TM: Una máquina objetivo simple

En el libro de texto [Louden, 2004], el autor y su equipo escribieron un simulador (emulador) de una máquina muy simple (el código en C del simulador está disponible en el apéndice del libro), llamada TM (por *Tiny Machine*) la cual se programa en ensamblador. El ensamblador de esta máquina, como todos los ensambladores, es especial para su procesador ya que trabaja para una arquitectura especial. Es necesario que conozcamos dicha arquitectura con propósito de poder hacer programas que funcionen correctamente en esa máquina. A continuación se explica la arquitectura y el lenguaje ensamblador.

Arquitectura básica de la máquina TINY

TM se compone de una memoria de instrucción sólo de lectura, un& memoria de datos, y un conjunto de ocho registros de propósito general. Todos éstos utilizan direcciones enteras no negativas que comienzan en 0. El registro 7 es el contador del programa y es el único registro especial, como se describe a continuación. Las declaraciones en C

```
#define IADDR_SIZE . . .
        /* tamaño de la memoria de la instrucción */
#define DADDR_SIZE ...
        /* tamafio de la memoria de datos */
#define NO_REGS 8 /* número de registros */
#define PC_REG 7

Instruction iMam[IADDR_SIZE];
int dMem[DADDR_SIZE];
int reg[NO_REGS];
```

se utilizarán en las descripciones que siguen.

La TM realiza un ciclo convencional de obtención-e.jecución:

```
do
  /* obtención */
  currentInstruction = iMem[reg[pcRegNo]++];
  /* ejecuta la instrucción actual */
  ...
while (!(halt || error));
```

Al comienzo, la máquina TINY establece todos los registros y memoria de datos a 0, luego carga el valor de la dirección legal más alta (a saber, **`DADDR-SIZE-1`**) en **`dMem[0].`** (Esro permite que la memoria se agregue fácilmente a la TM, puesto que los programas pueden averiguar durante la ejecución de cuánta memoria se dispone.) La TM comienza entonces a ejecutar las instrucciones comenzando en **`iMem[0].`** La máquina se detiene cuando se ejecuta una instrucción **`HALT.`** Las condiciones de error posibles incluyen a **`IMEM_ERR,`** que se presenta si **`reg[PC_REG]<0`** o si **`reg[PC_REG] ≥ IADDR_SIZE`** en el paso de

obtención anterior, y las dos condiciones **DMEM_ERR** y **ZERO_DIV,** las cuales se presentan durante la ejecución de instrucciones como se describió anteriormente.

El conjunto de instrucciones de la TM se proporciona en la figura 8.15, junto con una breve descripción del efecto de cada instrucción. Existen dos formatos de instrucciones básicos: sólo de registro, o instrucciones RO y de memoria de registro, o instituciones RM. Una instrucción s6lo de registro tiene el formato:

```
opcode r, s, t
```

Figura 8.15
Conjunto completo de instrucciones de la máquina TINY

Instrucciones RO

Formato:	*opcode r,s,t*
Opcode	Efecto
HALT	detener ejecución (operandos ignorados)
IN	reg[r] ← lectura de valor entero desde la entrada estándar (*s* y *t* ignorados)
OUT	reg[r] → la salida estándar (*s* y *t* ignorados)
ADD	reg[r] = reg[s] + reg[t]
SUB	reg[r] = reg[s] - reg[t]
MUL	reg[r] = reg[s] * reg[t]
DIV	reg[r] = reg[s] / reg[t] (puede generar ZERO_DIV)

Instrucciones RM

Formato: *opcode r,d(s)*

(*a* = *d* + reg[*s*]; cualquier referencia a dMem[*a*] genera DMEM_ERR si *a* < 0 o *a* ≥ DADDR_SIZE)

Opcode	Efecto
LD	reg[r] = dMem[a] (carga *r* con valor de la memoria en *a*)
LDA	reg[r] = *a* (carga dirección *a* directamente en *r*)
LDC	reg[r] = *d* (carga constante *d* directamente en *r*—*s* es ignorada)
ST	dMem[a] = reg[r] (almacena valor en *r* a localidad de memoria *a*)
JLT	if (reg[r]<0) reg[PC_REG] = *a* (salta a instrucción *a* si *r* es negativa, de manera similar para el siguiente)
JLE	if (reg[r]<=0) reg[PC_REG] = *a*
JGE	if (reg[r]>=0) reg[PC_REG] = *a*
JGT	if (reg[r]>0) reg[PC_REG] = *a*
JEQ	if (reg[r]==0) reg[PC_REG] = *a*
JNE	if (reg[r]!=0) reg[PC_REG] = *a*

donde los operandos **r, s, t** son registros legales (verificados en tiempo de carga). De este modo, tales instrucciones son tres direcciones, y las tres direcciones deben ser registros. Todas las instrucciones aritméticas se encuentran limitadas a este formato, como lo están las dos instrucciones primitivas de entrada/salida.

Una instrucción de memoria de registro tiene el formato

```
opcode r, d(S)
```

En este código ***r*** y **s** deben ser registros legales (verificados en tiempo de carga), y **d** es un entero positivo o negativo que representa un desplazamiento. Esta instrucción es una instrucción de dos direcciones, donde la primera dirección es siempre un registro y la segunda dirección es una dirección de memoria **a** dada por **a = d + reg[r]**, donde **a** debe ser una dirección legal (**0 ≤ a < DADDR_SIZE**). Si **a** está fuera del intervalo legal, entonces se genera **DMEM_ERR** durante la ejecución.

Las instrucciones RM incluyen tres instrucciones de carga diferentes que corresponden a los tres modos de direccionamiento "carga constante" (**LDC**), "dirección de carga" (**LDA**) y "memoria de carga" (**LD**). Además, existe una instrucción de almacenamiento y seis instrucciones de salto condicional.

Tanto en las instrucciones *RO* como en las RM, los tres operandos deben estar presentes, aun cuando algunos de ellos pueden ser ignorados. Esto se debe a la naturaleza simple del cargador, el cual sólo distingue las dos clases de instrucciones (RO y RM) y no permite diferentes formatos de instrucción dentro de cada clase.

La figura 8.15 y el análisis de la TM hasta este punto representan la arquitectura completa TM. En particular, no hay pila de hardware u otras facilidades de ninguna clase. Ningún registro, excepto el pc, es especial en modo alguno (no hay sp o fp). Un compilador para la TM debe, por lo tanto, mantener alguna organización del ambiente de ejecución de manera enteramente manual. Aunque esto puede ser algo poco realista, tiene la ventaja de que todas las operaciones se deben generar explícitamente a medida que sean necesarias.

Debido a que el conjunto de instrucciones es mínimo, algunos comentarios están dispuestos acerca de cómo se puede utilizar para conseguir casi todas las operaciones estándar de un lenguaje de programación (en realidad ésta es una máquina objetivo adecuado, si no es que hasta cómodo incluso para lenguajes muy sofisticados).

1. El registro objetivo en las operaciones aritméticas, **IN,** y de carga viene en primer lugar, y el (los) registro(s) fuente vienen en segundo, de manera similar a la arquitectura **80x86** y diferente a la de la SparcStation de Sun. No hay restricción sobre el uso de los registros para fuentes y objetivos; en particular, los registros fuente y objetivo pueden ser los mismos.
2. Todas las operaciones aritméticas están restringidas a registros. Ninguna operación (excepto las operaciones de carga y almacenamiento) actúa directamente en la memoria. En esto, la TM se parece a las máquinas RISC, tales como la SparcStation de Sun. Por otra parte, la TM tiene sólo 8 registros, mientras que la mayoría de los procesadores RISC tienen por lo menos 32.
3. No hay operaciones de punto flotante o registros de punto flotante. Aunque no sería muy difícil agregar un coprocesador a la TM con registros y operaciones de punto flotante, la traducción de los valores de punto flotante hacia y desde los registros regulares y la memoria requerirían de alguna precaución. Remitiremos al lector a los ejercicios.
4. No existen modos de direccionamiento especificables en los operandos como en algún código ensamblado (tal como **LD #1** para modo inmediato, o **LD @a** para indirecto). En vez de eso existen diferentes instrucciones para cada modo: **LD** es indirecto, **LDA** es directo y **LDC** es inmediato. En realidad, la TM tiene muy pocas opciones de direccionamiento.

5. No existe restricción para el uso del pc en ninguna de las instrucciones. Efectivamente, puesto que no hay instrucción de salto incondicional, ésta debe ser simulada mediante el uso del pc como el registro objetivo en una instrucción **LDA**:

```
LDA 7, d(s)
```

   Esta instrucción tiene el efecto de saltar a la localidad **a = d + reg [s].**

6. Tampoco hay instrucción de salto indirecto, pero también se puede imitar, si es necesario, mediante el uso de una instrucción **LD.** Por ejemplo,

```
LD 7,0(1)
```

   salta a la instrucción cuya dirección está almacenada en la memoria en la localidad a la que apunta el registro 1.

7. Las instrucciones de salto condicional **(JLT,** etc.) pueden hacerse relativas a la posición actual en el programa mediante el uso del pc como el segundo registro. Por ejemplo,

```
JEQ 0,4(7)
```

   provoca que la TM salte cinco instrucciones hacia delante en el código si el registro 0 es 0. Un salto incondicional también puede hacerse relativo al pc utilizando dos veces el pc en una instrucción **LDA**. De este modo,

```
LDA 7, -4(7)
```

   realiza un salto incondicional de tres instrucciones hacia atrás.

8. No existe llamada de procedimiento o instrucción **JSUB.** En su lugar debemos escribir

```
LD 7,d(s)
```

   que tiene el efecto de saltar al procedimiento cuya dirección de entrada es **dMem[d+reg[a]].** Naturalmente, debemos recordar grabar la dirección de retorno ejecutando primero algo como

```
LDA 0,1(7)
```

   que coloca e1 valor actual del pc más uno en **reg[0]** (que es a donde queremos regresar, suponiendo que la instrucción siguiente es el salto real al procedimiento).

El simulador de TM

El simulador de máquina acepta archivos de texto que contienen instrucciones de TM de la manera en que se describieron anteriormente, con las convenciones siguientes:

1. Una línea completamente en blanco se ignora.
2. Una línea que comienza con un asterisco se considera un comentario y se ignora.
3. Cualquier otra línea debe contener una localidad de instrucción entera seguida por un signo de dos puntos seguido por una instrucción legal. Cualquier texto que se presente después de la instntcci6n ***se*** considera un comentario y se ignora.

El simulador de TM no contiene otras características, en particular no hay etiquetas simbólicas ni facilidades de macro. En la figura 8.16 se proporciona un programa de muestra TM escrito manualmente que corresponde al programa TINY de la figura 8.1.

Estrictamente hablando, la instrucción **HALT** al final del código de la figura 8.16 no es necesaria, puesto que el simulador de TM establece todas las localidades de instrucción a **HALT** antes de cargar el programa. Sin embargo, es útil conservarla como un recordatorio, y como un objetivo para saltos que deseen salir del programa.

Figura 8.16
Un programa TM que muestra las convenciones de formato

```
* Este programa introduce un entero, calcula
* su factorial si es positivo,
* e imprime el resultado

0:     IN   0,0,0       r0 = read
1:     JLE  0,6(7)      if 0 < r0 then
2:     LDC  1,1,0         r1 = 1
3:     LDC  2,1,0         r2 = 1
                        * repite
4:     MUL 1,1,0            r1 = r1 * r0
5:     SUB 0,0,2            r0 = r0 - r2
6:     JNE  0,-3(7)       until r0 == 0
7:     OUT  1,0,0         write r1
8:     HALT 0,0,0       halt
* fin del programa
```

Tampoco es necesario que aparezcan localidades en secuencia ascendente como hicimos en la figura 8.16. Cada línea de entrada es efectivamente una directiva del tipo "almacena esta instrucción en esta localidad": si un programa TM fuera perforado en tarjetas, sería aceptable tirarlas y que se desordenaran en el piso antes de leerlas en la TM. Aunque esta propiedad del simulador de TM podría causar alguna confusión al lector de un programa, facilita ajustar saltos cuando no hay etiquetas simbólicas, ya que el código se puede ajustar sin apoyarse en el archivo del código. Por ejemplo, es probable que un generador de código genere el código de la figura 8.16 en la secuencia que se muestra a continuación:

```
0:     IN   0,0,0
2:     LDC  1,1,0
3:     LM   2,1,0
4:     MUL  1,1,0
5:     SUB  0,0,2
6:     JNE  0,-3(7)
```

```
7:     OUT  1,0,0
1:     JLE  0,6(7)
8:     HALT 0,0,0
```

Esto se debe a que el salto hacia adelante en la instrucción 1 no se puede generar hasta que se conozca la ubicación después del cuerpo de la sentencia if.

Si el programa de la figura 8.16 está en el archivo **`fact.tm`**, entonces este archivo se puede cargar y ejecutar como en la siguiente sesión de muestra (el simulador de TM automáticamente presupone una extensión de archivo **`.tm`** si no se proporciona alguna otra):

```
tm fact
TM simulation (enter h for help) ...
Enter command: g
Enter value for IN instruction: 7
OUT instruction prints: 5040
HALT: 0,0,0
Halted
Enter command: q
Simulation done.
```

El comando g representa "ir a", lo que significa que el programa se ejecuta comenzando en el contenido actual del pc (que es 0 justo después de 13 carga), hasta que se detecta una instrucción **`HALT.`** La lista completa de comandos del simulador se puede obtener mediante el uso del comando **`h`**, el cual imprime la lista que se muestra a continuación:

```
Los comandos son:*
   s(tep <n>          Execute n (default 1) TM instructions
   g(o                Execute TM instructions until HALT
   r(egs              Print the contents of the registers
   i(Mem <b <n>>      Print n iMem locations starting at b
   d(Mem <b <n>>      Print n dXem locations starting at b
   t(race             Toggle instruction trace
   p(rint             Toggle print of total instructions executed
                      ('go' only)
   c(lear             Reset simulator for new execution of grogram
   h(elp              Cause this list of commands to be printed
   q(uit              Terminate the simulation
```

El paréntesis izquierdo en cada comando indica el mnemónico a partir del cual se deriva la letra del comando (también es aceptable utilizar más de una letra, pero el simulador sólo examinará la primera). Los picoparéntesis < > señalan los parámetros opcionales.