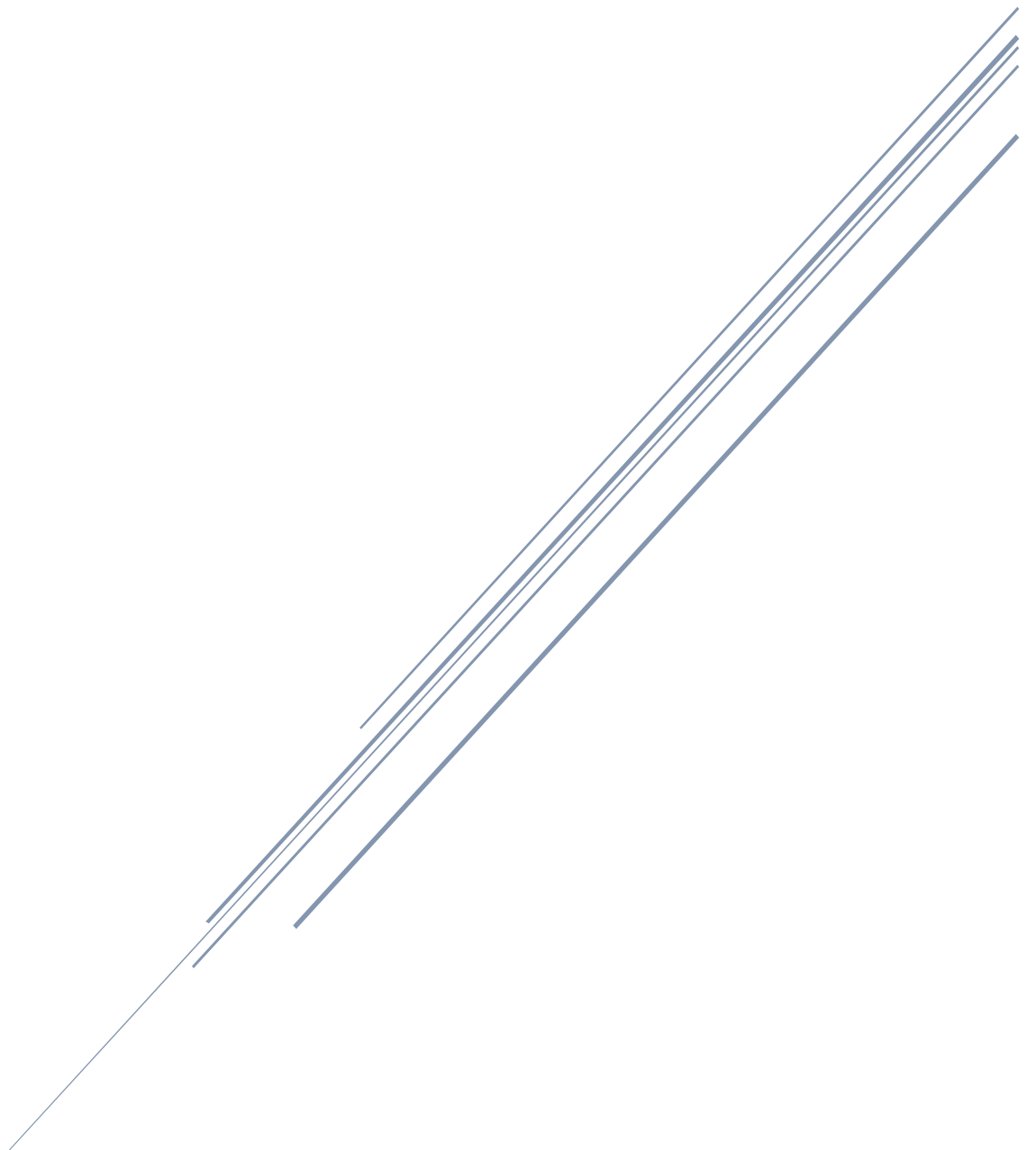


ARQUITECTURA Y ORGANIZACIÓN DE COMPUTADORES

Guion de practicas



Sergio Gómez Conde y Carlos Noé Muñoz Bastardo
Grupo 2, X4

Primera semana

17/09/19

1.

- a. Al hacer las llamadas al sistema correspondientes para imprimir la palabra y el byte que se cargan en `$s0` y `$s1` respectivamente (habiendo previamente almacenado estos valores en `$a0` para poder hacer dichas llamadas) vemos que la palabra la imprime entera y que el primer byte es el de más a la derecha, por lo que sigue el criterio de Little Endian. Esta conclusión la hemos sacado después de haber probado con números más grandes en el vector A. Los resultados que muestra el programa con las instrucciones del enunciado son un 1 en cada registro.

Las instrucciones `lw` y `lb` son instrucciones naturales ya que corresponden al formato I de instrucciones y no se pueden sustituir por otro conjunto de instrucciones que realicen su función.

- b. Al ser C un vector de ocho palabras (. space 32) debemos realizar un bucle que itere ocho veces, una por cada elemento a rellenar.
Hacemos la operación ($C_i = A_i + B_i - 1$).
Para ello cargamos en un registro la primera dirección de memoria donde se encuentra el vector A, en otro registro la de B y en otro la de C.
A continuación, creamos un índice (común a todos los vectores) y ese índice lo multiplicamos por cuatro y le sumamos la primera dirección de memoria de cada vector y vamos aumentando el iterador en una unidad cada vez que pasamos por el bucle.
Hacemos lo mismo con el vector B.
Cuando tenemos el elemento correspondiente de cada vector (A y B) hacemos las operaciones correspondientes y cargamos en la primera dirección de memoria donde está el vector C el resultado de dichas operaciones.
Reiteramos las operaciones anteriores hasta llegar al tamaño del vector que indicaría la finalización del bucle.

Para imprimir el vector C cargamos la primera dirección de memoria donde se encuentra el vector C en un registro.

Creamos un iterador `i`, de forma semejante a como hacíamos con los vectores A y B.

Multiplicamos `i` por cuatro y se lo sumamos a la primera dirección de memoria del vector C, traemos la palabra a registro y aumentamos el índice `i` en una unidad.

Cuando ya tenemos la posición correspondiente, almacenamos el contenido cargado en registro, y este, lo almacenamos en `$a0` para posteriormente ser impreso.

Ejecutamos los pasos anteriores hasta que finaliza el bucle.

18/09/19

- c. Las instrucciones sintéticas que utilizamos son la instrucción `la`, la cual se podría sustituir por:
- ```
lui $t, A_hi
ori $t, $t, A_lo
```

Y la instrucción `li` que se puede sustituir por:

```
ori $t, $Zero, A_lo
```

Bibliografía:

<https://github.com/MIPT-ILab/mipt-mips/wiki/MIPS-pseudo-instructions>

Algunos ejemplos de instrucciones que pueden ser naturales o sintéticas son `add`, `beq`, `lw`, `sw`...

- d. Detecta como errores las instrucciones sintéticas `la` y `li` además de las que también pueden ser naturales como `add` o `lw` debido a sus argumentos.

Al sustituir estas instrucciones sintéticas por naturales disminuye la legibilidad ya que se aumenta el número de instrucciones y la mayoría de estas nuevas son más difíciles de entender al verlas.

En concreto, las instrucciones que dan error son:

```
lw $s0, A
lw $s1, D
lw $s2, A
lw $s3, E
li $v0, 1
li $v0, 11
la $s0, A
la $s1, B
la $s2, C
la $s3, D
```

2.

- a. Sustituyendo en MARS las instrucciones sintéticas por las traducciones de QtSPIM la compilación del programa no da errores, por lo tanto, las instrucciones sintéticas se traducen igual, o por lo menos las que hemos utilizado nosotros.

Por ejemplo:

`lw $s0, A` se puede traducir por lo siguiente:

```
lui $1, 4097
lw $16, 0($1)
```

Se podrían traducir las instrucciones sintéticas de forma distinta, aunque el procesador que esté simulando es el mismo porque el compilador es software y puede variar según quien lo haya desarrollado.

20/09/19

- b. Las instrucciones de bifurcación no se traducen igual en ambos simuladores. En nuestro caso hemos usado la instrucción `bne $t0, $Zero, Buc1e`, la cual se traduce en MARS como `bne $8, 0$0, -17` y en QtSPIM como `bne $8, $0, -64`.

Creemos que la correcta es la de QtSPIM, ya que resta a la dirección de memoria lo necesario para que se ejecute de nuevo la primera instrucción del bucle:  $004000b4_{16} - 64_{10} = 400074_{16}$

3. Simplemente hemos hecho varias pruebas activando los puntos de ruptura después de distintas instrucciones de `load/store` para comprobar el estado de los registros después de que estas se ejecuten.

# Segunda Semana

26/09/19

1. Para realizar este ejercicio hemos usado dos bucles anidados controlados por dos contadores  $i$  y  $j$  con los que recorremos la matriz  $A$ . Mientras la recorremos, vamos obteniendo los valores  $A[i][j]$  mediante la fórmula  $A[i][j] = A + 4 * m * i + 4 * j$ . Esta fórmula la implementamos mediante las instrucciones `mul` para obtener  $i * n$ , `add` para obtener  $(i * n) + j$ , `sl` para multiplicarlo por cuatro y `add` de nuevo para añadirlo a la posición inicial del vector para posteriormente usar `sw` y así almacenar el valor en la posición  $B[j][i]$  que se obtiene de manera análoga a  $A[i][j]$ .

Antes de esto lo que hacemos es cargar las direcciones de las matrices en los registros  $\$a$  para usarlos como parámetros de las funciones.

En cada iteración de los bucles almacenamos un valor de  $A$  en su correspondiente posición en  $B$ , y los bucles finalizan cuando han iterado un número de veces igual al número de filas (y/o columnas ya que la matriz es cuadrada), lo cual comprobamos con instrucciones `beq` comparando los registros donde están almacenados los contadores con el número de filas.

2. En este ejercicio hemos utilizado de nuevo dos bucles con el mismo cuerpo que en el ejercicio anterior, pero con algunas modificaciones:  
Lo que hemos hecho es, en cada iteración, obtener los valores  $A[i][j]$  y  $B[i][j]$  de la misma forma que en el primer ejercicio, sumarlos y almacenarlos en  $C[i][j]$ , también de igual manera que en el anterior ejercicio.

27/09/19

3. En este ejercicio volvemos a realizar dos bucles anidados que usamos para recorrer la matriz de la misma manera que en los ejercicios anteriores. Estos bucles los controlamos con contadores cuyo valor es el número de filas que se pasa por parámetro.  
Esta vez, lo que hacemos dentro de los bucles es obtener los valores  $A[i][j]$  como siempre, pero una vez obtenidos, mediante la instrucción `add` los almacenamos en  $\$a0$  para imprimirlos realizando llamadas al sistema (en este caso, almacenando en  $\$v0$  el 1 con la instrucción `li $v0, 1`).

Cada vez que el bucle “interior” finaliza, lo que hacemos es imprimir un salto de línea con otra llamada al sistema, pero almacenando esta vez 11 en  $\$v0$  para que se imprima el valor almacenado en  $\$a0$  como carácter. Previamente habremos guardado en este último registro el número 10 que es el equivalente en la tabla ASCII al salto de línea.

28/09/19

4. Para este ejercicio lo primero que hacemos es pedir al usuario que introduzca los 25 valores que componen la matriz con la que vamos a operar, y esto lo hacemos con una llamada al sistema y con la instrucción `li $v0, 4`.  
Cada vez que el usuario introduce un valor, este es obtenido y guardado en  $A[i][j]$  de igual forma que en ejercicios anteriores. Cuando ya se han introducido todos los valores usamos la función del ejercicio 1 para hallar su traspuesta y almacenarla en  $B$ .

Después de esto usamos la función del ejercicio 2 para sumar ambas matrices (la original y su traspuesta) para, por último, imprimirla de forma tabulada utilizando la función del ejercicio 3.

La única variación es que, en este caso, al ser una matriz 5X5 sabemos el número de filas (5) que es el que controlara los bucles.

## Tercera Semana

03/10/19

1. En este ejercicio, tuvimos dos maneras de poder resolverlo, una de ellas era eficiente y la otra no tanto, pero funcionaba. Cuando probamos la forma eficiente, y vimos que funcionaba, nos quedamos con ella y así entregamos el fichero.

Antes de hacer el ejercicio tuvimos una fase de reflexión donde nos planteamos pasar el número que nos pedían a binario, pero luego nos dimos cuenta de que ese no era el camino correcto ya que ya lo podíamos tener en binario. También erramos al plantear el problema de esta manera:

Como 1 byte son 8 bits (1 posición de memoria global, 2 números en hexadecimal y 4 posiciones de memoria global formando una palabra) nos planteamos la posibilidad de hacer una máscara de 8 bits, así al hacer un `sb` (del `add` de la máscara con el número) quedarían almacenados 2 números en hexadecimal y cuadrarían perfectos en una posición de memoria global al hacer un `sb`.

Al ver la ejecución paso a paso vimos que metía bien los números, pero no nos dimos cuenta de que en realidad no lo estábamos codificando, sino que metíamos las cosas directamente y por ello, al imprimir la cadena aparecían cosas raras.

Reflexionando sobre el tema vimos que había que hacer necesariamente una máscara de 4 bits solamente que abarcara el número de 4 bits y más tarde lo interpretáramos como código ASCII, y una vez interpretado, y ahí sí, hacíamos un `sb` llevándolo a nuestro vector. Bien es cierto que somos más ineficientes, porque hacemos un `sb` de un número de "4 bits" cuando una posición de memoria podemos tener hasta 8 bits, pero de esta forma funcionaba.

También nos dimos cuenta de que al hacer un `sb` (no nos acordábamos) te coge los 8 últimos bits, así que tendríamos que desplazar el registro (el que íbamos a cargar a memoria) tantas posiciones como fuera necesario para que esos 4 bits quedaran los últimos (y a la izquierda todo 0's) y así poder hacer un `sb` correctamente.

04/10/19

La forma de poder hacerlo correctamente es la siguiente, escrito en código de alto nivel sería algo así (y gracias por el guiado):

```
for(int i = 0; i < 8; i++) {

 int a = numero_original;
 int b = resultado_tras_mascara;
 int mask = 0x0000000F;
 int c;

 b = a && mask;

 a = a << 4; // (desplazamos el número original)

 if (b < 10) {

 c = b + 48;

 } else {

 c = b - 10 + 'a'

 }

 //Aquí ya lo cargas en la cadena ASCII.

}
```

Y nos acordamos de finalizar la cadena con el `\0` (al final puede que interprete como que ha acabado porque siempre te encontrará eso en memoria).

Esto mismo codificado en MIPS, sería algo muy parecido: creamos una máscara (la cual no vamos a mover, lo que vamos a mover es el número original) y la razón de que movamos el número y no la máscara es que si movemos la máscara, entonces el número resultante (`add mask && numero`) quedaría cada vez más desplazado hacia la izquierda y lo tendríamos que ir desplazando en cada iteración para dejarlo al final del todo esos 4 bits para cuándo hagamos el `sb` poder hacerlo correctamente. Recordemos que el `sb` te coge los 8 últimos bits, en nuestro caso primeros 4 bits a 0 y los 4 últimos el número resultado de la máscara.

Como sacamos de 4 bits en 4 bits y nuestro registro es de 32 bits (porque así es nuestra arquitectura), tendremos que iterar 8 veces 4 bits para poder recorrer la palabra entera, así que conformaremos un bucle que englobe a nuestro código con esa condición.

Una vez sacado con la máscara nuestros 4 bits solo tenemos que compararlo con el número 10, así si es menor que 10 el número es tal cual y si es mayor que 10 estará entre el carácter 'A' y 'F'. Por lo que para poder codificar todo esto tuvimos que usar la tabla ASCII: para codificar los menores que 10 sumábamos el número más 48, lo que nos daba el número en ASCII; y si era mayor que 10, restábamos 10 y se lo sumábamos al carácter 'a'.

Hemos de decir que también hemos tenido problemas en el acceso a memoria porque no accedimos del todo bien pero ya lo logramos solucionar.

2.

- a. Aprovechando la función del apartado anterior, lo único que hay que hacer es imprimir la cadena cargando en `$v0` el valor de 4, y haciendo una llamada al sistema. (y antes cargamos la dirección donde se apunta a la cadena en `$a0`).
- b. Pruebas realizadas con los números indicados:

|                |               |
|----------------|---------------|
| 0              | 0x00000000    |
| 1              | 0x00000001    |
| -1             | 0xffffffff    |
| 130            | 0x00000082    |
| 511            | 0x000001ff    |
| -2148          | 0xffffffff79c |
| 2 147 483 647  | 0x7fffffff    |
| 536 870 911    | 0x1fffffff    |
| -2 147 483 648 | 0x80000000    |
| -1 073 741 824 | 0xc0000000    |

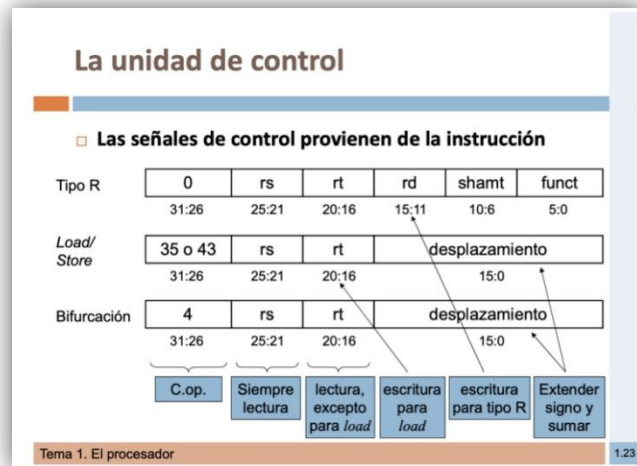
- c. En este apartado, simplemente hemos cogido el número que cogemos por pantalla y mediante la operación de desplazamiento hacia la izquierda (desplazamos dos veces hacia la izquierda), multiplicamos el número por 4.
- d. Estos son los resultados obtenidos tras multiplicar ese número por 4:

|                |             |
|----------------|-------------|
| 0              | 0x00000000  |
| 1              | 0x00000004  |
| -1             | 0xffffffffc |
| 130            | 0x00000208  |
| 511            | 0x000007fc  |
| -2148          | 0xffffde70  |
| 2 147 483 647  | 0xffffffffc |
| 536 870 911    | 0x7fffffff  |
| -2 147 483 648 | 0x00000000  |
| -1 073 741 824 | 0x00000000  |

Algunos resultados de los obtenidos no se esperaban como los 2 últimos donde dan todo 0s, pero creemos que es un problema de rango.

3. Con la opción de "MIPS X-Ray" hemos podido ver la ruta de datos del procesador. Para que funcionase, lo primero de todo hay que activar la opción de "connect to MIPS", luego ensamblar y más tarde ejecutar paso a paso.

Tenemos varios tipos de tipos de instrucciones, las vistas en teoría, que son, de tipo R de tipo lw/sw o de tipo bifurcación. Dependiendo del tipo de instrucción y de más factores haremos una ruta u otra en el procesador.



- a. Los diferentes colores que aparecen en la herramienta son los siguientes:
  - Rosa. Código de operación de la instrucción.
  - Verde. Codificación del registro rs.
  - Azul. Codificación del registro rt.
  - Naranja. Valor del inmediato.
  - Marrón. Inmediato con extensión de signo para poder operar en los multiplexores, en la ALU y poder ser desplazado a la izquierda.
  - Amarillo. Dirección actual para poder calcular la de la siguiente.
  - Salmón. Dirección de la siguiente instrucción (PC + 4).
- b. El camino crítico sería el siguiente: PC → memoria de instrucción → multiplexor → banco de registros → multiplexor → ALU → memoria de datos → multiplexor.
- c. Creemos que la representación es correcta pero no del todo, por lo que lo que modificaríamos para que fuese más correcta sería eliminar el multiplexor ubicado entre la memoria de instrucción y el banco de registros, la unidad ALU Control, el segundo multiplexor para calcular el PC y también el segundo Shift Left 2 de la misma operación.



4. Para hacer este ejercicio hemos aprovechado la función que hemos escrito en el ejercicio 1, pero ahora lo que le pasamos por parámetro a la función es un número en punto flotante.

Para leer un punto flotante hacemos un `li $v0, 6` y el resultado lo movemos a `$f0`, de tal forma que para pasárselo a `$a0` haremos un `move de $f0 → $a0`, pero no haremos un `move` normal, sino que utilizaremos la instrucción `mfc1 $a0, $f0` y entonces pasamos como parámetro el número en punto flotante a la función.

Los resultados de las pruebas son los siguientes:

|        |            |
|--------|------------|
| -1.0   | 0xbf800000 |
| 3072.3 | 0x454004cd |

## Cuarta Semana

08/10/19

1. Tenemos una función que se encarga de, una vez introducida la cadena en hexadecimal (codificada e interpretada en ASCII), poder convertirla en decimal. El nombre de dicha función es **“Convertidor”**.

Tras dar muchas vueltas a como poder hacerlo, se nos indicó hacer exactamente lo contrario de lo que hacíamos en la práctica anterior.

En la práctica anterior pedíamos un número por pantalla y lo transformábamos como una cadena de caracteres ASCII, por lo que en esta práctica al hacer lo contrario debemos tener en cuenta que la cadena que recogemos en hexadecimal ya la tenemos codificada en ASCII al ser una cadena. Este fue uno de los errores que más nos llevó comprender: aquí la codificación en ASCII ya la tenemos y lo que debemos es hacer lo inverso.

En pseudocódigo (más bien Java) en la práctica pasada, hacíamos algo así:

```
for(int i = 0; i < 8; i++) {

 int a = numero_original;
 int b = resultado_tras_mascara;
 int mask = 0x0000000F;
 int c;

 b = a && mask;
 a = a << 4; // (desplazamos el número original)

 if (b < 10) {
 c = b + 48;
 } else {
 c = b - 10 + 'a'
 }

 // Aquí ya lo cargamos en la cadena ASCII.
}
```

Pues bien, ahora tenemos que decodificar ese ASCII y devolverlo a un número (y realizar las operaciones necesarias para codificarlo en decimal):

```
for(int i = 0; i < 8; i++) {

 int a = numero_original;
 int b = resultado_tras_mascara;
 int mask = 0x0000000F;
 int c;

 b = a && mask;
 a = a << 4; // (desplazamos el número original)

 if (b < 10) {
 b = c - 48;
 } else {
 b = c + 10 - 'a'
 }

 // Parte para poder sacar el número en decimal:
 // Se tiene una máscara d que se va desplazando para poder
 // hacer el or con el número decodificado

 d = d * 4 // sll $t4, $t4, 4
 d "or" b // Hacemos el or entre la máscara y el número
}
```

Más adelante incluimos la posibilidad de que el usuario metiera la cadena por teclado. De la anterior forma podíamos hacer más rápidas las comprobaciones. El resultado en decimal lo devolvemos en el registro \$v0 como se indica en el guion.

10/10/19

2.

- a. Lo habíamos ya implementado en el ejercicio 1. Solo hacemos la llamada al sistema para poder imprimir el número en expresado en decimal.
- b. Comentados los resultados obtenidos con los enunciados indicados:

|            |             |
|------------|-------------|
| 0xff       | -48         |
| 0x00ffff   | -16         |
| 0xffffffff | -1          |
| 0x7fffffff | 2147483647  |
| 0x80000000 | -2147483648 |

- c. Ya lo implementamos en el ejercicio 1.  
Como indicación podríamos decir que, si metemos una cadena más larga de lo normal, caracteres inválidos o metemos las letras correctas en mayúsculas o minúsculas, daría un error en ejecución. Estos errores serán tratados más adelante en el guion.
- d. Cogemos por pantalla una cadena en hexadecimal, la convertimos en número y lo dividimos mediante las operaciones de desplazamiento entre 16. Es importante destacar que lo hacemos con la instrucción de desplazamiento srl y no lo hacemos con la instrucción sintética div.

3.

- a. Modificamos la función ya escrita en el apartado 1 para que nos muestre en `$v1` el resultado de la operación y en `$v0` el resultado del código de error. Utilizamos la siguiente lógica para lanzar los códigos de error: jugamos con los valores 0 y 1 del registro `$v0`.

Tenemos una instrucción sintética que nos va a poner el registro a `$v0` a 1 en el caso de que la cadena sea demasiado larga y va a poner, por el contrario, `$v0` a 0 si el carácter de la cadena es incorrecto. Si el resultado ha sido una cadena válida de entrada, entonces imprimimos el resultado.

Es importante indicar que todos estos errores no se lanzan desde nuestra función (en el caso de que haya error) sino que pasamos a la finalización del método y pasamos el control al main de nuevo, el cual determina, dependiendo de los valores de `$v0`, si ha habido algún error o no.

Si ejecutáramos el flujo del programa con una entrada correcta, (`$v0 != 0, 1`), entonces nos mostraría el resultado. Si en algún momento en el main `$v0 = 0` o bien `$v0 = 1` se mostrará el mensaje de error correspondiente.

La comprobación de errores que hicimos fue sencilla (después de haberlo planteado y hecho): Cogimos la tabla ASCII y metimos los números de la tabla ASCII de los caracteres que no nos interesaban de tal forma que cuando nos traemos el primer carácter al leer la cadena hacíamos un `lb`, iterábamos tantos caracteres ASCII como caracteres inválidos podríamos obtener. De esa forma (muy lenta) conseguimos comprobar que cada carácter era correcto, lanzando un mensaje de error en caso contrario y poniendo `$v0` a 0 o 1 según correspondiera.

- b. Para hacer este apartado debemos considerar el planteamiento hecho en las preguntas anteriores, así como la codificación que hicimos en Java de este problema en MIPS:

```
for(int i = 0; i < 8; i++) {

 int a = numero_original;
 int b = resultado_tras_mascara;
 int mask = 0x0000000F;
 int c;

 b = a && mask;
 a = a << 4; // (desplazamos el número original)

 if (b < 10) {
 b = c - 48;
 } else {
 b = c + 10 - 'a' // Aquí tendríamos que poner
 // para la letra 'A' en vez de
 // 'a'
 }

 // Parte para poder sacar el número en decimal:
 // Se tiene una máscara d que se va desplazando para
 // poder hacer el or con el número decodificado

 d = d * 4 // sll $t4, $t4, 4
 d "or" b // Hacemos el or entre la máscara y el
 // número
}
```

Para hacer bien esa sustitución de 'a' si el carácter mayor que 9 se encontraba en minúsculas o bien 'A' si el carácter mayor que 9 se encontraba en minúsculas tendríamos que hacer una comprobación de si el carácter estaba en mayúsculas o en minúsculas. Y ese proceso se vuelve a hacer con tabla ASCII en mano y en cuanto encontrase alguna letra que saltara a poder hacer la multiplicación y posteriormente hacer la máscara con "or".

|              |                        |
|--------------|------------------------|
| 0xFF         | -48                    |
| 0xFFFf       | -48                    |
| 0xfffffffffg | Carácter incorrecto    |
| 0xAfffffff   | -1342177281            |
| 0x80000000   | -2147483648            |
| 0xFFFFFFFFf0 | Cadena demasiado larga |
| 0xAfffffffg  | Cadena demasiado larga |

13/10/19

4.

- a. Solicitamos un número en hexadecimal por pantalla reciclando la función hecha en los apartados anteriores y hacemos la conversión de esa cadena en hexadecimal a un número en decimal.

Cuando obtenemos el número en decimal hacemos la división por 8 empleando la operación de desplazamiento hacia la derecha `sr1` con 3 posiciones. Resaltamos la importancia de hacer esta operación de la división mediante operaciones de desplazamiento y no mediante la instrucción sintética de `div`.

Una vez dividido el número en decimal por 8 tenemos que volver a mostrarlo en hexadecimal, así que, empleando la función de la práctica anterior, metemos el número ya dividido en decimal y, realizando las operaciones oportunas de la función, obtenemos el resultado de nuevo en hexadecimal.

- b. Obtención de los resultados con los enunciados propuestos:

|              |                                      |
|--------------|--------------------------------------|
| 0xc4033500   | -1006422784 (dec) / 0x188066a0 (hex) |
| 0x0001Be0    | 7136 (dec) / 0x0000037c (hex)        |
| 0xfffffffffg | Carácter incorrecto                  |

## Quinta Semana

15/10/19

1. Solicitamos un número y lo metemos en una cadena cuya dirección se encuentra en `$a0`, y devolvemos su valor en `$v0`.

En esta práctica, por fin, hemos decidido implementar los registros `$s` y salvaguardarlos en la pila (en anteriores practicas no lo guardábamos ella).

En la función lo que hacemos es, teniendo en la cadena el número almacenado, (por nosotros o porque lo introducimos por teclado), cogemos cada carácter codificado en ASCII (porque lo tenemos en una cadena y la interpretación es así) y le restamos 48 para obtener su valor como número (y no como carácter).

Sumamos a ese número el registro `$t3` (que será inicialmente 0), el cual será el resultado del producto que se hace en la línea siguiente, en nuestro caso el producto entre `$t2` y 10, donde `$t2` es el resultado de la suma (instrucción anterior).

Cuando volvamos a hacer el bucle ese \$t3 ya habrá tomado un valor y por lo tanto será distinto de 0.

Pongamos una imagen de cómo lo haríamos en lenguajes de alto nivel como puede ser Java o C:

```
for (int i = 0; i < hasta_que_haya_fin_de_cadena; i++) {

 char character; //una vez que tengamos el character
 // Para conseguir el carácter hacemos un lb de la primera posición
 // de memoria donde apunta a donde tenemos el String, aumentamos 1
 // posición de memoria cada vez que queramos conseguir el carácter
 // siguiente (avanzamos un byte que representa a un carácter). 1 byte
 // = 1 carácter en ASCII de 8 bits = 2 posiciones de memoria global.
 // Con el carácter, le restamos 48 (nos fijamos en la tabla ASCII)
 // para convertirlo a número. Una vez restado 48 al carácter, deja
 // de ser carácter y empieza a ser un numero normal.

 Numero_normal = carácter - 48;

 // Después sumamos ese número por el resultado del producto de más
 // adelante.

 Suma = Producto + Numero_normal;

 // Y posteriormente hacemos un producto

 Producto = Suma * 10

 // Con este algoritmo y esta última adicción de la suma y el
 // posterior producto podemos obtener el numero completo en decimal
 // (para pensar mejor la suma y el producto pensar cómo se pasaría
 // un número a base 10).

 // Debemos tener en cuenta detalles como la comprobación de
 // errores y la comprobación de si el número introducido es negativo,
 // por lo que para resolver esta última cogemos el primer carácter,
 // lo comparamos con 45 (codificación en ASCII del guion) y si
 // coincide sabemos que el número es negativo y por tanto tenemos
 // que tratar esa excepción. La manera de tratarla es ignorar ese
 // carácter (el signo menos), pasar a la siguiente iteración, y,
 // cuando acabemos de iterar, preguntamos si ha sido negativo para,
 // en ese caso, multiplicar por -1.

 // El método de hacer primero la suma y posteriormente el producto
 // nos llevó al problema en la última iteración con el producto, ya
 // que nos ponía un 0 al final. La fácil y rápida solución fue que
 // cuando terminase de iterar, dividimos entre 10 y solucionamos ese
 // error.
}
```

2.

- a. La función de leer por teclado el número para que lo meta el usuario, y mostrar el resultado en decimal por pantalla ya fue implementado en el ejercicio 1.

b.

|             |                                       |
|-------------|---------------------------------------|
| 25          | 25                                    |
| -048        | -48                                   |
| F024        | 22024 (sin implementación de errores) |
| 2147483647  | Se ha producido desbordamiento        |
| -5000000000 | Se ha producido desbordamiento        |
| -2147483648 | ERROR                                 |
| F024        | Character incorrecto                  |

- c. Para este ejercicio hemos recuperado el código implementado en la práctica 3. Nos piden introducir un número en decimal y devolverlo en hexadecimal. En esta práctica tenemos una cadena de caracteres, lo convertimos a decimal y ya tenemos el número en decimal. En la práctica 3 introducíamos un número en decimal y lo convertíamos a una cadena en hexadecimal. Haremos esos pasos entonces. Tras introducir el número en cadena y pasarlo a decimal y guardar ese número en un registro, solo tenemos que volcar ese número en decimal en el argumento de la función hecha en el ejercicio 3 y la función se encargará de pasarlo a hexadecimal.

3. Cuando el número es demasiado grande y se produce desbordamiento, lo que hacemos para comprobarlo es verificar si el registro HI es distinto de cero o si el registro LO es negativo después de cada operación. En el momento en que eso ocurre, se salta a una etiqueta donde se carga un 3 en el registro \$v0.

Otro posible error que comprobamos es que puede haber caracteres en la cadena introducida. Este error lo verificamos analizando si el valor del carácter en ASCII es menor de 48 o mayor de 57 y si se cumple una de estas dos condiciones, saltamos a la etiqueta donde cargamos un 2 en \$v0.

Si todo es correcto, el valor de \$v0 es cero.

4.

- a. Con la función que hicimos en el apartado anterior y la función desarrollada en la práctica 3 hicimos este apartado. Primero llamamos a la función que convertía nuestra cadena de caracteres a decimal y más tarde volvemos a llamar a la misma función pidiendo otro número al usuario. Cuando ya tenemos esos 2 números en decimal, los sumamos y el resultado de la suma lo metemos como argumento a la función hecha en la práctica 3. El resultado será la suma de los dos números introducidos por pantalla en una cadena en hexadecimal.

b.

|                 |                                   |
|-----------------|-----------------------------------|
| 25-048          | ffffffe9                          |
| 25 + F024       | Se ha producido un error de char  |
| 25 + 2147483647 | Se ha producido un desbordamiento |
| 25 - 5000000000 | Se ha producido desbordamiento    |
| 25 - 2147483648 | arithmetic overflow               |
| 25 + 25         | 00000032                          |

|             |                             |
|-------------|-----------------------------|
| -048 + 25   | ffffffe9                    |
| -048 + F024 | Se ha producido un error de |

|                   |                                   |
|-------------------|-----------------------------------|
|                   | char                              |
| -048 + 2147483647 | Se ha producido un desbordamiento |
| -048 - 5000000000 | Se ha producido desbordamiento    |
| -048 - 2147483648 | arithmetic overflow               |
| -048 - 048        | ffffffa0                          |

|                   |                                  |
|-------------------|----------------------------------|
| F024 + 25         | Se ha producido un error de char |
| F024 + F024       | Se ha producido un error de char |
| F024 + 2147483647 | Se ha producido un error de char |
| F024 - 5000000000 | Se ha producido un error de char |
| F024 - 2147483648 | Se ha producido un error de char |
| F024 + F024       | Se ha producido un error de char |

|                         |                                   |
|-------------------------|-----------------------------------|
| 2147483647 + 25         | Se ha producido un desbordamiento |
| 2147483647 + F024       | Se ha producido un desbordamiento |
| 2147483647+ 2147483647  | Se ha producido un desbordamiento |
| 2147483647- 5000000000  | Se ha producido desbordamiento    |
| 2147483647 - 048        | Se ha producido un desbordamiento |
| 2147483647 - 2147483648 | Se ha producido un desbordamiento |

|                          |                                   |
|--------------------------|-----------------------------------|
| -5000000000 - 048        | Se ha producido un desbordamiento |
| -5000000000 + F024       | Se ha producido un desbordamiento |
| -5000000000 + 2147483647 | Se ha producido un desbordamiento |
| -5000000000 - 5000000000 | Se ha producido un desbordamiento |
| -5000000000 - 2147483648 | Se ha producido un desbordamiento |
| -5000000000 + 25         | Se ha producido un desbordamiento |

|                          |                     |
|--------------------------|---------------------|
| -2147483648 - 048        | arithmetic overflow |
| -2147483648 + F024       | arithmetic overflow |
| -2147483648 + 2147483647 | arithmetic overflow |
| -2147483648 - 5000000000 | arithmetic overflow |
| -2147483648 - 2147483648 | arithmetic overflow |
| -2147483648 + 25         | arithmetic overflow |

Sexta Semana

**Importante:** Esta práctica 6, incluye las funciones pedidas en el ejercicio 1 y 2, la función pedida en la practica 4 (hexadecimal a decimal), y la función pedida en el ejercicio 6 b) (sumatorio pedido). Según el fichero entregado, primero se ejecuta la función del sumatorio del ejercicio 6 b), más adelante se piden dos número en hexadecimal para sumarlos y dar su suma en decimal (ejercicio 6 a) ), y por último se pide un numero para la función correspondiente a los ejercicios 1 y 2.

1. Para realizar esta función, lo que hemos hecho es que almacene el número a convertir en una cadena, pero al quedar almacenado al revés luego realiza su inversión para almacenarlo ordenado de forma definitiva.

Para guardar el número “al revés” lo que hacemos es dividirlo entre diez con la instrucción `div` (habiendo previamente guardado un 10 en `$t5` para operar) y guardamos el resto en `$t2` para sumar 48 a dicho valor, obteniendo así su carácter ASCII, y almacenarlo en la cadena mediante la instrucción `sb`.

Posteriormente, sustituimos el número introducido por el valor del registro `lo`, ya que es el cociente y es el valor que debemos volver a dividir entre diez para repetir el proceso anterior y obtener y almacenar el siguiente carácter.

El bucle finaliza cuando, después de sustituir el número por el valor del registro `lo`, este es menor que diez, por lo que no se puede realizar el proceso anterior. Cuando se da esta situación, lo que hacemos es sumar 48 a dicho valor para obtener su carácter ASCII y almacenarlo en la cadena.

Una vez hecho esto, ya tendríamos almacenado el número en una dirección de memoria, pero estaría totalmente invertido (si el número introducido es el 1234, el almacenado sería el 4321).

Para invertir esto, lo primero que hacemos es inicializar un contador que servirá para controlar el número de veces que se debe de realizar el bucle (el número de caracteres que tiene el número introducido).

Dicho bucle lee un carácter de la cadena “invertida”, lo almacena en `$t4`, avanza a la siguiente posición y repite la operación. Cuando ya no quedan caracteres por leer, finaliza dejando todos los caracteres almacenados en `$t4` y el contador con el número de caracteres que tiene el número introducido.

Después, otro bucle realiza la operación contraria, es decir, coge un carácter de `$t4`, lo guarda en la cadena “ordenada”, incrementa `$t4` en uno y decrementa el contador, repitiendo el proceso hasta que el contador es cero, indicador de que ya se han ordenado todos los caracteres.

2.
  - a. Para la realización de este ejercicio hemos utilizado la función del ejercicio 1 pero añadiendo una cadena que solicite al usuario que introduzca el número y la llamada al sistema encargada de imprimir el resultado por pantalla.

Además, debemos comprobar si es negativo, ya que en ese caso deberíamos multiplicarlo por -1 para obtener su valor absoluto. Para ello, mediante la instrucción `bgtz` comprobamos si es mayor que cero. Si lo es, saltamos directamente al bucle, pero en caso contrario, multiplicamos por -1 y ya iniciamos el primer bucle de la función.



b.

|                |            |
|----------------|------------|
| 10             | 10         |
| -2             | 02         |
| 65             | 65         |
| -524           | -524       |
| 63             | 63         |
| 2.147.483.647  | 2147483647 |
| -2.147.483.647 | 2147483647 |
| -1.073.741.824 | 1073741824 |
| 1.073.741.824  | 1073741824 |

25/10/19

3.

- a. Para realizar este ejercicio hemos utilizado la función realizada en la práctica 4, pero como teníamos varios errores en dicha práctica, hemos tenido que volver a hacer muchas partes de la función. En concreto, hemos vuelto a rediseñar todo el tema de los errores haciéndolo mucho más eficiente: antes la comprobación era carácter a carácter y ahora solo comparamos si está dentro un rango de números, lo que lo hace mucho más rápido y eficiente.

Para empezar, en su día hicimos este algoritmo en un lenguaje de alto nivel:

```
for(int i = 0; i < 8; i++) {

 int a = numero_original;
 int b = resultado_tras_mascara;
 int mask = 0x0000000F;
 int c;

 b = a && mask;
 a = a << 4; // (desplazamos el número original)

 if (b < 10) {
 b = c - 48;
 } else {
 b = c + 10 - 'a'; // Aquí tendríamos que poner
 // para la letra 'A' en vez de
 // 'a'
 }

 // Parte para poder sacar el número en decimal:
 // Se tiene una máscara d que se va desplazando para
 // poder hacer el or con el número decodificado

 d = d * 4; // sll $t4, $t4, 4
 d "or" b; // Hacemos el or entre la máscara y
 // el número
}
```

Una vez que obteníamos el número en decimal lo guardábamos en algún registro que sabíamos que no se iba a pisar. Una vez que teníamos ese registro a salvo llamábamos de nuevo a la función y, haciendo lo mismo, este resultado

lo guardábamos en otro registro donde también queríamos preservar el dato. Teniendo los dos datos en mano solo tenemos que realizar la operación, que en este caso era una suma y una vez teniendo la suma de los dos números en decimales podemos retornar el resultado ya en decimal.

- b. Para realizar este ejercicio tuvimos que hacer una función recursiva en alto nivel para poder comprenderlo mejor. Una vez visto como lo haríamos en alto nivel, lo podíamos implementar en ensamblador.

Lo primero de todo es resolver la operación  $-1^i$ , y se resuelve de la misma forma que  $2^i$ . Primero declaramos unos casos base para termine la función en algún momento. Para  $-1^i$ , si  $i = 0$ , el resultado es 1.

Análogamente pasa lo mismo para  $i = 0$  en  $2^i$  es 1.

El segundo caso base es para  $-1^i$ , si  $i = 1$ , el resultado es -1

Para el caso de  $2^i$ , si  $i = 1$  el resultado es 2.

Dejando atrás los casos base vamos a la recursividad.

- $-1^i = (-1^{i/2}) \cdot (-1^{i/2})$  si  $n > 1$  e "i" es par.
- $2^i = (x \cdot 2^{i-1})$  si  $i > 1$  e "i" es impar.

Tenemos dos funciones que hacen para las dos operaciones:  $2^i$  y  $-1^i$ .

Cuando tenemos ambos resultados del producto de ellos solo guardamos el resultado en un contenedor y hacemos esta operación tantas veces como el número que ha introducido el usuario por pantalla para hacer ese contenedor donde vamos acumulando las sumas de los productos de nuestros dos operandos.

## Práctica Final

29/10/19

Básicamente lo que hemos hecho esta primera semana es corregir los fallos que teníamos en las prácticas anteriores que podemos necesitar en esta práctica final, así como realizar los cambios oportunos en ellas para poder utilizarlas en esta situación, ya que algunas, tal y como estaban implementadas para los enunciados anteriores, no funcionarían correctamente en la práctica final.

Por ejemplo, en la práctica 6 lo que hemos hecho es eliminar todo lo sobrante para quedarnos exclusivamente con lo que se requería y, posteriormente, ir realizando las distintas fases de la práctica de la más simple a la más compleja para así no complicarnos e ir sabiendo lo que vamos consiguiendo implementar y lo que no.

Antes lo que hacíamos era algo raro, una vez que teníamos la cadena (en orden invertido) la metíamos en otra cadena para invertirla y hacíamos una función recursiva, algo muy lioso para lo que verdaderamente había que hacer.

Lo que hacemos es esta corrección, es una vez que tenemos la cadena al revés, tenemos otra cadena para poder volcar esa cadena al revés, pero al derechas.

En el bucle donde vamos guardando los caracteres en la cadena invertida tenemos un contador que nos va a indicar en números de elementos del vector y esa es la pista totalmente necesaria para ir colocando bien los caracteres en la cadena nueva.

Una vez que hemos colocado los caracteres en la cadena al revés, y nuestro contador está asignado un valor (tamaño de la cadena), podemos colocar nuestros bytes en nuestra nueva cadena en el orden correcto. La primera acción es acceder a la última posición de nuestro nuevo vector (donde va. Estar la cadena colocada) y la última posición vendrá determinada por el contador de antes.

Una vez comprendida esa lógica, vamos disminuyendo el puntero hasta llegar al principio.

De esta forma habremos logrado dar la vuelta a la cadena si necesidad de hacer la cosa rara que hacíamos anteriormente en esta práctica 6. No hemos podido hacer muchas as

10/11/19

Hemos podido avanzar muchas más cosas esta semana ya que no teníamos los exámenes. Hasta ahora hemos conseguido mejorar todas las practicas anteriores como se dijo, y recoger por pantalla la cadena pedida al usuario. La idea es tener un vector de 15 posiciones y cada posición se corresponde con un exponente. Es decir,  $3x^0$  se colocaría el 3 en la primera posición del array, y así sucesivamente hasta  $x^{15}$ . La lógica utilizada es sencilla y de momento funciona, se trata de coger los caracteres que nos interesen y los que no nos interesen (las x, o los ^) pasamos de ellos y accedemos al siguiente numero. Adentrados un poco más en la practica, nos dimos cuenta de que había que distinguir verdaderamente entre el primer número ( $3x^4$ ) en este ejemplo 3, que sería la base, y por lo tanto el número a colocar en el vector, y el 4 que sería el exponente, y sería la posición a la que acceder del array donde almacenamos el 3.

Para ello se nos ocurrió que cuando encontremos el carácter '^', entonces sabemos que el próximo número se refiere al exponente, y colocamos en esa posición la base que la tenemos previamente guardada en un registro (\$t2). Todos los demás caracteres en principio nos darían igual.

Tenemos también un método donde capturamos si el número es negativo o positivo, para ello tenemos un registro que, si aparece un menos, se activa (le ponemos a 1), y a la hora de tratar la base decimos que si ese registro esta a 1, multipliquemos el número de la base por -1. De esa forma tenemos los errores de signo paliados.

Respecto al tratamiento que se hacen para el tema de errores, tenemos algo implementado, pero el problema es que hay infinidad de formatos que se podrían meter y eso multiplica por mucho la posibilidad de error. De momento contemplamos la posibilidad de meter en orden que queramos los miembros del polinomio, también tenemos que, si hay varios miembros con el mismo exponente, entonces se sume a lo que ya había, de esa forma no pisamos un valor en el caso de que haya dos o más exponentes iguales.

Una vez rellenado el vector con los datos de entrada, estuvimos planteando la función de hacer la derivada, y simplemente es coger la posición del vector, (el exponente), y multiplicarlo por la base (el numero que no es el exponente), y elevarlo a un grado menos; es decir meterlo en otro array donde recolocaríamos exponentes (posiciones).

16/11/19

Esta semana hemos logrado avanzar más que la semana anterior. Hemos conseguido meter ya bien el polinomio que mete el usuario por pantalla en el array propuesto, para posteriormente hacer su derivada. Una vez que tenemos los datos en el array, es tan sencillo como multiplicar cada posición del array por el numero que hay dentro, y meterlo en nuevo array donde las posiciones se corresponden a una menos que el anterior array; es decir  $3x^2$  su derivada es  $6x^1$  por lo tanto en el array almacenado estará el 3 en la posición 2, y en la derivada estará el 3 en la posición 1.

Para meter la cadena introducida por el usuario en el array inicial, utilizamos la función de la práctica 5. Vamos leyendo una cadena de caracteres, y cuando encontramos una x, paramos y guardamos el numero introducido por pantalla a un numero de decimal para guardarlo en un registro, a continuación, cuando llega el '^' le dejamos pasar igual que la 'x', y el numero siguiente al elevado, se corresponde con la posición que va a ocupar en el array el numero que guardamos en un registro. Como dijimos en diarios anteriores, para diferenciar los dos números; la base y el exponente, primero guardamos la base, y cuando tengamos un '^' activamos una bandera y ya sabemos que el anterior numero es la base y va a posicionarse en el siguiente número que sigue al elevado.

Una vez colocados bien los elementos en el array, es hora de hacer la derivada. Para hacer la derivada como dicho antes, solo hay que multiplicar la base por la posición del array que ocupa (utilizamos un contador para ello). Hay que tener en cuenta un par de casos especiales que van a hacer la bajada de exponente.

Si tenemos, por ejemplo,  $3x^0 + 5x^1$ , ambos 2 van en la posición (en el array de derivadas) 0, porque el primer termino va a dar 0, y el segundo termino va a dar el numero base, por lo que esas dos primeras iteraciones van a dar lugar a que en la primera posición del array de derivadas se posiciones

la base del segundo termino (la base) del polinomio de ejemplo.

Los demás términos tienen que ir una posición menos metidos en el array de derivadas; por ejemplo, si tenemos  $4x^5$ , su posicionamiento sería en la posición 5 del array normal (y se pondría un 4 en esa posición), y en la posición 4 del array de derivadas (y se pondría un 20 en esa posición).

Ahora una vez que tenemos en el array normal los números de la cadena metida por el usuario, y en el array de derivadas, ahora solo tenemos que hacer la función inversa, imprimir de número a cadena, es decir, de decimal a binario, y eso lo hemos cogido de la práctica 6. Simplemente cuando nos cogemos un número del registro, lo codificamos y lo pasamos a cadena, y más adelante le añadimos la  $x$ , el elevado y la posición que ocupa en el vector.

Cosas a destacar donde hemos tenido problemas:

Cuando hemos introducido los números ya codificados (binario – decimal), lo metíamos en el vector como un `lw` (como debe ser), pero aumentábamos cada número en 1, y había que hacerlo en 4, y es por eso por lo que daba un fallo de alineación memoria. Hay que destacar el hecho de que cuando estas sacando cosas de la cadena del usuario, cuando decodificas a número (binario – decimal), lo haces cogiendo y tratando como `byte`; `lb` o `sb`, pero cuando ya lo tienes como número lo tratamos como `lw` o `sw` y el puntero se aumenta en 4 (por ser enteros) y en 1 cuando lo hacemos en bytes (porque es un byte).

24/11/19

Esta semana hemos logrado corregir los errores mas importantes (y otros no tan importantes) que arrastrábamos de las semanas anteriores. Hemos empezado a hacer mas pruebas con el servidor para lanzar pruebas y obtener mas puntuación. Por el mensaje que nos enviaste, miramos porque imprimía un 0 delante del número cuando el número era una sola cifra, lo que hacía que muchas pruebas al parecer fallaran. Después de hacer muchas pruebas con números diferentes, vimos que no solo pasaba que ponía un 0 delante, si no que a veces hacía cosas raras, como imprimir ":" en algún número random. Fuimos acotando el error y lo dejamos en que estaba en la traducción a cadena, y efectivamente era así. Modificamos por tanto la función de la práctica 6 (de decimal a binario) para que no cometiéramos ningún error. Hicimos un cambio y ya colocaba bien, pero a la hora de posicionarlo en memoria teníamos algunos desajustes que viéndolos tranquilamente logramos solucionar. Tuvimos algún problema al imprimir el exponente (cuando el exponente era mayor que 9), ya que, si era 10, imprimía el valor correspondiente al ASCII en 10, y en realidad queríamos la combinación en ASCII del 1 y del 0. Al final no tuvimos mas opción que llamar a la función hecha en la practica 6 (decimal a binario) pasarle el número del exponente y que lo colocara en la cadena. Con los números debajo del nueve hacemos lo mismo por simplicidad.

De tal forma que aplicamos la lógica de primero de todo cargamos el exponente de la manera indicada(en la cadena), es decir llamando a la función de la practica 6, mas tarde incluimos los caracteres '`x`', '`^`' (directamente, sin llamar a la función), y mas tarde el número a convertir, (esta vez si que llamamos a la función), y por ultimo de todo cargamos directamente un mas o un menos (carácter ASCII), dependiendo si el número contenido es positivo o negativo. Finalmente imprimimos la cadena. Este proceso le repetimos tantas veces como exponentes sean posibles, en este caso como el máximo es  $x^{15}$ , pues 16 veces iteramos sobre esa función. Se ha de resaltar, que como se utiliza siempre la misma cadena para escribir los resultados, cada iteración (de las 16) se resetea la cadena poniendo todos los bytes a `\0` para que no haya una sobre escritura en la cadena. Si no haríamos esto ultimo, el número mas largo (cadena mas larga) quedarían restos de el en la cadena y se imprimirá en la siguiente iteración. Por ejemplo,  $123456x^1 + 123x^2$  al imprimir el segundo, contendrá los registros no pisados del valor anterior. Para hacerlo mas eficiente aún, solo se ponen a `\0` aquellas iteraciones que tengan algún sentido, es decir, si tengo  $0x^{10}$  no voy a poner los bytes de la cadena como `\0` porque no habría nada y es volver a escribir `\0` sobre `\0`. Por lo demás estamos intentando ver que más cosas pueden fallar para poder avanzar mas puestos en el servidor.

31/11/19

Esta semana hemos implementado varias mejoras en la salida, reduciéndola al máximo para que sea lo más simple posible con acciones como eliminar el 'x^0' para simplemente mostrar la base del monomio o eliminar el 'x^1' para mostrar únicamente una 'x', así como eliminar el primer signo '+' que aparecía en ciertas salidas y que no debería aparecer en ninguna cuando el número es positivo. Además, hemos intentado, aunque aún no tenemos del todo implementado el que al introducir un número solo (p. ej., el 123) la salida sea 0, ya que hasta ahora en ese caso no imprimía nada, lo cual está mal.

Para poder quitar el mas al principio lo hemos hecho con un simple if, si se cumplía la condición de que estábamos en la primera iteración, es decir cuando empezábamos a escribir por pantalla ya sea el  $x^{14}$  o  $x^2$ , (el primer número que saliera) tenía que salir sin el símbolo '+'.  
Para el tema de poder hacer que cogiese un número sin la x y contemplara ya que es como un  $x^0$ , '1243' pe, nos ha resultado algo más complicado, ya que nuestra lógica era diferenciar los dos números que había, la base y el exponente. Cuando teníamos la base, nos la guardábamos hasta colocarla en la posición indicada por el exponente al colocar la cadena. Para saber diferenciar, teníamos una bandera que se ponía a 1 cuando encontraba el '^'. Ahora que no tenemos el "elevado" para números en  $x^0$ , tuvimos que adaptarnos a la situación para saber que el número "sin nada".

La conversión que hacemos de ese número a cadena, terminamos de iterar cuando tenemos una 'x', puesto que ahora no tenemos nada, aplicamos la lógica de cuando encontremos un '+', un '-', o un salto de línea, entonces sabemos que lo tenemos que posicionar en la posición 0.  
Para el  $x^1$ , solo podríamos poner 1234x por ejemplo, así que miramos si después de la 'x', hay un '+', un '-', o un salto de línea, y así sabemos que hay que posicionar ese número en la posición 1.

La conversión que hacemos de ese número a cadena, terminamos de iterar cuando tenemos una 'x', puesto que ahora no tenemos nada, aplicamos la lógica de cuando encontremos un '+', un '-', o un salto de línea, entonces sabemos que lo tenemos que posicionar en la posición 0.

Para el  $x^1$ , solo podríamos poner 1234x por ejemplo, así que miramos si después de la 'x', hay un '+', un '-', o un salto de línea, y así sabemos que hay que posicionar ese número en la posición 1.

Para diferenciar estos dos casos, y para "rellenar" el hueco del exponente (le tenemos que pasar un número al registro exponente) lo hacemos con banderas también. Si tenemos el caso que es exponente "número", ponemos la bandera a un valor y luego comprobamos si se ha lanzado esa excepción para cargar el registro exponente al número adecuado.

También hemos añadido más mejoras, como la posibilidad de poner un '+' en el exponente y que no salte ningún error de cadena no válida. Es decir, si te ponen  $3x^{+4}$  te lee  $3x^4$  y no te da ningún error de cadena. Lo hemos hecho diciéndole al programa que cuando vaya a recibir el exponente, si se encuentra un '+' o un '-', entonces tiene que iterar al siguiente elemento, el siguiente tiene que ser ya lógicamente un número. Si es otro 'x', el programa casca y te muestra cadena mal introducida.

Tuvimos en cuenta el tema de poner la X o x, y hemos hecho que la validación tanto de las mayúsculas o minúsculas sean válidas. El tema de poner dos veces '++' al principio del número, no daba error (al igual que con --), pero también hemos implementado que para dados esos casos indique que la cadena es incorrecta.

Hemos tenido en cuenta un error más que no teníamos en cuenta, y era el de MAL DERIVADA, y se daba en el caso de cuando multiplicábamos el exponente por la base y se salía de rango, para todo esto teníamos las condiciones de LO y HI, separando los casos de si el número era positivo o negativo.

Por el mismo camino, hemos solucionado los errores de Overflow que daban en la suma del parseamiento de la cadena, siguiente la siguiente lógica:

-Si el primero el positivo y el segundo el positivo, da desbordamiento cuando el resultado es negativo.

-Si el primero el positivo y el segundo el negativo, nunca da desbordamiento.

-Si el primero el negativo y el segundo el positivo, nunca da desbordamiento.

-Si el primero el negativo y el segundo el negativo, da desbordamiento cuando el resultado es positivo.

Es importante destacar que cuando haces la suma y después la comprobación anterior, esa suma la haces "addu".

Hemos añadido algunas mejoras más como puede ser el tema de que si introduces un espacio (carácter 32 en ASCII) ignore ese espacio y tome esa cadena como si no hubiese ningún espacio.

Esta ultima mejora simplemente hicimos que cuando se encuentre un 32 aumente el puntero en 1, y saque el siguiente termino.