TEC | Tecnológico de Costa Rica

Instituto Tecnológico de Costa Rica

Sede San Carlos

Escuela de Ingeniería en Electrónica

Emulador programado en Ensamblador x86_64 para sistemas operativos de la familia Linux

EL-4313 Lab. Estructura de Microprocesadores

Profesor:Ing. Ernesto Rivera Alvarado

Estudiantes:
Randall Durán Solano 2013027110
Francisco Elizondo Rodriguez 2013097310
Freddy Salazar Acosta 2013116449
José Eduardo Zúñiga Ramírez 2013099951

23 de Marzo de 2017, Semestre I

Índice

1. Descripción del diseño de software realizado en el proyecto

El simulador MIPS creado corre en sistema operativo Linux de 64 bits. El Software lee solamente un archivo de texto llamado ROM.txt, éste archivo tiene que estar en la misma ubicación en donde está el archivo ejecutable del código. Para ejecutar el programa se recomienda hacerlo desde la consola de linux mediante el comando ./simulador [arg0] [arg1] [arg2] [arg3], donde [argx] corresponde a los argumentos con los que se inicializan los valores de $a0 $a1 $a2 $a3 correspondientemente, si no se indican los valores de estos argumentos, se inicializarán en cero. El programa no reconocerá el excedente de argumentos, en este caso se verá como un error y se terminará la ejecución de programa.
Se recomienda evitar el uso de argumentos negativos, debido a que el programa interpretará un número diferente al deseado.

En el archivo ROM.txt se debe tener ciertos cuidados con las instrucciones, esto debido a que el programa cuando detecte un error en ellas avisará al usuario y se saldrá de la ejecución. Casos en el que el programa terminará la ejecución es tener un opcode inválido (que no esté dentro de las instrucciones MIPS), que presente un function incorrecto, tratar de operar con resgistros reservados o dar direcciones de memoria inválidos, tanto para memoria de datos como para stack.

El sistema fue diseñado de tal manera que el acceso a memoria de datos se de con números desde 0 hasta 400, donde cada espacio de memoria va de word en word (4 bytes en 4 bytes), si se trata de acceder a memoria con algún valor base más offset que esté fuera de este rango, el sistema le notificará y terminará su ejecución. Para el caso del stack, este debe editarse como la arquitectura lo indica para poder cargar y guardar datos en este, si se desea guardar o cargar un dato que está en una posición donde el Stack no está .ªbierto", no se permitirá la acción.

El programa cuando lee el documento compara los valores ASCII que tiene escrito cada línea, si es un 1, se hace un giro (los espacios en memoria de las instrucciones son de 64 bits, donde los primeros 32 bits son 1 y los restantes son 0) para intercambiar la posición de 0 en los 32 bits menos significativos por un 1, si detecta un 0 en ASCII de la instrucción, solo se hará un corrimiento para seguir leyendo el documento. Constantemente se compara con el caracter nulo para saber si la instrucción en el documento terminó para pasar a la siguiente. Al tener las instrucciones listas, se procede a hacer la lectura de estas donde se exxtrae el opcode y si este es cero se comparará el function (debido a que se asegura que es una instrucción tipo R), además se extraen los valores como el shamt, y los registros rt, rd y rs. De acuerdo al fuction que tengan, se direccionará a la rutina que ejecuta la instrucción deseada. Para el caso del opcode ser diferente de cero, se comparará el function de las demás instrucciones para poder determinar la rutina que se debe ejecutar, obteniendo a su ves los registros y el inmediate que se van a utilizar.

Luego de la ejecución de las rutinas de operación, se hace un PC+4, (o el address que corresponda según la instrucción que se ejecutó), esto para pasar a la siguiente instrucción. Cuando se terminan las instrucciones el sistema pasa a una rutina donde se escribe en pantalla el éxito de ejecución y se da la finalización del mismo.

2. ¿Cuáles fueron los principales retos a resolver y cómo se resolvieron?

El lw estaba agarrando un valor de 64 bits, cuando se creía que estaba asignado de 32 bits por lo que los otros 32 bits se estaban llenando con "basura". Para corregir esto, se debugueó los valores que tenían los registros y había uno que no concordaba. Revisando el código se notó un error en el uso de los registros donde se cargaba un valor de 32 bits en memoria a un registros de 64 bits por tanto se agregó una D después del número de registro (Ejemplo r10D) haciendo coincidir el número almacenado en la memoria con el registro.
Para obtener la información del CPU se encontró el inconveniente de que el código que se realizó sólo se podía compilar con gcc. Al momento de unirlo a la otra parte del código, ésto no se logra

acoplar porque se compila diferente. Pare corregir el problema, se optó por buscar otra manera de obtener la información del CPU. Investigando se llegó a otra solución para lograr compilar de la misma forma que el resto del código.
Para guardar en el archivo de texto los resultados e imprimir los números, se generaba un problema dependiendo de los registros que se utilizaban, no se guarda el valor correcto contenido en ese registro entonces se utilizó el registro r10 para guardar los valores a imprimir.
Dependiendo del orden de la declaración del .data se daba el caso de error, entonces primero se declara el texto y después las instrucciones.
El valor contenido dentro puntero no brinda el ACSII del argumento, se tiene que mover el valor contenido dentro del puntero a un registgro de 8 bits para obener el byte que contiene el código ASCII.
Para un argumento de más de 3 digitos, obtener el valor del dígito correspondiete al millar se debe mover el ACSII a un registro de 64 bits, ya que si se multiplica en uno de 8 bits, éste excede el tamaño del registro.

3. ¿Cuáles mejoras se sugieren para el programa y cómo se harían?

Se podría mejorar que los argumentos decodifiquen valores hexadecimales como letras y que no permita el ingreso de caracteres que no está dentro del grupo hexadecimal.
El uso de argumentos negativos hace que el programa interprete un número diferente al deseado. Esto se podría arreglar utilizando complemento a 2 en los argumentos.

4. ¿Cuáles son las conclusiones a las que llega el grupo de trabajo?

Con el software diseñado, se logró simular las instrucciones MIPS requeridas para la finalidad del proyecto.
A partir del programa creado, se desarrollaron las habilidades para el manejo de los recursos del hardware con la implementación del sorftware.

5. ¿Qué herramientas se usaron para completar el proyecto?

Las herramientas que se utilizaron fueron: computadoras con Sistema Operativo Linux de 64 bits con nasm y gdb instalados. Además se utilizaron libros y páginas de internet tales como gitHub para hacer el repertorio y tener los archivos guardados, también para que todos los integrantes del grupo tengamos los archivos actualizados cada vez que se trabaja en un código y se guarda. Otra página relevante fue Stackoverflow, de ahí se obtuvo muchos ejemplos de funciones de ensamblador x86_64.
Hoja de instrucciones MIPS.

6. Referencias bibliográficas

Duntemann, Jeff. Assembly Language Step by Step Programming with Linux. 2009. Third Edition.

Orenga, Miquel y Manonellas, Gerard. Programación en ensamblador (x86-64).

Easy x86-64 A tutorial on programming with the x86-64 in Assembly. http://ian.seyler.me/easy_x86-64/.

Patterson, David. Hennessy, John. Computer Organization and Design, The Hardware/Software Interface. 2012. Fourth Edition.

Stackoverflow. http://stackoverflow.com/questions.