30/11/2017

Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Organización del Computador (CI-3815)
Trimestre Septiembre-Diciembre 2017

Reporte del Proyecto 2: Instrumentador y planificador

Daniel Francis - Carnet: 12-10863
Javier Vivas - Carnet: 12-11067

Tabla de contenidos:

1. Introducción

En este proyecto, se pide implementar dos procesos complementarios para la arquitectura de procesador MIPS.

El primero es un instrumentador, el cual debe analizar las instrucciones de un programa de entrada y realizar los siguientes cambios:

- Si hay un `add`, debe insertar un `break 0x20` debajo del mismo

Al insertar el `break 0x20`, todas las instrucciones debajo del `add` deben ser trasladadas 4 bytes. Si hay un `beq` cuyo inmediato apunte a una dirección que está por encima del `add`, es necesario corregir el inmediato para que apunte a la instrucción correcta.

- Si hay un `syscall 10`, debe reemplazarlo por un `break 0x10`.

El segundo es un planificador de tiempo exclusivo (y compartido en el caso de la actividad adicional). El planificador requiere de un manejador de excepciones, en este caso una versión modificada del manejador para SPIM S20 MIPS.

El planificador requiere que el manejador de excepciones reconozca las siguientes interrupciones:

- Cuando el usuario presiona la tecla '`s`'. En este caso, el planificador pasará al siguiente programa.
- Cuando el usuario presiona la tecla '`p`'. En este caso, el planificador pasará al programa anterior.
- Cuando el usuario presiona la tecla '`esc`'. En este caso, el planificador cesará la ejecución e imprimirá el estado de finalización de cada programa, junto al número de `add` que contiene cada uno.
- Cuando se ejecuta un `break 0x10`. Esta instrucción finaliza el programa actual.

A continuación describimos el proceso a través del cual diseñamos e implementamos las funciones solicitadas, junto a las estructuras de datos que fueron necesarias.

2. Desarrollo y diseño

2.0.1 main - *Descripción:*

El proceso principal del proyecto o main se encarga de preparar las estructuras necesarias para ejecutar e instrumentar cada programa y luego ejecutarlos mientras se permite la intervención del manejador de excepciones.

2.0.2 Estructura de datos:

El main itera sobre la dirección PROGS y utiliza a NUM_PROGS para detener la instrumentación y pasar al planificador. PROGS lo manejamos como un arreglo cuyos elementos se encuentran a 4 bytes de separación.

La estructura más importante del proyecto es la de la **página de registros**. Funciona como un **arreglo** con secciones de tamaño 136 bytes, cada una con subsecciones de tamaño palabra de 4 bytes. RegActual mantiene la dirección de la página actual. Como no se sabe cuántos programas entrarán al planificador/instrumentador (proceso dinámico), hay que solicitar memoria de heap. Todas las páginas de registro se ubican en memoria de heap.

Cada subsección representa un registro o dato sobre el programa representado por la página. Cada programa tiene una página única. A continuación, la estructura de cada página. Cada dato se guarda en un espacio de 4 bytes:

- Dirección del programa (0)
- Número del programa (4)
- Program counter (8)
- Inutilizado (12)
- Número de adds (16)
- at (20)
- v0-1 (24-28)
- a0-3(32-44)
- t0-9(48-80)
- s0-7(84-112)
- gp (116)
- sp (120)

- fp (124)
- ra(128)
- Estatus de finalización (132)

En total, contabilizan 34 palabras = 34 * 4 bytes = 136 bytes en página por programa.

2.0.3 Planificación de registros:

- t0: Apuntador general para saber en qué programa se va a trabajar. También usado para prohibir interrupciones.
- t1: Iterador principal en lista de programas. Auxiliar a $t0, cargando datos y contadores para comparar con $t0.
- t2: Contador del ciclo principal.
- s0: Toma direcciones de PROGS.
- t3: Recupera datos apuntados por otros registros.
- s3: El número de programa sobre el cual se trabaja actualmente.

2.0.4 Diseño e implementación:

Después de inicializar las estructuras, los datos y la memoria en heap, comenzamos a guardar todo en su lugar respectivo sobre la página del programa actual. Utilizando contadores y comparando con NUM_PROGS, vemos cuándo es apropiado finalizar la rutina y pasar al instrumentador. El instrumentador entonces se llama para cada programa a través de un ciclo.

2.1.1 Instrumentador - Descripción**:**

Tomando las instrucciones de un programa de entrada en a0, realizamos las instrucciones previamente descritas en el caso de que haya un add, break, o syscall 10.

2.1.2 Estructura de datos:

Se actualiza y se hace uso de la estructura de página de registros descrita en la sección **main.**

2.1.3 Planificación de registros:

- t1: Iterador rincipal en la rutina principal.
- t2: Código de operación de la instrucción actual.
- t3: Resultado de enmascarar el código.
- t4: Iterador secundario.
- t5: Inmediatos de los beq. Código de break a insertar.
- s0: Es igual a 0x0a si encontré un li $v0 10

2.1.4 Diseño e implementación:

El instrumentador utiliza dos ciclos. El principal analiza cada instrucción en búsqueda de un add o de un syscall 10. Si no encuentra ninguno, no pasa nada.

En caso de que encuentre un add, procede al segundo ciclo o subrutina. El programa entonces parará el iterador principal y buscará un syscall 10. Al encontrarlo, comenzará a desplazar cada instrucción 4 bytes hacia abajo. El syscall 10 se identifica al buscar instrucciones li $vo X, guardando el inmediato asociado en memoria y sumándolo al resultado de enmascarar el código de instrucción de syscall. Seguidamente, el último syscall 10 será aquel que tenga una instrucción nop debajo de sí. El enunciado del proyecto asegura que hay espacio de nop para desplazar cada instrucción según se encuentren instrucciones add.

Si la instrucción a desplazar es un beq, se analiza el inmediato de la instrucción para saber si es necesario corregirlo (cuando la instrucción destino está por encima del add). Cuando el iterador de la subrutina alcanza al iterador principal, inserta el break en la posición deseada.

En caso de que el ciclo principal encuentre el último syscall 10, lo reemplaza por un break 0x10 y finaliza el instrumentador.

2.2 Manejador de excepciones:

Editando el manejador de excepciones de SPIM S20 MIPS, logramos adaptarlo a las exigencias del enunciado del proyecto. Comentamos las instrucciones que imprimen mensajes referentes a las excepciones estudiadas en este proyecto. También agregamos mensajes de inicio y finalización del instrumentador.

Fue necesario rescatar el pc del programa cuando éste sufre una excepción, para poder entonces guardarlo en la página de registros del programa y retomarlo

cuando se le asigne tiempo compartido.

Finalmente, leemos los datos del Transmitter para determinar si lo que se presionó en el teclado fue una `s`, una `p`, o la tecla `esc`.

Al pasar de un programa a otro, se realiza un proceso de almacenamiento y carga de registros, conservando el `pc` apropiado para continuar el programa al momento de la interrupción. En el caso de que se "apague la máquina" con `esc`, el manejador imprime los mensajes esperados y el estado de finalización de cada programa antes de terminar la ejecución del proyecto.

2.3.1 Planificador – Descripción:

El planificador ejecuta el primer programa y hace uso del manejador de excepciones para saltar de un programa al otro, todo mientras conserva los registros de cada uno y la posición del `pc` al momento de ocurrir la interrupción.

La mayor parte del trabajo la realiza el manejador de excepciones. El planificador solo inicializa los registros y obtiene la dirección del primer programa a ejecutar.

2.3.2 Estructura de datos:

El planificador se comporta como un algoritmo lineal. Después de inicializar los registros para usar el manejador, ejecuta el primer programa con una instrucción `j label`.

2.3.3 Planificación de registros:

- t0: Apuntador general. Contiene a RegActual (la página de registros sobre la que estamos trabajando) y luego la dirección del programa. En las instrucciones de `break`, muestra el número de programas ejecutándose. Con el registro también permitimos interrupciones.
- t1: Auxiliar a t0. Carga datos y realiza comparaciones con t0.
- t2: Durante las instrucciones de `break`, contiene a RegActual.
- t4: Muestra el estado de finalización del programa actual (1=finalizado). Luego muestra el número de adds del programa actual.

2.3.4 Diseño e implementación:

Cargamos los datos necesarios para conocer la página de registros del primer programa y la dirección de la primera instrucción. Luego, imprimimos un mensaje y permitimos interrupciones de teclado en `status` y `control` de Transmitter. Finalmente, limpiamos los registros usados hasta este punto.

2.4 Actividad adicional (Timer):

Se pedía implementar a través de una versión modificada de MARS, interrupciones de timer al programa, de manera que cuando el registro `count` fuese igual al registro `compare` ocurriese una interrupción.

Trabajamos con dos versiones del simulador MARS con un timer habilitado.

***2.4.1.a Versión 1 (Mars.jar)** - Comportamiento:*

El timer funcionó de forma satisfactoria. Realiza cambios en los registros count y status de coprocesador cuando es pertinente (a la hora de que el contenido de count llegase a ser igual al de compare). El bit correspondiente al interrupt level se comportaba de la manera esperada, de manera similar el bit de timer.

Notamos que la sección Tools no es visible en la interfaz del programa, por lo que las funciones de teclado del proyecto no se usan en esta versión (al menos de forma directa).

b. Manejo de interrupciones:

MARS interpreta las interrupciones del timer como interrupciones externas, similares a las de teclado que se implementaron en el proyecto. Esto permitió que fuese sencillo hacer que cuando ocurriese una interrupción de timer se colocara el count en 0x0 y se hiciese la rutina correspondiente a presionar la tecla `'s'` en el teclado.

Implementar una rutina que permitiese manejar colas de interrupciones es cuestión de planificación desde el comienzo del proyecto, por lo cual nuestro trabajo cuenta con una implementación modesta y práctica. Los escenarios donde podría usarse dicha rutina incluyen el caso de una interrupción de timer durante el manejo de otra interrupción (p.e. un `break`) o el caso `count = compare`.

No obstante, notamos que en esta versión ocurre lo siguiente:

Sea 'pi' un programa con gran cantidad de instrucciones `break`. Tenemos que durante la ejecución de una instrucción `break`...
1. Se llama al manejador de interrupciones para atender la interrupción.
2. **Apagamos los bits correspondientes a modo usuario y a interrupt enable** al entrar al manejador, de forma que no ocurran interrupciones mientras se trabaja en el segmento .ktext.
3. Cuando `count = compare`, ocurre una interrupción de timer.
4. A pesar de que es "ignorada", el registro `epc` es alterado.
5. Esto sabotea al simulador de manejador de excepciones SPIM20.

La solución a este problema sería modificar al manejador de excepciones para que sepa **administrar una cola de excepciones.** Adicionalmente, implementar este comportamiento después de haber finalizado las actividades del proyecto probablemente genere nuevos conflictos de casos borde. Estos podrían ser manejados de una manera más eficiente si la implementación de la cola de excepciones hubiese sido parte de las pautas o consideraciones iniciales.

Estando en esa posición, hubiéramos:

1. Estudiado casos pequeños de interrupciones individuales y consecutivas.
2. Determinado la relación timer/Uso de CPU/Run speed de MARS para saber un aproximado de cuánto tiempo real toma cada unidad de crecimiento de `count`.
3. Diseñado un esquema de flujo respecto al interrupt level y a cómo atender cada una de una manera efectiva y ordenada.
4. Finalmente, implementar la estructura en el proyecto.

2.4.2.a Versión 2 (MarsTimer.jar) - Comportamiento:

Esta versión no eleva su registro `count` como es de esperarse, pero este sí es visible y posible de editar.
La interfaz de MARS presenta un segundo botón de reset.

b. Manejo de interrupciones:

No fue posible evaluar este aspecto por las razones descritas en el punto anterior.

3. Conclusión.

Para realizar con éxito este proyecto fue necesario planificar con antelación cada una de las funciones. Esta estrategia es poderosa en el lenguaje de ensamblador, ya que permite escribir el código con relativamente poco esfuerzo.

La dificultad yace en los casos borde y en los conflictos entre subrutinas, ya que la simplicidad de la estructura obliga a que el programador sea quien vela por la correctitud del programa en el aspecto lógico-estructural de la implementación.

No podemos hacer suficiente énfasis en la importancia de conocer los parámetros dentro de los cuáles el programa actuará. En el caso del instrumentador y manejador, fue necesario encontrar el mejor momento para contar las instrucciones `add` y a la vez saber diferenciar los `syscall`, cuya relevancia para el instrumentador depende del valor de un registro.

Es buena práctica comentar el código mientras se programa, ya que más que una ayuda visual, ayuda al programador a pensar por segunda vez qué es lo que está ordenando en cada instrucción.

A pesar de todos los aprendizajes, trabajar en lenguaje ensamblador no deja de ser una tarea rigurosa, característica evidente en la ausencia de preguntas y participación en foros en línea por parte de usuarios que trabajan en ensamblaje. De igual forma, los creadores de MARS lo categorizan como un programa orientado a la educación, por lo cual se hace evidente que el principal objetivo de la programación en lenguaje ensamblador es aprender a apreciar la estructura básica del manejo de información en el procesador, siendo este capaz de representar estructuras abstractas complejas con un manejo inteligente y ambicioso de instrucciones, memoria y registros a través de operaciones lógicas con simples ceros y unos.

4. Anexos.

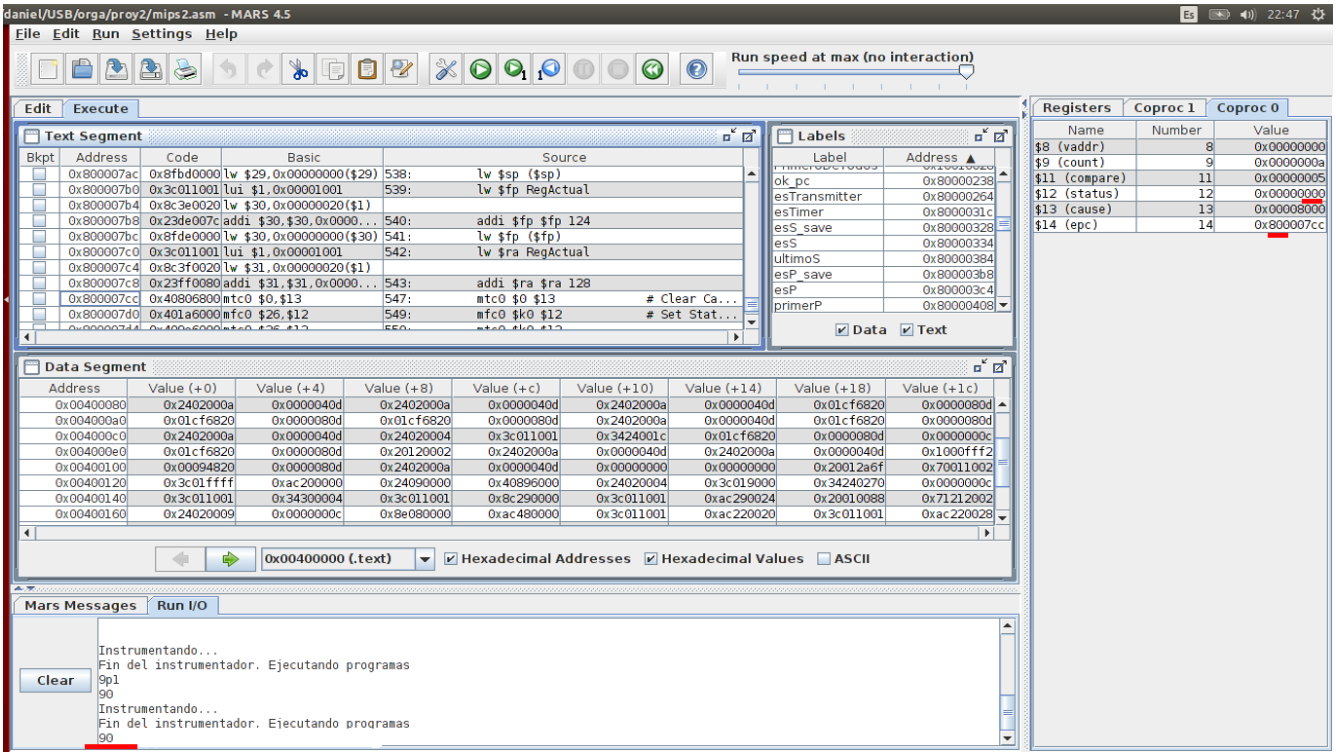

Figura 1. Captura del simulador Mars.jar durante la interrupción descrita en el punto 2.4.1.b. Nótese el registro `epc` y la sección Run I/O, donde el 9 subrayado repesenta una interrupción `break` y el 0 una interrupción externa (en este caso timer).
Ver también en https://imgur.com/TDNjei6.

5. Bibliografía.

- T. Altenkirch, L. Hu. Computer Systems Architecture – Lecture 12: Interrupts, Exceptions and I/O [Presentación]. Tomado de http://www.cs.nott.ac.uk/~psztxa/g51csa/l12-hand.pdf.