MEMORIA DE ARQUITECTURA Y ORGANIZACIÓN DE COMPUTADORES

Helguera López, Javier
Velasco Gil,

Índice

Índice Practica Final

PRÁCTICA 1 – Día 14/09/2016

1a)

Primera prueba a las 16:08 14/09/2016.

```
            .data
f:          word  45
G:.         word  10 24 55 67 89 90 110
H:          space 100
            .globl __start
            .text
__start:
            la $t0, f
            lw $s0,0($t0) #s0=f
            addi $s7, $zero, 0
bucle:
            la $t1, G
            lw $s1,0($t0) #s1=G
            la $t2, H
            lw $s2, 0($t2)
            add $s2, $s1, $s0
            li $v0, 1
            add $a0, $zero, $s2
            syscall
            sw $s2, 0($t2)
            addi $s7, $s7, 1
            addi $t1, $t1, 4
            addi $t2, $t2, 4
            bne $s7, 7, bucle
            li $v0 10
            syscall
```

```
            .data
f:          word  45
G:.         word  10 24 55 67 89 90 110
H:          space 100
            .globl __start
            .text
__start:
            la $t0, f
            lw $s0,0($t0) #s0=f
            addi $s7, $zero, 0
            la $t1, G
            la $t2, H
bucle:
            lw $s1,0($t0) #s1=G
            lw $s2, 0($t2)
            add $s2, $s1, $s0
            li $v0, 1
            add $a0, $zero, $s2
            syscall
            sw $s2, 0($t2)
            addi $s7, $s7, 1
            addi $t1, $t1, 4
            addi $t2, $t2, 4
            bne $s7, 7, bucle
            li $v0 10
            syscall
```

Fallaba $t0 que era $t1, un error al escribir. Luego al hacer la prueba solo cambiaba el primer valor H(0) debido a que los **la** debían estar fuera del bucle.
Una vez cambiado, ya funcionaba.

Segunda prueba a las 16:20 14/09/2016.

Aquí ya funciona correctamente.

1b)

Hemos utilizado **lui** y **ori** con la respectiva dirección de la instrucción para sustituir **la** *(Pseudoinstrucción).* En el caso de **li**, se ha sustituido también por **lui** (para limpiar el registro) y **ori**.

```
.data
f: .word  45
G: .word  10 24 55 67 89 90 110
H: .space 100
  .globl __start
  .text
__start:
  lui $t0, 0x1001
  lw $s0,0($t0) #s0=f
  lui $t1, 0x1001
  ori $t1, 0x0004
  lui $t2, 0x1001
  ori $t2, 0x0020
  addi $s7, $zero, 0
bucle:
  lw $s1,0($t1) #s1=G
  lw $s2, 0($t2)
  add $s2, $s1, $s0
  lui $v0,0x0000
  ori $v0, 0x0001
  add $a0, $zero, $s2
  syscall
  lui $v0,0x0000
  ori $v0, 0x000B
  addi $a0, $zero, 32
  syscall
  sw $s2, 0($t2)
  addi $s7, $s7, 1
  addi $t1, $t1, 4
  addi $t2, $t2, 4
  bne $s7, 7, bucle
  lui $v0,0x0000
  ori $v0, 0x000A
  syscall
```

Hay instrucciones duales, como por ejemplo **lw**, que se debería sustituir primero por **lui** y después poner **lw**, aunque sirve igualmente poner solo **lw.**

1c)

Hemos buscado en Internet para informarnos y no hay otra pseudo-instrucción que no hayamos nombrado anteriormente.
(https://en.wikipedia.org/wiki/MIPS_instruction_ set#Pseudo_instructions)

1d)

Ha disminuido bastante la legibilidad ya que se añaden bastantes instrucciones que pueden complicar la asimilación del programa.

2.

a) ✓

b) *¿Se traducen todas las instrucciones sintéticas de la misma forma que en QtSPIM?*

- No, no todas las instrucciones se traducen de la misma forma, véase que **li** lo traduce por **addiu** en *Mars*.

¿Por qué podrían traducirse las instrucciones sintéticas de forma distinta si el procesador que se está simulando es el mismo?

- Creemos que es debido a que *QtSPIM* está basado en formato Windows, pero en cambio *Mars* se basa en formato UNIX.

c) QtSPIM traduce a un **ori** con valor de 7 y *Mars* un **addi** con valor de 00000007.
- Nos parece más correcto lo realizado por *Mars* ya que se asegura que los bits de mayor peso están a 0, en cambio, QtSPIM, con el ori, simplemente añade 0007 en los de menor peso (pudiendo haber algo ya escrito en los de mayor peso).

PRÁCTICA 2 – Día 20/09/2016

Ejercicio 1:

a' La idea principal fue guardar un número (“1”) en un registro para posteriormente cargar los 8 primeros bits y comprobar si eran “0” o “1”. El error fue no darse cuenta que la instrucción **lbu** funciona con direcciones de memoria, no con registros.

Finalmente se determinó que el procesador funciona con el criterio Little-Endian.

b' Tanto QtSPIM como Mars dan el mismo resultado.

c' El bit de signo se almacena en el primer byte, es decir, en el de dirección más alta.

Para realizar este ejercicio primero se ha guardado un número en memoria. Lo importante es cargar su dirección (**la**) y de ésta extraer el primer byte por la derecha (**lb**). Una vez que se tienen estos datos se hace una comparación, si el primer byte es menor que 0xffffffff quiere decir que el número es negativo, por el contrario, si es mayor, el número es positivo. Se ha utilizado la instrucción “**slt**”.

En un principio tuvimos problemas al extraer el primer byte, ya que no escribíamos la instrucción correcta. También la comparación se intentó hacer sin **slt** pero finalmente nos dimos cuenta que no se podía, o no supimos cómo.

d' Trabajando ahora en punto flotante simple precisión, la solución es cargar el número que hay en memoria en el registro **$s0** y extraer el primer byte de este registro para saber si el número es positivo o negativo.(No hemos trabajado con variables en punto flotante). Con la instrucción “**lb**” hay que poner un 3 en el desplazamiento.

e' En punto flotante de doble precisión, en vez de guardarse el número en dos registros, se guarda entero en uno (de 64 bits), en este caso **$s0** (aunque no quepa entero). Se extrae el primer byte y se comprueba si es positivo o negativo. Con la instrucción "**lb**" hay que poner un 7 en el desplazamiento.

Ejercicio 2:

a' En este ejercicio tuvimos varios problemas. En primer lugar, el bucle para realizar la potencia no conseguíamos que funcionase bien. Para solventarlo se decidió crear dos etiquetas, una que estableciese el valor de un registro en "1" y otra que lo estableciese en "-1". Cada vez se utiliza una. Tiene el mismo resultado que la potencia del ejercicio.

El siguiente problema fue recorrer la matriz. La solución final fue establecer la fila en "0" y variar la columna hasta "3". Una vez que el número de la columna es el máximo, se pone la fila a "1" y se vuelve a variar la columna de 0 a 3.

Cada vez que se modifica la columna se realizan las operaciones pedidas en bucle y se guardan en memoria, en un vector previamente declarado, pero vacío. El resultado de sumar todo es "-8".

b' Para realizar este ejercicio simplemente se intercambiaron los registros de fila y columna, así en vez de ir rellenándose "00 – 01 – 02 …" se rellenan "00 – 10 – 20 …".

Ejercicio 3:

a'

Rojo → Señal Activa	Azul Oscuro→ Segundo Registro (rt)
Negro → Inactivo	Naranja → Immediate
Rosa → OpCode	Azul Claro → Immediate
Verde → Primer Registro (RS)	Marrón → Va a la ALU y al PC

b' El camino crítico es: Memoria Instrucciones – Banco Reg. – ALU – Memoria Datos – MUX y corresponde a la instrucción "**lw**".

PRÁCTICA 3 – Día 27/09/2016

Ejercicio 1:

En este ejercicio se ha hecho una pequeña variación a como se explicó en clase ya que nos resultó más fácil de entender así.

Se tiene un número de 32 bits cargado en un registro al cual hay que extraerle grupos de 4 bits. Para ello se utiliza la instrucción **and** con la máscara 0x0000000f. Esta es la diferencia con la solución propuesta en clase, nosotros hemos colocado la máscara en los 4 bits de menos peso.

Posteriormente se comprueba si el número es mayor o menor que 10. Si es mayor, se le suma 55 para que equivalga a su carácter en ASCII. Si es menor se le suma 48 por la misma razón. Esto se hace porque a partir del 9, en hexadecimal, se usan letras.

Finalmente se guarda el número en un vector previamente declarado.

Ejercicio 2:

a' Para realizar este programa se ha utilizado la función creada en el ejercicio anterior. Se ha colocado la instrucción **jal** que salta a dicha función. Una vez guardado todos los datos en el vector, el programa entra en un bucle (imprimir) que va extrayendo cada dato uno a uno del vector y a la vez lo va imprimiendo. Hay que tener en cuenta que nuestra función guarda los datos al revés en el vector, es decir, en vector[0] están guardado los 4 bits de menos peso y en vector[8] están guardados los 4 bits de más peso. Por ello, para imprimirlo lo que se ha hecho es adelantar la dirección hasta el final he ir restándole 4. En conclusión, recorrer el vector al revés.

b' √

c' En este apartado se ha añadido la instrucción **mul** después de pedir al usuario que introduzca el número por teclado. Con esto se consigue que el número que el programa va a convertir sea el del usuario multiplicado por dieciséis.

PRACTICA 4 – Día 4/10/2016

Ejercicio 1:

Para realizar este ejercicio hemos utilizado una dirección de memoria en la que hay almacenada una cadena en ASCII. Se extrae el primer elemento, se comprueba si es menor que 65 y se convierte a su equivalente en decimal, restando 48 o 55 dependiendo del caso.

El resultado se almacena en un registro vacio que previamente ha sido desplazado 4 bits a la izquierda (**sll**). Esto en un principio fue un problema porque primero guardábamos el número y luego desplazábamos, con lo que

al final se desplazaba una vez más, quedando el resultado final multiplicado por 16 (4bits). La operación se realiza 8 veces como marca el contador.

Ejercicio 2:

a' Primero se han cambiado los nombres de las variables para poder utilizar las funciones de syscall. En las últimas líneas se ha movido el valor al registro **$a0** y se ha imprimido. Los datos en los registros se guardan en binario aunque se impriman en decimal.

b' √

c' El problema en la realización de este ejercicio viene cuando el usuario introduce un número hexadecimal que no es de 8 caracteres. Se podría solucionar cambiando el contador del bucle principal por la longitud del vector introducido. En nuestro caso hemos supuesto que el usuario siempre va a introducir un número hexadecimal de 8 caracteres.

d' Se ha añadido la instrucción **sll** antes de imprimir el registro para multiplicar el número por 16

Ejercicio 3:

La cadena que introduce el usuario no pude ser mayor que 8 caracteres porque lo hemos limitado estableciendo $a1=9 en la función read-string.

Como nuestro programa solo acepta que la cadena sea de 8 caracteres exactos (en el momento que llega a 8, el número se lee directamente), al ir extrayendo cada carácter del string leído, si había un 10 quiere decir que el usuario ha presionado enter y por lo tanto se imprimirá el mensaje correspondiente, de que el numero introducido es demasiado corto.

También se ha implementado que el programa detecte si el usuario introduce un carácter no valido:

Lo primero que hay que limitar es que el carácter no se menor que 48 ni mayor que 102. Se hace con la instrucción **slti**. Lo siguiente será limitar que el carácter pertenezca a los rangos permitidos, los cuales son [48-57] [65-70] [97-102]. Para ello se ha utilizado una tabla de verdad:

Ejemplos	98	68	74	58
¿Válido?	SI	SI	NO	NO
<57?	0	0	0	0
<65?	0	0	0	1
<70?	0	1	0	1

<97?	0	1	1	1
TOTAL	0	2	1	3

Si nos fijamos en la tabla nos damos cuenta que los **no** validos siempre van a tener una suma de 1 o de 3.

Ejercicio 4:

Como el número más grande que se puede escribir es de 8 caracteres, siempre entrara en 32 bits. Esto es debido a que se ha limitado su longitud estrictamente a 8.

Ejercicio 5:

Para obtener el opuesto en hexadecimal hay que guardar lo que falte para llegar a 15 de cada grupo de 4 bits del registro donde está el numero introducido por el usuario, excepto para el último grupo que habrá que guardar lo que falte para 16. La implementación de esto ha sido hacer un bucle que guarde siempre lo que falta para 15. Una vez sale del bucle suma 1 al resultado.

Para imprimir de nuevo en hexadecimal, se extraen 4 bits del registro resultado con la máscara 0xf0000000, se transforma a su carácter ASCII y se imprime.

Ejercicio 6:

En este ejercicio una vez se extrae el byte de lo introducido por teclado se comprueba en que rango de valores está. Puede ser, o un número, o una letra minúscula, o una letra mayúscula. Nuestro programa trabaja con letras mayúsculas por lo que si se detecta que el usuario ha introducido una letra minúscula sólo habrá que transformarla a su equivalente en mayúscula. Esto se hace restándole 32.

Nota: en la realización del programa completo, con todas las funciones. El principal problema que tuvimos fue limitar los rangos de carácter válidos ya que buscábamos la manera de hacerlo lo más eficiente posible y sin que el código se extendiese demasiado.

Otro problema a tener en cuenta fue que el número resultado (ya en hexadecimal) se imprimía al revés. La solución fue utilizar la máscara 0xf0000000 en vez de 0x0000000f. De esta manera, los bits que se extraían podían ser imprimidos directamente sin tener que ser ordenados.

PRACTICA 5 – Día 11/10/2016

Ejercicio 1:

Para realizar este ejercicio se ha ido cargando un cierto número en ASCII de la memoria a través de la dirección guardada en **$a0**. Una vez cargado el número empieza el bucle: se comprueba que sea diferente de 10, ya que indicaría el fin de la cadena, y posteriormente, se le resta 48 para pasar de carácter en ASCII a número en decimal. Al poder ser de varias cifras, vuelve a empezar el bucle donde se vuelve a comprobar que el siguiente sea diferente de 10 en ASCII, para después multiplicar por 10 al obtenido anteriormente, y sumarle el número de ahora, al que se le ha restado 48.

Ejercicio 2:

a' La lectura de un número como String e impresión como entero se ha conseguido asignando un buffer para cargar el string, y una longitud máxima de 20 caracteres, por poner un límite simbólico. Posteriormente, se ha asignado el buffer a una dirección guardada en **$a0**, para extraer carácter a carácter, procesándolo como en el ejercicio anterior. En este caso se comprueba que el ASCII no sea de valor 10, ya que eso significaría el fin de cadena, al ser un salto de línea.

b' Para poder comprobar si el número es positivo o negativo, con introducción del símbolo, se ha tratado de forma que se comprueba el primer carácter, viendo si es “-“ (valor 45 en ASCII) para asignar a **$t3=1** y obtener al final el opuesto del numero positivo..

c' La suma de dos números introducidos por teclado ha sido bastante fácil, simplemente se ha llamado dos veces a la función que leía el número y aplicaba todas las operaciones, pero entre medias de las dos funciones se ha guardado el primer número en otra variable para que no se sobrescribiese. Como la función ya interpreta si es positivo o negativo, y lo guarda como tal, no ha surgido problema para sumar ambos números.

Ejercicio 3:

En este ejercicio se ha sustituido el nombre de las variables, siendo **$v1** el registro en el que se guardaba el número final, y **$v0** donde se guardaba información sobre el número introducido.

Se ha creado una función **errores** para imprimir en pantalla, y dependiendo del valor de $v0, imprime una cosa u otra:

$v0	**Labor**
0	Imprime la suma
1	Carácter invalido
2	El numero introducido no cabe en

	32 bits

El de tipo 0 viene de forma predeterminada, que cambia en caso de error.

El error de tipo 1, se obtiene cuando un carácter es menor que 48 o mayor que 57, por lo que no representaría un número, sino que otro símbolo. Se ha hecho la excepción únicamente en el primer carácter, permitiendo el valor 45 (-).

El error de tipo 2 se explica en el siguiente ejercicio, ya que es debido a la introducción de un número que no entra en 32 bits.

Ejercicio 4:

Para detectar el error de que el usuario introduzca un número que no quepa en 32 bits se ha implementado el siguiente código:

La base es comprobar si el número es mayor o menor que 2147483647. Obviamente no se puede hacer directamente porque si el número fuese mayor ya habría saltado un error antes de poder comprobarlo. Para ello lo que se hace es que cada vez que se extrae un carácter del string entrada y se sume a la variable donde se almacenara el resultado se le restará 2147483640. Si el resultado es negativo, no hay problema, la función sigue su curso. Si el resultado es “0” quiere decir que dependiendo del próximo número que se lea habrá overflow o no.

Para ello se lee el siguiente carácter y se comprueba si es mayor o menor que 7. Si es menor, no habrá overflow y la función seguirá su curso pero si es mayor que 7 **$v0** valdrá 2 y la función terminara.

Este valor se pasa a la función **errores** que determinará el mensaje a imprimir.

En un principio se intentó comparar el número con 2147483640 en vez de restárselo pero en la instrucción de comparación sólo caben número de 16 bits, por lo que esta opción se tuvo que descartar.

PRACTICA 6 – Día 18/10/2016

Ejercicio 1:

Este ejercicio nos pedía una función que convirtiese un número a ASCII, de forma que se quedase en la memoria. Para elaborarlo, hemos creado una función que se basa en invertir primero el número dado, para después pasarlo a carácter. Si no se invirtiese, el algoritmo nos dejaría el número

invertido al pasarlo a carácter, visto al revés, y por ese motivo se necesita invertirlo con anterioridad.

Al ir a pasar el código a una única función, hemos tenido el problema de que el **jr $ra** no volvía, porque perdía la dirección, que se ha solucionado guardando una dirección de retorno con:

addi $sp, $sp, -4
sw $ra, 0($sp)

Y recuperando luego la dirección perdida antes del **jr $ra** con:

lw $ra, 0($sp)
add $sp, $sp, 4

Ejercicio 2:

a' Se ha creado una función que lee el buffer del String, de modo que coge el primer número, le resta 48, se multiplica por 10 el registro donde se va a guardar, y se suma ese número al registro. Esto se repite en bucle hasta detectar el fin de cadena, ya que es un carácter de valor 10, que indica salto de línea.

   Este número ahora obtenido, se multiplica por 2 y se realiza lo mismo que en el ejercicio anterior: se invierte y se pasa a carácter.

b' Se ha conseguido implementar todo introduciendo limitadores de caracteres, con la excepción de que funcione con número negativos.

Ejercicio 3:

a' Para la realización de este ejercicio se ha ido aprovechando partes de códigos de prácticas anteriores.

b' Se introduce un número en hexadecimal de 8 caracteres.
   Aprovechamos la función **cargar** creada en una práctica anterior para guardar el número en decimal en el registro **$s4** (solo números positivos)**.** Luego se llama a la función **invertir** creada en el primer ejercicio, que llama a su vez a **toChar** para guardar ese número decimal como si fuese una cadena en una dirección de memoria. De este modo, en la dirección de memoria queda el número guardado de forma invertida y en decimal representado en ASCII. Este ciclo **se realiza 10 veces**, independientemente de la longitud del número que estemos tratando**, rellenando los huecos sobrantes con 0** (48 en ASCII) debido a que el máximo número representable en 32bits, tiene 10 caracteres en decimal.
   En la función de invertir, en cada bucle se suma 1 al registro **$s7** para contabilizar cuantas cifras tiene el número decimal.
   Esta variable luego nos indicará el **número de cifras a imprimir** de las 10 que se han guardado en memoria.
   Hemos detectado que cuando el número introducido en hexadecimal es muy grande, al invertirlo, ya en decimal, sobrepasa el máximo número representable, devolviendo la función un número que no es el correcto.

Conclusión: el programa no funciona con números negativos ni con números grandes. La detección de caracteres incorrectos (con la función de la practica 4) y que el número no sea de 8 caracteres está implementado. El programa admite letras en minúscula o mayúscula indiferentemente.

PRACTICA FINAL

Día 25 de octubre de 2016:

Visualizando las 3 posibilidades nos hemos dado cuenta que una vez conseguido realizar la primera opción, las otras dos restantes no suponen mucho más trabajo a parte.

Al tener los días que han pasado de una fecha a otra (opción uno), aplicando el módulo de 7 se puede ver cuantos días de la semana han pasado, y, por tanto, se sabría en qué día de la semana se está (opción 2). Implementando esto, se puede obtener la opción 3, de imprimir el mes entero, partiendo de una fecha ya conocida y almacenada.

Para comenzar, se ha implementado una función que imprime el mes dependiendo el número introducido. Este método se aprovechará para imprimir también el día de la semana (Lunes -> Viernes).
Esta basado en guardar los meses ocupando un total de 13 espacios en memoria, de forma que si el mes tiene 6 letras *(enero)* se le asigna después 7 espacios mediante la declaración **space**. Para imprimirlo, se accede al mes **restando 1** y después multiplicando por 13 a la dirección inicial donde se guardan los meses.

*Enero = 1; Se imprime desde posición 0 = (1-1)*13*

El bucle termina de imprimir en el momento que se encuentra el valor ASCII 0, ya que indica que ahí hay un espacio nulo, y por tanto, se ha acabado lo que se deseaba imprimir.

Día 8 de noviembre de 2016:

Como se explicó la semana pasada, nuestra idea es hacer tres programas en uno. Es decir, que imprima tres datos.

A lo largo de estas dos semanas se han ido añadiendo pequeños fragmentos al programa hasta completar el primer apartado, la distancia entre dos fechas.

El algoritmo para ello es totalmente de nuestra invención, no se ha consultado ninguna página ni seguido ningún método matemático concreto. Pongamos un ejemplo:

Fecha 1: 25/5/1995 Fecha 2: 2/2/2000

Lo primero es calcular la distancia entre la primera fecha y el 31 de diciembre de ese mismo año, es decir, entre 25/5/1995 y 31/12/1995. Para ello se ha definido un vector que contiene los días de cada mes. Se van sumando esos días desde el mes 5 hasta el 12. Finalmente se resta 25, por los días que ya tenía la primera fecha.

Para la segunda fecha se calcula los días de la misma manera, pero esta vez desde 1/1/2000 hasta 2/2/2000. Restando al resultado final los días máximos del mes + los días introducidos, es decir, resultado-(28+2).

Hay que tener en cuenta si alguna de las dos fechas es un año bisiesto y si llega a febrero, por ejemplo, la segunda fecha es un año bisiesto, pero no llega al 28 de febrero, asi que no influye. Esto en un principio fue un problema porque sumábamos “1” al resultado sólo con ver si el año era bisiesto.

Por ultimo sólo queda sumar los años que haya de por medio. Se hace mediante un bucle que recorre cada año de por medio comprobando si es bisiesto, si lo es suma 366 al resultado, si no, 365.

Tuvimos un problema en la función “bisiesto” porque dividíamos el año entre 10, no entre 100, por lo que cuando se introducían fechas muy separadas daba unos días de menos en el resultado. Tuvimos que ir ejecutando el programa paso a paso viendo el valor de los registros hasta que dimos con el fallo.

El usuario puede introducir una fecha más reciente como primera fecha ya que el programa lo comprobará y si es así las intercambiará. El resultado siempre será un número positivo. Se ha considerado que una distancia siempre es positiva.

Lo último que se ha añadido ha sido la entrada de datos y la detección de errores. El usuario puede introducir la fecha 2/2/1999 como 02/02/1999 o 0002/0002/1999. Funcionará igual. Lo importante es que el separador sea una barra lateral. Es importante decir que el programa traduce la entrada (dia, mes, año) introducida en string a entero y lo guarda en tres registros. Da igual lo que el usuario introduzca, se guardará en los registros. Será aquí donde se compruebe si realmente todos los datos son válidos. Si no lo son se imprime un mensaje y se acaba el programa.

Respecto a la introducción de datos por el usuario tuvimos un problema porque cada carácter ocupa 8 bits, pero por ejemplo, el año 1999 no entra en 8 bits. La solución fue leer la entraba mediante “**lb**” y almacenarlo en memoria mediante “**lw**”.

Se pretende mejorar en futuras semanas el programa para arreglar el cambio de calendario juliano a gregoriano.

Día 15 de noviembre de 2016:

Esta semana se ha trabajado en el ejercicio 2. Ahora, el programa funciona de la siguiente manera:

> Introduce la primera fecha: 25/5/1995
> Introduce la segunda fecha: 6/10/2002
> Entre las dos fechas hay 2691 dias
> El calendario del Domingo 6 de Octubre, que es la fecha mayor, es:
> (Aquí se implementará el calendario).

Para calcular el día exacto que es la fecha mayor (no la segunda introducida) se ha creado una función que calcula (utilizando la función del primer ejercicio) la distancia entre la fecha 15/10/1582 y la fecha de la cual se quiere saber el día. El programa sabe que el 15/10/1582 es Viernes, por lo tanto, se calcula el resto de dias_pasados_entre_fechas/7. Si el resto es 0, es

viernes, si el resto es 1, es Sábado etc. También se ha creado una función que toma como parámetro un número entre 1 y 7 e imprime el día de la semana equivalente. Esta misma función se reutiliza para imprimir los meses ya que están guardados de la siguiente manera en memoria:

```
semana: .asciiz "Viernes Sabado Domingo Lunes Martes Miercoles Jueves Enero Febrero Marzo Abril
Mayo Junio Julio Agosto Septiembre Octubre Noviembre Diciembre "
```

Así, para imprimir el mes habrá que sumar 7 al número de mes. Si se quiere imprimir “Febrero” habrá que pasarle a la función como parámetro el numero 9 (7+2). De esta manera se consigue reutilizar la función y el vector “semana”.

De momento el programa sigue sin funcionar con el cambio de calendario. Sólo calculara los resultados bien si las fechas son mayores que 15/10/1582.

Día 22 de noviembre de 2016:
Esta semana hemos ido ya a la implementación de la parte 3 de la práctica.
Para poder imprimir el calendario, nos hemos basado en la parte 2, de la que sabíamos que día de la semana era un cierto día. Para ellos hemos comprobado que día de la semana es el primer día de ese mes.
Como la primera semana del mes no tiene por qué empezar en Lunes, se ha guardado en **$t1** los días que quedan para acabar la primera semana, por ejemplo, si el día 1 es Viernes, esa primera semana tiene 3 días, y (7-3) 4 primeros días de la semana que no pertenecen, que serían L-M-X-J. Este número de días se guarda en **$t0.**
Se van imprimiendo 3 espacios y restando 1 al valor de $t0 hasta que $t0=0, teniendo impreso en blanco los días de la semana que no pertenecen a este mes.
Posteriormente, **$t2** llevará la cuenta del día en el que estamos.
Se realiza el bucle de los días que quedan de la primera semana:

- Se resta 1 a $t1, se suma 1 a $t2, se imprime $t2, y se imprime un espacio.
- Cada vez que se vaya a imprimir $t2 se comprueba si es menor que 10 para imprimir otro espacio, de esta forma hacemos que no se descuadre el calendario impreso.

Cuando $t1 sea 0, significa que la primera semana se ha acabado, por lo que se imprime un salto de línea.
A partir de ahora habrá otro contador, **$t5**, que llevará la cuenta hasta 7, y cuando sea 7 se imprimirá un salto de línea y se restablecerá a 0.
Ahora el bucle va imprimiendo $t2, suma 1 a $t2, imprime espacio (2 espacios si es menor que 10) y comprueba que $t5 no haya llegado a 7.
Este bucle finaliza cuando $t2 llegue al día máximo del mes, almacenado en **$t7**. Se hará una comprobación de si ese mes es bisiesto, para imprimir hasta 29.

Día 24 de noviembre de 2016:
Haciendo varias pruebas se ha comprobado que en algunos casos fallaba, por lo que nos hemos dado cuenta de que estaba mal el algoritmo para calcular el primer día del mes.

Como teníamos asignado que 0=V, 1=S, 2=D, 3=L, … , 6=J, comprobamos cómo es este resultado del módulo. Se ha realizado un ajuste:

- Si es menor que 3, se le suma 5 (Viernes=5, D=7)
- Si no es menor que 3, que se le reste 2 (L=1, J=4)

Tras esto, aparentemente ya todo funciona a la perfección:

- EL programa te dice la distancia entre dos fechas.
- Te dice en forma de texto en que día de la semana cae, y que mes es la fecha mayor de las dos.
- Te imprime el calendario de la fecha mayor.

Día 25 de noviembre de 2016:

En el transcurso de las pruebas, nos hemos percatado de que en la introducción de fechas, no aceptaba los días 29 del mes bisiesto (29/2/2000). En el código de validación de números introducidos, se ha ajustado que compruebe si ese año es bisiesto, y que le sume 1 a una variable que estaba a 0 si ese año es bisiesto.
A la hora de comprobar la fecha, si vemos que se está en febrero, se le suma a los días máximos esa variable, que era 0 si no era bisiesto, pero 1 si sí que es.
Aparentemente no se encuentran más fallos y la práctica ya debería estar terminada.

Día 1 de diciembre de 2016:

Practica finalizada.