El repertorio de instrucciones de los procesadores MIPS: instrucciones y pseudoinstrucciones (I)

Escuela Técnica Superior de Ingeniería Informática
Universidad de La Laguna

Marzo 13, 2013

Esquema de la lección

Introducción

- Hemos estudiado en la lección anterior la codificación de las instrucciones y hemos podido ver ya el funcionamiento de algunas instrucciones como:
  - Formato R: **add** (suma), **addu** (suma sin signo), **sll** (desplazamiento lógico a la izquierda).
  - Formato I: **addi** (suma inmediata), **lw y sw** (carga y almacenamiento de palabra), **lui** (carga superior inmediata), **beq** (branch on equal).
  - Formato J: **j** (salto incondicional), **jal** (jump and link).
- Una vez conocemos la importancia del tipo de codificación en relación a las capacidades de una instrucción, haremos un repaso de las instrucciones de MIPS por criterios funcionales.

Función y tipos de instrucciones de carga

- Recordemos que las instrucciones de carga sirven para transferir un contenido a un registro.
- Todas las instrucciones de carga usan la codificación I (inmediata).
- El contenido puede provenir de:
  - La memoria. En este caso la parte inmediata de la instrucción se refiere a un desplazamiento respecto a un registro base que apunta a una ubicación de la memoria.
  - Una constante. Tenemos una instrucción para cargar una constante de 16 bits en la parte alta de un registro. Hay otras posibilidades de carga de constantes en registros, mediante **pseudoinstrucciones** de carga. Explicaremos el concepto de pseudoinstrucción más adelante.
  - Un registro. Nuevamente hablaremos en este caso tanto de instrucciones como de **pseudoinstrucciones** de carga.

Carga desde la memoria (I)

- Carga de una palabra: **lw**. Formato I. Operación: $R[rt] = M[R[rs] + SignExtImm]$. Obsérverse que la dirección final se obtiene como una suma binaria donde los operados se interpretan en C2.

Ensamblador MIPS:

```
1   lw $t0,32($t1)
```

El orden de llenado del registro depende de una característica propia de la máquina denominada **endianness**.

- Si la máquina es **big endian** se llenará el registro desde su parte menos significativa a la más significativa comenzando por el byte con dirección más alta.
- Si la máquina es **little endian** se llenará el registro desde su parte menos significativa a la más significativa comenzando por el byte con dirección más baja.

Carga desde la memoria (II)

- Carga de media palabra: **lh**. Formato I. Operación: $R[rt] = SignExt(M[R[rs] + SignExtImm](0:15))$. Se cargan cuatro bytes de la memoria en el orden establecido por el endianness de la máquina. El signo de la media palabra cargada es extendido en el registro.
- Carga de media palabra sin signo: **lhu**. Formato I. Operación: $R[rt] = ZeroExt(M[R[rs] + SignExtImm](0:15))$. La diferencia con el anterior es que no se extiende el signo de la media palabra obtenida de la memoria.
- Carga de un byte: **lb**. Formato I. Operación: $R[rt] = SignExt(M[R[rs] + SignExtImm](0:7))$. Se carga un byte de la memoria. El signo del byte cargado es extendido en el registro.
- Carga de un byte sin signo: **lbu**. Formato I. Operación: $R[rt] = ZeroExt(M[R[rs] + SignExtImm](0:7))$. No se extiende el signo.

Instrucción de carga inmediata

- Carga inmediata de media palabra en la parte superior: **lui**. Formato I. Operación: $R[rt] = \{imm, 16'b0\}$. Como vemos esta instrucción tiene dos efectos. Los 16 bits menos significativos del registro quedan a 0. Los 16 bits más significativos quedan con el valor de la constante.

Instrucciones de carga desde registro

- Se trata de instrucciones especiales, puesto que sólo permiten la carga desde registros internos particulares a registros accesibles por el programador.
- En particular, se puede realizar la carga desde los registros internos Hi y Lo utilizados en las operaciones aritméticas de multiplicación y división.
- Carga desde Hi: **mfhi**. Formato: R. Operación: $R[rd] = Hi$.
- Carga desde Lo: **mflo**. Formato: R. Operación: $R[rd] = Lo$.

Pseudoinstrucciones

- Una pseudoinstrucción es una instrucción que el fabricante define para el ensamblador de la máquina pero que no tiene una traducción directa a una instrucción del procesador.
- Teniendo en cuenta la definición de la pseudoinstrucción dada por el fabricante, el compilador deberá traducir el ensamblador a las instrucciones máquinas que logren dicha funcionalidad.
- Así pues, una pseudoinstrucción puede requerir sólo una instrucción máquina en algunos casos o varias en otras.
- Al utilizar depuradores debemos tener en cuenta esto, ya que aunque en nuestro código hayamos usado la pseudo instrucción, en la versión compilada ya no la encontraremos (sólo habrá instrucciones del repertorio del procesador).

Pseudoinstrucciones de carga (I)

- Carga de constante inmediata: **li**. Operación: carga en el registro destino una constante. Veamos algunos ejemplos de como el compilador puede traducir la pseudoinstrucción a instrucciones dependiendo del valor de la constante (carga de $t0$ con la constante k)

Ensamblador MIPS:

```
2  li $t0,k #es equivalente a....
3  ori $t0,$zero,k #(si 0<=k<=65353)
4  addiu $t0,$zero,k #(si -32768<=k<=0)
5
6  lui $t0,hi16(k)
7  ori $t0,$t0,lo16(k) #(para otra posible constante de 32
       bits)
```

Pseudoinstrucciones de carga (II)

- Carga de dirección: **la**. Operación: carga en el registro destino una constante que se asume es una dirección de memoria.

Ensamblador MIPS:

```
8  la $t0, mietiqueta #carga en t0 mietiqueta
```

- Carga desde un registro: **move**. Operación: carga en el registro destino, el contenido de otro registro.

Ensamblador MIPS:

```
9  move $t0,$t1 #carga el contenido de $t1 en $t0
```

Intrucciones de almacenamiento (I)

- Las instrucciones de almacenamiento permiten transferir el contenido de los registros a la memoria.
- Al igual que con las instrucciones de carga el orden de los bytes transferidos depende del endianness de la máquina.
- La dirección de memoria se calcula mediante la suma de un desplazamiento a un registro que actúa como dirección base.
- Tenemos instrucciones para transferencia de palabra, media palabra y byte.

Instrucciones de almacenamiento (II)

- Almacenamiento de palabra: **sw**. Operación: $M[R[rs] + SignExtImm] = R[rt]$. Ejemplo:

Ensamblador MIPS:

```
10
11    sw  $t0,32($t1)
```

- Almacenamiento de byte: **sb**. Operación: $M[R[rs] + SignExtImm] = R[rt](7:0)$. Almacena en memoria el byte más bajo del registro.
- Almacenamiento de media palabra: **sh**. Operación: $M[R[rs] + SignExtImm] = R[rt](15:0)$. Almacena en memoria la media palabra más baja del registro.

Sumas y restas no inmediatas

- Para cada operación tenemos una variante capaz de provocar excepción de "overflow" y otra que no.
  - **add** (Formato R). $R[rd] = R[rs] + R[rt]$. Entiéndase que el operador $+$ es la suma binaria. En caso de overflow bajo la interpretación de complemento a 2 produce una excepción de overflow.
  - **addu** (Formato R). $R[rd] = R[rs] + R[rt]$. Entiéndase que el operador $+$ es la suma binaria. No genera excepción por overflow.
  - **sub** (Formato R). $R[rd] = R[rs] - R[rt]$. Entiéndase que el operador $-$ es la suma binaria con el **opuesto** del segundo operando, codificado conferme a la interpretación de complemento a 2. En caso de overflow bajo la interpretación de complemento a 2 produce una excepción de overflow.
  - **subu** (Formato R). $R[rd] = R[rs] - R[rt]$. Versión de sub que no genera excepción por overflow.

Sumas inmediatas

- Hay versiones inmediatas para las dos variantes de la suma:
  - Suma inmediata: **addi** (Formato I).
    $R[rd] = R[rs] + SignExtImm$. Entiéndase que el operador $+$ es la suma binaria y *SignExtImm* es la extensión del signo de la parte inmediata hasta 21 bits. En caso de overflow se producirá excepción.
  - Suma inmediata sin signo **addiu** (Formato I).
    $R[rd] = R[rs] + SignExtImm$. Entiéndase que el operador $+$ es la suma binaria y *SignExtImm* es la extensión del signo de la parte inmediata hasta 21 bits. En caso de overflow no se producirá excepción.
- Recuerden que la instrucción de suma inmediata puede utilizarse con propósitos diferentes a los de la suma, como por ejemplo cargar la parte baja de un registro con una constante o copiar un registro en otro.

Multiplicación y división (I)

- Una multiplicación de dos números de 32 bits, da un resultado que puede requerir hasta 64 bits (dos palabras). Se necesitan dos lugares de almacenamiento para el resultado. MIPS ofrece dos registros internos denominados respectivamente **Lo** y **Hi**:
  - Multiplicación: **mult** (Formato R). $Hi, Lo = R[rs] \times R[rt]$. La parte "alta" del resultado de la multiplicación se guarda en Hi, mientras que la parte baja se guarda en Lo. Esta variante de la multiplicación considera que los números deben interpretarse en complemento a 2.
  - Multiplicación sin signo: **multu** (Formato R). $Hi, Lo = R[rs] \times R[rt]$. La parte "alta" del resultado de la multiplicación se guarda en Hi, mientras que la parte baja se guarda en Lo. Esta variante de la multiplicación considera que los números deben interpretarse en binario natural.

Multiplicación y división (II)

- Por otra parte, la división produce como resultado un cociente y un resto. Se necesitan dos lugares de almacenamiento para el resultado. MIPS ofrece dos registros internos denominados respectivamente **Lo** y **Hi**:
  - División: **div** (Formato R). El cociente a Lo: $Lo = R[rs]/R[rt]$. El resto a Hi: $Hi = R[rs]\,\%R[rt]$. Se interpretan los números en complemento a 2.
  - División sin signo: **divu** (Formato R). El cociente a Lo: $Lo = R[rs]/R[rt]$. El resto a Hi: $Hi = R[rs]\,\%R[rt]$. Se interpretan los números en binario natural.

Introducción a las instrucciones lógicas

- Las instrucciones lógicas implementan operacions lógicas al nivel de bit entre dos palabras.
- Las operaciones lógicas más frecuentes son:
  - **And**. La operación $x\&y$ es 1 sólo si ambos operandos son 1 y 0 en caso contrario.
  - **Or**. La operación $x|y$ es 1 si alguno de los operandos es 1 y 0 cuando ambos operandos son 0.
  - **Nor**. Se define como la negación del operador Or.
  - **Xor**. Se denomina Or exclusivo. $x\hat{y}$ es 1 si uno solo de los operandos es 1 y 0 en caso contrario.

Ejemplos de uso de las operaciones lógicas

- Poniendo a 0 la mitad superior de una palabra, sin alterar la otra mitad. Sea x una palabra de 32 bits. Obtenemos $y = x\&0 \times 0000FFFF$
- Poniendo a 1 la mitad superior de una palabra, sin alterar la otra mita. Sea x una palabra de 32 bits. Obtenemos $y = x|0 \times FFFF0000$.
- Mutando los bits que ocupan posiciones pares en una palabra. Sea x una palabra de 32 bits. Obtenemos $y = x\hat{}0 \times 55555555$.

Las instrucciones lógicas no inmediatas en MIPS

- Instrucción **and**. Formato R. Se define $R[rd] = R[rs] \& R[rt]$.
- Instrucción **or**. Formato R. Se define $R[rd] = R[rs] | R[rt]$. Obsérves que la instrucción or puede usarse para copiar un registro en otro:

Ensamblador MIPS:

```
13
14   or $t0,$t1,$zero
```

- Instrucción **nor**. Formato R. Se define $R[rd] = not(R[rs] | R[rt])$.
- Instrucción **xor**. Formato R. Se define $R[rd] = R[rs] \veebar R[rt]$.

Las instrucciones lógicas inmediatas en MIPS (I)

- Instrucción **andi**. Formato I. Se define $R[rt] = R[rs] \& ZeroExtImm$. Observen como podemos usar la instrucción andi para ponera 0 la parte alta de una palabra en un registro:

Ensamblador MIPS:

```
16   andi  $t0,$t0,0xFFFF
```

- Instrucción **ori**. Formato I. Se define $R[rt] = R[rs] | ZeroExtImm$. Observen que la instrucción ori puede ser utilizada para cargar un valor constante en la parte baja de un registro:

Ensamblador MIPS:

```
17   ori  $t0,$zero, 0x4A00
```

Las instrucciones lógicas inmediatas en MIPS (II)

- Instrucción **xori**. Formato I. Se define $R[rt] = R[rs] \veebar ZeroExtImm$. Observen como podemos usar xori para mutar todos los bits de la parte baja de una palabra sin alterar el resto:

Ensamblador MIPS:

```
18   xori $t0,$t0,0xFFFF
```

Introducción (I)

- En general las instrucciones de desplazamiento de bits, sirven para desplazar un número de posiciones determinado el conjunto de bits de una palabra.
- El desplazamiento puede ser a la izquierda o a la derecha.
- En el desplazamiento a la izquierda, el bit que "entra" en la palabra por la derecha es un 0. Siendo así, se denomina **desplazamiento lógico a la izquierda**.
- En el desplazamiento a la derecha, hay dos posibiliades para el bit que "entra" en la palabra por la izquierda.
  - Si entra un 0 siempre, se denomina **desplazamiento lógico a la derecha**.
  - Si por el contrario, entra el bit de signo de la palabra antes de efectuar cada desplazamiento, se denomina **desplazamiento aritmético a la derecha**.

Introducción (II)

Para comprender la diferencia entre los tipos de desplazamiento, obsérvese la siguiente codificación de complemento a dos de una celda de 4 bits:

$(7)_{10}$	0	1	1	1
$(6)_{10}$	0	1	1	0
$(5)_{10}$	0	1	0	1
$(4)_{10}$	0	1	0	0
$(3)_{10}$	0	0	1	1
$(2)_{10}$	0	0	1	0
$(1)_{10}$	0	0	0	1
$(0)_{10}$	0	0	0	0
$(-1)_{10}$	1	1	1	1
$(-2)_{10}$	1	1	1	0
$(-3)_{10}$	1	1	0	1
$(-4)_{10}$	1	1	0	0
$(-5)_{10}$	1	0	1	1
$(-6)_{10}$	1	0	1	0
$(-7)_{10}$	1	0	0	1
$(-8)_{10}$	1	0	0	0

- Desplazamiento lógico la izquierda de 2 es $\{0, 1, 0, 0\}$ que se corresponde con el 4 en la tabla. Equivale a multiplicar por dos, y el resultado en este caso es válido para binario natural y C2.
- Desplazamiento lógico la izquierda de 4 es $\{1, 0, 0, 0\}$ que se corresponde con el -8 en la tabla. Equivale a multiplicar por dos pero sólo en binario natural, puesto que el resultado queda fuera del rango de C2.
- Desplazamiento lógico a la izquierda de -3 es $\{1, 0, 1, 0\}$ que equivale a -6 en la tabla. Equivale a multiplicar por dos en complemento a 2.
- Desplazamiento lógico a la izquierda de -5 es $\{0, 1, 1, 0\}$ equivale 3 en la tabla. El resultado no es correcto ya que cae fuera del rango de complemento a 2.

Introducción (III)

Para comprender la diferencia entre los tipos de desplazamiento, obsérvese la siguiente codificación de complemento a dos de una celda de 4 bits:

$(7)_{10}$	0	1	1	1
$(6)_{10}$	0	1	1	0
$(5)_{10}$	0	1	0	1
$(4)_{10}$	0	1	0	0
$(3)_{10}$	0	0	1	1
$(2)_{10}$	0	0	1	0
$(1)_{10}$	0	0	0	1
$(0)_{10}$	0	0	0	0
$(-1)_{10}$	1	1	1	1
$(-2)_{10}$	1	1	1	0
$(-3)_{10}$	1	1	0	1
$(-4)_{10}$	1	1	0	0
$(-5)_{10}$	1	0	1	1
$(-6)_{10}$	1	0	1	0
$(-7)_{10}$	1	0	0	1
$(-8)_{10}$	1	0	0	0

- Desplazamiento lógico a la derecha de 6 es $\{0, 0, 1, 1\}$ que se corresponde com el 3. Equivale a dividir por dos. El resultado es válido para este caso tanto en complemento a 2 como en binario natural.
- Desplazamiento lógico a la derecha de -6 es $\{0, 1, 0, 1\}$ que se corresponde con el 5. El resultado no es equivalente a dividir por dos.
- Desplazamiento aritmético a la derecha de -6 es $\{1, 1, 0, 1\}$ que se corresponde con el -3. El resultado es equivalente a dividir por dos.

Introducción (IV)

- En resumen, entendiendo un desplazamiento a la izquierda (derecha) como multiplicación (división) por dos, y asumiendo que el resultado de la operación está en el rango:
  - El desplazamiento lógico a la izquierda da el resultado correcto.
  - El desplazamiento lógico a la derecha da el resultado correcto para binario natural y números positivos en C2.
  - El desplazamiento aritmético a la derecha da el resultado correcto.

Operaciones de desplazamiento constante

El desplazamiento es un valor entre 0 y 31 constante.

- Desplazamiento lógico a la izquierda **sll**. Formato R. Resulta en $R[rd] = R[rs] << shamt$.
- Desplazamiento lógico a la derecha **srl**. Formato R. Resulta en $R[rd] = R[rs] >> shamt$.
- Desplazamiento aritmético a la derecha **sra**. Formato R. Resulta en $R[rd] = R[rs] >>> shamt$-

Operaciones de desplazamiento variable

En este caso el desplazamiento se establece en el registro rt.

- Desplazamiento lógico a la izquierda variable **sllv**. Formato R. Resulta en $R[rd] = R[rs] << R[rt]$.
- Desplazamiento lógico a la derecha variable **srlv**. Formato R. Resulta en $R[rd] = R[rs] >> R[rt]$.
- Desplazamiento aritmético a la derecha variable **srav**. Formato R. Resulta en $R[rd] = R[rs] >>> R[rt]$.