DIARIO DE PRÁCTICAS

Darío de la Torre Guinaldo, Mario Garrido Tapias, Grupo 112

Sesión 1

Ejercicio 1:

a) Después de las instrucciones del enunciado que hay que incluir, copiamos el contenido de $s0 en $a0 para poder imprimirlo por pantalla, y después repetimos el proceso con $s1. Ambos imprimen un 1 porque es el valor de la primera posición del vector A, con la instrucción "lb" al imprimir un 1, deducimos que MIPS es Big Endian, ya que el bit más significativo se almacena en la dirección más baja. Las dos instrucciones son sintéticas porque cargan la dirección de un vector del área de datos, que es

b) Para realizar el intercambio de los vectores:

Cargamos el vector A y el vector B en $s3 y $s4 respectivamente mediante la instrucción sintética la. Después ponemos una etiqueta de Bucle1 a la que volveremos en cada iteración.

En cada iteración cargamos mediante "lw" el contenido de la dirección de los registros $s3 y $s4 en $t2 y $t3. Después comprobamos si uno de los registros es igual a 0 o no. Si es igual a 0 sale del bucle y si no es igual a 0 continúa la ejecución. Almacena el contenido de estos registros temporales mediante "sw" en los registros $s3 y $s4, pero intercambiándose. De esta forma, el contenido que antes estaba en el vector A pasa al vector B y viceversa.

Por último, se suman 4 a $s3 y $s4 para avanzar a la siguiente posición del vector y se añade una instrucción de salto incondicional al inicio de Bucle1.

Para imprimir por pantalla los componentes del nuevo vector B:

Ponemos una etiqueta de Bucle2 al principio y cargamos en $s5 mediante "lw" el contenido de la dirección del registro $s3. Comparamos mediante "beq" $s5 con $s0. Si es igual a 0 significa que ya ha imprimido todos los valores y sale del bucle. Si no continúa ejecutando y copia el contenido de $s5 en $a0 y lo imprime. Después desplazamos 4 a $s3 para avanzar hasta la siguiente posición del vector. Por último, realiza un salto incondicional a la etiqueta de Bucle2.

c) Instrucciones sintéticas usadas:

a. lw $s0, A se sustituye por lui $1, 4097 lw $16, 0($1)
b. lb $s1, A se sustituye por lui $1, 4097 lb $s17,0($1)
c. li $v0, 1 se sustituye por addiu $2, $0, 1

d. la $s3, A se sustituye por lui $1, 4097 ori $19, $1, 0

Después repetimos algunas de estas instrucciones con registros diferentes. En cuanto a las instrucciones propuestas por el guión.

- move $t0, $t1 → add $t0, $zero, $t1
- blt $t0, $t1, L → slt $at, $t0, $t1 + bne $at, $zero, L

d) Al desactivar esta opción aparecen el mismo error en todas las líneas en las que hemos usado instrucciones sintéticas. Al cambiar las instrucciones sintéticas por instrucciones máquina el programa sí compila, pero resulta más difícil de leer porque son instrucciones menos intuitivas que las sintéticas propias del simulador Mars.

Ejercicio 2:

a) Después de cargar y ejecutar el programa hemos observado una instrucción que QtSPIM traduce distinto a MARS, esta es "li", ya que MARS la traduce como un "addiu" y QtSPIM como un "ori". La razón por lo que es la transcripción es distinta puede ser que la instrucción "addiu" no se encuentre dentro del nuevo simulador.

b) Una diferencia que hemos conseguido apreciar en cuanto a las instrucciones de bifurcación es que, por ejemplo, la instrucción: "`j Bucle1`", MARS la traduce con "`j 0x00400034`" y el otro simulador con "`j 0x00400030 [Bucle1]`"; no sabemos la causa de esto, pero nos ha resultado curioso que haya una diferencia de 4 en las instrucciones de salto ya que se cumple en algunos saltos que tenemos. Por lo demás no hemos encontrado ninguna desigualdad de gran importancia. En cuanto a cual es mas correcta hemos llegado a la resolución de que la de QtSPIM ya que utiliza mayormente registros en hexadecimal y desplazamiento, no etiquetas como en el otro.

Ejercicio 3:

Hemos utilizado la función "breakpoint" para varias situaciones en el ejercicio 1, para ir depurando fallos que teníamos y ver el contenido de los registros durante los intercambios, es muy útil, ya que puedes centrarte en la zona donde consideras que esta el error y encontrarlo más fácilmente y no ir instrucción a instrucción para llegar a ese punto de ejecucción.

Bibliografía:

- Tema 2 – El lenguaje del computador (trasparencias 10 y 68): para verificar el endianness de MIPS y para ver las traducciones de move y blt.

- http://courses.missouristate.edu/KenVollmar/mars/Help/MarsHelpIntro.html : Manual de MARS para utilizarlo como apoyo para identificar instrucciones sintéticas y sus traducciones.

Sesión 2

Ejercicio 1:

Primero cargamos la dirección de la matriz A, la dirección de la matriz B y el número de filas, n, de las matrices en tres registros e inicializamos los índices i y j a 0 en los registros $t0 y $t1 respectivamente.

A continuación, ponemos una etiqueta BucleCol, una instrucción addi que pondrá a 0 el registro $t1, que contiene el índice de las columnas, y otra etiqueta BucleFil. Dentro de este bucle realizamos varias operaciones:

1. **Calculamos A[i][j]:** Para ello debemos realizar la operación 4·i·n+4·j+A. Para calcular 4·i usamos la instrucción sll para desplazar el contenido del registro $t0 dos lugares a la izquierda que es lo mismo que multiplicarlo por cuatro. Después este registro los multiplicamos por el registro que contiene n, que es el número de filas, mediante la instrucción mul. Para calcular 4·j volvemos a usar sll pero con el registro $t1. Luego sumamos los resultados mediante add y ya tenemos el desplazamiento. Ahora el desplazamiento hay que sumarlo a la dirección de A para conocer la dirección del elemento A[i][j]. El valor de este elemento lo metemos mediante lw en un registro.
2. **Calculamos A[j][i]:** Para ello debemos realizar la operación 4·j·n+4·i+A. Hacemos exactamente los mismo que en la operación anterior, pero cambiando el registro $t0 por $t1 y viceversa.
3. **Restamos A[i][j]-A[j][i]:** Restamos los valores mediante la instrucción sub y almacenamos el valor con sw en la dirección de la matriz B.
4. **Incrementamos las columnas:** mediante addi $t1, $t1, 1
5. **Avanzamos a la siguiente posición de B:** Para ello, sumamos 4 al registro en el que guardamos la dirección de la matriz B.

Después de realizar estas operaciones comprobamos si $t1, que contiene el índice de columnas, es menor que el número de filas. Si es menor vuelve a la etiqueta BucleFil, y si es mayor continúa con la ejecución e incrementa el registro $t0, que contiene el índice de filas. Por último, hace otra comparación con la instrucción blt. Si $t0 e menor que el número de filas, vuelve a la etiqueta BucleCol, y si es mayor ya ha terminado el proceso.

Ejercicio 2:

Primero cargamos la dirección de la matriz A que queremos imprimir en un registro y el número de filas en otro. Después inicializamos a 0 $t1 y $t2, que serán i y j respectivamente y escribimos una etiqueta BucleCol.

A continuación, escribimos una instrucción que reinicie las columnas como en la función anterior, y una instrucción que sume a $a0 el valor 10 para después hacer una llamada al sistema con li $v0 11 para que escriba el carácter que contiene $a0, que es el de nueva línea.

Luego escribimos la etiqueta BucleFil y dentro de este bucle nos desplazamos hasta cada elemento A[i][j] mediante la fórmula 4·i·n+4·j+A como en el apartado anterior, y una vez que tenemos el valor de A[i][j], lo pasamos al registro $a0 y llamamos al sistema con li $v0, 1 para que escriba el entero que contiene esa dirección de la matriz. Ahora cargamos en $a0 el valor 9 y llamamos al sistema para que imprima el carácter, que es el tab horizontal, para que los números queden bien espaciados.

Por último, se hacen las comprobaciones de los dos bucles igual que en el ejercicio anterior.

Ejercicio 3:

Primero reservamos espacio en la zona de datos para la matriz, A, que queremos introducir, para la matriz, B, que queremos crear para imprimir por pantalla, para el número de filas y columnas, n, que es 5 y los distintos strings que se van a imprimir durante la ejecución.

En el área de código hacemos un bucle dentro de un bucle similar al de los apartados anteriores para ir pidiendo cada elemento de la matriz. Primero inicializamos $t0 y $t1 a 0, que llevarán el índice de filas y de columnas respectivamente. Después ponemos una etiqueta PeticionF, una instrucción addi para incrementar el índice de columnas y otra etiqueta PeticionC, y en cada iteración:

1. Imprime un string que pide que se introduzca un número.
2. Imprime el número de fila.
3. Imprime un espacio.
4. Imprime el número de columna.
5. Imprime otro string para completar la frase.
6. El usuario introduce el número.
7. Incrementa el índice de columnas.

Después hace una comparación, incrementa las filas y hace otra comparación igual que en los apartados anteriores.

Una vez que tenemos la matriz, restamos 100 al registro que contiene la dirección del último elemento de la matriz para volver a la dirección inicial de la matriz. Ahora utilizamos el método del ejercicio 1 para calcular la traspuesta y colocarla en la dirección de B. Después utilizamos el método del ejercicio 2 para imprimir los elementos de la matriz traspuesta calculada.

Al terminar la ejecución se pregunta al usuario si quiere repetir el proceso con otra matriz 5x5.

Bibliografía:

Estructura y diseño de computadores. La interfaz hardware/software. David A. Patterson, John L. Hennessy. Sección 2.7 y 2.8.

Sesión 3

Martes, 2 de octubre de 2018

Ejercicio 1:

Para empezar, inicializamos un registro con el número 15, que es F en hexadecimal y en la siguiente instrucción lo desplazamos 28 posiciones a la izquierda para conseguir una máscara con la que extraer los bits del parámetro en grupos de 4 bits en el orden correcto.

Ahora hacemos un bucle en el que:

1. Hacemos la operación and entre el número parámetro y la máscara.
2. Desplazamos este resultado 28 lugares a la derecha.
3. Corremos el número parámetro 4 posiciones a la izquierda para en la siguiente iteración comparar el siguiente grupo de 4 bits.
4. Creamos una bifurcación en la que, si el resultado del and está entre 0 y 9 se le suma 48, y si está entre 9 y 15 se le suma 55, para obtener el carácter ASCII correspondiente.
5. Almacenamos el resultado del and más la suma en la dirección donde debe estar la cadena en ASCII.
6. Sumamos 4 a esta dirección para recorrer la cadena guardando cada uno de los caracteres.

Si hemos recorrido el número entero sale del bucle. Después vuelve al inicio de la cadena para iniciar el bucle en el que va imprimiendo cada carácter. Cuando detecta el carácter de fin de línea termina la ejecución.

Miércoles, 3 de octubre de 2018

Ejercicio 2:

a) En la zona .data reservamos espacio con .space para ir guardando los caracteres y en la zona de código cargamos en $a0 la dirección de la cadena. Después pedimos la entrada por teclado de un entero y lo copiamos en el registro $a1 para pasarlo como parámetro en la función. Esta vez, en la función almacenamos los valores en ASCII con store byte para después poder usar la instrucción de imprimir cadena.

b) Después de probar el programa con los diferentes valores, obtuvimos los siguientes resultados:

- 1: El programa escribe 00000001
- -1: El programa escribe FFFFFFFF
- 100: El programa escribe 00000064
- -1024: El programa da error
- 255: El programa escribe 000000FF
- 2147483647: El programa escribe 7FFFFFFF que es el máximo número positivo que cabe en 32 bits
- -2147483648: El programa da error
- -1073741824: El programa da error

c) El programa empieza imprimiendo un texto de petición para que el usuario introduzca cualquier número decimal, el cual se guarda en parámetro $a1, para luego trasladarlo al registro $t0 con el que trabajará la función. Ya en la función, el número introducido es multiplicado por 4, es inicializado un registro con el número 15, que es 0xF en hexadecimal para en la siguiente instrucción desplazarlo 28 posiciones a la izquierda para así conseguir una máscara con la que aislar los bits en grupos de 4 en 4 para después imprimirlos. Las siguientes líneas de código consisten en la construcción de un contador, para que controle el número de iteraciones al ir guardando cada grupo de 4, este llegara hasta 9, que son las iteraciones necesarias para guardar los números extraídos más el salto de línea. Después hay dos variables locales, una inicializada con el valor 10, para realizar comparaciones y para guardar el carácter de fin de línea, y la otra con el valor 8, para realizar la confrontación con el contador anterior.

La función realiza las mismas instrucciones que en el ejercicio 1, realiza un "and" para aislar el primer valor, lo suma 48 o 55 dependiendo si esta entre 0 y 9 o no, y va desplazando el numero hasta conseguir los 8 caracteres que es cuando guarda un fin de línea. Por último, guarda la cadena resultando en un registro de retorno y salta a la dirección de retorno, donde se imprime un mensaje para indicar el número y lo que se ha hecho con él.

Cabe mencionar un cambio en el algoritmo, respecto al ejercicio uno ya que, probándole, nos dimos cuenta de que colocando un 12 que multiplicado por 4, es 48_{BN} – $0x00000030_{HEX}$ imprimía el ultimo 0, ya que el algoritmo bifurcaba cuando $t5 era 0, pero con el nuevo contador explicado anteriormente solucionamos esto.

d) Después de probar el programa con los diferentes valores, obtuvimos los siguientes resultados:

- 1 → 00000004, el resultado es correcto.
- -1 → FFFFFFFC, el resultado es correcto.
- 100 → 00000190, el resultado es correcto.
- -1024 → FFFFF000, el resultado es correcto.
- 225 → 00000384, el resultado es correcto.
- 2.147.483.647 → FFFFFFFC, el resultado es correcto.
- -2.147.483.648 → 00000000, el resultado debería ser FFFF FFFE 0000 0000, pero se sale de rango.
- -1.073.741.824 → 00000000, el resultado debería ser FFFF FFFF 0000 0000, pero al igual que el anterior, el rango se sale del especificado.

Viernes, 5 de octubre de 2018

Ejercicio 3:

a)

Lleva al CONTROL el código de operación, para que se activen las señales correspondientes.
Corresponde al campo “rs” de la instrucción.
Corresponde al campo “rt” de la instrucción.
Todas las señales de control que están activadas dependiendo de la instrucción
Corresponde al resto de señales, es decir las inactivas, que salen de control y se distribuyen por toda la ruta de datos.
Lleva el valor que escribiremos en el Banco de Registros, es decir el resultado.
Primero en él se encuentra el campo “funct” de la instrucción que llega al ALU Control, en el que se junta a la señal ALUOp.
Contador de programa (PC).
Contador de programa ya incrementado.
Instrucción bifurcada.
El desplazamiento en las instrucciones tipo I, con su correspondiente extensión de signo y suma.

b) Según la representación que aparece en el apartado de herramientas MIPS X-Ray, el camino crítico sería el de las instrucciones de carga de datos, es decir, “lw”. Ya que si nos fijamos en la ruta de datos de la diapositiva 1.26 del Tema 1 de la asignatura, es igual.

c) Algunos de los cambios son los representados en la diapositiva 1.29 que incrementan las instrucciones de salto, de una manera más sencilla.

Bibliografía:

Estructura y diseño de computadores. La interfaz hardware/software. David A. Patterson, John L. Hennessy. Sección 2.9

Arquitectura y organización de computadoras. Tema 1, transparencias 1.26 y 1.29.

Sesión 4

Martes, 9 de octubre

Ejercicio 1:

Primero cargamos en un registro la constante 57, que servirá para diferenciar si el carácter que se está tratando es un número o una letra.

Después hacemos un bucle en el que:

1. Cargamos en un registro un byte de la dirección de la cadena, que contendrá un carácter en ASCII.
2. Hacemos una bifurcación dependiendo del valor del carácter. Si es mayor que 57 entonces es una letra y se le resta 55. Si es menor que 57 entonces es un número y se le resta 48.
3. Realizamos la operación or entre el número calculado y otro registro que en la primera iteración contendrá un 0 y se guarda en este último registro.
4. Sumamos 1 a la dirección de la cadena para colocarnos en la posición del siguiente carácter.
5. Cargamos ese carácter con lb para comprobar si se ha acabado o no la cadena. Si el valor del carácter es 0, significa que es el fin de cadena y sale del bucle. Si es distinto de 0 desplaza el número 4 posiciones a la izquierda y vuelve a hacer el bucle.

Por último, copiamos este número en el parámetro de retorno.

Jueves, 11 de octubre

Ejercicio 2:

a) Definimos en la zona .data una cadena de caracteres y en la zona .text, cargamos la dirección de la cadena en un registro. Después llamamos a la función del ejercicio 1, que devuelve en $v0 el valor del número y copiamos este valor en $a0 para poder imprimirlo por pantalla.
b) Cadenas:
    a. 000000ff: El programa devuelve 255 y es correcto.
    b. 0000ffff: El programa devuelve 65535 y es correcto.
    c. ffffffff: El programa devuelve -1 y es correcto.
    d. 7fffffff: El programa devuelve 2147483647 y es correcto.
    e. 80000000: El programa devuelve -2147483648 y es correcto.
c) Ahora en vez de inicializar la cadena en el área de datos, lo pedimos por teclado con li, pero antes tenemos que cargar en $a1 la longitud máxima del string que se va a introducir por teclado. El problema de pedir por teclado la cadena es que el último carácter de la cadena es un carácter de fin de línea (“\n”) en vez de un carácter de fin de cadena (“\0”). Para solucionarlo hemos creado una función a la que llamamos justo después de introducir la cadena por teclado para que la modifique.
   En esta función primero inicializamos un registro con la constante 10, que es el valor del carácter de fin de línea y otro registro a 0 que se utilizará como contador del número de caracteres. Después hay un bucle en el que leemos el primer carácter de la cadena,

sumaos 1 a la dirección de la cadena y al contador. Si el valor del carácter es distinto de 10 se repite el bucle, y si es distinto se termina el bucle y se cambia ese 10 por un 0. Antes de devolver el control al programa principal se resta al registro que contiene la dirección de la cadena el registro contador para volver al principio de la cadena.

d) Para dividir por 4 simplemente usamos la instrucción srl para desplazar el número devuelto por la función 2 posiciones a la derecha, que es equivalente a dividir entre 4.

Ejercicio 3:

Dentro de la función en el apartado inicializaciones cabe destacar la adicción de 4 nuevas variables locales: $t6, el cual es un contador del número de caracteres que tiene el String introducido por pantalla, este será comparado una constante con el valor 8, que se localiza en el registro $t9, número máximo de caracteres que puede tener la codificación en hexadecimal; $t4 y $t5, corresponde a otra constante con el valor 9 y 15 respectivamente, que nos servirán para verificar que los caracteres analizados entran dentro del rango válido: 48-57 [0-9], 65-70 [A-F], 97-102 [a-f]; con respecto al código ASCII. En lo que respecta al código que detecta si la cadena insertada es demasiado larga consiste en una comparación de $t6 y $t9 para ver si son iguales para cuando no lo sean, salte hacia un apartado de la función en el que carga en $v0 el valor 1 que pertenece al error de haber introducido una cadena más larga de lo permitido, y en $v1 deja un 0. Para verificar si los caracteres son válidos, lo que hemos hecho es en *Menorque10* tenemos una bifurcación que si después de restar 48, para establecer el número en binario, verifica que el número no es menor que 0, es decir serian [0-47] en ASCII, en *Mayuscula* realiza 2 comparaciones un para ver si es menor o igual a 9, es decir [58-64], y otra para ver si son mayores que 15, el intervalo [71-96], igual que cuando entramos dentro de las minúsculas que comprobamos el intervalo no válido [103-fin de ASCII]. Si alguna de las condiciones se cumple bifurcará a un lugar en que $v0 pasa a contener el valor 2, que representa carácter no válido, e de igual modo $v1 será 0, para indicar que no hicimos ninguna operación. Si ninguna de las condiciones anteriores se cumple en $v0 sacaremos un 0, lo que significara que todo está correcto, y por $v1, el resultado en decimal. Las impresiones se realizan al recuperar la dirección de retorno, fuera de la función como se indicaba en los requisitos.

Cabe destacar que dedicamos cierto tiempo a la depuración de este código, ya que en algunas ocasiones las comparaciones necesitaban un mayor o menor estricto y en otras un mayor/menor e igual. Esto lo fuimos comprobando con una paleta de pruebas, con un ejemplo de cada uno de los intervalos no válidos que eran: $, =, W y w.

Ejercicio 4:

Para poder distinguir entre mayúscula y minúsculas definimos otra constante en el inicio de la función con valor 97. Ahora al tratar un carácter primero averiguamos si es menor que 57 (número) o mayor que 57(letra), si es número le restamos 48 y si es letra tenemos que averiguar si es menor que 97 (mayúscula) o mayor que 97 (minúscula). Si es mayúscula se le resta 55 y si es minúscula se le resta 87.

Ejercicio 5:

- 0x000000FF → 255. El resultado es correcto: 15*16+15.
- 0x0000FFff → 65535. El resultado es parcialmente correcto ya que coge distintos tipos de caracteres, mayúsculas y minúsculas, lo analiza como si fueran iguales, es decir: $15*16^3+15*16^2+15*16+15$.
- fffffffg → Carácter incorrecto. La "g" no está dentro de nuestros intervalos válidos.
- Affffffff → 268.435.455. Ocurre lo mismo que en el apartado b.
- 80000000 → El programa devuelve -2.147.483.648, el resultado es correcto.
- FFfffffffff0 → Cadena demasiado larga. El código hexadecimal debe tener solo 8 dígitos y este número introducido tiene 11.
- Affffffffg → Carácter incorrecto. Debido al orden del código detecta antes un carácter no valido a una cadena demasiado larga.

Ejercicio 6:

Sobre el código realizado en el ejercicio 4, lo primero que hicimos fue guardar en el apartado ".data" tres Strings con cada uno de los mensajes de error, llamados *TodoOk*, *CaracErr*, *CadLarga*. El otro cambio que habría que hacer es, en los trozos de código que asignábamos a $v0 su código de error correspondiente, cargar las direcciones de las 3 cadenas mencionadas anteriormente. Y en vez de poner el código 1 para imprimir enteros, colocar el 4 que nos imprime strings.

Ejercicio 7:

En la zona .data inicializamos con .asciiz todas las cadenas que vamos a necesitar durante la ejecución del programa.

En la zona .text primero cargamos en dos registros de tipo s los valores 1 y 2 que servirán para tratar el código de error que devuelva la primera función. A continuación, pedimos la entrada por teclado de un número hexadecimal y llamamos a la función LimpiarCadena, para que cambie el carácter de fin de línea por el fin de cadena. Después llamamos a la función HexToBin, que transforma el número hexadecimal en binario y que devuelve en $v1 el número en binario y en $v0 un código de error. Si vale 0 no pasa nada y sigue con la ejecución. Si vale 1, imprime por pantalla que la cadena es demasiado larga y vuelve a pedir la entrada. Si vale 2, imprime por pantalla que se ha introducido un carácter erróneo y también vuelve a pedir la entrada.

Si no ha habido errores, divide el número entre 4 con la instrucción srl y llama a la función BinToHex, que transforma un número binario en hexadecimal. La cadena devuelta se imprime por pantalla y el programa termina.

Bibliografía:

Estructura y diseño de computadores. La interfaz hardware/software. David A. Patterson, John L. Hennessy. Sección 2.4.

Sesión 5

Martes, 16 de octubre de 2018

Ejercicio 1:

Copiamos la dirección de la cadena que se pasa como parámetro a otro registro para trabajar con ella y cargamos la constante 10 en un registro. Con lb cargamos el primer carácter de la cadena en ASCII para comprobar si es un signo negativo (45 en ASCII), y en ese caso guardamos un 1 en el registro $t9. Después hacemos un bucle para recorrer toda la cadena. En cada iteración:

1. Cargamos un byte de la cadena. Si se detectó que era un signo negativo entonces si es la primera iteración se pasa al segundo byte.
2. Se le resta 48 al carácter para hallar su valor real.
3. En el registro $t3, se suma este número con los anteriores resultados y se van acumulando.
4. Se comprueba si el carácter siguiente es el fin de cadena, y en ese caso se sale del bucle. Si todavía hay otro número, el registro $3 se multiplica por 10.

Al terminar el bucle se comprueba si el número es negativo gracias al bit guardado en el registro $t9. Si es positivo devuelve el resultado tal y como está. Si es negativo lo complementa antes de devolverlo.

Ejercicio 2:

a) Reservamos espacio en el área de datos para la cadena que pedimos por teclado. Pero antes de llamar a la función que la convierte en binario, primero llamamos a la función que creamos en la práctica anterior que limpia la cadena y cambia el carácter de fin de línea por el de fin de cadena. Cuando tenemos el resultado final lo imprimimos.
b) Cadenas:
    a. 25 → El resultado es 25 y es correcto.
    b. -048 → El resultado es -48 y es correcto.
    c. F024 → El resultado es 22024 porque se introduce un carácter que no es un número.
    d. -5 → El resultado es -5 y es correcto.
    e. 2147483647 → El resultado es 2147483647 y es correcto porque es el número positivo más grande que cabe en 32 bits.
    f. -5000000000 → El resultado es -705032704 y es incorrecto porque ha habido desbordamiento ya que este número no cabe en 32 bits.
    g. -2147483648 → Se produce desbordamiento porque ese número en positivo no cabe en 32 bits, aunque en negativo sí que cabe.
c) Después de que la función de convertir de decimal a binario devuelva el resultado llamamos a la función de convertir de decimal a hexadecimal e imprimimos por pantalla este resultado.

Viernes, 19 de octubre de 2018

Ejercicio 3:

En esta práctica el tratamiento de errores era más sencillo, ya que para el caso de "carácter erróneo" con dos instrucciones de comparación, "*blt*" y "*bgt*", una para ver si es menor que 0 y otra para comprobar si es mayor que 10 basto. Si alguna de estas condiciones se cumple bifurca a una etiqueta en la que en *$v0* se guarda un 1. En cambio, para detectar una cadena introducida de mayor tamaño que el permitido existen 2 ocasiones en las que puede detectarlo; una es después de haber hecho la instrucción "*addu*" en la que comprobamos que el resultado obtenido no es negativo, ya que en ese caso daría desbordamiento. Y el otro caso es gracias a verificar que el apartado "*hi*" no es distinto de 0 [1], en ambos casos bifurca y carga un número 2. Después devuelve la dirección de retorno, imprime el código de error y volver a pedir números o en su defecto tira al caso de *SinErrores*.

Ejercicio 4:

a) Para este apartado tuvimos que añadir 2 comparaciones "*beq*" después de recuperar la dirección de retorno para identificar el código de error e imprimir su mensaje correspondiente. En este momento nos dimos cuenta de un fallo del código, el cual consistía en introducir caracteres aceptables junto con otros que no, por tanto, en *$t3* donde almacenamos el resultado, se quedaban restos y si después de varios errores introducíamos algo correcto el resultado difería del correcto. Como solución reiniciamos el valor del registro nada más comenzar la función.
   Además, esta vez no almacenamos constantes en un registro para después usarlas en las comparaciones, sino que usamos las constantes directamente en las instrucciones de comparación, transformándolas en instrucciones sintéticas.
b) Cadenas:
   a. 25 → 00000019. El resultado es correcto 1*16+9 = 25.
   b. -048 → FFFFFFD0. El resultado es correcto
   c. F024 → Carácter incorrecto. El programa detecta la "F" como carácter no valido ya que es mayor que 10.
   d. -5 → FFFFFFFB. El resultado es correcto.
   e. 2147483647 → 7FFFFFFF. El resultado correcto ya que es el mayor valor positivo permitido.
   f. -5000000000 → Cadena demasiado larga. El programa detecta bien que el valor es más grande del que se puede meter.
   g. -2147483648 → Cadena demasiado larga. Da fallo ya que al eliminar el guion lo desarrolla como si fuera positivo por tanto es mayor que el caso e.

Bibliografía:

Estructura y diseño de computadores. La interfaz hardware/software. David A. Patterson, John L. Hennessy. Sección 2.4.

[1] Resumen ensamblador MIPS32.pdf, para buscar una instrucción que nos permitiera obtener el valor de "*hi*".

Sesión 6

Martes, 23 de octubre de 2018

Ejercicio 1:

En cuanto al apartado *.data* tenemos 2 reservas de espacio, una para el número en binario y otra para la cadena en ASCII.

El programa consta de 2 funciones, la primera es *BinToDec*, a esta la llegan 2 parámetros: $a0 con la dirección donde dejaremos la cadena de caracteres en ASCII, y $a1 con el número en binario introducido por el usuario. Para empezar, movemos a registros temporales estos parámetros y establecemos una constante de división, con el valor 10. Después entramos en el bucle en el cual dividimos el numero en binario entre la constante con ello logramos que en "*hi*" se guarde el resto y en "*lo*" el cociente, el primero será al que sumaremos 48 para pasarlo al código requerido, el segundo será el nuevo numero que dividiremos, esto se realizara hasta que el cociente sea 0. En cada iteración tras cambiar de código, guardamos el número transformado en la dirección pasada como parámetro, después incrementamos esta para prepararle para una nueva incorporación. Lo explicado anteriormente esta basado en el "Algoritmo de Cambio de Base"[1], que básicamente es dividir el número a cambiar por base a la que queremos cambiar y en el resto nos va quedando los dígitos del numero en la base deseada.

El problema de esta función es que nos devuelve la cadena invertida, aquí entra la otra función *InvertirCad*, consiste en hacer 2 copias a la cadena en ASCII, una de ellas se recorrerá hasta el final, mas tarde comenzara un bucle en el que se irán intercambiando bytes para colocar la cadena correctamente, su funcionamiento consiste en cargar, guarda y, incrementar o decrementar, dependiendo de la copia.

Ejercicio 2:

a) Para esta parte, tuvimos que añadir 3 líneas de código al realizado en el ejercicio 1, estas consistían en una instrucción "*slti*", en la que colocaríamos un 1 en *$t9*, si el numero fuera negativo, y un 0 en caso contrario, mediante una condición si se cumple la primera condición hacemos *$0* – el numero a convertir, para así conseguir su valor absoluto.
b) Cadenas:
    a. 1 → 1, el resultado es correcto
    b. -1 → 1, el resultado es correcto, valor absoluto satisfactorio.
    c. 100 → 100, el resultado es correcto
    d. -1024 → 1024, el resultado es correcto, valor absoluto satisfactorio.
    e. 255 → 255, el resultado es correcto
    f. 2147483647 → 2147483647, el resultado es correcto
    g. -2147483648 → Aritmetic Overflow, al intentar hacer el valor absoluto del número, salta un error de overflow, ya que el máximo positivo es del del apartado anterior.
    h. -1073741824 → 1073741824, el resultado es correcto, valor absoluto satisfactorio.
    i. 1073741824 → 1073741824, el resultado es correcto

Viernes, 26 de octubre de 2018

Ejercicio 3:

Primero inicializamos en .data todas las cadenas que vamos a usar y reservamos espacio para las dos cadenas de hexadecimales que se piden y para la cadena en cifras decimales del resultado final.

En la zona de código pedimos que se introduzca por teclado el primer hexadecimal, llamamos a la función que limpia esta cadena y después a la función que lo transforma en binario y el resultado lo almacenamos en un registro salvado. Si el hexadecimal introducido no es válido, se mostrará un error por pantalla y se volverá a pedir la entrada de datos. Este mismo proceso se repite para el segundo número hexadecimal.

Ahora, antes de pasar la suma de los dos números a decimal hay que comprobar si esa suma produce desbordamiento usando la instrucción addu, que no da error si hay desbordamiento al sumar. Para ello analizamos el signo de los dos números. Si son de distinto signo no habrá riesgo. Si son los dos positivos, habrá desbordamiento si la suma da negativa y si los dos son negativos, habrá desbordamiento si la suma da positiva. Si se produce desbordamiento, se mostrará un mensaje por pantalla y se vuelve a pedir la entrada de datos.

Si no se ha producido desbordamiento, llamamos a la función que transforma este número en decimal y lo imprimimos por pantalla.

Anexo:

Para completar el diario anterior, explicaremos el "Algoritmo de Horner" [2] brevemente, se basa en colocar los coeficientes del polinomio en una tabla junto con el valor de x que quiere evaluarse. Bajamos el primer coeficiente y lo multiplicamos por el valor de x colocando el resultado debajo del siguiente coeficiente en la tabla, sumamos los dos valores obteniendo un nuevo resultado parcial y repetimos esa operación para cada coeficiente.

Bibliografía:

Estructura y diseño de computadores. La interfaz hardware/software. David A. Patterson, John L. Hennessy. Sección 2.4.

[1]http://www.disfrutalasmatematicas.com/numeros/base-cambio-metodo.html

[2] https://juncotic.com/metodo-de-horner-algoritmos-antiguos/

Sesión 7

Martes, 30 de octubre de 2018

Hoy hemos comenzado a estructurar la Práctica Final, lo primero que hicimos fue un esquema general, de cómo sería nuestro programa, este fue algo tal que así:

En cuanto al apartado *.data*

- Espacio para 2 *String*, en los que almacenaríamos las 2 fechas a introducir por el usuario y que denominaríamos *CadFecha1* y *CadFecha2*.
- Varios *arrays* para guardar: distintos trozos de cadenas, los enteros ya transformados para operar, o el resultado final.
- Cadenas de impresión para: pedir entrada, sacar los distintos errores o en caso contrario el resultado.

En cuanto al apartado *.text* seleccionamos las necesidades que teníamos para ver el número de funciones que nos requerirían, algunas de las que apuntamos son estas:

- Separar cada fragmento de tiempo, es decir, días, mes, año, hora, … cada uno por su lado. Para ello necesitaríamos una funcionan que nos fuera dividiendo el *String* y que lo fuera guardando en *arrays* ordenados, teniendo en cuenta los separadores (*).
- Validación de cada ámbito por separado, o lo que es lo mismo, una función para cada uno de ellos con el objetivo de comprobar:
    - Año → 0-Infinito
    - Mes → 1-12
        - Comprobación de Si/No es BISIESTO
          En una función que lo compruebe, devolverá un 1 si es cierto y 0 si no lo es.
        - Comprobación de 30/31/28/29 días de MES
    - Días → 1-DiasDeMes
    - Hora → 0-23
    - Minutos → 0-59
    - Segundos → 0-59
- Transformación de DECIMAL-BINARIO para operar con ellos, momento en el cual utilizaríamos la función hecha en práctica anterior.

(*) Caracteres válidos como separadores → 32 (espacio), 45 (guion), 47 (barra), 58(dos puntos). [1]

Después de toda esta estructuración, comenzamos a programar, nos centramos primero en tratar de conseguir los objetivos con una única fecha, para ello comenzamos con un *.data* muy parecido al planteado. Al pedir la cadena con la primera fecha, nos dimos cuenta de que deberíamos de utilizar la famosa función "*LimpiarCadena"* de prácticas pasadas, para eliminar el retorno de línea (\n) y cambiarlo por un fin de cadena (\0).

Con todo esto comenzamos la función "*SepararFecha"*, pero el tiempo no nos dio más que para plantear que argumentos teníamos que pasarla y comenzar los bucles para la división.

Jueves, 1 de noviembre de 2018

Al dejar a medio empezar la función, hoy planteamos el algoritmo de esta: consistiría en una serie de 3 partes, la primera recorrería encontrando "/" o "–", y por tanto sacando el día, mes y

año, otro buscaría el espacio, y el último se encargaría de los dos puntos, para obtener hora, minuto y segundos.

El primero y el último se realizarán 2 veces, controlados gracias a un contador $t9, después de cada bloque separado, se llama a la función *DecToBin* que va guardando en un *array* estructurado los valores ya convertido y en otro array las secciones excluidas.

No logramos acabarla del todo ya que encontramos algún fallo, que corregiremos a la semana que viene.

Bibliografía:

[1] https://elcodigoascii.com.ar/caracteres-ascii-control/desplazamiento-afuera-codigo-ascii-14.html

Estructura y diseño de computadores. La interfaz hardware/software. David A. Patterson, John L. Hennessy. Sección 2.8 y 2.9.

Sesión 8

Martes, 6 de octubre de 2018

Hoy nos dedicamos íntegramente a finalizar y depurar de errores la función que comenzamos el otro día; encontramos multitud de errores, pero los más destacables fueron: la modificación de registros temporales, ya que dentro de *SepararFecha* llamamos a *DecToBin* por tanto los registros *$t* se sobrescribían con las llamadas, desplazamientos incorrectos o almacenamientos insuficientes.

El primer error lo solucionamos con el uso de la **pila** que, tras preguntar al profesor, recordamos como se utilizaba correctamente, entonces nuestros registros importantes fueron cambiados por *$s* y son guardados en la pila antes de cada llamada. Asique al comienzo de la función reservamos espacio con un *addi $sp, $sp, -16*.

El algoritmo de esta función consiste en la carga de bytes del *String* introducido por el usuario hasta encontrar el primer separador ("/", 47 en código ASCII), simultáneamente se guardan en un array de caracteres cada uno de los dígitos que vamos leyendo, ya que este array será el parámetro que pasemos a función que pasa de decimal a binario. Como mencionamos, precedentemente a la llamada de esta guardamos los tres registros *$s* que tenemos, que corresponden a: el *String* introducido, el array de caracteres y otro array donde guardamos los valores en binario. Después cargamos de nuevo estas variables, guardamos el dato transformado en el vector de enteros y lo desplazamos.

Este algoritmo se repite 6 veces, pero esta dividido en 4 bloques, uno en el que buscamos "/", que se realiza 2 veces, otro para el " ", otro para ":", que también se realiza 2 veces, y por el ultimo "\0". Para controlar el numero de veces que se realizan los que se repiten más de una vez utilizamos un contador $t7, que se reinicia después de encontrar el espacio, y que elegimos este temporal ya que era el único registro no usado por la función de cambio de base.

Sábado, 10 de noviembre de 2018

Hemos hecho las diferentes funciones que se necesitan para validar una fecha. Hemos creado la función principal *ValidarFecha*, que va llamando a la función que, valida una parte diferente de la fecha, si alguna de estas detecta un error, la función principal transmite el error al *main* para que el usuario introduzca de nuevo una fecha.

Funciones:

- *ValidarAño*: Analiza si un año dado es válido, y si lo es, halla si es bisiesto o no. Para ello hemos buscado información sobre cómo determinar si un año es bisiesto. Un año es bisiesto si es divisible entre 4 pero no entre 100, o si es divisible entre 400.
    - Parámetros -> *$a0*: Número que representa el año.
    - Return -> *$v0*: 0 si es válido, 1 si no es válido. *$v1*: 0 si no es bisiesto, 1 si es bisiesto.

- *ValidarMes*: Analiza si un mes dado es válido. Es válido si está entre 1 y 12. Para hallar el número de días hacemos una especie de *switch* evaluando cada número de mes.
    - Parámetros -> *$a0*: Número que representa el mes, *$a1*: 0 si no es año bisiesto, 1 si es año bisiesto.
    - Return -> *$v0*: 0 si es válido, 1 si no es válido. *$v1*: El número de días del mes.
- *ValidarDia*: Analiza si un día dado es válido. Será válido si está entre 1 y el número de días del mes en el que está.
    - Parámetros -> *$a0*: Número que representa el día del mes, $a1: Número de días que tiene el mes al que pertenece ese día
    - Return -> *$v0*: 0 si es válido, 1 si no es válido.
- *ValidarHora*: Analiza si una hora es válida. Si es menor que 0 o mayor que 23 entonces no es una hora válida.
    - Parámetros -> *$a0*: Número que representa la hora.
    - Return -> *$v0*: 0 si es válida, 1 si no es válida.
- *ValidarMinutoSegundo*: Analiza si un minuto o un segundo es válido. Ambos se pueden analizar de la misma forma. Si es menor que 0 o mayor que 59 entonces no es válido.
    - Parámetros -> *$a0*: Número que representa el minuto o el segundo.
    - Return -> 0 si es válido, 1 si no es válido.

Domingo, 11 de noviembre de 2018

Debido al gran tamaño en cuanto a líneas de código de la función *SepararFecha*, nos pusimos a intentar reducir esto.

La solución ha sido concatenar las 4 condiciones para encontrar los separadores principales, para juntar todo en un mismo bucle ya que los 4 anteriores eran prácticamente idénticos. Ahora cuando encuentra "/" o ":" bifurca a un mismo lugar, cuando encuentra el espacio bifurca a otro distinto y con el fin de cadena igual. Las diferencias se encuentran en los bytes que mandamos como parámetros a *DecToBin*, ya que con "/" y ":" mandamos 16 bits y con el espacio 32 bits ya que el año se compone de 4 dígitos. Diferenciamos el fin de cadena ya que cuando le encontramos debemos devolver al sistema *$ra* que guardamos en la pila nada más empezar.

Bibliografía:

https://elcodigoascii.com.ar

http://www.enrique7mc.com/2016/02/determinar-si-es-bisiesto/

Sesión 9

Martes, 13 de noviembre de 2018

Hemos añadido la validación de fechas a la función DecToBin por si se introduce una marca de tiempo con algún campo con algún carácter no válido. La función DecToBin detecta si el número en ASCII que se le pasa contiene algún carácter no válido y se lo comunica a la función SepararFecha, que es la que la llama. De este modo, si al separar la marca de tiempo se detecta que algún campo contiene algún carácter incorrecto, se retorna al main, que imprime el mensaje de error. Si todos los campos se han separado correctamente y transformado en binario, la función ValidarFecha (explicada en el diario de la semana anterior) se encarga de detectar si la fecha es válida.

Para hacer la diferencia entre dos fechas tratamos los campos día/mes/año y hora/minuto/segundo de forma diferente.

Para calcular la diferencia en horas, minutos y segundos entre las dos marcas de tiempo hemos creado la función DifHorMinSeg, que calcula la diferencia que existe entre los campos hora, minuto y segundo.

- Parámetros -> $a0: Primera Fecha (campo correspondiente)
  -> $a1: Segunda Fecha (campo correspondiente)
- Return -> $v0: acarreo de la operación
  -> $v1: resultado de la operación

Si el campo de la segunda fecha es mayor que el de la primera habrá que restarle 1 al campo siguiente: segundos->minutos, minutos->horas, horas->días.

Para calcular la diferencia de años, meses y días, pasamos los tres datos de ambas fechas a días y los restamos. Para ello hemos creado la función ToDias, que transforma años, meses y días de una fecha en días

- Parámetros -> $a0: Vector de enteros correspondiente a la fecha
- Return -> $v0: Número de días totales

Primero cargamos el campo AÑO, que se encuentra en la posición 8 del vector donde se encuentran los campos separados. Este número los multiplicamos por 365 y le sumamos el cociente de la división del año entre 4 (de esta forma hallamos el número de años bisiestos que han pasado y le sumamos el número de días extra por los años bisiestos que ha habido).

Después cargamos el campo MES, que se encuentra en la posición 4, y creamos un bucle para ir acumulando el número de días que tiene cada mes desde enero hasta el que cargamos. Para ello en cada iteración llamamos a la función ValidarMes (creada anteriormente), que recibe en $a0 el número que representa el mes y devuelve en $v1 el número de días que tiene ese mes.

Por último, cargamos el campo DÍA, que se encuentra en la posición 0, y se suma junto con los otros dos resultados calculados para hallar el total de días y retornarlo en $v0, para que el main calcule la diferencia de días entre las dos fechas.

Bibliografía:

http://www.sunshine2k.de/articles/coding/datediffindays/calcdiffofdatesindates.html

Sesión 10

Martes, 20 de noviembre de 2018

El día de hoy nuestro cometido fue sacar por pantalla el resultado correspondiente a la resolución básica de la práctica. Para ello creamos 10 cadenas con el nombre Fin_i, siendo i=1, 2, …, 10, en el que cada uno se correspondía con fragmentos del mensaje correspondiente a la diferencia entre las fechas introducidas. Y para lograr imprimir los diferentes campos que teníamos almacenados en un array, necesitamos la función utilizada en alguna de las prácticas anteriores, *BinToDec*. Pero para llamar a esta, tuvimos que reservar espacio para cadenas en el apartado *.data* para cada uno de los campos. Así que cuando queremos imprimir uno de los valores calculados movemos el campo del array al parámetro correspondiente y cargamos la dirección de la cadena en el otro parámetro, después de la transformación, con una llamada al sistema con el código 4, como en la impresión de los fragmentos Fin_i.

También nos dimos cuenta de que estábamos calculando los años bisiestos pasados de forma errónea. Por eso creamos una nueva función, BisiestosPas:

- Parámetros -> $a0: Año correspondiente
- Return -> $v0: Número de años bisiestos pasados

Esta función calcula el número de años bisiestos pasados de la siguiente forma:

Bisiestos pasados = Años/4 - Años/100 + Años/400

Por lo tanto, la función ToDias, que pasa los campos AÑO, MES y DIA a días, llama a la función BisiestosPas para saber el número de días extra que tiene que añadir.

Después de la depuración de algunos fallos conseguimos el objetivo, el cual se lo enseñamos al profesor para que nos guiara en las cosas a mejorar, en vista a la entrega final. A ello nos dedicaremos esta semana, con el propósito de dejarla acabada.

Jueves, 22 de noviembre de 2018

Hemos implementado la funcionalidad para que calcule la diferencia de años ordinarios y bisiestos. Para ello, cargamos el campo año de la primera fecha y hallamos el número de años bisiestos mediante la función *BisiestosPas* y repetimos el proceso con la segunda fecha. Después hacemos la diferencia entre estos dos resultados. Ahora dividimos la diferencia total de días hallada anteriormente entre 365. El cociente de esta operación será el número de años pasados y el resto el número de días sueltos, pero a estos días sueltos hay que restarles la diferencia de bisiestos que equivaldría al número de días extra que han pasado por los años bisiestos. Por último, para saber el número de años ordinarios, restamos la diferencia de años totales menos la diferencia de años bisiestos.

Estos tres valores hallados los guardamos en el vector de resultados para después imprimirlos por pantalla.

Viernes, 23 de noviembre de 2018

Nos hemos percatado de que la diferencia de horas, minutos y segundos la estábamos calculando de forma errónea, así que hemos hecho varias modificaciones en esta parte. Ahora, antes de empezar a calcular la diferencia, comparamos las dos fechas introducidas para saber cuál es mayor de las dos y después restar la mayor menos la menor para evitar fallos.

Hemos creado una nueva función llamada *Comparacion*:

- Parámetros -> $a2: Vector de enteros correspondiente a las partes de la fecha 1.
  $a3: Vector de enteros correspondiente a las partes de la fecha 2.
- *Return* -> $v0: 0 si la fecha 1 es mayor que la fecha 2 o si son iguales, 1 si la fecha 2 es mayor que la fecha 1.

Esta función compara las fechas para saber cuál es mayor. Llama a la función que convierte a días totales los campos AÑO, MES y DÍA para cada fecha y compara los resultados. Si son iguales compara los campos HORA, si estos son también iguales compara los campos MINUTO, y si también son iguales los campos SEGUNDO. Si en alguna comparación resulta que hay una que es mayor que la otra, se carga el valor correspondiente en $v0 y se retorna al *main*.

Al volver al *main*, si $v0 contiene un 0, entonces hay usar la fecha 1 como minuendo y la fecha 2 como sustraendo, y si $v0 contiene un 1, hay que usar la fecha 2 como minuendo y la fecha 2 como sustraendo.

Para calcular la diferencia de minutos o de segundos entre fechas usamos la función *DifMinSeg*:

- Parámetros -> $a0: Fecha mayor (campo correspondiente)
  -> $a1: Fecha menor (campo correspondiente)
- *Return* -> $v0: acarreo de la operación
  -> $v1: resultado de la operación

Esta función resta el campo correspondiente de la fecha mayor menos la menor. Si el resultado es negativo, se vuelve a hacer la operación después de haberle sumado 60 al minuendo y se carga un 1 en $v0 para indicar que hay un acarreo que hay que restar al minuendo del campo inmediatamente superior para calcular la diferencia.

Para calcular la diferencia de horas entre fechas usamos la función *DifHoras*, que es idéntica a la función *DifMinSeg*, pero en vez de sumar 60 cuando la diferencia es negativa, suma 24.

Sábado, 24 de noviembre de 2018

El día de hoy nos hemos dedicado al perfeccionar la flexibilidad a la hora de introducir las fechas a analizar, las mejoras a implementar son: añadir algún separador más, como el guion, y los datos con forma 0x, siendo x un número del 0-9, poderlos introducir sin la necesidad del 0 a la izquierda.

La primera mejora, fue bastante asequible, con la adicción de la instrucción: "*beq $t3, 45, Separacion*", en la función *SepararFecha*, junto a las otras comprobaciones para la búsqueda de separadores.

La segunda mejora, implicó una reducción de la misma función anterior, ya que añadimos un contador del número de iteraciones que se hacen al buscar el separador, ya que así descontaríamos ese número para dejar la dirección de memoria en la posición correcta, para mandársela a la función *DecToBin*. Después del cambio de base del dato dividido se reinicia el contador.

La rebaja de código se debe a que antes teníamos 3 apartados distintos, uno para "/" y ":", otro para el espacio, y otro para el fin de cadena, ahora es un único bucle que se realiza 6 veces buscando los separadores y cuando acaba realiza las operaciones para dejar la pila como estaba y volver a la dirección de retorno.

Durante el día de hoy, depuramos un error, probando las nuevas implementaciones, ya que en el caso de que fuese el mes de enero entraba en un bucle infinito, debido a unas asignaciones incorrectas del parámetro que mandábamos en la función *ToDias*, para ir sumando los días de cada mes pasado.

Bibliografía:

http://www.titulosnauticos.net/calculadoras/index.htm?/calculadoras/sexagesimal.htm

https://elcodigoascii.com.ar/caracteres-ascii-control/desplazamiento-afuera-codigo-ascii-14.html

Sesión 11

Martes, 27 de noviembre de 2018

Para el día de hoy nuestro cometido fue añadir comentarios aclaratorios de distintos fragmentos de código y depurar los fallos que fuéramos encontrando con la ayuda de una batería de prueba.

Después de añadir suficientes explicaciones, para facilitar la comprensión del código, estuvimos corrigiendo errores menores, el que nos supuso la mayor del tiempo era, que teníamos en cuenta el acarreo de las horas, pero al restárselo al campo días de la fecha superior, no modificábamos ese número en el array.

Jueves, 29 de noviembre de 2018

Hemos realizado una pequeña batería de pruebas para comprobar que los resultados de nuestro programa eran correctos. Nos hemos ayudado de una página web [1] que realiza este mismo cálculo para comparar las respuestas. También hemos añadido a la batería de pruebas los principales casos en los que una fecha no es válida:

- No están todos los campos necesarios.
- Hay algún carácter incorrecto.
- Algún campo no corresponde a una fecha o a una marca de tiempo válida.

ENTRADAS		ESPERADO	OBTENIDO
FECHA 1	FECHA 2	Diferencia	Diferencia
01/11/2018 01:20:05	28/09/2015 17:01:15	2 años ordinarios, 1 bisiestos, 33 días, 8 horas, 18 minutos, 50 segundos	2 años ordinarios, 1 bisiestos, 33 días, 8 horas, 18 minutos, 50 segundos
3-11-2018 1 20:5	28-9/2015 17:1-15	2 años ordinarios, 1 bisiestos, 35 días, 8 horas, 18 minutos, 50 segundos	2 años ordinarios, 1 bisiestos, 35 días, 8 horas, 18 minutos, 50 segundos

1/1-2014 0:13 59	31:12/2009 1-12:01	3 años ordinarios, 1 bisiestos, 0 días, 23 horas, 1 minutos, 58 segundos	3 años ordinarios, 1 bisiestos, 0 días, 23 horas, 1 minutos, 58 segundos
19/2/2013 9:0:1	20/2/2013 8:1:0	0 años ordinarios, 0 bisiestos, 0 días, 23 horas, 0 minutos, 59 segundos	0 años ordinarios, 0 bisiestos, 0 días, 23 horas, 0 minutos, 59 segundos
1/03-2017 17:8:56	29-02/2016 12:10 50	1 años ordinarios, 0 bisiestos, 0 días, 4 horas, 58 minutos, 6 segundos	1 años ordinarios, 0 bisiestos, 0 días, 4 horas, 58 minutos, 6 segundos
31/12/2015 23:55:16	01/1-2017 13:33:9	0 años ordinarios, 1 bisiestos, 0 días, 13 horas, 37 minutos, 53 segundos	0 años ordinarios, 1 bisiestos, 0 días, 13 horas, 37 minutos, 53 segundos
19/2 1999 15/23-48	29/11:2018 20:15:48	14 años ordinarios, 5 bisiestos, 285 días, 4 horas, 52 minutos, 0 segundos	14 años ordinarios, 5 bisiestos, 285 días, 5 horas, 52 minutos, 0 segundos
28/2/2012 13:23:19	1/3/2012 13:23:19	0 años ordinarios, 0 bisiestos, 2 días, 0 horas, 0 minutos, 0 segundos	0 años ordinarios, 0 bisiestos, 2 días, 0 horas, 0 minutos, 0 segundos
01/01/1999	-	Fecha no válida	Fecha no válida
a/01/1999 00:00:00	-	Fecha no válida	Fecha no válida
01/01/1999 00:a0:00	-	Fecha no válida	Fecha no válida
40/01/1999 00:00:00	-	Fecha no válida	Fecha no válida
1/13/1999 00:00:00	-	Fecha no válida	Fecha no válida
1/1/0 00:00:00	-	Fecha no válida	Fecha no válida

1/1/1999 24:00:00	-	Fecha no válida	Fecha no válida
1/1/1999 00:60:00	-	Fecha no válida	Fecha no válida
1/1/1999 00:00:60	-	Fecha no válida	Fecha no válida

Bibliografía:

[1] https://www.timeanddate.com/date/timeduration.html -> Utilizada para verificar los resultados de las diferencias entre fechas utilizadas.