Introducción al simulador WinMips64

Salvador Carrillo Fuentes

Noviembre 2016

1 Primer ejercicio

Editar y ejecutar *p1_ej1.s* en el simulador. Responde a las cuestiones siguientes:

1.1 ¿Qué registros se utilizan en el programa para trabajar con las variables `i` y `j`?

Para la variable `i` usa el registro `R2` y para `j` el registro `R3`.

1.2 ¿Qué hace la instrucción `slt R6, R2, R5`? ¿Qué ocurre si intercambias los dos últimos registros de la instrucción?

Este tipo de instrucciones compara el contenido de dos registros y deja el valor 1 en un tercer registro si el primero es menor que el segundo (en caso contrario, se actualiza con valor 0).

En la instrucción que nos concierne, el registro `R6` se cargará con el valor 1 si el valor de `R2` es menor que el de `R5`, de otro modo, `R6` se cargará con el valor 0. Este tipo de instrucciones se leen *set on less than*.

El cometido de esta instrucción en el programa es indicar cuándo tenemos que ir la etiqueta `ENDWHILE` y salir del bucle. Eventualmente, `R6` se cargará con el valor 0 y la instrucción `beqz` tomará el salto hacia la dirección indicada por la etiqueta `ENDWHILE`.

El momento en el que `R6` tomará valor 0 se debe a que `R5` mantiene en 10 su valor, mientras que `R2` se incrementa en uno en cada paso por el bucle.

1.3 ¿Qué registro tiene almacenado el número de veces que tiene que ejecutarse el bucle?

`R5`. Este registro se inicializa con el valor 10 antes de entrar en el bucle y es el que marcará el número de veces que se entra, ayudado por el contador, que se irá incrementando en uno a cada paso por el bucle para llevar el cómputo de las veces que se ha entrado en el bucle. Así, cuando el contador llegue a ser igual a 10, no se cumplirá la condición, cargándose un 0 en `R6` y tomándose el salto del `beqz` hacia `ENDWHILE`.

1.4 Comprueba en la memoria qué valores tienen las variables `i` y `j` tras finalizar la ejecución del programa

La variable `i` tiene el valor `a` en base hexadecimal, que equivale al valor 10 en base decimal. Es consistente con el planteamiento realizado en cuestiones anteriores, ya que se incrementa una vez por cada entrada en el bucle y se entra diez veces, puesto que en el intento onceavo `i` ya tiene el valor 10 y no es menor que el valor con el que se inicializó `R5`.

La variable `j` acaba con el valor 32 en hexadecimal, que equivale a 50 en base diez. También sigue el mismo razonamiento que con la variable `i`, pero en este caso el incremento por cada entrada en el bucle no es de uno, sino de cinco. Por tanto, al entrar 10 veces en el bucle, el resultado es 50 (base 10).

La posición de memoria reservada para almacenar el valor de la variable `i` es la 0 (la primera). La siguiente posición es la 8 en base hexadecimal. Es decir, 8 bytes después (64 bits), ya que el contenido de las direcciones se muestra en palabras de 64 bits.

1.5 ¿Qué estrategia de salto se está utilizando para los saltos condicionales e incondicionales? ¿En qué etapa se produce la terminación del salto en ambos casos?

En los saltos condicionales se usa predicción de salto no tomado, puesto que entra la instrucción siguiente en vez de la instrucción destino. Es lo más lógico en este caso puesto que el salto normalmente no se tomará.

De hecho, se tomará 1 de cada 10 veces. El salto se resuelve en `ID` (etapa de decodificación) puesto que sólo se observa un ciclo de pérdida cuando se falla en la predicción (cuando tendría que haber saltado sólo entra una instrucción).

Igualmente, con la configuración actual del simulador no tendría sentido optar por la predicción de salto tomado, ya que la dirección del salto se conoce al final de `ID`. Esto implica que, aunque acertemos en la predicción (había que saltar y ha saltado), habrá un ciclo de pérdida igualmente.

Sin embargo, en la predicción de salto no tomado, cuando acertamos no perdemos ningún ciclo. En los saltos incondicionales siempre se salta y el salto también se resuelve en `ID`.

2 Segundo ejercicio

En el fichero *p1_ej2.s* encontrarás un programa que almacena en las 20 posiciones de un vector `A` el valor 2^4. La función `POWER` realiza el cálculo de la potencia de dos realizando sucesivos productos (`dmul`) por 2. El tamaño de los elementos del array es de 32 bits.

2.1 Comprobar el tiempo de ejecución (consulta los ciclos en la ventana *statistics* empleado en la ejecución total del programa, así como el que corresponde a instrucciones de la función `POWER`

El tiempo de ejecución es

$$\begin{aligned}\texttt{Tcpu} &= \text{nº instrucciones} \cdot \texttt{CPI} \cdot \text{tiempo de ciclo} \\ &= \text{nº ciclos} \cdot \text{tiempo de ciclo} \\ &= 247 \cdot \texttt{tck} = 158 \cdot 1.563 \cdot \texttt{tck}\end{aligned}$$

El tiempo de ciclo en el `MIPS` segmentado viene marcado por la unidad funcional más lenta, que en nuestro caso es 2 *ns*.

No es de mayor importancia calcular el número concreto ya que a la hora de compararlo con otro programa tendrá el mismo `tck` y podremos dejarlo indicado.

Para conocer el tiempo empleado sólo por las instrucciones de la función `POWER`, nos fijamos en el nº de instrucciones que se ejecutan en la función, o en el número de ciclos desde que se entra a la función hasta que se sale de ella.

No tenemos en cuenta la llamada a la función. La primera instrucción de la función entra en el ciclo 6 y la instrucción `jr` para volver a la instrucción siguiente a la de la llamada acaba en el ciclo 57.

Por tanto, las instrucciones de la función están activas durante 51 ciclos. Es decir, $\texttt{Tpower} = 51 \cdot \texttt{tck}$.

2.2 Si sustituimos la función `POWER` por el uso de la instrucción `dsll` (`dsll R3, R3, 4`) emplearíamos un sólo ciclo en el cálculo de la potencia 2. Determina los parámetros de la *Ley de Amdhal* y aplícala para predecir cuál será la aceleración que conseguimos al ejecutar el programa

Los parámetros usados en la *Ley de Amdhal* son:

- $\texttt{Fm} = \frac{\texttt{Tsub}}{\texttt{Tcpu}} = \frac{51 \cdot \texttt{tck}}{247 \cdot \texttt{tck}} = 0.2064$
- $\texttt{Sm} = \frac{\texttt{Tsub}}{\texttt{Tsub'}} = \frac{51 \cdot \texttt{tck}}{1 \cdot \texttt{tck}} = 51$

2.3 Cambia el programa sustituyendo la llamada a `POWER` por la instrucción de desplazamiento a la izquierda `dsll` (`dsll R3, R3, 4`). Comprueba el tiempo de ejecución (ciclos) para esta segunda versión. ¿Es consistente con la predicción que realizaste en el apartado anterior?

Al realizar el cambio en el código, las estadísticas de esta segunda implementación son:

- 197 ciclos
- 130 instrucciones
- `CPI` de 1.515

El `Tcpu'` = 197·`tck`. Por tanto la ratio de tiempos es

$$\frac{\texttt{Tcpu}}{\texttt{Tcpu'}} = \frac{247 \cdot \texttt{tck}}{197 \cdot \texttt{tck}} = 1.25 = \texttt{S}$$

Es un 25% mejor cuando sustituimos la instrucción `dsll` por la función `POWER`. Este resultado es coherente con lo obtenido en el apartado anterior:

$$\texttt{S} = \frac{1}{\frac{51 \cdot \texttt{tck}}{247 \cdot \texttt{tck}} + (1 - \frac{51 \cdot \texttt{tck}}{247 \cdot \texttt{tck}})} = 1.25$$