**PRÁCTICA 1 DE PRINCIPIOS DE COMPUTADORES**

**Programa 1:**

* La parte del usuario dispone de 262144 direcciones. De la 1000000 a la 10040000, por lo que tendríamos (40000)16 que nos daría las 262.144 direcciones. Cada una de ellas ocupa un byte, por lo que en total 1 \* 262.144 = 262.144 bytes = 256 kilobytes.
* La palabra etiquetada como “midato1” (0xA2B1C3E0) ocupa las direcciones desde 10010024 hasta la 10010027, ambas inclusive. El esquema de ordenamiento que sigue es Little Endian, ya que al pasar del registro $t0 al $t1, vemos como el último dato que almacena es el más grande, es decir, ordenado de más pequeño a más grande.
* En primer lugar, tenemos que tener en cuenta que se almacena de manera Little Endian, es decir del más pequeño al más grande. Además, cada letra estará almacenada por 4 bytes, 2 números hexadecimales. La etiqueta “micadena” se almacena desde 10010000 hasta 10010023, ambas inclusive. También debemos de tener en cuenta que no tendremos 24 direcciones, si no 36, ya que este valor está en hexadecimal.

|  |  |  |  |
| --- | --- | --- | --- |
| **Dirección** | **Byte en hexadecimal** | **Byte en decimal** | **Carácter representado** |
| 10010000 | 64 | 100 | d |
| 10010001 | 65 | 101 | e |
| 10010002 | 52 | 82 | R |
| 10010003 | 0a | 10 | LF (Salto de línea) |
| 10010004 | 64 | 100 | d |
| 10010005 | 65 | 101 | e |
| 10010006 | 63 | 99 | c |
| 10010007 | 75 | 117 | u |
| 10010008 | 73 | 115 | s |
| 10010009 | 6e | 110 | n |
| 1001000a | 49 | 73 | I |
| 1001000b | 20 | 32 | (Espacio) |
| 1001000c | 63 | 99 | c |
| 1001000d | 75 | 117 | u |
| 1001000e | 72 | 114 | r |
| 1001000f | 74 | 116 | t |
| 10010010 | 6e | 110 | n |
| 10010011 | 6f | 111 | o |
| 10010012 | 69 | 105 | i |
| 10010013 | 74 | 116 | t |
| 10010014 | 74 | 116 | t |
| 10010015 | 65 | 101 | e |
| 10010016 | 53 | 83 | S |
| 10010017 | 20 | 32 | (Espacio) |
| 10010018 | 6d | 109 | m |
| 10010019 | 6f | 110 | n |
| 1001001a | 43 | 67 | C |
| 1001001b | 20 | 32 | (Espacio) |
| 1001001c | 65 | 101 | e |
| 1001001d | 74 | 116 | t |
| 1001001e | 75 | 117 | u |
| 1001001f | 70 | 112 | p |
| 10010020 | 00 | 0 | NULL |
| 10010021 | 00 | 0 | NULL |
| 10010022 | 0a | 10 | LF (Salto de línea) |
| 10010023 | 72 | 114 | r |

* Encontramos 79 palabras comentadas en la parte de Kernel Text Segment. Las instrucciones de la máquina MIPS se almacenan en las direcciones de 800.000.000 a 800.100.000, o sea, ocupan 10.000 direcciones. 10.000 direcciones de un byte = 10.000 bytes = 9 Kilobytes y 784 bytes.
* **Programa 2:**
* Con 32 bits en Complemento a Dos se puede representar desde el -2147483648 hasta el 2147483647, ambos inclusive, es decir, un total de 42944967296 números. He utilizado los valores -2147483647 y -30. De la primera suma (addu, enteros sin signos) he obtenido 2147483619 y la segunda (add, complemento a dos) ha dado overflow, “Exception occurred at PC=x00400038”, “Arithmethic overflow”. Lo que ocurre es que addu evita el overflow, de manera que si se supera el límite de los 32 bits (2147483647 o -2147483647) cambia el signo de los enteros, y por ello el resultado de la suma de antes da positivo y no negativo como realmente debería ser. En el caso de la suma en complemento a dos, no se ignora el signo y por ello da un error de overflow.
* Si tenemos 32 bits podemos representar hasta 4294967296 números sin signos, del 0 al 4294967295. He utilizado los valores 2147483647 y 50. En el caso de la primera suma (addu, enteros sin signos) nunca podremos causar overflow, pero si podemos sumar números que superen el 232. El resultado de esta suma ha dado -2147483599. En el caso de la segunda suma (add, complemento a dos), para que el número esté fuera del rango de la suma de enteros sin signos, dará error de overflow. Los mensajes de error han sido los mismos que en el apartado anterior: “Exception occurred at PC=x00400038”, “Arithmethic overflow”. Lo que ha ocurrido en el caso de la suma sin signo es que a 2147483647 le ha sumado 50, pero llega hasta el límite de números (2147483648), y después resta los números que quedaban, en este caso 49, y pone un menos para indicar que ha llegado al límite, si no, ignoraría el signo que tuviese delante.
* A) Falso. Almacena la dirección de la instrucción que se va a ejecutar justo después.
* B) Falso. Almacena la dirección de la instrucción que se va a ejecutar justo después.
* C) Verdadero. Siempre saltará una cantidad determinada, excepto que existan saltos, bifurcaciones, llamadas, retornos, etc.
* **Programa 3:**
* Tenemos dos números decimales (0.67 y -1.23) en flotante simple, y queremos pasarlos a binario. 0.67 en binario, con 1 bit de signo, 8 para el exponente y 23 para la mantisa sería:

0|01111110|01010111000010100011110

La salida del QtSpim no es la misma, puesto que el último número es un ‘0’ y el programa tiene un ‘1’:

0 01111110 010101110000101000111111

-1.23 en binario, con 1 bit de signo, 8 para el exponente y 23 para la mantisa sería:

1|01111111|00111010111000010100011

La salida del QtSpim tampoco es la misma, ya que no coinciden los últimos tres números:

1 011111110 0111010111000010100100

Es cierto que al calcular el número en decimal es muy parecido. El 0.67 sería 0.6699999570847 en el caso del binario calculado, y 0.6700000166893 en el caso del binario utilizado por el QtSpim. El -1.23 sería -1.2299998998642 en el caso del binario calculado, y -1.2300000190735.

* La suma de los números en precisión simple (0.67 y -1.23) da como resultado -0.56, siendo 1 01111110 00011110101110000101001, tal y como expresa el QtSpim, ya que este número es decimal sería -0.5600000023842.
* En el caso de la suma de precisión doble (1.9889230 y 1.6762158-27) da como resultado 908906998.82821381, siendo

0 00100000101 1100111111100101010001000110001110010001101010001100, tal y como expresa el QtSpim ya que este número decimal sería 908906998.82821381. Es igual al primer operando, ya que este es muy grande y el segundo operando es muy pequeño. Por ello, cuando se le suma el segundo, lo que añade es insignificante.