PROGRAMANDO EN MIPS

Principio de Computadores

Sesión académicamente dirigida nº 2

INTRODUCCIÓN: PROGRAMANDO EN ENSAMBLADOR

- Cuando uno desea desarrollar un algoritmo normalmente se hace en un lenguaje de alto nivel como C o Java. Estos programas podrán servir como el pseudocódigo para implementar el algoritmo en ensamblador.
- En esta sesión se explicarán algunos de los elementos básicos que son necesarios para poder realizar los algoritmos de nuestras prácticas:
  - el uso de los registros
  - las estructuras de control.

USO DE LOS REGISTROS

- Nuestros programas de alto nivel usan variables para almacenar la información (masa, radio, etc.).
- Estas variables no son más que identificadores que hacen referencia a una posición de memoria donde se almacena el dato correspondiente.
- Ya se ha visto que en ensamblador del MIPS declaramos nuestras variables bajo la directiva .data
- Sin embargo, cuando tratamos los datos en MIPS (como en la mayoría de las CPUs), no podemos operar directamente con lo que se encuentra en la memoria, sino que los datos tienen que estar en los registros.

USO DE LOS REGISTROS

- Muchas de las variable, principalmente las locales que tienen una vida corta, basta con almacenarlas en un registro. No necesitan almacenarse en memoria (no tienen que definirse en la sección `.data`). Esta es la solución más eficiente.
- Cuando se codifica un algoritmo, es necesario mantener una tabla de referencia con los registros que se están usando en el mismo y a qué variable de tu algoritmo en lenguaje de alto nivel hace referencia cada registro.
- Esta información debe aparecer en los comentarios de tu código fuente.

LOS REGISTROS DEL MIPS

- MIPS dispone de 32 registros enteros $0-$31
  - Cada uno de 32 bits → tamaño palabra (word).
- Para la CPU todos los registros son equivalentes, es decir, todos tienen las mismas capacidades y funcionalidades.
- Pero para facilitar la programación, existe un convenio que asigna funcionalidades específicas a cada registro.
- Tenemos que respetar el convenio si queremos que nuestros programas puedan ser entendidos y usados de forma correcta por otros programadores.
- El programador de ensamblador es responsable de hacer un uso adecuado de los registros.
- Se definen unos nombres simbólicos (alternativos a $0-$31) según la funcionalidad del registro: $t0, $t1, … $s0, $sp, …
- En nuestros programas de ensamblador podemos y debemos usar los nombres simbólicos.

RESUMEN FUNCIONALIDAD DE LOS REGISTROS

Num	Nombre	Descripción
0	$zero	Constante valor cero
1	$at	Temporal reservado para el ensamblador (no utilizar)
2-3	$v0-$v1	Valores resultantes de funciones y evaluaciones
4-7	$a0-$a3	Argumentos para subrutinas. No se preservan a través de llamadas a subrutinas.
8-15	$t0-$t7	Temporales. No se preservan a través de llamadas de subrutinas, por lo que la rutina que llama debe salvarlos si los quiere conservar.
16-23	$s0-$s7	Valores salvados. Una subrutina que los use debe salvar sus valores antes de usarlos, y restaurarlos al salir.
24-25	$t8-$t9	Continuación a los $t0-t7. Temporales. No se preservan a través de llamadas de subrutinas, por lo que la rutina que llama debe salvarlos si los quiere conservar
26-27	$k0-$k1	Reservados para el kernel (manejo de interrupciones)
28	$gp	Puntero global. Apunta al medio del bloque de 64K en el segmento de datos estáticos
29	$sp	Puntero de pila (stack pointer)
30	$s8/$fp	Valor salvado. Puntero de marco. Se preserva a través de llamadas.
31	$ra	Dirección de retorno (return address)

REGISTROS: CONVENIO USO

- $zero: su contenido siempre debe estar a valor cero. No se debe modificar.
- $at: usado por el ensamblador al implementar pseudoinstrucciones. No se debe utilizar por el programador.
- $v0-$v1: cuando una subrutina (función) tiene que devolver pocos resultados deberá utilizar estos registros para depositarlos. Por ello no se garantiza que su valor se conserve al llamar a una subrutina.
- $a0-$a3: cuando una subrutina tenga pocos parámetros de entrada se usarán estos registros para pasar los valores. Por ello no se garantiza que su valor se conserve al llamar a una subrutina.

REGISTROS: CONVENIO USO

- $s0-$s7: son registros “salvados” (saved) y se garantiza que su contenido se conservará cuando llamemos a una subrutina.
  - Cuando escribamos una subrutina, si necesitamos utilizar alguno de estos, tendremos que encargarnos de salvarlos al comienzo y restaurarlos antes de regresar.

- $t0-$t9: son registros “temporales” y no se garantiza que se conserve su valor cuando llamemos a una subrutina.
  - Solo debemos usarlos si durante su "vida útil" no hay una llamada a subrutina
  - Si estamos programando una subrutina o función no es responsabilidad de la subrutina restaurar el valor de los mismos a la salida.

REGISTROS: CONVENIO USO

- $sp: puntero de pila (stack pointer). Pila es zona de memoria donde las rutinas almacenan valores temporalmente. Debe manejarse de manera rigurosa según convenios que veremos cuando se presenten las subrutinas. Uso incorrecto puede provocar errores difíciles de depurar.
- $ra: Almacena la dirección a la que debe regresar la subrutina llamada. Veremos su uso cuando se presenten las subrutinas.

ESTRUCTURAS DE CONTROL

- Está demostrado que cualquier algoritmo se puede implemtar mediante la combinación de las siguientes estructuras:
  - estructuras secuenciales → se ejecuta una instrucción detrás de otra
  - estructuras alternativa → según condición se ejecuta unas intrucciones u otras
  - estructuras iterativas → permite la repetición de un conjunto de instrucciones
- La programación estructurada solo permite el uso de esas estructuras. En C y C++ estas estructructuras son:
  - secuenciales → bloque de instrucciones (entre { })
  - alternativas → if-else y switch
  - iterativas → while, do-while, for

ESTRUCTURAS DE CONTROL

- En ensamblador solo existe la estructura secuencial: al terminar una instrucción se ejecuta la siguiente en memoria.
- El resto de estructuras se deben replicar mediante el uso de saltos.
- Los saltos modifican el contador de programa de manera que la siguiente instrucción a ejecutar no sea la siguiente en memoria.
- La dirección de la instrucción a la que se salta se indica mediante una etiqueta definida en nuestro código fuente
  - unos pocos saltos no usan etiqueta, sino dirección almacenada en registro
- En todas las CPUs existen dos tipos de saltos:
  - incondicionales → simpre saltan
  - condicionales → saltan si se cumple condición

SALTOS EN MIPS

- Saltos incondionales:
  - `j etiqueta   # salta a línea de código con 'etiqueta:'`
  - `b etiqueta   # salta a línea de código con 'etiqueta:'`
  - `jr $t3   # salta a la dirección de programa contenida en $t3`
- Saltos condicionales:
  - unos tienen la forma `bCC $t0,$t1,etiqueta`
    - Si se cumple la comparación `CC` entre `$t0` y `$t1` se salta a `etiqueta`
    - caso contrario se continúa con siguiente instrucción
  - Las comparaciones dependen si datos con signo o sin signo:

Con Signo	Comparación	Sin Signo	Para acordarse
blt	<	bltu	Less Than
ble	<=	bleu	Less or Equal
beq	==	beq	Equal
bne	!=	bne	Not Equal
bge	>=	bgeu	Greater or Equal
bgt	>	bgtu	Greater Than

SALTOS EN MIPS

- Saltos condicionales: comparación con cero
  - tienen la forma `bCCz $t0,etiqueta`
    - Si se cumple comparación `CC` entre `$t0` y cero se salta a `etiqueta`
    - caso contrario se continúa con siguiente instrucción
  - Las comparaciones posibles son:

	Comparación	Para acordarse
bltz	< 0	Less Than Zero
blez	<= 0	Less or Equal Zero
beqz	== 0	EQual Zero
bnez	!= 0	Not Equal Zero
bgez	>= 0	Greater or Equal Zero
bgtz	> 0	Greater Than Zero

ALTERNATIVA IF

- En C → se ejecutan instrucciones si se cumple condición
- En ensamblador → **SALTAMOS** si se cumple condición
- Queremos **mantener orden de las instrucciones y de los operandos** de la comparación → hay dos alternativas:
  - SIN invertir la comparación

```
if ($s0 != $s1) {
    // si se cumple condición → then
    $s3 = $s1;
}
// después if
```

```
if:
    bne     $s0,$s1,if_then
    b       if_fin
if_then:
    # si se cumple condición → then
    move    $s3,$s1
if_fin:
    # después if
```

  - INVERTIR la comparación y evitamos el salto extra

```
if ($s0 != $s1) {
    // si se cumple condición → then
    $s3 = $s1;
}
// después if
```

```
if:
    beq     $s0,$s1,if_fin
    # si se cumple condición → then
    move    $s3,$s1
if_fin:
    # después if
```

ALTERNATIVA IF-ELSE

- Si existe parte `else` ⇒ ensamblador necesario salto incondicional al final parte **`then`** para evitar ejecutar parte **`else`**.
  - SIN invertir comparación:

```
if ($s0 != $s1) {
    // si se cumple condición → then
    $s3 = $s1;
} else {
    // si no se cumple condición
    $s3 = $s0;
}
// después if
```

```
if:
    bne     $s0,$s1,if_then
    b       if_else
if_then:
    # si se cumple condición → then
    move    $s3,$s1
    b       if_fin # evitar parte else
if_else:
    # si no se cumple condición
    move    $s3,$s0
if_fin:
    # después if
```

  - INVERTIR la comparación y evitamos el salto extra

```
if ($s0 != $s1) {
    // si se cumple condición → then
    $s3 = $s1;
} else {
    // si no se cumple condición
    $s3 = $s0;
}
// después if
```

```
if:
    beq     $s0,$s1,if_else
    # si se cumple condición → then
    move    $s3,$s1
    b       if_fin # evitar parte else
if_else:
    # si no se cumple condición
    move    $s3,$s0
if_fin:
    # después if
```

ITERATIVA WHILE

- En C → permanece `while` mientras se cumple condición
- En ensamblador → **SALTAMOS** si se cumple condición

```
suma = 0;
std::cin >> numero;

while (numero > 0) {
    // interior while
    suma += numero;
    std::cin >> numero;
}
// después while
```

```
# solución SIN invertir comparación
# suma → $s0
# numero → $v0
    move    $s0,$zero

    # leo entero por consola
    li      $v0,5
    syscall # entero en $v0

while:
    #salimos si numero <= 0
    bgtz    $v0,while_dentro
    b       while_fin
while_dentro:
    # interior de while
    add     $s0,$s0,$v0
    # leo entero por consola
    li      $v0,5
    syscall # entero en $v0

    b       while # vuelvo condición
while_fin:
    # después while
```

```
# solución INVERTIR comparación
# suma → $s0
# numero → $v0
    move    $s0,$zero

    # leo entero por consola
    li      $v0,5
    syscall # entero en $v0

while:
    #salimos si numero <= 0
    blez    $v0,while_fin

    # interior del while
    add     $s0,$s0,$v0
    # leo entero por consola
    li      $v0,5
    syscall # entero en $v0

    b       while # vuelvo condición
while_fin:
    # después while
```

ITERATIVA WHILE

- Otra opción poner condición al final
  - se CAMBIA orden de las instrucciones
  - SIN invertir comparación

```
suma = 0;
std::cin >> numero;

while (numero > 0) {
    // interior while
    suma += numero;
    std::cin >> numero;
}
// después while
```

```
# suma → $s0
# numero → $v0
    move    $s0,$zero

    # leo entero por consola
    li      $v0,5
    syscall # entero en $v0

    # Saltamos a condición del while
    b       while_condicion
while:
    # interior del while
    add     $s0,$s0,$v0
    # leo entero por consola
    li      $v0,5
    syscall # entero en $v0

while_condicion:
    #repetimos si numero > 0
    bgtz    $v0,while
while_fin:
    # después while
```

ITERATIVA FOR

- En C un `for` es totalmente equivalente a un `while`
  - usar cuando se sabe número de iteraciones al comenzar → bucles con contador

```
// sumar 100 enteros
suma = 0;

for(int i=0; i<=100; i++) {
    suma += i;
}
// después for
```

```
// sumar 100 enteros
suma = 0;

int i=0; // inicilación
while(i<=100){ // condición
    suma += i;
    i++; // actualización
}
// después for
```

```
# solución SIN invertir la comparación
# suma → $s0
# i → $t0 no hay llamadas syscall ni funciones
    move    $s0,$zero

    move    $t0,$zero #inicilización i=0
for:
    #condición → entramos si i<=100
    ble     $t0,100,for_dentro
    b       for_fin
for_dentro:
    add     $s0,$s0,$t0

    addi    $t0,$t0,1   # actualización i++
    b       for # vuelta condición
for_fin:
    # después for
```

```
# Solución INVIRTIENDO la comparación
# suma → $s0
# i → $t0 no hay llamadas syscall ni funciones
    move    $s0,$zero

    move    $t0,$zero #inicilización i=0
for:
    #condición invertida → salimos si i> 100
    bgt     $t0,100,for_fin
    add     $s0,$s0,$t0

    addi    $t0,$t0,1   # actualización i++
    b       for # vuelta condición
for_fin:
    # después for
```

TRATA DE HACERLO TU

- Iterativa do-while
  - pista: es la más directa

```
suma = 0;
maximo = 100;

do {
    std::cin >> numero;
    suma += numero;
} while (suma < maximo);
// después do-while
```

TRATA DE HACERLO TU

- Alternativa switch
  - rescribirla como serie de `ifs`
  - implementar directamente pensando el algoritmo que sigue

```
// sumar 100 enteros
std::cin >> eleccion;

switch(eleccion) {
case 1:
    valor *= 2;
    break;
case 2:
    valor /= 2;
    break;
case 3:
    valor += 5;
    break;
case 4:
    valor *= 3;
    break;
default:
    valor++;
}
// después switch
```

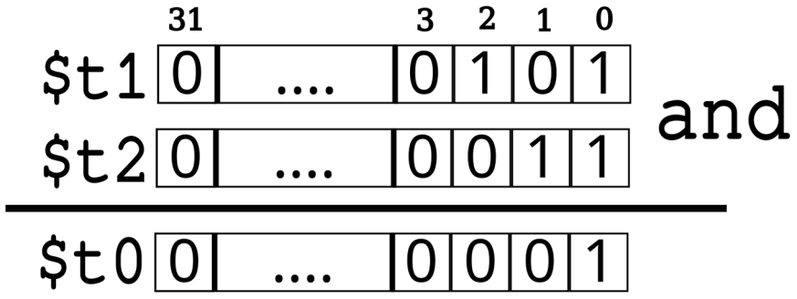
CONDICIONES MÁS COMPLEJAS

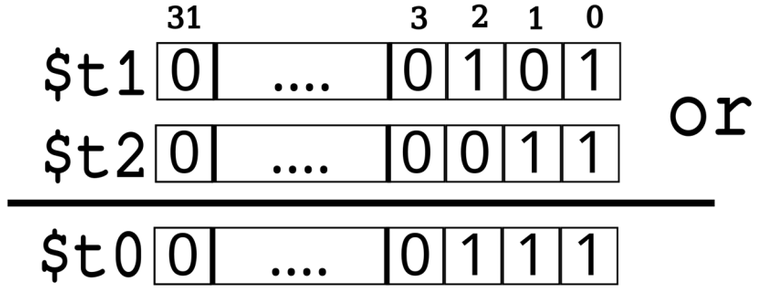
- En ocasiones las condiciones están compuestas de varias comparaciones
- Se pueden usar registros para almacenar el valor booleano del resultado de una comparación
  - Instruciones SET de la forma `sCC $t0,$t1,$t2`
    - Si se cumple comparación `CC` entre `$t1` y `$t2` se pone `$t0` a `1`
    - caso contrario `$t0` se pone a `0`
  - Como en los saltos, las comparaciones dependen si datos con signo o sin signo:

Con Signo	Comparación	Sin Signo	Para acordarse
slt	<	sltu	Less Than
sle	<=	sleu	Less or Equal
seq	==	seq	Equal
sne	!=	sne	Not Equal
sge	>=	sgeu	Greater or Equal
sgt	>	sgtu	Greater Than

CONDICIONES MÁS COMPLEJAS

- Los valores booleanos se pueden combinar utlilizando operaciones booleanas bit a bit:
  - `and $t0,$t1,$t2  # y-bit-a-bit entre $t1 y $t2, resultado en $t0`
    `# lo que en C sería $t0 = $t1 & $t2`
  - `or $t0,$t1,$t2  # o-bit-a-bit entre $t1 y $t2, resultado en $t0`
    `# lo que en C sería $t0 = $t1 | $t2`
  - `xor $t0,$t1,$t2  # o-exclusiva-bit-a-bit entre $t1 y $t2,`
    `# resultado en $t0`
    `# lo que en C sería $t0 = $t1 ^ $t2`
  - `nor $t0,$t1,$t2  # o-negada-bit-a-bit entre $t1 y $t2,`
    `# resultado en $t0`
    `# lo que en C sería $t0 = ~($t1 | $t2)`
  - `not $t0,$t1  # no-bit-a-bit de $t1, resultado en $t0`
    `# lo que en C sería $t0 = ~$t1`
- Estas instruciones operan con todos los 32 bits del registro
  - Para las condiciones solo nos interesa el bit 0





CONDICIONES MÁS COMPLEJAS

- Ejemplo

```
if( (($s1>$s2) && ($s1<=$s3)) || ($s4<0)) {
    // se cumplió la condición
    $s1++;
}
// después if
```

```
if:
    sgt     $t0,$s1,$s2   # $t0 = ($s1>$s2)
    sle     $t1,$s1,$s3   # $t1 = ($s1<=$s3)
    and     $t0,$t0,$t1   # $t0 = ($s1>$s2) && ($s1<=$s3)

    slt     $t2,$s4,$zero  # $t2 = ($s4<0)

    or      $t0,$t0,$t2    # $t0 = (($s1>$s2) && ($s1<=$s3)) || ($s4<0)

    beqz    $t0,if_fin     # si $t0 es 0 ⇒ NO se cumplío condición

    # se cumplió la condición
    addi    $s1,$s1,1

if_fin:
    # después if
```

CONDICIONES MÁS COMPLEJAS

- También se puede resolver con saltos condicionales bien elegidos

```
if( (($s1>$s2) && ($s1<=$s3)) || ($s4<0)) {
    // se cumplió la condición
    $s1++;
}
// después if
```

```
# posible solución
# SIN invertir condiciones
if:
    bgt     $s1,$s2,if_mirarS3
    b       if_mirarS4
if_mirarS3:
    ble     $s1,$s3,if_then
if_mirarS4:
    bltz    $s4,if_then
    b       if_fin
if_fhen:
    # se cumplió la condición
    addi    $s1,$s1,1

if_fin:
    # después if
```

```
# posible solución eligiendo
# condición más adecuada
if:
    ble     $s1,$s2,if_mirarS4
    ble     $s1,$s3,if_then
if_mirarS4:
    bgez    $s4,if_fin

if_then:
    # se cumplió la condición
    addi    $s1,$s1,1

if_fin:
    # después if
```