Si se considera la instrucción:

**subi $8, $8, 0x2**

1. Investiga y averigua cuántos bits son necesarios para representar la instrucción en código máquina.   
    Para cualquier instrucción en MIPS en código maquina se requieren 32 bits.
2. Traduce la instrucción a código máquina. ¿Una instrucción es suficiente? Proporciona tus conclusiones y respuesta en formato hexadecimal y binario.

Si es posible con una sola instrucción, la instrucción “addi” puede realizar una operación de una suma de números negativos en complementos a dos. Dentro del mapa de bits la sección de la constante los 16 bits deben tener el siguiente valor 0xFFFE para tener un -2 en complementos a dos.

**addi $8, $8, 0XFE** - 001000000000100111111111111111110 b = 0x2008FFFE h

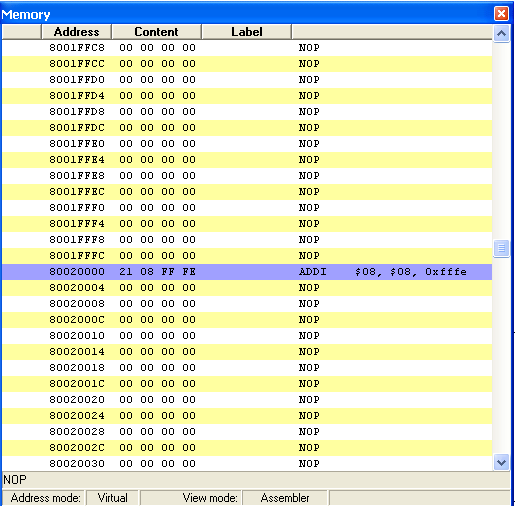
|  |  |  |  |
| --- | --- | --- | --- |
| Op | Rs | Rt | Immediate |
| addi | $8 | $8 | -0x02 |
| 0x08 | 0x08 | 0x08 | 0xFE |
| 00 1000 | 0 1000 | 0 1000 | 1111 1111 1111 1110 |

Referencias:

<https://opencores.org/projects/plasma/opcodes>

<https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MIPS_Architecture_microMIPS64_InstructionSet_AFP_P_MD00594_06.04.pdf>

1. Ya que hayas averiguado cómo se representa la instrucción escrita en hexadecimal, en la ventana ‘Memory’ del módulo RAM de nuestro simulador Mips.exe escribe ese número que obtuviste en la dirección de memoria 0x80020000 y establece el Program Counter en 0x80020000 (en la ventana de los registros del módulo de CPU).



* 1. ¿Qué es lo que en realidad se almacena en memoria? Anota las distintas representaciones que toma el contenido de memoria 0x80020000 (ASCII, integer, floating point).

Lo que se guarda en memoria es la codificación de la intraccion en un lenguaje que la computadora pueda entender.

* ASCII. **!...**
* Integer. **554237950**
* Floating Point. **4.64174e-019**

Lo que se almacena en memoria son instrucciones, cada campo de la instrucción conlleva a ir a un registro donde dependiendo del campo de operación, el tamaño de los demás registros varía para completar la instrucción de manera adecuada.

* 1. ¿Cómo se interpreta cada representación?

Cada representación se interpreta como una dirección de registro o de operación.

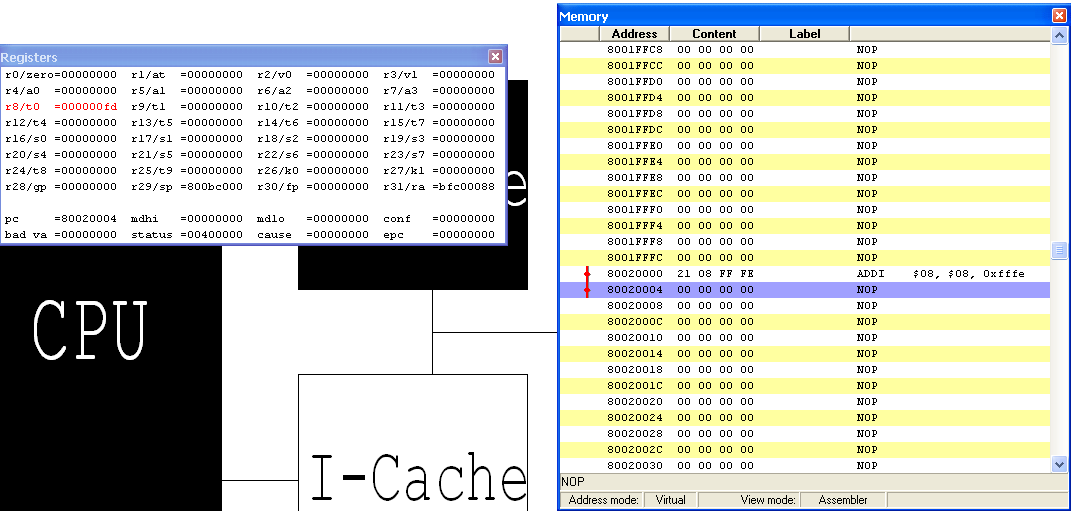
* 1. ¿Cuántos bits observas?

32 bits.

* 1. ¿Cómo puede la computadora saber que el patrón de bits almacenado es una instrucción?

Por el primer campo de operación (los primeros 6 bits), este campo puede indicar desde el formato de la instrucción y el tipo de operación a realizar.

1. Ahora, resetea el CPU en el menú homónimo del Mips.exe Cpu => reset . Escribe en el registro $08 (en la ventana de CPU) el valor de 0xFF y en la localidad de memoria 0x80020000 de la ventana de Memory vuelve a colocar las instrucciones del principio. Ejecuta paso a paso las instrucciones con el botón.



* 1. ¿Cuál es el nuevo valor del “program counter”? ¿Por qué?

En un inicio 0x80020000, al finalizar 0x80020004

Porque cada palabra es de 32 bits, si el largo de la memoria es de 4 bytes, para avanzar 1 palabra del cpu requiere leer 4 palabras de la memoria.

* 1. ¿Cuál es el nuevo valor del registro $8? ¿Por qué?

**0xfd,** ya que el valor con el cual inicializamos el registro es -1 en complemento a dos y la suma del numero negativo -2 en el mismo formato nos da como resultado -3, cuya conversión es 0xFD.

1. Dado el siguiente código de ensamblador:

#include <iregdef.h>

.set noreorder

.text

.globl start

.ent start

start: jal wait # Wait for button click

nop

lui s0, 0xbf90 #Load switch port address

lb s1, 0x0(s0) #read first number from switches

nop

jal wait #Wait for button click

nop

lb s2, 0x0(s0) #Read second number from switches

nop

addu s3, s1, s2 #Perform an arithmetic operation

sb s3, 0x0(s0) #Wirte the result to LEDs

b start #Repeat all over again

nop

###Add code for wait subroutine here! ###

wait:

lui t0, 0xbfa0

lb t1, 0x0(t0)

nop

andi t1, t1, 0x1

nop

beq $0, t1, wait

nop

wait2:

lui t0, 0xbfa0

lb t1, 0x0(t0)

nop

andi t1, t1, 0x1

nop

bne $0, t1, wait2

nop

jr ra

.end start

* 1. ¿Qué hace el programa?
     1. Lee los valores que representan en binario (representación sin signo) la combinación de voltajes altos y bajos de los switches.
     2. Lee los valores que representan en binario (representación sin signo) la combinación de voltajes altos y bajos de los switches.
     3. Para cada iteración espera una interrupción.
     4. Suma los valores obtenidos.
     5. Muestra el resultado en los leds.
  2. ¿Para qué funciona la librería incluida?

Para no tener que escribir el símbolo $, esto puede funcionar para hacerlo mas amigable para el programador y codificar más rápido.

* 1. ¿Qué notas sobre el puerto de I/O?

Esta conectada a una interrupción del puerto del CPU I/O el cual afecta la ejecución del código.

De igual manera, simula el programa previo y utiliza los switches y LEDs del simulador (ventana de I/O) para investigar la suma entera definida por la instrucción addu (add unsigned). Esta operación suma enteros positivos. Prueba con las sumas 1+2, 0x0F + 2 y algunas otras que creas apropiadas para entender mejor la instrucción. Trata de cargar el valor de 0xFF con los switches usando la instrucción ‘lb’.

* 1. ¿Qué sucede con los datos en el registro? ¿Por qué?

Cambian dependiendo de los estados de los switches, porque están ligados a puertos I/O del CPU y al almacenar lo que captura de esos puertos, cambia dependiendo de los estados de los switches.

* 1. ¿Cuándo aparecen resultados especiales o inusuales en la suma?

Cuando hay un desbordamiento (el resultado de la suma es más grande que el tamaño de memoria donde se va a almacenar)