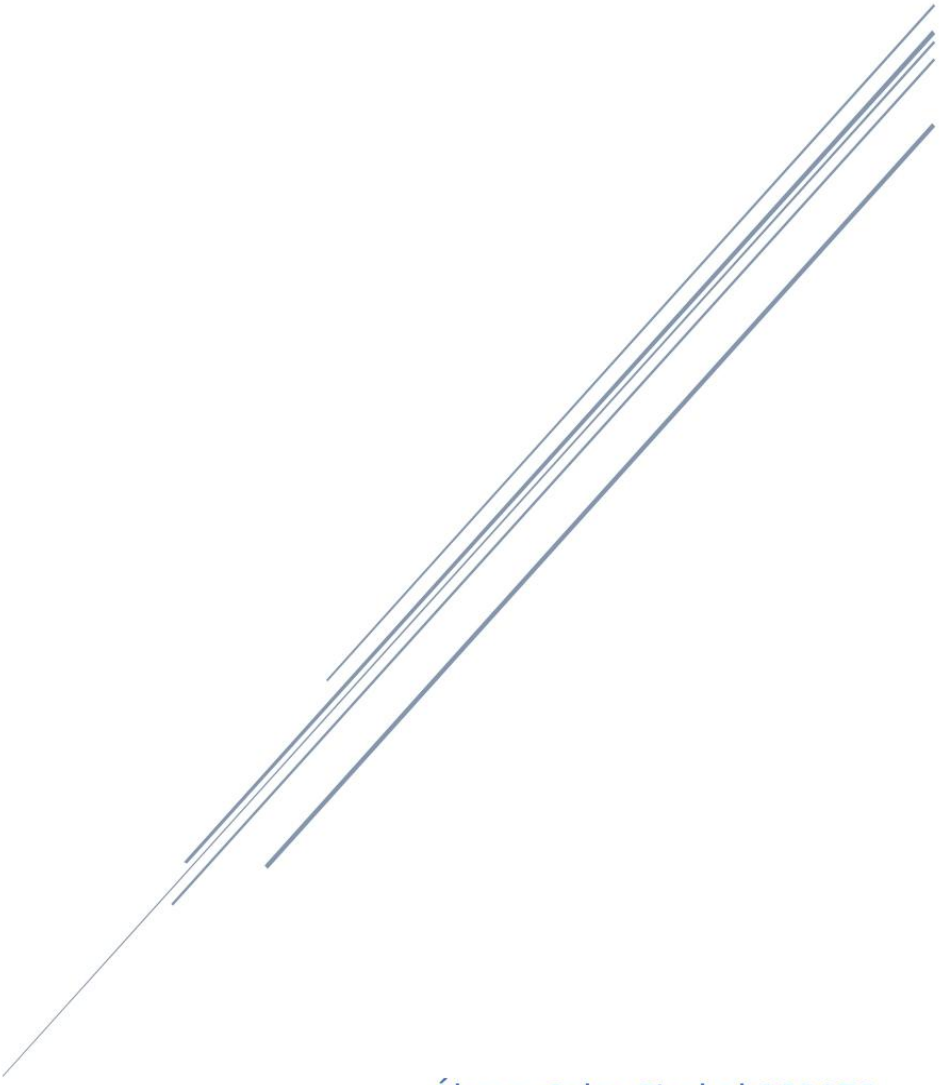
MEMORIA

Proyecto Ensamblador 2021/22



Álvaro Cabo Ciudad 200172
Pablo Fernández de Gracia 200068

1. Índice

2. Histórico del desarrollo del Programa

Dentro de este este apartado puede encontrar las anotaciones realizadas durante el desarrollo del programa, divididas en versiones correspondientes a las 3 “grandes fases” por las que ha pasado el proyecto hasta su entrega final el 20 de Junio.
Cada versión cuenta con subapartados que describen cómo se realizaban los avances en las respectivas subrutinas de forma paralela, el día de inicio, el tiempo empleado en estos y la organización entre nosotros a la hora de repartirnos las tareas.

V1. (Ordinaria)

1.1. Primeros pasos (06/11/2021)

Definición de Macros y Subrutinas:

MACROS (0,5 horas): Implementación de las macros LEA, LOAD, DBNZ, PUSH, POP, VALOR y PILA usadas a lo largo del proyecto tanto para la elaboración de las subrutinas como para los casos de prueba.

Subrutina LongCad (1,5 horas): Se implementa la subrutina LongCad. En la depuración de la misma, no se encuentran fallos para los casos de prueba planteados por nosotros.

Subrutina BuscaCar (1 hora): Se implementa la subrutina BuscaCar. En la depuración de la misma, encontramos fallos para los casos de prueba planteados por nosotros mismos, también en los casos de prueba del corrector.

Subrutina CoincidenCad (2 horas): Se implementa la subrutina CoincidenCad. En la depuración de la misma, no se encuentran fallos para los casos de prueba planteados por nosotros.

Subrutina BuscaMax (3 horas): Se implementa la subrutina BuscaMax. En la depuración de la misma, no obtenemos los resultados esperados para los casos de prueba planteados por nosotros mismos, tampoco pasa las pruebas del corrector.

1.2. CheckSum + Comprime (08/11/2021)

Subrutina CheckSum (3 horas): Se implementa la subrutina CheckSum. En la depuración de la misma, obtenemos los resultados esperados para los casos de prueba planteados por nosotros pero en el corrector solamente conseguimos pasar 4/7 casos de prueba.

Subrutina Comprime (4 horas): Planteamos un “esqueleto” de la

subrutina Comprime sin mucho éxito ya que las demás subrutinas necesarias no funcionaban al 100%.

Hemos decidido dejar el proyecto para intentarlo en la convocatoria extraordinaria ya que no contamos con el tiempo y la práctica necesaria para llegar a tiempo en la convocatoria ordinaria.

V2. (Hito + Descomprime)

2.1. Primeras modificaciones (11/06/2022)

Cambio en la forma de realizar las pruebas y reestructuración de las cabeceras de las subrutinas (2 horas): Lo primero que nos hemos dado cuenta al "desempolvar" el código desde el primer semestre es que la forma en la que introducimos los parámetros en la pila en las llamas a la subrutinas no están bien hechas, puesto que introducíamos los parámetros en orden inverso.
Además decidimos prescindir de las macros PILA y VALOR ante la mayor simplicidad de utilizar el esquema dado en la guía.
- La consecuencia de haber realizado mal las pruebas es que en las subrutinas en las que se leía más de un parámetro éstos se metían en registros no deseados, por lo que las pruebas del gestor no coincidían con el resultado que obtenemos nosotros.

BuscaCar (0.5 horas): Examinamos el código para encontrar el error que hacía que no pasara ninguna de las pruebas. Se modifica la línea 1 para guardar el carácter como palabra en vez de como byte para evitar errores de alineamiento. Además de simplificar el código evitando control de errores innecesario y eliminando "Fracaso_Buscacar" ya que su función consistía en `if(r2==to) {r2<- to}`, por lo que no tenía utilidad.

Replantear Checksum (2 + 4 horas): Tras dos horas intentando cambiar nuestro código original para arreglar los fallos que nos lanzaba el corrector, decidimos reconstruir Checksum siguiendo un esquema mucho más simple:

- Eliminando el bucle suma y sustituyendo toda la operación de añadir el carácter en su posición dentro del registro por un mak 0x100 en vez de un mulu, operación que causaba errores tanto de resultado como de alineamiento.
- Eliminando el proceso de añadir caracteres arbitrarios puesto que solo era necesario comprobar que el carácter que leíamos del texto era nulo e interpretar como carácter arbitrario el 00 que queda en su lugar.
- También se elimina el sumador de palabra puesto que no vamos a operar nada a nivel de palabra, solo el puntero dentro de las mismas para añadirlas en la posición correcta, el resto se guardaría en un checksum general

Descomprime (6 horas): En esta entrega, trabajamos de manera paralela en Comprime y Descomprime, ya que a pesar de que Comprime exigía una mayor dificultad, teníamos el tiempo en contra para poder realizar la corrección de las 21:00, por lo que decidimos trabajar de manera simultánea en Comprime y Descomprime.

Fuimos siguiendo los pasos de la documentación del proyecto aportada y las dificultades encontradas en esta subrutina se resumen en el paso 3 de la documentación debido al tratamiento del mapa de bits, que fue lo que más tiempo llevó ya que al principio no sabíamos cómo movernos dentro del mismo, pero la operación *extu* facilitó bastante el código para la realización de los desplazamientos. También encontramos dificultades a la hora de tratar los los bits de los bytes almacenados en la variable temporal ya que al principio no nos dimos cuenta de que estaban ordenados de manera decreciente (bit7...bit0) por lo que obtuvimos resultados no esperados en los casos de prueba. Tras esta entrega, solo obtuvimos una prueba que fallase en el corrector, la cual no supimos arreglar durante las demás entregas ya que el fallo era pasando como parámetro una cadena de más de 256 carácteres con solo 2 diferentes (fallo por Bucle Infinito), pero al realizar nosotros esa prueba no encontrábamos ningún bucle infinito por lo que suponemos que no supimos realizar esa prueba de manera correcta, ver casos de prueba para consultar la cadena que nosotros pasamos.

Entrega el día 13: Pasa hito 1 y un fallo en descomprime

2.2. Recodificación de Comprime (15-16/06/2022)

Primer intento de Comprime (6 horas): Se modifica todo lo que teníamos de comprime:

- Se quita el guardado en pila innecesario de r21 en pila (es un parámetro lo podemos recuperar con *ld r31, 12* cuando queramos)
- Reservamos r2*4 bytes (r2 palabras)
  Se respeta la inicialización de punteros de texto y escritura y además se añaden los punteros necesarios para trabajar con el mapa de bits.
- Se modifica el Bucle_cmprimir a -> Pre_cmpr para un mejor distinción con el algoritmo y además se escribe en memoria en formato Little Endian, por lo que se empieza desde r31 -1 -> r28 y se escribe “de derecha a izquierda” respecto a esa posición
- La posición que se pasa como parámetro jj al Buscamax pasa a ser la que está en la cima de la pila en ese momento en lugar de r0, que se guarda en r17.
- Se hace un primer intento de completar el algoritmo de compresión, siguiendo el esqueleto que se tenía hecho, se crea un sistema que distingue entre los dos grandes submétodos de este algoritmo:
  - Para r29<4: Copia el carácter en la zona reservada en pila en formato little endian (r31-r6) y escribe un 0 en bitmap
  - En caso contrario, se lee P de la dir del jj (r30) y se guarda en 3 bytes:

    - 2 bytes para P (high[r31-r6] y low[r31-r6-1])
    - 1 byte para L devuelta [r31-r6-2]

  Aumenta el puntero L unidades y pone un 1 en el mapa bit

- Por último se crea un sistema de comprobación (*bitmap*) para saber si hemos acabado el byte de trabajo (usando el puntero de bit) y, si esto ha sucedido, comprobar si hemos acabado la palabra en la que se encontraba el byte (usando el puntero al mapa)

  Ante la falta de tiempo para entregar la subrutina completa, se declara cont, se escribe una línea arbitraria para que compile y se deja como incompleta. La prueba que se realiza es el caso 2 de comprime y arroja buenos resultados a la espera de terminar la subrutina.

2.3. Verifica (15/06/2022)

En esta entrega, a pesar no tener completada la subrutina Comprime, decidimos entregar una implementación de la subrutina Verifica aunque no pudiéramos probarla ya que aún no teníamos la subrutina Comprime terminada, porque creíamos que el corrector probaba Verifica con su propia subrutina Comprime, lógicamente los resultados obtenidos no fueron los esperados ya que no era como nosotros creíamos.

V3. (Final)

3.1. Entrega repetida (16-17/06/2022)

Esta fue la última de las entregas seleccionables, pero tras un despiste entregamos el mismo fichero .ens que en la anterior entrega por lo que no pudimos comprobar los cambios que obtendríamos tras modificar las subrutinas Comprime y Verifica, por lo que solamente nos quedaba la corrección final del 20 de Junio.

Finalizando Comprime (5 horas): Se retoma donde se dejó en la anterior entrega. Siguiendo los pasos descritos en el enunciado, se guarda el mapa de bit de trabajo en ese momento, si no está vacío.

Después se empieza a crear en la dirección designada como Output la cabecera del texto.

- [0] y [1] la longitud del texto (MAX: 65535)
- [2] Guardamos un 1
- [3] y [4]: low y high de la longitud del mapa de bits (en bytes)

Para guardar tanto el mapa de bits como el texto comprimido se va a utilizar el mismo sistema: Se define un inicio y un fin, y se itera utilizando un puntero a la pila que extrae el texto y un contador que desplaza r22 para escribir.

Esta parte se implementa sin más complicaciones que errores de tipeo, un mal uso de las instrucciones extu, que se sustituyen por 2 divus para desplazar los registros 16 bits a la derecha. Tras estas modificaciones, se obtiene el resultado esperado a excepción de un vacío en el mapa de bits, que hemos solucionado reservando una palabra extra en pila en *Inicializo.*
Se ha enviado un correo a los profesores preguntado si la solución es válida.
Pasa ambas pruebas.

Finalizando Verifica (6 horas): Tras haber conseguido terminar y obtener los resultados esperados en la subrutina Comprime, pudimos realizar los casos de prueba para la subrutina Verifica. Vimos varios fallos una vez probada la subrutina ya que estábamos creando las zonas de la pila de Comprime (*ZonaCom*) y Descomprime (*ZonaDes*) de manera errónea, ya que se producían desalineamientos en los accesos a memoria debido a que no habíamos ajustado por exceso al múltiplo de 4 mas cercano como se indica expresamente en el enunciado de la subrutina aportado en la documentación del proyecto.
Para solucionarlo introducimos las operaciones `and r8, r29, 0xFFFC` y `and r8, r29, 0xFFF8` (equivalente a multiplicar y dividir por el mismo número y después comprobar si el resultado es el mismo lo cual significaría que están en módulo, pero de esta manera era mucho más simple y corto) de esta manera podemos ver si la longitud de la cadena obtenida en r29 por *LongCad* está en módulo 4 y 8 para que así no se produzcan desalineamientos en los accesos a memoria y con lo cual, conseguimos obtener los resultados esperados en los casos de prueba planteados por la documentación del proyecto y por nosotros mismos.

3.2. Entrega Final (20/06/2022)

Cambio LongCad: Las llamadas a longcad en otras subrutinas provocaban una alteración del puntero de texto devuelto ya que se operaba con este (r20++) en lugar de usar el contador que contaba la longitud de programa (r2) para cargar el carácter en la posición indicada:

```
ld.b r3, r20, r2; addu r2, r2, 1
```

Cambio mulu: Al probar la subrutina *Verifica*, nos dimos cuenta que estábamos reservando mal el espacio en pila para el texto comprimido en *Comprime*: Quitamos el mulu r2,r2, 4 y reservamos directamente r2 bytes en memoria para r2 caracteres

Cambio de módulo 4 Comprime: Debatimos cual era la mejor forma de comprobar si un registro contenía un valor en módulo 4, ya que cada uno había utilizado una solución en las subrutinas de las que se encargaba. Y decidimos decantarnos por utilizar la solución de comp.

3. Juego de ensayo

- **Subrutina LongCad:**
  - **Caso 1:** Llamada a LongCad pasándole como parámetro la cadena `"Estructura de CompuHad"`. Se obtiene como resultado en r29 = 22 (0x16)
  - **Caso 2:** Llamada a LongCad pasándole como parámetro la cadena `"Tstructura de ComputadoreS"`. Se obtiene como resultado en r29 = 26 (0x1A)
  - **Caso 3:** Llamada a LongCad pasándole como parámetro la cadena `"Hola que\0"`. Se obtiene como resultado en r29 = 8 (0x8)

- **Subrutina BuscaCar:**
  - **Caso 1:** Llamada a BuscaCar pasándole como parámetros: C = 0x69, REF = `"*2345*78*0\0"`, FROM = 0 y TO = 0xA. Se obtiene como resultado en r29 = 10 (0xA)
  - **Caso 2:** Llamada a BuscaCar pasándole como parámetros: C = 0xA0, REF = 0x8250 , FROM = 0x4 y TO = 0xA. Se obtiene como resultado en r29 = 9 (0x9)

- **Subrutina CoincidenCad:**
  - **Caso 1:** Llamada a CoincidenCad pasándole como parámetros las cadenas `"Estructura de CompuHad"` y `"Tstructura de ComputadoreS"`. Se obtiene como resultado en r29 = 0 (0x0) ya que el primer carácter de ambas es diferente.

- **Subrutina Checksum:**

  Para el checksum intentamos crear un programa en java que nos permitía obtener el checksum de una cadena, pero lo máximo que conseguimos era obtener los 2 últimos dígitos de este debido a los continuos problemas de trabajar en hexadecimal y al borde del límite de tamaño del tipo int. No se llegó a completar por falta de tiempo.
  - **Caso 1:** Llamada a checksum con la cadena en ASCII:`"AAAA_BBBBCCCCDDDDEEEE_Fin.__.\0"` obtiene como resultado en r29 -> (0x)18F5DE4A

prueba_checksum.java 1

C: > Users > alvar > OneDrive - Universidad Politécnica de Madrid > UNIVERSIDAD > 2º AÑO > Estructura de Computac

```java
import java.lang.*;
public class prueba_checksum{
    static String palabra;
    //static int checksum;
    Run | Debug
    public static void main(String[] args) {
        palabra = "AAAA_BBBBCCCCDDDDEEEE_Fin.__.\0";
        short checksum=0;
        int tam=palabra.length();

        for(int i=0; i<tam; i++){ //Bucle char
            int j = i%4; char letter = palabra.charAt(i);
            if(letter==0x0||i>= palabra.length())
                break;
                double desplazamiento= Math.pow(a: 0x100,j);
                checksum+=(letter*desplazamiento);

                System.out.println(Integer.toHexString(checksum));
        }
    }
}
```

```
b25
b69
5069
5069
5069
50ae
ffffafae
ffffafae
ffffafae
ffffb01c
ffffde1c
ffffde1c
ffffde1c
ffffde4a
```

Un ejemplo del funcionamiento del programa en java

- **Caso 2:** Llamada a checksum con la cadena en hexadecimal

CASO9: data 0xF1F1F1F1, 0xF2F2F2F2, 0xF3F3F3F3, 0xF4F4F4F4
data 0x01010101, 0x02020202, 0x03030303, 0x04040404
data 0xFFFFFFFF, 0x03030303, 0xDDDDDDDD, 0xCCCCCCCC
data 0xFF00CCDD

obtiene como resultado en r29 -> (0x)8586525C

- **Caso 3:** Llamada a checksum con la cadena en hexadecimal

```
; ;CASO 8
CASO8: data 0x01010101, 0x02020202, 0x03030303, 0x04040404, 0x00
```

con resultado en r29 -> (0x)06060606

- **Subrutina BuscaMax:**
  - **Caso 1*:** Llamada a BuscaMax con la cadena `"aaaaabbbbbbbbbbbbbbbbbbbbbbbbbbb\0"`, JJ = 5500* y MAX = 10. Se obtiene como resultado en r29 = 24 (0x18)

```
88110> v 0x5500 1

    21760       05000000
```

*Este caso fue corregido en un documento distinto a la entrega pero no fue incluido en el CDV final. Se cambio el LEA(r21, JJ) por LOAD(r21, JJ) para utilizar el parámetro no la dirección de este.

  - **Caso 2:** Llamada a BuscaMax con la cadena `"tres tristes tigres comen trigo en un trigal, el primer tigre que...\0"`, JJ = 0x810D y MAX = 0xA. Se obtiene como resultado en r29 = 3 (0x3)

```
88110> v 0x8300 1

    33536       05000000
```

- **Subrutina Comprime:**
  - **Caso 1:** Llamada a Comprime con la cadena `"0123456789\0"`. Se guarda el resultado en OUT: 0x12100

Se obtiene como resultado en r29 = 16 (0x10)

```
88110> v 0x12100 4

    73984       0A000106    00003031    32333435    36373839
```

  - **Caso 2:** Llamada a Comprime con la cadena `"tres tristes tigres comen trigo en un trigal, el primer tigre que...\0"`.

Se obtiene como resultado en r29 = 70 (0x46)

```
88110> v 0x12100

    73984       4400010B    00241010    00400074    72657320
    74000       74726973    74020004    69670100    04636F6D
    74016       656E0400    04676F20    656E2075    18000661
    74032       6C2C2065    6C207072    696D6572    0C000620
    74048       7175652E    2E2E0000    00000000    00000000
```

- **Subrutina Descomprime:**
  - **Caso 1:** Llamada a Descomprime con la cadena:

```
CMPR:   data 0x07010014, 0x30000000, 0x34333231, 0x38373635
        data 0x37383939, 0x33343536, 0x00303132
```

Se obtiene como resultado en r29 = 20 (0x14)

```
88110> v 0x12100 5

    73984       30313233    34353637    38393938    37363534
    74000       33323130
```

  - **Caso 2:** Llamada a Descomprime con la cadena:

```
CMPR2:  data 0x0b010044, 0x10102400, 0x74004000, 0x20736572
        data 0x73697274, 0x04000274, 0x00016769, 0x6d6f6304
        data 0x00046e65, 0x206f6704, 0x75206e65, 0x61060018
        data 0x65202c6c, 0x7270206c, 0x72656d69, 0x2006000c
        data 0x2e657571, 0x00002e2e
```

Se obtiene como resultado en r29 = 68 (0x44)

```
88110> v 0x12100 18

    73984       74726573    20747269    73746573    20746967
    74000       72657320    636F6D65    6E207472    69676F20
    74016       656E2075    6E207472    6967616C    2C20656C
    74032       20707269    6D657220    74696772    65207175
    74048       652E2E2E    00000000
```

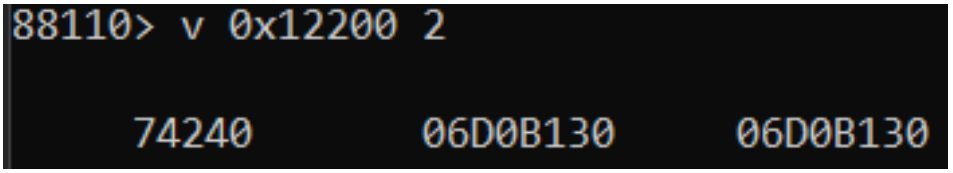
- **Subrutina Verifica:**
  - **Caso 1:** Llamada a Verifica con la cadena `"0123456789\0"`. Se obtiene como resultado en r29 = 0 (0x0) y en las direcciones de memoria:

```
88110> v 0x12200 2

    74240       9C9F686A    9C9F686A
```

  - **Caso 2:** Llamada a Verifica con la cadena `"tres tristes tigres comen trigo en un trigal, el primer tigre que...\0"`. Se

obtiene como resultado en r29 = 0 (0x0) y en las direcciones de memoria:



4. Conclusiones

Nuestro entendimiento acerca de la programación en ensamblador y, en concreto, del proyecto en sí, ha sido muy diferente al que teníamos en la evaluación ordinaria. Para esta entrega hemos desarrollado un método de trabajo eficiente:

1. Antes de ponernos a trabajar en el proyecto, dedicamos tiempo a recordar las instrucciones más específicas, especialmente las que trabajan con campo de bits, las instrucciones de desplazamiento y, para nuestra sorpresa, ya que pensábamos que dominábamos correctamente, las instrucciones lógicas y los sufijos de tamaño (.b, .h, .d) de las instrucciones de almacenamiento.

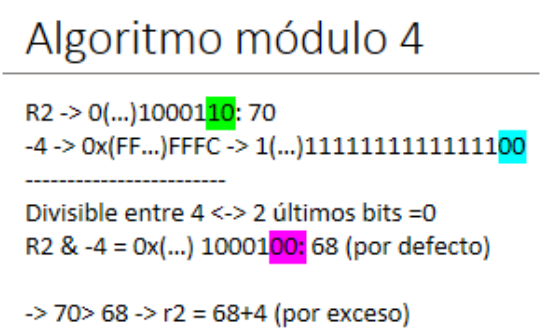
2. Después decidimos, antes de tocar el código, rediseñar las pruebas de acuerdo a nuestro "nuevo entendimiento del proyecto", para que pasasen parámetros de acuerdo a cómo lo hacía el corrector. Además empezamos a utilizar pruebas con datos en hexadecimal, separación por org para una mejor organización de los datos y evitar errores de alineamiento. Este proceso mejoró en gran medida nuestro entendimiento de la memoria y la pila en el 88110.

3. A la hora de trabajar en el código, dividir el trabajo de forma que cada uno enfocase todo su tiempo en una subrutina y no antes de terminar una jornada de trabajo poner en común los resultados, para evitar perder "la inmersión" en la subrutina en la que se está trabajando. Esto nos permitió un gran aumento de productividad y sacar adelante el proyecto en menos de 2 semanas.

El proyecto ha demandado un total de 44.5 horas, el código fuente de "CDV.ens" final, consta de 1012 LOC, siendo 695 de estas correspondientes a código y el resto a los 17 casos de prueba que hemos decido incluir.

Algunas de las dificultades que nos hemos encontrado durante el desarrollo:

- **Comprobar correctamente que un registro == 0 (mod x)**: Llegamos a una primera implementación utilizando un bucle que comparaba el resultado de multiplicar el registro y dividirlo por x y comparar si coincidían, en caso contrario, aumentar el número hasta llegar a 0(mod x). Después, descubrimos una forma de comprobarlo para x siendo potencia de 2, utilizando la operación lógica AND. En nuestro caso x solo sería 4 y 8 por lo que nos podría funcionar.



Pablo decidió utilizarla en comprime y verifica. Al ser esta mucho más fácil de implementar, más eficiente y exenta de los errores que provocaba utilizar el divu, nos decantamos por esta última.

- **Varios elementos de comprime**
    - Entender correctamente la escritura y lectura little endian, algo que obviamos completamente en la entrega ordinaria, aunque fuera solo un esqueleto, y que nos obligó a repetir todo a partir de pre_cmpr.

    - Encontrar una forma de confeccionar el mapa de bits: En la evaluación ordinaria, pasamos por varias versiones hasta decantarnos por los 3 punteros, que es con la que hemos finalizado el proyecto.

    - Como ya hemos dicho, las instrucciones de desplazamiento supusieron una gran dificultad a la hora de entender e implementar, por eso en algunos casos ante la falta de tiempo hemos optado por usar mulus y divus para desplazar bits en un registro y olvidamos cambiarlo.

Este proyecto ha mejorado notablemente nuestra comprensión del lenguaje ensamblador y, en mayor medida, de la gestión de memoria y pila del procesador.
Consideramos que es un proyecto que demanda considerablemente más esfuerzo y dedicación que el resto de los que hemos realizado hasta ahora, especialmente ante el reto de comprender elementos no vistos en clase como la gestión de un programa principal, la reserva en pila, o confeccionar casos de prueba correctamente.