Arquitectura de las Computadoras

Assembly de la arquitectura Intel x86

Stefano Mazziotta

Índice

La consigna

Sintetizar la arquitectura x86 es un trabajo titánico que el Dr. Paul Carter ha llevado a cabo con sus limitaciones, motivo por el cuál la cátedra no ha optado por su uso para entender los aspectos prácticos de la programación de bajo nivel de procesadores. Sin embargo, dado la masividad de su presencia en el mercado y debido a las enormes diferencias que ostenta con MIPS R2000, se hace necesario establecer un contraste.

La primera parte de la consigna consiste en realizar una breve práctica usando los ejemplos provistos en la máquina virtual y en clase. Se le solicita que reconstruya las sesiones de trabajo con los comandos utilizados para compilar y linkear.

Finalmente debe programar su propio driver en C, para invocar una rutina llamada det escrita en assembly de Intel x86 con el prototipo int det(int a, int b, int c, int d) que devuelva la resta de los productos cruzados a.c – b.d2.

Luego se le pide que revise con detalle todo lo realizado e informe las diferencias sustanciales entre arquitecturas por escrito. Es importante que en esta segunda parte no acopie el material del libro o internet sino que realice sus propias conclusiones. Si dicha tarea tuvo éxito, entonces deberá ser capaz de abordar conceptualmente los siguientes ítems:

- *Diferencias en cantidad y uso obligatorio o no de los registros.*
- *Importancia del registro acumulador.*
- *Cantidad y forma de los operandos en las instrucciones.*
- *Diferencias en las instrucciones aritméticas.*
- *Diferencias en cantidad y modos de direccionamiento. Modo implícito.*
- *Complejidad del formato de las instrucciones.*
- *Importancia de los FLAGS y su análogo en MIPS (ALU).*
- *Diferencias de evaluación de condiciones y su implicación en los saltos.*
- *Diferencias en el manejo de pila.*
- *Diferencias en la gestión de subrutinas.*

Sus conclusiones deben estar justificadas a través de la ejemplificación mediante código de la práctica. A pesar de su simplicidad, esta práctica tiene un alcance conceptual muy importante y vasto. Estos interrogantes bien pueden encontrarse en la extensa literatura que existe sobre el tema, pero sus respuestas debieran surgir sólo del análisis del código utilizado y del libro de cátedra.

Código

det.asm

```
%include "asm_io.inc"
segment .text
    global cal_det

cal_det:
    ; int a, int b, int c, int d -> int result
    enter 0, 0        ; Crea el stack frame

    mov eax, [ebp+8]      ; eax = a
    imul eax, [ebp+20]    ; eax = a * d

    mov ebx, [ebp+12]     ; ebx = b
    imul ebx, [ebp+16]     ; ebx = b * c

    sub eax, ebx          ; eax = a*d - b*c - eax valor de retorno int

    leave                 ; Limpia el stack frame
    ret                   ; Retorna con el resultado en EAX
```

det.c

```c
#include <stdio.h>
int calc_det(int, int, int, int) __attribute__((cdec1));

int main (void){
  int a, b, c, d;
  int result;

  printf("A: ");
  scanf(%d, &a);

  printf("B: ");
  scanf(%d, &b);

  printf("C: ");
  scanf(%d, &c);

  printf("D: ");
  scanf(%d, &d);

  result = calc_det(a, b, c, d);
  printf("Det is %d\n", result);
  return 0;
}
```

Scripts de compilación

```
nasm -f elf det.asm
gcc -o finaldet det.c det.o asm_io.o
```

Diferencias en cantidad y uso obligatorio o no de los registros

Intel x86

Registros Generales	**Segmentos de registros**	**Registros de estado**	**Instruction Pointer**
EAX (AX, AH, AL)	CS (code segment)	EFLAGS	EIP
EBX (BX, BH, BL)	SS (stack segment)		
ECX (CX, CH, CL)	DS (data segment)		
EDX (DX, DH, DL)	ES (extra segment)		
EBP (BP) base pointer	FS		
ESP (SP) stack pointer	GS		
ESI (SI)			

Cantidad de registros

En comparación con MIPS, intel x86 posee menor cantidad de registros, esto implica una menor flexibilidad en la manipulación de los datos en tiempo de ejecución, dando lugar, un mayor uso del stack pointer y la técnica de desplazamiento.

Otro factor diferencial, es la **dirección de retorno**. En intel x86 se gestiona desde el stack *[esp]*, en cambio, en MIPS se utiliza un registro dedicado *$ra*.

Uso obligatorio de registros

Mediante la convención cdecl, las llamadas a funciones utilizan los siguientes registros:

EAX: Almacena el valor de retorno, acumulador.
ESP: El registro ESP siempre apunta a la parte superior de la pila. Cada vez que se agrega (push) o se retira (pop) un dato, ESP se actualiza automáticamente.
EBP: apuntar al comienzo del stack frame de una función. Esto ayuda a acceder a parámetros y variables locales de manera ordenada.

Importancia del registro acumulador

En x86, el registro EAX (Extended Accumulator) se usa en operaciones aritméticas y en llamadas a funciones (retorno de valores).

En MIPS, no existe un registro acumulador estricto, y el valor de retorno se almacena en el registro $v0, sin imponer restricciones en operaciones generales.

Cantidad y forma de los operandos en las instrucciones

x86: Permite de 0 a 3 operandos, depende de la instrucción dada (instrucción destino origen). Las formas de los operandos, son:

- **Registro:** Ej. EAX, EBX, ECX, etc.
- **Memoria:** Acceso a memoria directa o indirecta con desplazamiento.
- **Inmediatos:** valores constantes.

MIPS: Suelen tener 3 operandos (*instrucción destino, origen1, origen2*). La diferencia a destacar es que el acceso a memoria no es directo ya que MIPS es una arquitectura load/store, es decir, en caso de manipular un dato en memoria, se debe cargar en un registro (*lw*), operar y luego almacenarlo para impactar el cambio (*sw*).

Diferencias en las instrucciones aritméticas

Intel x86: Permite operar directamente con memoria y registros.
Ejemplo: `imul eax, [ebp+20]`

MIPS: No permite operaciones aritméticas directas con memoria.

```
# Asignar un ID incrementado al nuevo nodo
 lw          $t3, 4($t2)          # $t7 = id del último nodo
 addi        $t3, $t3, 1          # $t3 = nuevo id
 sw          $t3, 4($s6)          # nuevo->id = $t4
```

Diferencias en cantidad y modos de direccionamiento. Modo implícito.

La arquitectura CISC de Intel x86 ofrece una amplia variedad de modos de direccionamiento.

La principal diferencia es el direccionamiento directo, donde una instrucción acepta como origen un valor proveniente de memoria, sin previamente cargarlo en un registro, lo cual es impensado en MIPS.

Respecto al modo implícito, Intel x86 posee instrucciones que operan de forma implícita.
Por ejemplo, en base a lo desarrollado utilizamos:

```
leave                   ; Limpia el stack frame
ret                     ; Retorna con el resultado en EAX
```

¿Qué hace **LEAVE**?
Restablece el puntero de pila (ESP) y el puntero base (EBP).
¿Qué hace **RET**?
Extrae la dirección de retorno de la pila y la carga en el puntero de instrucción (EIP).
En estos casos no necesitan especificar registros o direcciones.

MIPS no posee el modo implícito ya que es una arquitectura RISC.
Los modos de direccionamiento que comparten entre ambas arquitecturas son:

- Inmediato.
- Registro.
- Base+Desplazamiento.

Complejidad del formato de las instrucciones

Desde el punto de vista del programador, me resulta compleja la lectura de las instrucciones que proporciona Intel x86 por la resolución implícita de los argumentos.
Un ejemplo básico es la suma.
x86:

```
MOV EAX, [var1]  ;Cargar var1 en EAX
ADD EAX, [var2]  ;Sumar var2 al valor en EAX
```

MIPS:

```
LW $t0, var1  ;Cargar var1 en el registro $t0
LW $t1, var2  ;Cargar var2 en el registro $t1
ADD $t2, $t0, $t1 ; Sumar $t0 y $t1, y guardar el resultado en $t2
```

Si bien, MIPS posee mayor líneas de código, es más explícito que x86 reduciendo su complejidad de lectura.

Importancia de los FLAGS y su análogo en MIPS (ALU)

En x86, los **flags** permiten tomar decisiones basadas en el estado de la CPU.

En el contexto de **Intel x86 de 32 bits**, los flags más comunes son:

- **ZF (Zero Flag)**: Indica si el resultado de una operación es cero.
- **SF (Sign Flag)**: Indica el signo del resultado.
- **OF (Overflow Flag)**: Señala si ocurrió un desbordamiento en una operación aritmética.
- **CF (Carry Flag)**: Indica si hubo un acarreo o préstamo en operaciones aritméticas.

Mientras que en MIPS, se usa la lógica de saltos condicionales (beq, beqz, bne) sin un registro explícito de flags. Estos saltos condicionales están basados en los resultados de las operaciones realizadas por la ALU.

Diferencias de evaluación de condiciones y su implicación en los saltos

- **x86**:
    - Dependencia de los **FLAGS** para evaluar condiciones.
    - Evaluación implícita de las condiciones.
    - Mayor dependencia en el estado global del procesador.
    - Introduce cierta complejidad al depender del estado global del procesador y los **FLAGS**, lo que puede llevar a comportamientos menos predecibles y más difíciles de depurar.
- **MIPS**:
    - Evaluación explícita mediante instrucciones de comparación directa de registros.
    - Flujo de control más claro y predecible.
    - Menos dependencia del estado global del procesador.
    - Al hacer las comparaciones explícitas y centrarse en los registros, ofrece un control más transparente sobre los saltos, lo que hace que el código sea más fácil de seguir y depurar.

Diferencias en el manejo de pila

x86:

- **Instrucciones dedicadas a la pila**:
  - **PUSH**: Empuja un valor en la pila y ajusta automáticamente el puntero de pila (ESP).
  - **POP**: Extrae un valor de la pila y ajusta automáticamente el puntero de pila.
  - **CALL**: Llama a una función, empuja la dirección de retorno en la pila, y ajusta automáticamente el puntero de pila.
  - **RET**: Retorna de una función, recuperando la dirección de retorno de la pila y ajustando el puntero de pila.
- **Automatización**: El puntero de pila (ESP) se actualiza automáticamente con estas instrucciones, lo que simplifica la gestión de la pila y reduce la necesidad de manipulación manual.

MIPS:

- **Manipulación manual**: Las instrucciones que se utilizan para trabajar con la pila en MIPS son:
  - **SW (Store Word)**: Almacena un valor en la memoria, que se utiliza para guardar datos en la pila.
  - **LW (Load Word)**: Carga un valor desde la memoria, que se utiliza para obtener datos de la pila.
- **Control del puntero de pila**: En MIPS, el puntero de pila (generalmente en **$sp**) debe ser gestionado manualmente. Las instrucciones de la pila no modifican automáticamente el puntero, por lo que el programador debe ajustarlo explícitamente antes y después de cada operación (por ejemplo, moviendo **$sp** hacia abajo para reservar espacio en la pila o hacia arriba para liberar espacio).
- **Flexibilidad y complejidad**: MIPS proporciona mayor flexibilidad en la manipulación de la pila, también requiere un manejo más explícito y cuidadoso, lo que puede aumentar la complejidad y la probabilidad de errores si no se gestiona correctamente.

Diferencias en la gestión de subrutinas

x86: Utiliza **CALL** para llamar a una subrutina y **RET** para regresar. **CALL** guarda automáticamente la dirección de retorno en la pila. En el caso de subrutinas anidadas, cada **CALL** guarda una nueva dirección de retorno en la pila sin sobrescribir las anteriores. Al ejecutar **RET**, la dirección más reciente se recupera para regresar correctamente a la subrutina llamadora, sin necesidad de intervención manual.

MIPS: Usa **JAL** (Jump and Link) para guardar la dirección de retorno en el registro **$ra** y **jr $ra** para regresar. Si hay subrutinas anidadas, el programador debe gestionar manualmente el valor de **$ra** para evitar sobreescrituras.