6 de Octubre de 2020

MIPS IT Simulador – Programación

Laboratorio de arquitectura de Computadoras

ITESM Campus Monterrey

Profesor
Diego Fernando Valencia Martínez

Miguel Morales de la Vega
Javier Mondragón Martín del Campo

A00821541
A01365137

Introducción

Durante el desarrollo de la práctica observaremos las diferentes etapas por las cuales pasa un programa en ensamblador hasta llegar a su ejecución por el equipo de computo. La herramienta principal a utilizar será MIPS Simulator. La cual nos permite tener una visión abstracta de los componentes principales de una arquitectura MIPS, estos componentes son CPU, Memoria, Caché, Consola y I/O.
Al momento de ejecutar un código de ensamblador podremos observar como cada instrucción es procesada por la el CPU, convertida a lenguaje maquinal y guardada en los registros internos de CPU para su ejecución. La herramienta nos habilita la configuración sin necesidad de código en ensamblador de los registros internos tanto del CPU como los de Memoria para realizar las simulación y operaciones deseadas,

Procedimiento

Durante esta sección vemos como es usado la herramienta de simulación para el código mostrado en la figura 2.1. El código realiza una carga de valores a los registros del CPU y después realiza la operación de suma de estos valores.

```
#Lab Exercise 1
.set noreorder        #Avoid reordering instructions
.text                 #Start generating instructions
.global start         #The label should be globally known
.ent start            #The label marks an entry point
start: li $8, 0x1     #Load the value 1
li $9, 0x2            #Load the value 2
add $10, $8, $9       #sum the values
.end start            #Marks the end of program
```

Figura 2.1
Código ejemplo de ensamblador

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 máquina se requieren 32 bits.

2. Traduce la instrucción a código máquina. ¿Una instrucción es suficiente? Proporciona tus conclusiones y respuestas 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 - 00100000000010011111111111111110 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

3. 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).
    a. ¿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 interacción 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.

    b. ¿Cómo se interpreta cada representación?

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

    c. ¿Cuántos bits observas?

        32 bits.

d. ¿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.

4. 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.
   a. ¿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.

   b. ¿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 número negativo -2 en el mismo formato nos da como resultado -3, cuya conversión es 0xFD.

5. 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) #Write 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
beq $0, t1, wait
nop
wait2:
lui t0, 0xbfa0
lb t1, 0x0(t0)
nop
andi t1, t1, 0x1
bne $0, t1, wait2
nop
jr ra
.end start
```

Justificacion de NOP:

1. lb: Cada vez que se requiere leer de la memoria es requerido una espera, en este caso rs (o t0) va a acceder a una dirección de memoria, la instrucción tomará un ciclo de reloj para escribir la dirección deseada y otro para leerla.
2. beq/bne: Esta instrucción requiere dos ciclos de reloj, el primero corresponde a la comparación de dos registros y el segundo ciclo corresponde al salto a la instrucción deseada.

    a. ¿Qué hace el programa?
        i. Lee los valores que representan en binario (representación sin signo) la combinación de voltajes altos y bajos de los switches.
        ii. Lee los valores que representan en binario (representación sin signo) la combinación de voltajes altos y bajos de los switches.
        iii. Para cada iteración de captura de datos espera una interrupción de botón.
        iv. Suma los valores obtenidos.
        v. Muestra el resultado en los leds.

    b. ¿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.

c. ¿Qué notas sobre el puerto de I/O?

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

d. ¿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.

e. ¿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)

Resultados

Los resultados de la simulación son la carga de los valores 1 y 2 a los registros 8 y 9 del CPU, en la figura 3.1 podemos observar cómo estos registros tienen el valor correspondiente y también como el registro PC (Program Counter) obtiene valores correspondiente a la dirección de memoria donde reside la instrucción (Figura 3.1).

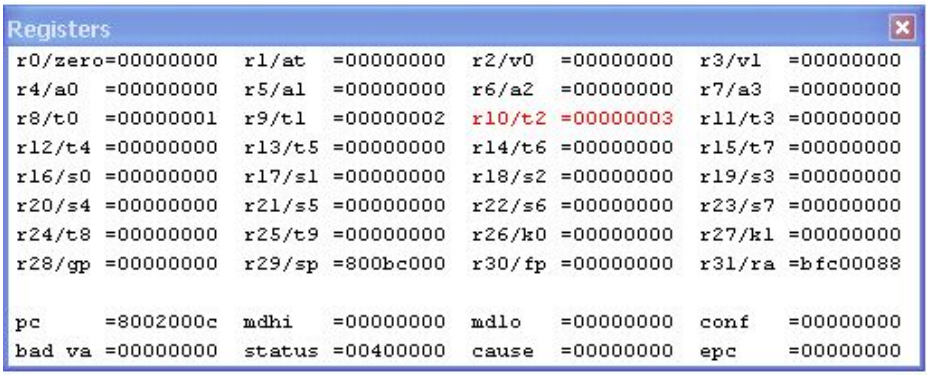

Figura 3.1
Ventana de registros del CPU

En la figura 3.2 observamos la memoria interna de la arquitectura MIPS en esta se codifica cada una de las instrucciones que el procesador va a realizar, para el ejemplo de la carga y suma de registros se observan 3 instrucciones cada una con su correspondiente codificación.

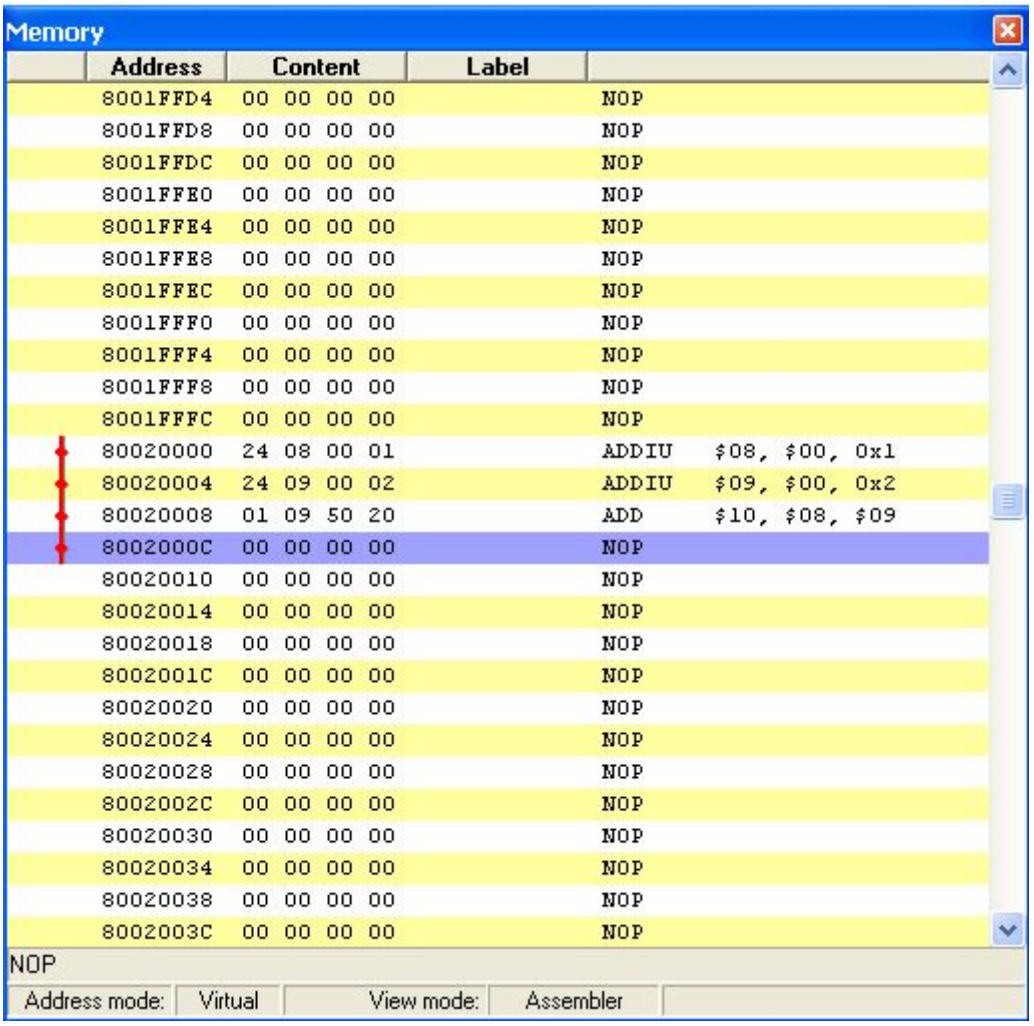

Figura 3.2
Instrucciones del programa en memoria

Conclusión

Durante esta práctica diseñamos y codificamos para la arquitectura MIPS32, realizando código en lenguaje máquina y ensamblador. Con esta práctica apreciamos el funcionamiento interno de un procesador y aclarar cómo realiza la lógica para hacer las instrucciones. Esta arquitectura se basa en los conceptos generales de una arquitectura de computadora y al comprender esto nos da las bases para poder interpretar y entender procesadores de otras arquitecturas, así como, ARM, x86, x64, entre otras.

Para la realización del código ensamblador, tuvimos diferentes soluciones para realizar un problema, una más eficiente que otra, con esto podemos concluir que podemos realizar los mismos problemas con distintas instrucciones y a pesar que en el procesador se van a ejecutar de manera distinta, el resultado será el mismo.

Reflexion Individual

Miguel Morales:

Durante la práctica logre recapitular los conceptos de programación en ensamblador e investigar las instrucciones específicas para la arquitectura MIPS. Al momento de realizar el último ejercicio con el uso de interrupciones pude identificar los conceptos generales de lectura de un push button o como se conoce en la práctica “polling”. Esta práctica aunque tuvo un nivel muy bajo de dificultad, el programar en ensamblador siempre es un reto, ya que requiere pensar como una computadora.

Javier Mondragon:

Durante esta práctica tuvimos que documentar, realizar investigaciones y codificar para entender de una manera más profunda como una computadora trabaja y cómo se ejecutan las tareas en un procesador. Cada procesador es distinto pero en la arquitectura MIPS, ayuda a realizar las instrucciones con un ISA reducido y esto nos puede ayudar a hacerlo más compacto, entre otras cosas pero esto nos perjudica en el número de instrucciones que se deben de generar y ejecutar por programa. Para la realización de el push button tuvimos que trabajar en equipo, a pesar que tenía baja dificultad, nuestra experiencia en código ensamblador es poca pero esto nos muestra que para dominar esta área se requieren más horas de estudio y práctica.

Referencias

- https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MIPS_Architecture_microMIPS64_InstructionSet_AFP_P_MD00594_06.04.pdf
- https://opencores.org/projects/plasma/opcodes
- https://syssec.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/system-security-group-dam/education/Digitaltechnik_14/21_Architecture_MultiCycle.pdf