Módulo 10
Programación en Lenguaje Ensamblador (Pt. 1)

Organización de Computadoras
Depto. Cs. e Ing. de la Comp.
Universidad Nacional del Sur

Copyright

Contenidos

- Estructura de un programa
- Las arquitecturas **i386** y **OCUNS**
- Estructuras de control
- Invocación de los servicios del **SO**
- Pasaje de parámetros
- Pila del programa y reentrancia
- Proceso de ensamblado, vinculación y carga

Estructura de programa

- El código fuente, como suele ser el caso en los lenguajes de programación de alto nivel, son en esencia archivos de texto
- Los programas en la arquitectura **OCUNS** constan únicamente de una secuencia de instrucciones
  - Una instrucción puede opcionalmente estar etiquetada para facilitar su posterior referencia
- En otras arquitecturas los programas fuente puede tener una estructuración más compleja

Etiquetas

- En assembler, las etiquetas son básicamente direcciones de memoria
  - En los lenguajes de alto nivel... ¡también!
  - Pueden apuntar a una locación que contiene datos o bien que contiene código

```
        call rutina, R0
           :
rutina: hlt
```

Registros del procesador

- La arquitectura **OCUNS** cuenta con **16** registros de propósito general
  - Recordemos que el registro **F** está “cableado a cero”, es decir, no importa que valor se le asigne siempre termina conteniendo un cero
  - No cuenta con un registro que apunte a la pila
- Entre los registros internos contamos con los tradicionales registros **IR** y **PC**
  - Los registros internos no son accesibles de manera directa por el programador

Registros del procesador

- En contraste, la arquitectura **i386** tiene **8** registros a disposición del programador:
  - **ESP** y **EBP**: suelen tener un rol específico
  - **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI**: son registros de propósito general
- En ambas arquitecturas la memoria funciona como un gran arreglo
  - Se direcciona al byte
  - En el caso de **i386**, con ordenamiento little-endian

Instrucción MOV (CISC)

- Sintaxis: **mov dest, origen**

```
mov ebx, 3    ; guarda un 3 en EBX
mov eax, ebx  ; copia EBX en EAX
mov eax, CONT ; guarda CONT en EAX
```

- Restricciones:
  - El destino no puede ser una constante
  - Se puede hacer referencia de a lo sumo una dirección de memoria
  - Origen y destino tienen que ser compatibles

Instrucción MOV (RISC)

- La arquitectura **OCUNS** no cuenta con una instrucción específica para mover información
- Para mover información desde o hacia memoria usaremos las instrucciones **`load`** y **`store`**
- Para mover información entre registros podemos usar cualquier operación aritmética que nos retorne el resultado deseado
  - En general, resulta práctico usar el registro **`F`** como segundo operando (si es que el cero se comporta como neutro de la operación en cuestión)

Accediendo a memoria (CISC)

- Los corchetes denotan que se debe aplicar un nivel de indirección:

```
mov eax, 5        ; guarda un 5 en EAX
mov ecx, [eax]    ; guarda el contenido de
                  ; la locación 5 en ECX
mov ecx, [5]      ; inst. equivalente
```

- La arquitectura **i386** permite acceder a memoria mediante el siguiente modo:

[base + escala × índice + offset]

Accediendo a memoria (RISC)

- En una arquitectura **RISC** por lo general hay muy pocas instrucciones capaces de referenciar una dirección de memoria completa
- La arquitectura **OCUNS** cuenta con solo una instrucción a tal efecto: **lda**
  - Si bien la intención es cargar una dirección en un registro en preparación a la ejecución de un **load** o de un **store**, puede ser usada para cargar un valor arbitrario

```
lda R5, 80
```

Operaciones (CISC)

- Operaciones aritméticas disponibles:
  - **add destino, operando**
  - **sub destino, operando**
  - **inc destino**
  - **dec destino**
  - **not destino**
  - **neg destino**
  - **mul operando** / **imul operando**
  - **div divisor** / **idiv divisor**

Operaciones (RISC)

- Operaciones aritméticas disponibles:
  - **`add destino, operando, operando`**
  - **`sub destino, operando, operando`**
  - **`inc destino`**
  - **`dec destino`**

Estructuras de control

- Los lenguajes de alto nivel cuentan con diversas estructuras de control para dictaminar el flujo de ejecución de las instrucciones
- El lenguaje ensamblador es más elemental, sólo cuenta con:
  - Los flags del procesador para recordar el resultado de la última operación
  - Las instrucciones de salto para alterar el flujo de ejecución de forma condicionada o no

Principales flags (CISC)

- La arquitectura **i386** cuenta con diversos flags, los cuales reflejan el resultado de la última operación aritmética
  - Para los enteros no signados se usan los flags zero (**ZF**) y carry (**CF**)
  - Para los enteros signados se usan los flags overflow (**OF**) y sign (**SF**)
- Cabe destacar que el resultado de la mayoría de las operaciones afectan a los flags

Principales flags (RISC)

- La arquitectura **OCUNS** también cuenta con diversos flags, si bien por tratarse de una arquitectura **RISC** solo opera con números signados
  - Como se trata de una arquitectura de papel y lápiz, podemos imaginar que cuenta con todos los flags que concibamos, más allá de su utilidad práctica
- Al igual que antes, el resultado de las distintas operaciones afectan a estos flags

Instrucción CMP (CISC)

- Sintaxis: **`cmp primero, segundo`**
- Esta instrucción computa **`primero – segundo`**, modificando los flags de manera acorde.
  - Si resultado = 0 (**`primero = segundo`**):
    `ZF = 1; CF = 0;`
  - Si resultado > 0 (**`primero > segundo`**):
    `ZF = 0; CF = 0; SF = OF;`
  - Si resultado < 0 (**`primero < segundo`**):
    `ZF = 0; CF = 1; SF != OF;`

Instrucción JMP (CISC)

- Sintaxis: **`jmp dest`** / **`jmp short dest`**

  `jmp infinite-loop`

  `jmp short label-cercano`

- Los saltos incondicionales siempre transfieren el control a una cierta dirección de memoria sin tener en cuenta el estado de los flags
  - La instrucción a continuación de un **`jmp`** jamás será ejecutada...
  - ...salvo que sea ¡el destino de un salto!

Saltos condicionales (CISC)

- En los saltos condicionales no siempre se produce la transferencia de control, habida cuenta que el salto se realiza o no dependiendo del estado de uno o más flags
  - Los saltos más sencillos dependen del estado de sólo un flag (no son tan frecuentes)
  - Los saltos más complejos dependen del estado de múltiples flags a la vez (son más frecuentes)

Saltos condicionales (CISC)

- Saltos simples:
  - **`jz dest`** y **`jnz dest`**: dependen de **ZF**
  - **`jo dest`** y **`jno dest`**: dependen de **OF**
  - **`js dest`** y **`jns dest`**: dependen de **SF**
  - **`jc dest`** y **`jnc dest`**: dependen de **CF**
  - **`jp dest`** y **`jnp dest`**: dependen de **PF**
- Si el flag está activo/inactivo se produce el salto
- Caso contrario, la ejecución continúa en la instrucción siguiente al salto condicional

Saltos condicionales (CISC)

- Saltos condicionales para valores signados (se asume que se acaba de ejecutar la instrucción **`cmp eax, ebx`**):
  - **`je dest`**: salta si **`eax = ebx`**
  - **`jne dest`**: salta si **`eax ≠ ebx`**
  - **`jl dest`** y **`jnge dest`**: saltan si **`eax < ebx`**
  - **`jle dest`** y **`jng dest`**: saltan si **`eax ≤ ebx`**
  - **`jg dest`** y **`jnle dest`**: saltan si **`eax > ebx`**
  - **`jge dest`** y **`jnl dest`**: saltan si **`eax ≥ ebx`**

Saltos condicionales (CISC)

- Saltos condicionales para valores no signados (se asume que se acaba de ejecutar la instrucción **`cmp eax, ebx`**):
  - **`je dest`**: salta si **`eax = ebx`**
  - **`jne dest`**: salta si **`eax ≠ ebx`**
  - **`jb dest`** y **`jnae dest`**: saltan si **`eax < ebx`**
  - **`jbe dest`** y **`jna dest`**: saltan si **`eax ≤ ebx`**
  - **`ja dest`** y **`jnbe dest`**: saltan si **`eax > ebx`**
  - **`jae dest`** y **`jnb dest`**: saltan si **`eax ≥ ebx`**

Instrucción JMP (RISC)

- Sintaxis: **`jmp registro`**

  `jmp R0`

- Los saltos incondicionales siempre transfieren el control a la dirección de memoria contenida en el registro indicado
- Esta instrucción se complementa con la invocación a procedimiento (**`call`**), la cual preserva en un registro la dirección a la cuál se debe retornar al finalizar el procedimiento

Saltos condicionales (RISC)

- En el caso de la arquitectura **OCUNS**, los saltos condicionales son mucho más simples:
  - **`jz registro, dest`**: si el valor del registro es cero
  - **`jg registro, dest`**: si el valor del registro es positivo
- Al igual que en la arquitectura **i386**, el salto se produce si la condición es satisfecha, continuando la ejecución en la instrucción inmediata siguiente al salto en caso contrario

Estructuras de control (CISC)

- La estructura de control condicional se puede codificar con facilidad en **i386**:

```
; if (a > 15) { b = 32; } else { a = a + 1; }
mov eax, [a]
cmp eax, 15         ; comparo a con 15
jng else            ; ir a “else” si ≤ 15
mov [b], 32         ; brazo “then”
jmp seguir          ;
else: inc eax       ; brazo “else”
seguir: ...         ; resto del programa
```

Estructuras de control (CISC)

- Las restantes estructuras de control también se pueden codificar de una manera similar
- Otra posibilidad es inspeccionar el código ensamblador generado por un compilador de **C** ante las distintas estructuras de control
  - El parámetro opcional **-S** le solicita al compilador de **C** que no descarte el archivo intermedio conteniendo el código en lenguaje ensamblador

Estructuras de control (RISC)

- La misma estructura de control también se puede capturar en la arquitectura **OCUNS**:

```
; if (a > 15) { b = 32; } else { a = a + 1; }
lda R0, a          ; preparación de los registros
lda R1, b          ; auxiliares para acceso a mem.
lda R2, 15         ; preparación de los registros
lda R3, 32         ; auxiliares usados para simular
                   ; el modo literal
load RA, 0(R0)     ; RA contiene el valor actual
                   ; de la variable a
```

Estructuras de control (RISC)

```
; if (a > 15) { b = 32; } else { a = a + 1; }
  sub R4, RA, R2          ; comparo a con 15
  jg R4, then             ; ir a “then” si a > 15
  inc RA                  ; brazo “else”
  call seguir, RF         ; es un call…
                          ; ¡pero no es un call!
  then: store R3, 0(R1)   ; brazo “then”
  seguir: …               ; resto del programa
```

¿Preguntas?