Taller de MIPS básico

Miguel Ángel

March 20, 2024

Contenidos

¿Qué es MIPS?

- Un lenguaje de ensamblado.
- Nos permite entender el código máquina (Binario puro)
- Microprocessor without Interlocked Pipeline Stages.
- Es un poco difícil al principio...
- Hay otras arquitecturas similares: ARM, x86_32, x86_64...

¿Para qué se usa?

- Para sistemas donde es necesaria alta eficiencia y velocidad.
- Esto es porque MIPS usa formatos de instrucción de tamaño fijo, que son más rápidas.
- Lo usan algunos supercomputadores y consolas (como la PlayStation o Nintendo64)

Figure: Procesador MIPS

Contenidos

Instrucciones Aritméticas

- Hay dos tipos de instrucciones aritméticas:
  - Tipo R (Registro)
  - Tipo I (Inmediato, son las que terminan en 'i')
- Las más importantes son:
  - add, addi, addu, addiu, add.s, add.d
  - sub, subu, sub.s, sub.d
  - mul, mult, mul.s, mul.d
  - div, div.s, div.d
  - and, andi
  - or, ori
  - xor, xori
  - sll, sllv, srl, srlv (en este caso las tipo R son las que terminan en 'v' de variable)
  - slt, slti

Instrucciones de Memoria

Todas las instrucciones de acceso a memoria tienen este formato:

lw \$s0, <Desplazamiento>(\$s1)

- lw, sw → cargan/guardan una palabra (4 Bytes en MIPS) en memoria
- lb, sb → cargan/guardan un Byte en memoria (por ejemplo caracteres de una cadena)
- lh, sh → cargan/guardan media palabra en memoria (2 Bytes)
- ld, sd → cargan/guardan dos palabras en memoria (8 Bytes)
  Para este último caso hay que usar registros dobles (por ejemplo los registros usados para los números de punto flotante)

Instrucciones de Saltos

Los saltos son instrucciones que nos llevan a otra zona del código. En esta diapositiva solo se verán los saltos incondicionales.
Estas son todas las instrucciones de saltos incondicionales:

- j (Salto incondicional)
- jal (Salto incondicional + enlace a dirección de memoria)
- jr (Salto incondicional a la dirección en un registro)

Instrucciones Condicionales

Las instrucciones condicionales son también saltos, pero que es necesaria cumplir una condición para realizarlo.
Las instrucciones más usadas son:

- beq (Bifurca si son iguales)
- bne (Bifurca si no son iguales)

Si queremos comparar si un número es mayor o menor que otro combinaremos las instrucciones slt con las bifurcaciones

Llamadas al S.O.

La palabra reservada es 'syscall'
Para realizar la llamada al sistema operativo cargaremos en el registro $v0 el valor de la llamada de sistema.
En el simulador MARS hay más de 50 códigos de llamada, pero muchos no los utilizaremos.
Códigos de escritura por salida estándar:

- 1 $\rightarrow$ imprimir entero en $a0
- 2 $\rightarrow$ imprimir float en $f12
- 3 $\rightarrow$ imprimir double en $f12
- 4 $\rightarrow$ imprimir string en $a0
- 11 $\rightarrow$ imprimir char en $a0

Llamadas al S.O.

Códigos de lectura por entrada estándar:

- 5 $\rightarrow$ leer entero por teclado y lo guarda en $v0
- 6 $\rightarrow$ leer float por teclado y lo guarda en $f0
- 7 $\rightarrow$ leer double por teclado y lo guarda en $f0
- 8 $\rightarrow$ lee cadena de la entrada y guarda el primer elemento en $a0, el resto en las direcciones de memoria continuas. Hay que especificar el tamaño de lo que vamos a leer y guardarlo en $a0.
- 12 $\rightarrow$ lee char por teclado y lo guarda en $v0

Llamadas al S.O.

Por último, tenemos otras llamadas de interés:

- 9 $\rightarrow$ función de kernel sbrk(), sirve para reservar memoria en el heap.
- 10 $\rightarrow$ fin de programa, esta llamada de sistema se hace siempre.
- 13 $\rightarrow$ abrir un fichero.
- 14 $\rightarrow$ leer de un fichero.
- 15 $\rightarrow$ escribir en un fichero.
- 16 $\rightarrow$ cerrar un fichero.
- 17 $\rightarrow$ fin de programa pero devolviendo un código de error. Normalmente 0 si ha salido todo bien, o un número distinto de 0 si ha ocurrido algún error del programa.

Otras Instrucciones

Hay otras instrucciones muy usadas en MIPS:

- la $\rightarrow$ carga en un registro la dirección de memoria de una etiqueta
- nop $\rightarrow$ literalmente no hace nada (más relevante en Procesadores con Segmentación)
- li $\rightarrow$ realmente es una pseudoinstrucción, pero es muy utilizada. Carga un inmediato en un registro

Contenidos

Primer ejercicio

Al principio del programa pediremos al usuario un número entero positivo, si es negativo o cero debe mostrar un mensaje de error y salir del programa. Una vez recibido ese número entero *n*, pediremos al usuario *n* enteros por teclado, tanto positivos como negativos o cero.
A todos esos números habrá que añadir el contenido de un vector de enteros predefinido en la sección .data. Este vector lo podéis elegir vosotros.
El programa deberá devolver la media aritmética de los números introducidos y los números del vector.

Contenidos

¿Cómo sabe el procesador dónde está la función?

- Tenemos una instrucción muy importante que es "jal" (jump and link)
- Esta instrucción saltará a la dirección de una etiqueta guardando la dirección de la siguiente instrucción a "jal" en el registro $ra.

```
.data
    inputText:  .asciiz "Introduce un número entero: "
    outputText: .asciiz "El resultado es: "
.text
    li $v0, 4
    la $a0, inputText
    syscall                 # Imprime inputText
    li $v0, 5
    syscall                 # Pide entero por teclado
    move $s0, $v0           # Movemos $v0 a $s0 para no perder el dato
    li $a0, '\n'
    li $v0, 11
    syscall                 # Imprimimos el salto de línea
    add $a0, $s0, $zero     # Argumento de la función
    jal suma2
    add $s0, $v0, $zero
    li $v0, 4
    la $a0, outputText
    syscall                 # Imprime outputText
    add $a0, $s0, $zero     # Movemos el resultado de la función a $a0 para imprimirlo
    li $v0, 1
    syscall
    li $v0, 10
    syscall
suma2:
    addi $v0, $a0, 2 # Suma dos y lo deja en $v0. $a0 es el registro de parámetro de función
    jr $ra
```

¿Y si hay más de una llamada seguida?

- Pues perderíamos el contenido del anterior $ra, así que para no perderlo tenemos que guardarlo en memoria, pero... ¿Dónde exactamente?
- Lo guardamos en una zona de memoria especialmente dedicada a las llamadas a funciones: la Pila o el Stack.

Contenidos

¿Qué es la Pila o Stack?

Si lo estudiamos desde un punto de vista de la programación es una Estructura de Datos que forma una estructura tipo LIFO (Last In First Out). Es decir, el último dato que entra en la pila será el primero que saquemos.
Pero es un enfoque que no se entiende...
Más fácil de pensar: la ropa que dejas en la silla de tu habitación, la vas apilando y la primera prenda que coges es la que está en lo alto.

¿Cómo se utiliza?

- Tenemos un registro especial para la pila, el $sp (stack pointer)
- La pila al comienzo del programa SIEMPRE está en la posición de memoria más alta. (0x7fffff...)
- Si queremos introducir un dato primero decrementamos el tamaño de lo que se quiera guardar y se guarda en un espacio libre de la pila.
- Si recuperamos ese dato de la pila, tendremos que incrementar el puntero de pila ($sp) tanto como el tamaño que hayamos recuperado.

Contenidos

Segundo Ejercicio

En la sección .data habrá 4 vectores de enteros, todos ellos de tamaño distinto.
Hay que escribir una función que reciba como parámetros en $ai con $i \in \{0, 1, 2, 3\}$ cada vector v_i, es decir, el vector 0 en el $a0, el vector 1 en el $a1...
Esta función debe calcular la media de cada vector a través de otra función que reciba en $a0 la dirección de comienzo del vector y en $a1 el tamaño del vector.
La función principal deberá devolver la media de las medias de cada vector en $v0.

Segundo Ejercicio

Funciones a programar:

- Función media4: recibe en $a0, $a1, $a2 y $a3 las direcciones de cada vector. Devolverá en $v0 la media de las medias de cada vector. Esta función deberá llamar a la función media para calcular la media de cada vector.
- Función media: recibe en $a0 la dirección de un vector y en $a1 el tamaño del vector. Devolverá en $v0 la media de los elementos del vector.