-¿Cuántos Kilobytes se usan para el segmento de datos del usuario?.

El segmento de datos del usuario, según muestra la interfaz de QT-Spim, comprende las direcciones de la [10000000] (incluida) a la [10040000] (sin incluir) expresadas en hexadecimal. Por tanto, contiene 40.000 (hexadecimal) bytes, esto es 262144bytes, es decir 256KB.

Si la interpretación a la pregunta hecha por el compañero fuera cuántos bytes se están realmente usando dentro de ese segmento, y su respuesta fuera 34 bytes de la cadena "mi cadena" y los 4 de "midato1", es decir 38/1024KB, ante la duda de la interpretación de la pregunta, se le debe dar por válido.

También, si se cuentan los bytes desde la primera dirección de la cadena hasta la última de midato1 se ve que hay 40 bytes (hay dos nulos después de la cadena), por lo que debe darse también por válida esta opción (1 0010027-10010000+1=28 direcciones en hexadecimal = 40 bytes en decimal).

-¿Cuál es la dirección de cada uno de los bytes de la palabra etiquetada como "midato1"?.

Byte	Contenido	Dirección (HEX)
0(ms)	E0	10010024
1	C3	10010025
2	B1	10010026
3(MS)	A2	10010027

- ¿Qué tipo de esquema de ordenamiento se sigue, Big Endian o Littlle Endian?. Razona tu respuesta. Ayuda: ejecuta paso a paso el programa en el emulador y fíjate lo que ocurre cuando transfieres el byte sin signo situado en la posición midato1 al registro $t1 (tras ejecutar la instrucción "lbu $t1,($t0)").

El byte menos significativo se almacena en la dirección más baja, por lo que la máquina es little-endian

-¿Cuál es el código ASCII de los caracteres que componen la cadena etiquetada como "micadena"?. Escribe una tabla indicando la dirección de cada byte, su valor en decimal y hexadecimal y el carácter que presenta.

Direccion (HEX)	Valor Dec	Valor Hex	Caracter
10010000	10	0a	\n
10010001	82	52	R
10010002	101	65	e
10010003	100	64	d
10010004	117	75	u

10010005	99	63	c
10010006	101	65	e
10010007	100	64	d
10010008	32	20	' '
10010009	73	49	I
1001000A	110	6e	n
1001000B	115	73	s
1001000C	116	74	t
1001000D	114	72	r
1001000E	117	75	u
1001000F	99	63	c
10010010	116	74	t
10010011	105	69	i
10010012	111	6f	o
10010013	110	6e	n
10010014	32	20	' '
10010015	83	53	S
10010016	101	65	e
10010017	116	74	t
10010018	32	20	' '
10010019	67	43	C
1001001A	111	6f	o
1001001B	109	6d	m
1001001C	112	70	p
1001001D	117	75	u
1001001E	116	74	t

1001001F | 101 | 65 | e

10010020 | 114 | 72 | r

10010021 | 10 | 0a | \n

-¿En cuantas palabras se traducen las instrucciones en ensamblador que componen el programa p1_1.s?

Las palabras del programa se cargan comenzando cada una de ellas en las direcciones de la [00400000] a la [0040004c]. El último byte de la última instrucción terminará en la dirección [0040004f]. Por tanto ocupa 50 (hex) bytes, es decir, 80 bytes, y por tanto 20 palabras.

Debe admitirse como válido también el cómputo desde la dirección [00400024] a la [0040004c] 11 palabras (si se entendió que el código puesto por el loader no es parte del programa p1_1.s

Ejercicio 2:
- Antes de ejecutar la primera suma del programa, modifica los registros $t1 y $t2 para que contengan valores tales que al realizar la suma bajo la interpretación de complemento a 2 den lugar a un resultado fuera de rango, mientras que si se suman bajo la interpretación en enteros sin signo, el resultado esté en el rango. Recuerda que el tamaño de todos los registros es 32 bits. Ejecuta paso a paso el programa y observa lo que ocurre. Responde a este ejercicio con: el rango de los números en complemento a 2 para una celda de 32 bits, los valores que has utilizado, el resultado obtenido de las sumas, la descripción de lo que ocurre y tu interpretación tratando de dar una explicación a todo lo que has observado.

En 32 bits el rango de números en complemento a 2 es: [-2^31 .. 2^31-1]
En 32 bits el rango de enteros sin signo es: [0 .. 2^32-1]

Dos números cuya suma estará fuera de rango en complemento a 2, y en rango interpretando el resultado como entero sin signo serán, por ejemplo: 0x7FFFFFFF y 0x00000001

La operación realizada internamente por el simulador es la misma para los registros.

Con addu, el simulador no detecta desbordamiento, siendo el usuario el que debe realizar la comprobación de que el resultado sea correcto.

Sin embargo, al realizar un add, el simulador detecta el overflow en el rango de los enteros en complemento a 2.

- Antes de ejecutar la primera suma del programa, modifica los registros $t1 y $t2 para que contengan valores tales que al realizar la suma bajo la interpretación de enteros sin signo den lugar a un resultado fuera del rango. Recuerda que el tamaño de todos los registros es 32 bits. Ejecuta paso a paso el programa y observa lo que ocurre. Responde a este ejercicio con: el rango de los números enteros sin signo para una celda de 32 bits, los valores que has utilizado, el resultado obtenido de las sumas, la descripción de lo que ocurre y tu interpretación, tratando de dar una explicación a todo lo que has observado.

Los números utilizados deberán tener ambos el primer bit a 1, por lo que se tendrán que introducir en el simulador como números negativos para que permita fijar esos valores.

Los números a utilizar serán, por ejemplo, 0xffffffff (-1) y 0x00000007.

La instrucción addu hace una operación que provoca un desbordamiento, pero sin embargo no se produce una interrupción por desbordamiento en el simulador.

La instrucción add, al hacer la interpretación en complemento a 2, no provoca un desbordamiento al interpretar la operación como -1 + 7 = 6.

- Vuelve a realizar la ejecución conforme a las condiciones del apartado anterior, pero esta vez fíjate en los valores que va tomando el registro Program Counter (PC). Responde verdadero o falso a las siguientes afirmaciones, y razona la respuesta:
a) El registro PC almacena el código de la instrucción que se acaba de ejecutar.

Falso. Almacena la dirección de la siguiente instrucción a ejecutar.

b) El registro PC almacena la dirección de la instrucción que se acaba de ejecutar.

Falso. Almacena la dirección de la siguiente instrucción a ejecutar.

c) El registro PC siempre varía la misma cantidad tras la ejecución de una instrucción.

Verdadero (Para el código de ejemplo), siempre se incrementa en el tamaño de la palabra de instrucción que es 4. Si en el código hubiera instrucciones de salto esto no sería así.

3. El programa p1_3.s, contiene en el segmento de datos dos flotantes en formato simple (myfl1 y myfl2) y dos flotantes en formato extendido (myfl3 y myfl4). Debes resolver los siguientes ejercicios:
–
Utiliza el emulador para comprobar la representación interna de cada uno de los flotantes. Tienes la dirección de cada uno de ellos en los comentarios del programa. Demuestra que la representación se realiza conforme al estándar IEEE 754. Nota: La representación en los registros es en decimal, y es posible que no esté funcionando bien en esta versión de QTSpim, por eso céntrate sólo en los resultados que puedes ver en el área de memoria para esta pregunta. Debes incluir en el informe la representación binaria del número extraída del emulador y los cálculos que te permiten concluir que es correcta.
–
---
myfl1:
Representación Binaria: 0_01111110_01010111000010100011111
Signo: 0 → Positivo
Exponente: 01111110 = 126 → 126 – 127 = -1
Mantisa: 01010111000010100011111 →
1.01010111000010100011111 (binario) = 11.240.735 * 2^-23
Número = + 1.01010111000010100011111 (bin) * 2^-1 = 11240735 * 2^-23 * 2^-1 = 0.670000016689
---
myfl2:
Representación Binaria: 1_01111111_00111010111000010100100
Signo: 1 → Negativo
Exponente: 01111111 = 127 → 127 – 127 = 0
Mantisa: 00111010111000010100100 →
1.00111010111000010100100 (binario) = 10.317.988 * 2 ^-23
Número = - 1.00111010111000010100100 (bin) * 2^-23 * 2^0 = -1.23000001907

---
myfl3:
Repr. Binaria: 0_10001100011_10010001101010001100 00010000010111001111111001010100
Signo: 0 → Positivo
Exponente: 10001100011 = 1123 – 1023 = 100
Mantisa: 10010001101010001100 00010000010111001111111001010100 →
1.10010001101010001100000100000101110011111111001010100 = 1.56898123161
Número = + 1.56898123161 * 2^100 = 1,98892e30
---
myfl4:
Repr Binaria: 0_01110100110_00001001100110110110 01000001000010101100100011001000
Signo: 0 → Positivo
Exponente: 01110100110 = 934 – 1023 = -89
Mantisa: 00001001100110110110 01000001000010101100100011001000 →
1.00001001100110110110010000010000101011001000110010000 = 1.03752732665
Número = + 1.03752732665 * 2^-89 = 1.6762158e-27

Observa el resultado de la suma de los dos flotantes de precisión doble. La suma en precisión simple se almacena en res1 y la suma de los operandos en precisión doble se almacena en res2. La instrucción que transfiere el resultado de la la primera suma a memoria es:
swc1 $f0, res1
y la instrucción que transfiere el resultado de la segunda suma a memoria es:
sdc1 $f0,res2
Comprueba revisando el valor en memoria que res1 es correcto. En el caso de precisión doble, ¿por qué el resultado de la suma es igual al de uno de los operandos a pesar de que ninguno de ellos es 0?

res1:
Repr. Binaria: 1_01111110_00011110101110000101001
Signo: 1 → Negativo
Exponente: 01111110 = 126 → 126 – 127 = -1
Mantisa: 00011110101110000101001 →
1.00011110101110000101001 (binario) = 11.240.735 * 2^-23
Número = + 1.01010111000010100011111 (bin) * 2^-1 = 1.12000000477 * 2^-1 =
0.560000002384
---
res2:
Repr. Binaria: 0_10001100011_10010001101010001100 00010000010111001111111001010100
Signo: 0 → Positivo
Exponente: 10001100011 = 1123 – 1023 = 100
Mantisa: 10010001101010001100 00010000010111001111111001010100 →
1.10010001101010001100000100000101110011111111001010100 = 1.56898123161
Número = + 1.56898123161 * 2^100 = 1,98892e30

Los dos números a sumar en doble precisión tienen exponentes muy distintos. El primero tiene exponente +30 y el segundo -27. El resultado de la suma tendrá también exponente 30, sin embargo la mantisa no tendrá suficiente resolución para albergar los cambios debidos a que el segundo sumando es 57 órdenes de magnitud más pequeño. Por ello, la representación en doble precisión del resultado es igual al primer sumando.