destino);
    Hanoi (auxiliar, origen, destino, n-1);
  }
}

int main(void)
{
    int n = 3;     // Número de discos
    Hanoi(varilla1, varilla2, varilla3, n);
    return 0;
}
```

Primera llamada recursiva

Observa como se intercambian auxiliar y destino

Segunda llamada recursiva

Observa como se intercambia origen y auxiliar

El ensamblador/compilador `MipsIt.exe` se encarga de generar automáticamente código ensamblador a partir del código C que le damos, si abrimos el editor de ensamblador (CTRL+F9) podemos verlo[9].

Al principio aparecen unas directivas muy similares a las que utilizábamos en el programa del palíndromo, en las que se reserva espacio e inicializan los elementos del segmento de datos. El código propiamente dicho comienza en el segmento de texto con la función (etiqueta) Hanoi, si se examina cuidadosamente este código (puedes hacer uso de la tabla de la página 7 para ver la correspondencia entre los registros por nombre funcional y su número) se aprecian claramente unas secciones del código en las que se están guardando y sacando de la pila los registros, así como la preparación de los datos y su colocación en los registros para argumentos antes de llamar a la función, siguiendo las indicaciones de la sección anterior.

El código que genera el compilador es muy conservador (introduce más operaciones de las que son realmente necesarias), de ahí que la ejecución del mismo en el simulador en modo *tracking PC* sea tan lenta.

Hay algunos comentarios generados automáticamente, pero el código resultante no tiene la claridad suficiente como para ilustrar todas las cosas importantes que "esconde". Por ello y con ánimo de centrarnos en los aspectos didácticos hemos escrito una nueva versión de las Torres de Hanoi en ensamblador MIPS.

[9] Tienes también una versión de este código **en Pascal** en el Apéndice de la práctica, para que te ayude a entenderlo, aunque en ambos lenguajes las operaciones son sencillas, muy similares a pseudocódigo.

```
##############################################################################
# Descripción:     Torres de Hanoi
# Autor:           David Miraut
# Asignatura:      ETC-II (ITIG) - URJC
# Fecha:           Febrero 2005
# Web:             http://dac.escet.urjc.es/docencia/ETC-Gestion/
##############################################################################

#include <iregdef.h>
```

Segmento de datos

```
### Segmento de datos
.data
   .globl  cadena1
introduccion:
   .asciz  "Práctica 1 -> Torres de Hanoi\n"
cadena1:
   .asciz   "(%d) Mueve un disco del palo %d al palo %d \n"
```

Segmento de texto (código)

```
### Segmento de texto (programa)
.text                   # Comienzo de seccion de codigo de usuario
.align 2                # Alineamiento en formato palabra (multiplo de 4)
.globl main             # La etiqueta "main" se hace conocida a nivel global
.ent main               # La etiqueta "main" marca un punto de entrada
```

Antes de llamar a la función prepara los parámetros

```
main:   li   a0, 6          # Num. de discos
        li   a1, 1          # Palo de inicio (donde estan los discos)
        li   a2, 2          # Palo auxiliar (vacío)
        li   a3, 3          # Palo destino (adonde deberían acabar los discos)
        li   t7, 0

        jal  TdeH

        j    _exit          # Saltamos a la rutina de salida para terminar
.end  main                  # Final de la seccion "main"
```

Hace sitio en la pila y guarda los parámetros de llamada (incluido ra)

```
##      Procedimiento de las Torres de Hanoi
TdeH:
        addi sp, sp, -20    # Hace hueco en la pila para 5 elementos
        sw   ra, 16(sp)     # Guarda la dirección de llamada arriba de la pila
        sw    a0,  0(sp)     # Guarda el num. de discos
        sw   a1,  4(sp)     # Guarda origen
        sw   a2,  8(sp)     # Guarda destino
        sw   a3, 12(sp)     # Guarda auxiliar
```

Si no quedan discos acaba

```
        slt  t1, a0, 1
        bne  t1, 1, Recursion
        j ending
```

Recupera los parámetros de la pila y prepara los nuevos parametros para la llamada recursiva

```
Recursion:
        lw      a0, 0(sp)       # Recuperamos los valores de la pila
        lw      a1, 4(sp)
        lw      a2, 12(sp)      # Intercambiamos el de enmedio con el final
        lw      a3, 8(sp)
        add     a0, a0, -1      # Dismi   nuye en uno el número de discos
                                # que verá la función llamada
```

Llamada recursva

```
        jal TdeH
```

Imprime el resultado de este movimiento

```
        addi    t7, t7, 1
        la      a0, cadena1     # indicamos la cadena y aprovechamos
                                # los otros argumentos
        move    a1, t7
        lw      a2, 4(sp)
        lw      a3, 12(sp)
        jal   printf            # llama a printf que está en la libreria I/O
```

Recupera los parámetros de la pila y prepara los nuevos parametros para la llamada recursiva

```
        lw      a0, 0(sp)       # Recuperamos los valores de la pila
        lw      a3, 12(s  p)
        lw      a1, 8(sp)       # Intercambiamos el primero con el de enmedio
        lw      a2, 4(sp)
        sub     a0, a0, 1       # Disminuye en uno el número de discos
                                # que verá la función llamada
```

Llamada recursva

```
        jal TdeH
```

Restaura el valor de los registros y la posición del puntero de la pila

```
ending:
        lw      a0, 0(sp)       # carga de vuelta el num. de discos en a0
        lw      a1, 4(sp)       # E igualmente deshace la operación con el resto
        lw      a2, 8(sp)       # de los registros
        lw      a3, 12(sp)
        lw      ra, 16(sp)
        addi sp, sp, 20         # Vuelve a colocar el puntero de pila
```

Devuelve el control

```
        jr ra
```

Esta nueva versión se encuentra en el proyecto/directorio `hanoi`[10], ábrelo con MipsIt.exe, compílalo (F7) y ejecútalo en el simulador (F5). Al igual que antes, prueba recompilar y ejecutar cambiando el número de discos para ver qué sucede.

Una vez entendáis el funcionamiento general del programa, leedlo más detenidamente (con ayuda de la chuleta de instrucciones anexa al enunciado de esta práctica).
Ejecutad paso a paso esta versión de las Torres de Hanoi (para un número pequeño de discos, por ejemplo 3) tratando de comprender cada uno de sus detalles: uso de la pila en cada sección del programa, el porqué de la elección de esos registros de la CPU, el uso de las expresiones condicionales, la preparación de los argumentos de las funciones, el significado de las expresiones condicionales, el flujo de ejecución del programa (podéis ayudaros de la opción *PC tracking*)… tal y como se explicó con el entorno del simulador en la práctica anterior.

3.3 Ejercicios de la tercera parte de la memoria

1. Diferencias y semejanzas entre el código generado automáticamente y ésta versión de las torres de Hanoi. ¿Por qué la nueva versión es más rápida?
2. ¿Qué ha ocurrido con el puntero a marco? ¿es desaconsejable no hacer uso de él? ¿por qué?
3. ¿Qué registros se guardan en la pila cada vez que se ve invocada la función `hanoi`? ¿Es importante el orden en el que se hace?
4. ¿Por qué al preparar los argumentos antes de llamar a las funciones recursivas el programa coge los valores de la pila y no de los propios registros después de la etiqueta `Recursion`?
5. ¿Y por qué lo vuelve a hacer después de haber impreso el mensaje por pantalla?
6. El uso de `t7` para contar el número de pasos –desde el punto de vista de la programación en lenguajes de alto nivel- es una chapuza ¿crees que es un error en este caso? ¿si este proyecto fuera más grande (o creciera) cómo lo arreglarías?
7. Dibuja en una tabla todas las llamadas a la función Hanoi en el caso de 3 discos, indicando los argumentos que recibe y cuales de esas llamadas son "llamadas hoja" en la recursión (y por tanto no producen movimiento de discos).

Nº de llamada	Parametros que recibe la función				¿Es "llamada hoja"?
	Origen	Destino	(Auxiliar)	Nº de discos	

8. **OPCIONAL**[11]: Si observas detenidamente el movimiento indicado para los discos en la consola, apreciaras un cierto patrón en el movimiento de los discos que tiene cierta similitud con los códigos cíclicos de Gray vistos en el tema 2. Implementar la versión iterativa de las Torres de Hanoi teniendo en cuenta esta relación. Más información en: http://www.juntadeandalucia.es/averroes/iesarroyo/matematicas/taller/aspectosweb/aspectosweb.htm

[10] Sin 'c' a diferencia del proyecto anterior en el que el código estaba en lenguaje C
[11] No es necesario entregarlo, pero es un buen ejercicio. El arte de la programación se aprende con la práctica.

Apéndice A: Código fuente de los programas

Código fuente de palindromo.s

```
###############################################################################
#
# Fichero:          palidromo.s
# Descripción:      Verifica si una expresión forma un palidromo
# Autor:            David Miraut
# Asignatura:       ETC-II (ITIG) - URJC
# Fecha:            Febrero 2005
# Lenguaje:         MIPS-assembler, GNU/AT&T-syntax
# Web:              http://dac.escet.urjc.es/docencia/ETC-Gestion/
#
###############################################################################

## Uso de los registros:
## t1 - Dirección del extremo inicial de la cadena
## t2 - Dirección del extremo final de la cadena
## t3 - Carácter en el extremo inicial de la cadena
## t4 - Carácter en el extremo final de la cadena

### Segmento de datos

                            # Incluimos los alias de los registros
#include <iregdef.h>

    .data
    .globl   posible_palindromo
posible_palindromo:
    .asciiz "reconocer"    # Hay miles de palindromos, puedes probar unos cuantos
                           # cambiando el valor de la cadena para comprobar el
                           # funcionamiento del programa (como: rapar, orejero...)
msg_positivo:
    .asciiz "La cadena <<%s>> es un palindromo.\n"
msg_negativo:
    .asciiz "La cadena <<%s>> no es un palindromo (%c) != (%c).\n"

### Segmento de texto (programa)

    .text                 # Comienzo de seccion de codigo de usuario
    .align 2              # Alineamiento en formato palabra (multiplo de 4)
    .globl main           # La etiqueta "main" se hace conocida a nivel global
    .ent   main           # La etiqueta "main" marca un punto de entrada

main:

    la t1, posible_palindromo  # Colocarnos en el extremo inicial es fácil

       ## Lo ideal sería tener una función a parte que contara la longitud de la
       ## cadena para luego posicionar un puntero (t2) al final de ésta
       ## En esta versión inicial del programa lo calculamos dentro de main

    la t2, posible_palindromo  # Lo ponemos al principio
                               # y lo movemos hacia el final
bucle_longitud:
    lb   t3, (t2)              # Carga en t3 el byte al que apunta t2
```

```
    beqz t3, fin_bucle_long    # si t3 es igual a 0, hemos llegado
                               #   al final de la cadena
    addu t2, t2, 1             # sino nos movemos un caracter en la cadena
    b bucle_longitud           # y repetimos el bucle.
fin_bucle_long:
    subu t2, t2, 1             # La cadena además de terminar en el caracter
                               # nulo tiene dos caracteres que no nos interesan
                               # De modo que nos movemos "2 bytes" hacia atrás
bucle_palindromo:
    bge  t1, t2, es_palin      # Condición de final de bucle (explicar)

    lb   t3, (t1)              # Carga el byte en el extremo inicial (desplazado)
    lb   t4, (t2)              # Carga el byte en el extremo final   (desplazado)
    bne  t3, t4, no_es_palin   # Condición de final de bucle (explicar)
     # Si no se sale del bucle (explicar)
    addu t1, t1, 1             # Desplazamos el extremo inicial
    subu t2, t2, 1             # Desplazamos el extremo final
    j bucle_palindromo         # Y repetimos el bucle

es_palin:                      # Imprime el mensaje
    la  a0, msg_positivo
    la  a1, posible_palindromo
    jal printf
    j   fin                    # Saltamos a la rutina de salida para terminar

no_es_palin:                   # Imprime el mensaje
    la  a0, msg_negativo
    la  a1, posible_palindromo
    move a2, t3                # caráteres "responsables"
    move a3, t4                # de la falta de simetría
    jal printf
    j   fin                    # Saltamos a la rutina de salida para terminar

fin:                           # finaliza el programa (explicar su importancia)
    jal     _exit
    .end    main               # Final de la seccion "main"
```

Código fuente de hanoi.c

```
#include <stdio.h>

static char varilla1[] = "fuente";
static char varilla2[] = "aux";
static char varilla3[] = "destino";
int num = 0;

void Hanoi (char * origen, char * auxiliar, char * destino, int n)
{
  if (n > 0)
  {
    Hanoi (origen, destino, auxiliar, n-1);
    num++;
    printf("(Mov. %d) Mueve el disco de %s a %s \n", num, origen, destino);
    Hanoi (auxiliar, origen, destino, n-1);
  }
}

int main(void)
{
    int n = 3;     // Número de discos
    Hanoi(varilla1, varilla2, varilla3, n);
    return 0;
}
```

Código fuente de hanoi.pas

```
(* Las Torres de Hanoi en Pascal *)
(* Como referencia para aquell@s que no programen aún en lenguaje C. *)
(* 1 representa la varilla inicial, 2 la auxiliar y 3 la destino *)

PROGRAM HanoiPascal(input, output);

VAR N:integer;
VAR Num:integer;

PROCEDURE hanoi(Torigen, Tauxiliar, Tdestino, N : integer);
BEGIN
  if N > 0 THEN
  BEGIN
    hanoi(Torigen, Tdestino, Tauxiliar, N-1);
    writeln('(Mov. ',Num, ') Mueve el disco de', Torigen:1, ' a ', Tdestino:1);
    hanoi(Tauxiliar, Torigen, Tdestino, N-1);
  END
END;

BEGIN
  Num := 0;
  N := 3; (* Número de discos *)
  hanoi(1, 2, 3, N)
END.
```

Código fuente de hanoi.s

```
##############################################################################
#
# Fichero:          hanoi.s
# Descripción:      Torres de Hanoi
# Autor:            David Miraut
# Asignatura:       ETC-II (ITIG) - URJC
# Fecha:            Febrero 2005
# Lenguaje:         MIPS-assembler, GNU/AT&T-syntax
# Web:              http://dac.escet.urjc.es/docencia/ETC-Gestion/
#
##############################################################################

                          # Incluimos los alias de los registros
#include <iregdef.h>

### Segmento de datos
        .data
        .globl  cadena1
introduccion:
        .asciiz  "Práctica 1 -> Torres de Hanoi\n"
cadena1:
        .asciiz  "(%d) Mueve un disco del palo %d al palo %d \n"

### Segmento de texto (programa)
.text                       # Comienzo de seccion de codigo de usuario
        .align 2            # Alineamiento en formato palabra (multiplo de 4)
        .globl main         # La etiqueta "main" se hace conocida a nivel global
        .ent main           # La etiqueta "main" marca un punto de entrada

main:   li   a0, 6          # Num. de discos
        li   a1, 1          # Palo de inicio (donde estan los discos)
        li   a2, 2          # Palo auxiliar (vacío)
        li   a3, 3          # Palo destino (adonde deberían acabar los discos)
        li   t7, 0

        jal  TdeH

        j    _exit          # Saltamos a la rutina de salida para terminar
        .end  main          # Final de la seccion "main"

##      Procedimiento de las Torres de Hanoi
TdeH:

        addi sp, sp, -20    # Hace hueco en la pila para 5 elementos
        sw   ra, 16(sp)     # Guarda la dirección de llamada arriba de la pila
        sw   a0,  0(sp)     # Guarda el num. de discos
        sw   a1,  4(sp)     # Guarda origen
        sw   a2,  8(sp)     # Guarda destino
        sw   a3, 12(sp)     # Guarda auxiliar

        slt  t1, a0, 1
        bne  t1, 1, Recursion
        j fin

Recursion:

        lw      a0, 0(sp)       # Recuperamos los valores de la pila
```

```
        lw      a1, 4(sp)
        lw      a2, 12(sp)      # Intercambiamos el de enmedio con el final
        lw      a3, 8(sp)
        add     a0, a0, -1      # Disminuye en uno el número de discos
                                # que verá la función llamada
        jal TdeH


        addi    t7, t7, 1
        la      a0, cadena1     # indicamos la cadena y aprovechamos
                                # los otros argumentos
        move    a1, t7
        lw      a2, 4(sp)
        lw      a3, 12(sp)
        jal     printf          # llama a printf que está en la libreria I/O


        lw      a0, 0(sp)       # Recuperamos los valores de la pila
        lw      a3, 12(sp)
        lw      a1, 8(sp)       # Intercambiamos el primero con el de enmedio
        lw      a2, 4(sp)
        sub     a0, a0, 1       # Disminuye en uno el número de discos
                                # que verá la función llamada

        jal TdeH

fin:
        lw      a0, 0(sp)       # carga de vuelta el num. de discos en a0
        lw      a1, 4(sp)       # E igualmente deshace la operación con el resto
        lw      a2, 8(sp)       # de los registros
        lw      a3, 12(sp)
        lw      ra, 16(sp)
        addi sp, sp, 20         # Vuelve a colocar el puntero de pila
        jr ra
```