Eduardo García Asensio

Santiago Blasco Arnaiz

Sesión 1

19/09/2017

Ejercicio 1 apartado a)

Creamos una variable que ejerce de contador y límite para el bucle, utilizamos el contador para incrementar las direcciones de los vectores, esto lo conseguimos multiplicando por 4 el contador y guardándolo en otra variable, cargamos las direcciones de los vectores A y B, las sumamos y las guardamos en la misma posición pero en el vector C, cuya dirección ya ha sido incrementada. Mostramos por pantalla los números separados por una coma escrita en una cadena de texto que cargamos, el bucle termina cuando el contador supere el límite establecido, en este caso la longitud del vector, que es 7.

Ejercicio 1 apartado b)

Hemos encontrado en nuestro programa una única instrucción que puede ser sintética o natural, la instrucción "lw", que en nuestro caso se usa de forma natural puesto que cargamos una variable con desplazamiento en vez de una etiqueta.

Ejercicio 1 apartado c)

La instrucción "`la`" se sustituye por una "`lui`" y una "`ori`".

La instrucción "`li`" se sustituye por una "`addiu`".

La instrucción "`move`" se sustituye por una "`addu`".

21/09/2017

Ejercicio 1 apartado d)

Si ejecutamos el código sin modificar nos aparecen los errores correspondientes a las instrucciones "la" y "li".

Al reescribir en el código las instrucciones sintéticas por sus correspondientes naturales y ejecutar el código los errores ya no aparecen. La legibilidad del programa disminuye considerablemente.

Ejercicio 2 apartado a)

No se traducen de la misma forma, esto puede deberse a que MARS realiza la simulación más fielmente, mostrando incluso el proceso de compilación que realmente ocurriría.

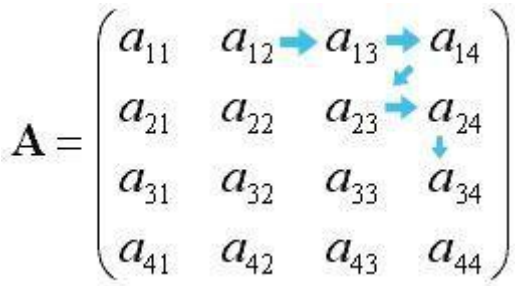
Ejercicio 2 apartado b)

La traducción es distinta entre los dos programas. Para la instrucción `beq $s6,$s7,Exit` QtSpim codifica como `0x12d70002` y MARS como `0x12d70001`, con bne ocurre de igual forma al codificar difieren en los últimos dos bytes, correspondientes a los 16 bits de desplazamiento, comprobamos que la traducción correcta es la de QtSpim.

Sesión 2

26/09/2017

Para trasponer la matriz solo recorremos su parte triangular superior como indicamos en la imagen:



Comenzando el valor i en 0 y el valor j en 1 (las posiciones de la imagen comienzan a contar en 1 en vez de en 0), para hallar la dirección de la posición que queremos utilizamos la fórmula que nos proporciona el enunciado A + 4* i*m + 4*j. El elemento traspuesto de a_{ij} es a_{ji}, por tanto con la fórmula A + 4* j*m + 4*i hallamos la dirección del elemento traspuesto, cargamos los valores almacenados en ambas direcciones en dos variables y posteriormente almacenamos el valor almacenado en la primera dirección en la segunda dirección y viceversa.

Esta acción la realizamos dentro de un bucle que aumenta el valor j para avanzar en la fila columna a columna y cuando acaba de recorrer la fila pasa a la siguiente dándole a i valor i + 1 y a la j el valor de i ya aumentado + 1, es decir al acabar la fila 0 el valor de i pasa a valer 1 y el de j 2.

Para sumar dos matrices en una tercera cargamos la dirección de las 3 matrices y con la fórmula del ejercicio anterior 4*i*m+4*j más la dirección de cada matriz accedemos al elemento ij de cada matriz para sumar los elementos de las dos primeras matrices y almacenar el resultado en el elemento de la tercera. Vamos aumentando los valores de i y j en un bucle para recorrer toda la matriz.

28/09/2017

Para que el usuario introduzca por teclado los valores de los elementos de las matrices mostramos por pantalla una serie de cadenas y caracteres para guiar al usuario y mediante un bucle vamos rellenando las direcciones de la matriz que tenemos reservada, esto se realiza para las dos matrices.

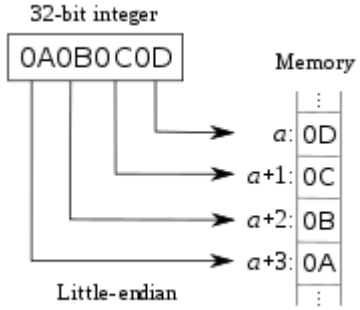
Por último para imprimir la matriz simplemente la recorremos y vamos imprimiendo e valor del elemento seguido de un espacio, cada cuatro elementos imprimimos un sato de línea para que quede mejor estéticamente.

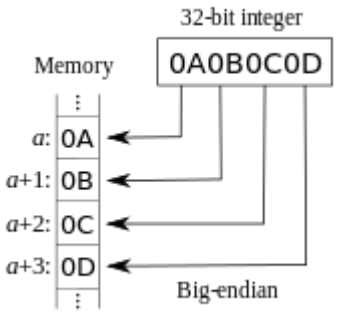
Sesión 3

03/10/2017

Ejercicio 1 apartado a)

Para comprobar si el procesador MIPS trabaja con LE o BE reservamos 32 bits de espacio y guardamos en él el número 1 y cargamos el primer byte almacenado en la dirección, si este byte es 0 significa que trabaja en BE, sino querrá decir que es LE.





El procesador MIPS trabaja en LE, esto lo sabemos porque al cargar el primer byte almacenado en la dirección en la que guardamos el número 1 el resultado fue 1, eso quiere decir que carga el byte menos significativo en la dirección más baja.

La información que nos proporcionan las ventanas de MARS no nos indica el endianness ya que carga palabras enteras.

Ejercicio 1 apartado b)

Tanto QtSPIM como MARS dan el mismo resultado, ya que el endianness depende del procesador, no del simulador.

Ejercicio 1 apartado c)

Almacenamos un número negativo en un espacio de 32 bits y cargamos byte a byte la palabra, mostrando por pantalla el contenido de cada byte. Descubrimos que el byte almacenado en la dirección más alta (el byte más significativo, como esperábamos) alberga el signo. Para saber si un número es positivo o negativo mediante un programa evaluamos el bit más significativo del byte más significativo (el cuarto) utilizando esta máscara 0x80000000, la instrucción and y el byte más significativo del número a evaluar, si el resultado nos diese 0 querrá decir que el número es positivo, en caso contrario será negativo.

04/10/2017

Ejercicio 1 apartado d)

El formato IEEE 754 de punto flotante en simple precisión utiliza el bit más significativo para el signo, los 8 siguientes para el exponente y los 23 restantes para la fracción, en el caso de doble precisión son 11 bits de exponente y 52 bits de fracción. Para conocer el signo de un número representado en simple precisión lo hacemos de la misma forma que para el apartado anterior, en el caso de doble precisión se hace de igual forma pero el byte más significativo será el octavo, no el cuarto.

Ejercicio 2 apartado a)

Declaramos una matriz por filas 4 x4 con elementos de 32 bits y la inicializamos con los valores que indica la fórmula, mediante un programa similar al utilizado la semana pasada con la fórmula *"Dirección de la matriz" + 4* i*m + 4*j* recorremos la matriz y sumamos todos los elementos, obteniendo como resultado -122.

Ejercicio 2 apartado b)

El mismo procedimiento que en el apartado anterior excepto por la declaración de la matriz que se hace por columnas.

Ejercicio 2 apartado c)

Al cambiar la declaración de filas a columnas también cambia el orden de los valores almacenados en memoria, siendo en el primer caso de derecha a izquierda y de en el segundo de arriba a abajo.

Ejercicio 2 apartado d)

Para este programa cambia la reserva de espacio para la matriz (utilizamos .byte en vez de .word) y la fórmula que utilizamos (cambia ligeramente *"Dirección de la matriz" + i*m + j"* ya que cada elemento de la matriz solo ocupa 8 bits, no 32).

06/10/2017

Ejercicio 3 apartado a)

Para este ejercicio ejecutamos el programa del apartado 2 a). La leyenda de colores es la siguiente:

Código de operación: Rosa

Señales de control activas: Rojo

Señales de control inactivas: Gris

rs: Verde

rt: Azul

Inmediato: Cian

Ejercicio 3 apartado b)

El camino crítico perteneciente a la instrucción bne es:

Banco de registros, MUX, ALU, Branch, MUX.

Ejercicio 3 apartado c)

La representación tiene recorridos erróneos, como por ejemplo que se bifurca y se desplaza a la izquierda antes de llegar al banco de registros. La señal debería llegar antes al banco de registros, pasar por el MUX, la ALU y bifurcar después.

Sesión 4

10/10/2017

Ejercicio 1)

Para pasar un número de decimal a hexadecimal codificado en ASCII hay que sumarle 48 si el número se encuentra entre 0 y 9(ambos incluidos) y sumarle 55 si se trata de números superiores a 9, siendo letras mayúsculas entre la A y la F codificadas en ASCII.

Para ello creamos una máscara 0xF0000000 y realizamos la operación lógica and sobre la máscara y el número a convertir (así leeremos su byte más significativo, lo hacemos de tal forma para así almacenarlo en la memoria en el mismo orden que se imprimirá), tras aplicar la máscara realizamos un desplazamiento de 28 bits sobre el resultado para que quede situado en el byte menos significativo. Comparamos si es menor o igual que 9 con la instrucción ble, si es menor se le suma 48, si es mayor se le suma 55, almacenamos el byte en la posición de memoria dada por $a1y aumentamos esta en 1 para así almacenar el siguiente byte a continuación, realizamos esto 8 veces valiéndonos de un bucle para ello.

Ejercicio 2 apartado a)

Imprimimos un mensaje solicitando que se introduzca un número por teclado y lo recibimos con li $v0 5, para imprimirlo en hexadecimal simplemente imprimimos el número hexadecimal codificado en ASCII como un carácter cargando su dirección en $a0 y ejecutando la instrucción li $v0 4.

Ejercicio 2 apartado b)

Funciona tanto para número positivos mostrando los bytes de signo con 0, como para número negativos mostrando los bytes de signo con una F, para números excesivamente grandes (Superiores a los 32 bits) el programa dará error.

Ejercicio 2 apartado c)

Seguimos el mismo procedimiento que en el apartado a de este ejercicio pero incrementando en 1 el número decimal introducido antes de convertirlo a hexadecimal.

13/10/2017

Ejercicio 3)

Reutilizamos el código del ejercicio 2ª) , eliminando la entrada del número por teclado y declarando dos números de 32 bits en memoria, num1 con los bits más significativos y num2 con los menos.

Se pasan num1 y num2 a la función conversor, almacenando sus conversiones en el espacio reservado. Tras ser convertidos se resetea la dirección del convertido al principio y, de nuevo con li $v0 4 se imprime el número en hexadecimal, pero esta vez en 64bits.

Sesión 5

17/10/2017

Ejercicio 1)

Esta función es la inversa de la hecha en la sesión anterior, por lo tanto seguiremos un procedimiento similar.

Cargamos byte a byte el número hexadecimal guardado en memoria, si el byte es menor de 57 significa que tenemos una cifra, y restando 48 obtenemos su valor en decimal. Por el contrario, si es mayor tendremos una letra (entre la A y la F), y restando 55 obtenemos su valor en decimal. Hacemos este mismo procedimiento en bucle hasta detectar que se ha acabado la cadena. Para desplazarnos por los bytes del número usamos la instrucción or y vamos desplazándolo con sll.

Ejercicio 2 apartados a y b)

Se usa la función anterior, almacenando un numero hexadecimal en memoria, las correspondientes etiquetas y haciendo la llamada necesaria al sistema para imprimir en pantalla su conversión. Este programa funciona perfectamente con todos los números positivos, negativos, pequeños y grandes.

Ejercicio 2 apartado c)

En este apartado se reserva espacio en memoria, se efectúan las llamadas al sistema para tomar el número por teclado y hay que hacer un tratamiento a la cadena. Al pedir el número por teclado la cadena va con un salto de línea. Hay que detectarlo y suprimirlo, simplemente poniendo la condición de que si el byte tiene valor 10 se borra.

Ejercicio 2 apartado d)

Exactamente igual al apartado anterior, pero antes de imprimir el número generado por la función sumarle 4.

20/10/2017

Ejercicio 3

Valiéndonos de las instrucciones ble y bge restringimos los valores codificados en ASCII que admitimos creando estos intervalos validos: [48,57],[65,90],[97,122]. Si el carácter se encuentra fuera de esos intervalos será invalido y $v1 tomará valor 1. Con un contador limitamos a 8 el número de carácteres que puede admitir por cadena (dado que los registros son de 32 bits), si la cadena es mayor pondrá $v1 con valor 2, si ninguno de los dos casos anteriores se da el código se ejecutará con normalidad y $v1 tomará valor 0.

Ejercicio 4

Dentro de la función se comprueba si la letra es mayúscula o minúscula. En el primer caso al byte se le resta 55 y en el segundo 87 para hallar su valor decimal.

Ejercicio 5

Usando el control de errores del ejercicio 3, si en $v1 tenemos como resultado un 1 o un 2 llamamos al sistema para que imprima una de las etiquetas almacenadas en memoria avisando del error.

Ejercicio 6

Primero usamos la función para convertir el número de hexadecimal a binario. Para hallar el opuesto restamos 2 veces el propio número; una vez hecho esto llamamos a la función de la anterior práctica (convierte de binario a hexadecimal) e imprimimos dicho número, siendo este el opuesto del primero en hexadecimal.

Sesión 6

24/10/2017

Ejercicio 1)

Para la realización de este ejercicio reutilizamos en gran medida el código de la práctica anterior. Declaramos en memoria la cadena que contiene el número decimal. Inicializamos t5 con valor 1, si t5 vale 1 quiere decir que el número de la cadena es positivo, si vale -1 quiere decir que es negativo. Cargamos el primer carácter ASCII almacenado en la dirección a0, si su valor es 10 (salto de línea) o 0 (fin de línea) la función termina, si su valor es 45 quiere decir que el número es negativo por tanto el registro t5 pasa a valer -1 y si vale 43 que es positivo por tanto t5 pasa a valer 1, si no se da ninguno de estos casos comprobamos que esté en el intervalo [48,57] si lo está es un número asique seguimos con la función, le restamos 48 para obtener el valor decimal del número codificado y se lo sumamos al resultado final. Antes de pasar al siguiente ciclo del bucle multiplicamos el número por 10 para que el número adquiera el valor que debe, lo hacemos antes de pasar al siguiente bucle y después de comprobar el contador del bucle, para que no multiplique el último número por 10. Cuando el valor que evaluamos sea 10 o 0 terminamos el bucle.

Ejercicio 2 a)

En vez de declarar la cadena en memoria la pedimos por teclado y la pasamos como parámetro a la función, además el resultado de la función devuelto en v1 lo mostramos por pantalla.

Ejercicio 2 b)

El programa funciona correctamente con todo tipo de números.

Ejercicio 2 c)

Modificamos un poco el flujo y las etiquetas de este y llamamos dos veces a la función, almacenando el número devuelto la primera vez que llamamos a la función en un registro y sumándolo más tarde con el devuelto por la segunda llamada a la función y mostramos el resultado por pantalla.

Ejercicio 3)

Al igual que en la práctica anterior utilizamos las instrucciones ble y bge restringimos los valores codificados en ASCII que admitimos creando estos intervalos validos: [48,57]. Si el carácter se encuentra fuera de esos intervalos será invalido y v0 tomará valor 2. Con un contador limitamos a 9 el número de carácteres que puede admitir por cadena (dado que los registros son de 32 bits), si la cadena es mayor pondrá v0 con valor 1, si ninguno de los dos casos anteriores se da el código se ejecutará con normalidad y v0 tomará valor 0.

Ejercicio 4)

Utilizamos el programa confeccionado para el ejercicio 2c) y le añadimos la función del ejercicio 3), si en v0 tenemos como resultado un 1 o un 2 llamamos al sistema para que imprima una de las etiquetas almacenadas en memoria avisando del error.

Sesión 7

31/10/2017

Ejercicio 1)

Primero evaluamos si el número es negativo o positivo, si es negativo almacenamos un guion (Codificado como 45) en la dirección pasada a la función en el registro $a1, a continuación invertimos el número para trabajar con él en positivo. Para efectuar el paso del número a cadena dividimos este entre 10 y le sumamos al resto (contenido en el registro HI) 48 para pasar a su codificación ASCII y este valor lo almacenamos en otro espacio de memoria que utilizamos para contener la cadena del revés, realizamos esta tarea hasta que el cociente sea 0. Cuando el bucle finaliza cargamos el contenido almacenado en la última dirección de memoria utilizada (de nuestra cadena invertida) y lo almacenamos en la dirección que se pasa a la función en el registro $a1, cuando la dirección sea igual a la inicial donde almacenábamos la cadena del revés finaliza el bucle y le añadimos un 0 a la cadena para que señale el fin de línea.

Ejercicio 2 a y b)

Para leer la cadena del número decimal utilizamos la función que creamos en la práctica anterior, multiplicamos el número que nos devuelve esa función por dos y lo imprimimos con a función confeccionada en el ejercicio anterior.

Ejercicio 3 a)

Implementamos el programa con la función que creamos en la práctica 4 para pasar números de binario a hexadecimal y la función del ejercicio 1 para leer la cadena decimal.

Ejercicio 3 b)

Implementamos el programa con la función que creamos en la práctica 5 para pasar números de hexadecimal a binario y la función del ejercicio 1 para leer la cadena decimal.

Trabajo final

7/11/2017

Tras leer el guion de la práctica final y escuchar la explicación del profesor comenzamos a pensar maneras de abarcar el problema. Nuestro objetivo es implementar un programa que respete la precedencia tanto de los operandos como de los paréntesis, para ello se nos ocurre confeccionar una función recursiva que actue dentro de un paréntesis y si encuentra otro dentro de este vuelva a llamarse a sí misma, acompañando a esta función estaría otra que asignaría una prioridad a la operación leída y así solucionaríamos la precedencia. Estudiamos la

posibilidad de utilizar la notación polaca para crear la función que estudia la precedencia, esta consiste en poner los dos operandos y a continuación el símbolo de la operación, por ejemplo:

A + B sería escrito como + A B, y A + B * C sería + A B * C que al operar por primera vez quedaría * AB C siendo AB el resultado de A + B.

Esto no soluciona el problema de la precedencia, pero podríamos solucionarlo leyendo la cadena entera primero y reordenándola escrita ya en notación polaca, dedicamos un tiempo a comprender el funcionamiento de la notación polaca y a decidir qué hacer y cómo tratar la precedencia.

11/11/2017

Recolectamos las funciones que realizamos en prácticas anteriores para leer e imprimir cadenas de números y así utilizarlas en el programa. Encontramos una posible solución al problema de la precedencia, el algoritmo de **shunting yard.** Este algoritmo reordenada expresiones algebraicas pasándolas a la notación polaca inversa, que se basa en la notación que nos planteábamos utilizar con un ligero cambio. Para entender el funcionamiento del algoritmo buscamos información en internet, gracias a este vídeo, y a la información que encontramos en Wikipedia y StackOverflow entendemos la totalidad del algoritmo. Comparando la sencillez y eficacia de este algoritmo con nuestra idea de realizar una función recursiva decidimos no utilizarla ya que trae muchos problemas de carga y almacenamiento en memoria, cuando se ejecuta la función y vuelve a llamarse a sí misma habría que guardar el contador de programa y además todas las variables a las que afecta la función.

14/11/2017

Modificamos el diario ya que faltaban los enlaces en los que encontramos la información.

Una vez aclarado el algoritmo a utilizar procedemos a implementarlo. Lo primero es leer la cadena de caracteres, usando las instrucciones beq ble y bge acotamos los rangos de los caracteres ASCII válidos (0-9, +, -, (,) , / y *) mostrando un mensaje de error si alguno no lo es. A continuación, hay que distinguir si el carácter es un número o un operando, si es un número lo tratamos con la función conversor que recibe la dirección en la que se leyó el carácter número y a partir de ella lee la cadena de números y la interpreta como un número decimal que devuelve, ese valor obtenido lo introducimos en PilaNum, que es el espacio de memoria reservado para hacer de pila. Si es un operando debemos comprobar si su precedencia es menor que la del último operando almacenado en PilaOp (espacio de memoria reservado para almacenar los operandos), esta comprobación la realizamos con un registro que guarda el valor de la precedencia del último operando, en caso de que sea menor deberán sacarse todos los operandos de PilaOp y una vez vaciada almacenar este último operando de menor precedencia. La asignación de la precedencia la hacemos mediante saltos condicionados con la instrucción beq.

16/11/2017

Si se detecta una apertura de paréntesis se almacena en PilaOp y se continua hasta encontrar un cierre de paréntesis, cuando se encuentre se vacía la pila hasta la dirección de memoria donde se insertó la apertura de paréntesis.

Vaciar la PilaOp consiste en sacar un operando, determinar de qué operación se trata y realizarla sobre los dos primeros números de la PilaNum (Los dos con direcciones más bajas),

el resultado de la operación se guarda en la dirección más baja de la PilaNum y se mueven todos los demás elementos de la pila una posición para llenar el hueco que hemos dejado. Este proceso se realiza hasta vaciar la PilaOp (Al principio de la función añadimos en la dirección más baja de la pila un 0 para poder utilizarlo como fin de la pila). Tras leer toda la cadena de operaciones tendremos como resultado un único número que se encontrará en la dirección más baja de la PilaNum, gracias a la función que confeccionamos en prácticas anteriores pasamos este número como parámetro y nos lo convierte a una cadena de caracteres que representan el número decimal para poder mostrarlo por pantalla.

17/11/2017

Realizando pruebas en el programa vemos que funciona bien para cadenas de operaciones simples, aunque contengan paréntesis, pero para cadenas más complejas que contienen varios cocientes o cálculos con números negativos falla. Creemos que uno de los errores puede ser la instrucción que utilizamos para dividir.

Conseguimos solucionar uno de los fallos, ocurría que al mover la PilaNum una posición no poníamos la última posición a cero, por tanto siempre dejábamos al final de la pila un número duplicado. Proseguimos con la depuración del programa.

20/11/2017

Hemos conseguido que el programa funcione, incluso admite la introducción de números negativos (Aunque estos deben ir entre paréntesis), eso lo conseguimos creando una función que elimine los espacios que contenga la cadena de operaciones y comprobando si a continuación de un "(" se encuentra un "-", entones quiere decir que el número que metamos en la PilaNum será negativo. Para eliminar la precondición de que los números negativos deban ir entre paréntesis se nos ocurre modificar la función que elimina los espacios de la cadena para que la trate antes de leerla, consiste en que si encuentra un "-" precedido de un "(" o una operación reescriba en la cadena entre paréntesis una resta del número a continuación del guion y el cero.

Uno de los errores que cometíamos era el de la asociatividad de los números según su operando, cuando realizábamos el cálculo 4 – 8 + 4 no le dábamos preferencia a la resta, esto ocurre también con la multiplicación y la división. Para solucionarlo cuando leemos un operador de precedencia 1 o 2 (resta/suma o multiplicación/división) si el siguiente operador tiene la misma precedencia y es asociativo hacia la izquierda con precedencia (resta o división) introducimos un "!" (codificado en ASCII como 33) en la pila de operaciones para así poder reutilizar el método SacaPilaOp que ya tenemos hecho, lo modificamos con una instrucción beq que detecta si el carácter leído es "!" y entonces deja de sacar operaciones de la pila, así conseguimos sacar un solo operador, el último, sin tener que escribir más código.

21/11/2017

Enseñamos el programa al profesor y nos sugirió implementar algunos añadidos, como detectar desbordamientos o trabajar con números en hexadecimal. Para poder trabajar con números en hexadecimal estos deben ir precedidos de 0x, entonces cuando el carácter leído sea un 0 cargamos el siguiente, si es una x o una X pasamos la dirección siguiente al conversor de hexadecimal codificado en ASCII a binario y lo almacenamos en la pila de números, si el siguiente carácter no fuese una x/X quiere decir que es un número, entonces pasaría la dirección anterior al conversor de números decimales.

Para el desbordamiento en multiplicación miramos si al realizar la multiplicación el registro HI contiene un número distinto de 0x00000000 o 0xFFFFFFFF que serían las extensiones de signo, si el registro HI es 0xFFFFFFFF y el LO es 0x00000000 también se habrá producido desbordamiento. Para el desbordamiento en la suma realizamos la operación con addu, si los signos de los operandos eran distintos no se producirá desbordamiento, si eran iguales y el signo después de la operación ha cambiado querrá decir que se ha producido desbordamiento.

También anulamos la posibilidad de dividir entre 0, para ello comprobamos si el divisor es 0 entonces realizamos un salto e informamos del error.

Para cada uno de estos errores damos un valor a $v1 e interrumpimos la ejecución, a continuación según el valor que hayamos dado a $v1 imprimimos un mensaje de error informando del error.

28/11/2017

Implementamos la detección de desbordamiento en la resta de igual forma que en la suma, la operación la realizamos sin signo y sin desbordamiento, si los dos operandos tienen el mismo signo no puede haber desbordamiento, si tienen distinto signo y el signo del resultado y el primer operando son diferentes quiere decir que ha desbordado.

Hacemos que el programa reconozca que se ha introducido el número 0x80000000 puesto que este no tiene inverso y provoca un fallo en el programa.

05/12/2017

Implementamos los últimos retoques, evaluamos el caso especial en el que se introduce el número 0x80000000 que no tiene opuesto por ser el más grande de los negativos e imprimimos un mensaje de error, también evaluamos que se hayan emparejado correctamente los paréntesis, si se cierra un paréntesis sin haberlo abierto o si no está correctamente emparejados se imprimirá un mensaje de error, esto lo hacemos en la función que utilizamos para eliminar los espacios entre caracteres, si se encuentra un “)” resta 1 a una variable, si se encuentra “(“ suma 1 a esa misma variable, si la variable en algún momento adquiere valor negativo querrá decir que están mal emparejados y si al acabar de leer la cadena la variable es distinta de 0 también . Si se ha pulsado el salto de línea sin ningún otro carácter solicitamos que se introduzca de nuevo la cadena.

Fin del diario de prácticas.