MINI VM

Rodríguez Jeremías
R-3887/3

TRABAJO FINAL - ARQUITECTURA DEL COMPUTADOR

6 de febrero de 2016

Índice

1. Introducción

El trabajo consiste en la implementación de una máquina abstracta, modificando el código provisto por la cátedra.

En este informe se detallará la estructura de la máquina y su implementación en C; incluyendo una especificación de sus instrucciones y dificultades con las que me encontré al implementarlas.

Luego se describen algunas extensiones que realicé, se muestran ejemplos de testeo y finalmente se proponen nuevas extensiones.

2. Mini VM

2.1. Estructura General

La máquina abstracta tendrá como objetivo ejecutar una secuencia de instrucciones (programa), donde todos los operandos son enteros de 32 bits. Esta VM consiste en:

- 8 registros de 32 bits: ZERO (siempre vale cero), PROGRAM COUNTER (linea del programa que se está ejecutando), STACK POINTER (tope de la pila), R0, R1, R2, R3 (propósito general) y RFLAGS (banderas de estado).
- Una única memoria,que puede ser leída y escrita mediante algunas instrucciones, y está formada por 1024 celdas de 8 bits (numeradas de 0 a 1023). En las celdas más altas se guardará una pila.
  El formato de almacenamiento en memoria es little endian. Por ejemplo, si se pushea el entero 239054 en la pila vacía, representado con el binario 00000000 0100100 01111010 00010001, se obtiene:

  memory [1020] = 00010001 (bit 0)
  memory [1021] = 01111010 (bit 1)
  memory [1022] = 0100100 (bit 2)
  memory [1023] = 00000000 (bit 3)

- Un juego de instrucciones (detallado en la sección siguiente) que consta de instrucciones de movimiento/carga de datos , aritméticas ,entrada/salida, comparaciones, saltos, y de manejo de pila. Adicionalmente, agregué instrucciones que manejan bits, llamados a funciones y bucles.

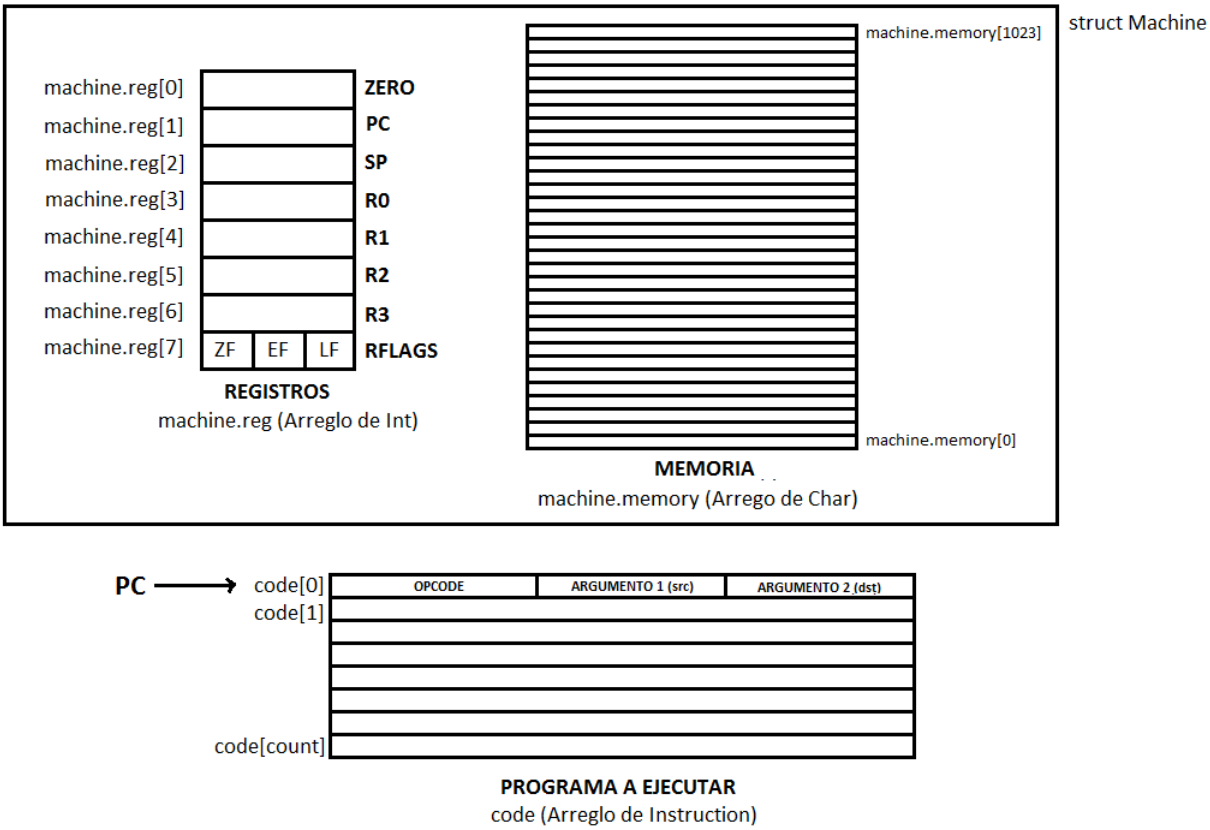
2.2. Implementación en C

En machine.h podemos ver los tipos de datos utilizado. Básicamente, una estuctura Machine está compuesta por un arreglo de bytes (memoria) y un arreglo de enteros (registros).

El código del programa entrada, luego de ser parseado, se guarda en un arreglo de instrucciones code; donde la posición i del arreglo guarda la instrucción de la línea i-ésima.

Una instrucción es una estructura compuesta de un Opcode (ADD, MOV, etc) y dos operandos (src y dst).

Un operando es una estructura que guarda un valor, un char y tiene un tipo (IMM: inmediato, MEM: memoria, REG: registro o LABELOP: etiqueta)



2.3. Funcionamiento

Para llevar a cabo la ejecución de un programa, la máquina realizará los siguientes pasos:

1. Parsear la entrada, que será un archivo de texto .asm con una lista de instrucciones, como por ejemplo:

   ```
   mov $0, %r0
   mov $7, %r1
   add %r0, %r1
   push %r1
   hlt
   ```

   El parser se encargará de traducir este archivo de texto en un arreglo de instrucciones.

2. La función processLabels modifica este arrego de instrucciones, reemplazando todas las etiquetas por los valores numericos que representan. (ver funciones JMP, CALL, etc).

3. Seteando el tope de la pila en la dirección de memoria más alta, y el contador de programa en la línea 0, comienza la ejecución del programa del siguiente modo:

Se ejecuta la instrucción de la línea PC, modificando adecuadamente (según el tipo de instrucción) el contador; y repetir hasta llegar a una instrucción hlt. De esto se encarga la función run.

Ejecutar cada instrucción consistirá en modificar entradas en registros o en memoria, de acuerdo al código de la instrucción y sus argumentos. De esto se encarga la función runIns.

4. Se muestra el estado de los registros y de memoria una vez terminada la ejecución del programa.

2.4. Compilación y Ejecución

El primer problema que tuve fue que machine.c utilizaba la entrada estándar para leer el programa. Esto ocasionaba un problema al querer implementar la instrucción read. En Análisis de Lenguajes de programación, cuando estudiamos parsers, vimos utilidades similares a Bison y lex para general parsers en Haskell.

La solución fue leer un poco sobre Bison y Flex. En la referencia [1] se describe la variable yyin. La utilicé para que el programa .asm a ejecutar fuera pasado como argumento a main; y parseado utilizando yyin desde el archivo.

Luego, para ejecutar ejemplo.asm en la máquina virtual, debemos teclear ./machine ejemplo.asm en la consola.

3. Juego de Instrucciones

A continuación describiré las distintas instrucciones y su comportamiento.

En general, para implementarlas, tuve que hacer análisis por casos en el tipo de sus argumentos; utilizando la funcion raise cuando era erróneo para manejar el error.

La instrucción **HLT** indica el fin de programa.

3.1. Instrucciones de Carga/Almacenamiento

Las siguientes instrucciones son las únicas que pueden acceder a memoria. Su función es copiar datos desde registro/memoria/valor inmediato hacia un registro/memoria.

Para mover datos desde/hacia memoria, dado que un registro o valor inmediato es de 4 bytes y una celda de memoria es de 1 byte, utilizo una función copy4bytes que realiza la copia correctamente.

MOV src dst : Copia src en dst. No pueden estar simultáneamente en memoria.

```
mov $45235, %r0     # Guarda en el registro r0 el valor 45235
mov %r0, %r1        # Copia el valor de r0 en r1 (r1 <- 45235)
mov %r1, 9          # Guarda a partir de la celda de memoria 9 el valor contenido en r1.
hlt
```

SW src dst : guarda el valor del registro src en la dirección apuntada por dst (dst puede ser MEM, o un REG que guarda una dirección de memoria).

```
mov $45235, %r0     # Guarda en el registro r0 el valor 45235
sw %r0, 7           # Guarda el valor de r0 a partir de la celda 7
mov $245, %r1
sw %r0, %r1         # Guarda el valor de r0 a partir de la celda 245
hlt
```

LW src dst : Si src es de tipo MEM, guarda en el registro dst el int guardado en esa dirección. Si src es de tipo REG, guarda en el registro dst el int guardado en la dirección apuntada por ese registro.

```
mov $145, 5
lw 5, %r0           # r0 <- 145
mov $5, %r1
lw %r1, %r2         # r2 <- 145
hlt
```

3.2. Instrucciones Aritméticas

Las instrucciones aritméticas reciben sus argumentos y realizan alguna operación, guardando el valor resultante en dst. src puede ser un valor inmediato o registro, y dst debe ser un registro.

A continuación enumero las instrucciones y un equivalente en C, las instrucciones INC y DEQ las incorporé para mayor comodidad (no estaban en el enunciado original).

ADD src dst : dst += src
MUL src dst : dst *= src
DIV src dst : dst = div dst src
SUB src dst : dst -= src
CMP src dst : Realiza una resta ficticia (SUB), modificando el registro RFLAGS pero no dst.
INC src : src ++
DEC src : src –

3.3. Instrucciones de Manejo de Pila

Permiten apilar/desapilar enteros (guardados en registros/o valores inmediatos) en la pila, que crece hacia las direcciones más bajas de memoria. Utilizo nuevamente la función que copia 4 bytes.

```
push $10     // apila el entero 10
push $24     // apila el entero 24
push $-1452  // apila el entero -1452 ; pila = -1452 ; 24 ; 10
pop %r1      // r1 <- 1452    pila = 24 ; 10
pop %r1  // r1 <- 24      pila = 10
pop %r1      // r1 <- 10      pila = empty
pop %r1      // lanza un error pila vacía
hlt
```

3.4. E/S

Las instrucción print imprime su argumento por pantalla. La instrucción read lee un entero y lo guarda en el registro argumento.

```
mov $14, %r1
mov $1556, 12
print $15           # Imprime 15
print %r1           # Imprime 14
print 12            # Imprime 1556
read %r1            # Lee un entero y lo guarda en r1
```

```
print %r1           # Lo imprime
hlt
```

3.5. Saltos

Análogamente a las instrucciones que vimos en x8664, la instrucción jmp "salta"(cambia el flujo del programa) a una etiqueta. jmpe y jmpl son saltos condicionales, de acuerdo a las flags equal y lower, respectivamente.

La implementación fue sencilla pues, en el código provisto por el enunciado, una función llamada processLabels modifica todos los argumentos que son etiquetas, reemplazándolos por los enteros correspondientes a las líneas de las etiquetas (valores IMM).

Luego, lo único que tuve que hacer fue definir estas funciones para argumentos del tipo IMM; cambiando el PC apropiadamente.

```
mov $23, %r1    # r1 <- 23
dec %r1         # r1 <- 22
cmp $23, %r1     # "22 - 23" prende la bandera L
jmpl menor      # Como L esta prendida, salto
print $25    ## No se realiza
menor:
print $0        # Imprimo 0
jmp fin         # Salto
print $1     ## No se realiza
fin:
hlt             # Fin de programa
```

4. Extensiones

Las extensiones que realicé consisten en agregar algunas instrucciones más al conjunto de instrucciones de la máquina. Esto requirió extender el parser, los tipos de datos y las distintas funciones de la máquina para considerar estas instrucciones nuevas.

Utilizando como modelo las instrucciones ya implementadas, extender el parser fue muy sencillo. También las funciones de machine.c, donde la única complicación que encontré fue extender processLabels.

4.1. Instrucciones de Desplazamiento de Bits

Estas instrucciones reciben en src un desplazamiento (guardado en un registro, o como valor inmediato); y un registro destino.

La implementación no resultó muy complicada utilizando operaciones de bits, máscaras en caso de las rotaciones. El problema que se me presentó es que no sabía exáctamente si el shift de C, era un shift aritmético o lógico. Para solucionar esto, utilicé casteos a unsigned cuando fue necesario.

SHR src dst : desplaza src bits a la derecha el elemento dst, rellenando con ceros
SHL src dst : desplaza src bits a la izquierda el elemento dst, rellenando con ceros
SAR src dst : desplaza src bits a la derecha el elemento dst, rellenando con el bit de signo
SAL src dst : exáctamente igual a SHL (Wikipedia)
ROR src dst : Rota src bits a la derecha a dst. (análogo al cursado)

ROL src dst : Rota src bits a la izquierda a dst. (análogo al cursado)

4.2. Instrucciones Lógicas

Análogo a las instrucciones aritméticas, reciben dos argumentos src (valor inmediato o registro) y dst (registro); y realizan una operación bit a bit.

La implementación es análoga a la suma, resta, etc.

AND src dst : dst <- dst & src
OR src dst : dst <- dst | src
XOR src dst : dst <- dst ^ src
NOT dst : dst <- dst

4.3. Call y Ret

Las instrucciones call y ret nos permiten implementar llamado a funciones. El funcionamiento es el que vimos en en clase.

Ejecutar una instrucción call (seguida de una etiqueta), es pushear en la pila la dirección de retorno (PC+1) y cambiar el flujo del programa (saltar).

Cuando se ejecuta una instrucción ret, simplemente se realiza un pop (suponiendo que, previamente, se había pusheado una dirección de retorno) y se salta a la dirección obtenida de la pila.

```
# FACTORIAL RECURSIVO
read %r1     # Leo un entero
call fact    # Llamo a factorial. El argumento es %r1
print %r1    # Imprimo el resultado
hlt

fact:        # Argumento en r1. Retorno en r1.
cmp $0, %r1  # fact (0) = 1
jmpe cero
cmp $1, %r1  # fact (1) = 1
jmpe uno
push %r1
dec %r1
call fact     # r1 <- (n-1)!
pop %r2       # r2 <- n
mul %r2, %r1  # r1 <- n * (n-1)!
ret

cero:
mov $1, %r1
uno:
ret
```

4.4. Loop

La instrucción LOOP es análoga a la instrucción LOOP que vimos en x86. El argumento de LOOP es una etiqueta.

Cuando se ejecuta una instrucción LOOP, primero se decrementa en 1 el valor del registro r3. Luego, si r3 vale cero, se continúa ejecutando la siguiente instrucción (no se hace nada). SI r3 no es cero, se salta a la etiqueta argumento.

Nos sirve para implementar bucles.

```
# FACTORIAL ITERATIVO
read %r1
call fact
print %r1
hlt

fact:          # argumento: %r1.  retorno: %r1
cmp $0, %r1
jmpe cero      #  fact (0) = 1

mov %r1, %r3
mov $1, %r1    # ans <- 1  , acumulador

rep:           # for (i=n ; i>0 ; i--) { ans = ans * i }
mul %r3, %r1
loop rep
ret

cero:
mov $1, %r1
ret
```

5. Problema de Tests

A continuación los problemas de test planteados en el enunciado:

5.1. Valor Absoluto

```
read %r0          # Lee un entero x
cmp $0, %r0       # Si r0<0, prende la bandera L
jmpl negativo
print %r0         # Imprime si x >= 0
jmp fin
negativo:
mul $-1, %r0      # r0 <- |x|
print %r0         # Imprime
fin:
hlt
```

5.2. Contador de bits

```
mov $32, %r3    # Número de repeticiones del loop
xor %r2, %r2    # Contador de bits encendidos
read %r0        # Valor a analizar

rep:  # En cada repetición del bucle, analizo el bit r3-ésimo

   # Formo una máscara para extraer el bit r3-ésimo en r1
push %r3
mov $1, %r1
dec %r3
shl %r3, %r1   #  r1 <-  1 << ( r3 - 1)
pop %r3

and %r0, %r1
cmp $0, %r1    # me fijo si el bit r3-ésimo esta encendido
jmpe apagado
inc %r2
apagado:
loop rep       # r3 -- ; repito si r3 > 0
print %r2      # imprimo el contador final
hlt
```

5.3. Sum para arreglos

```
#inicializo el arreglo: comienza en la dirección 120, [5,3,1,-1,-5]
#inicializo la pila: comienza en la  dire 120, tiene 5 elementos.
mov $5, 120
mov $3, 124
mov $1, 128
mov $-1, 132
mov $-5, 136
push $5
push $120


pop %r0         # r0 <- direccion del arreglo
pop %r3         # r3 <- cantidad de elementos
xor %r1, %r1    # r2 <- 0, acumulador
rep:            # do
lw  %r0, %r2            # leo un nuevo elemento del arreglo
add %r2, %r1            # lo sumo al acumulador
add $4, %r0             # me posiciono en el siguiente elemento
loop rep                # si quedan elementos, repido
print %r1       # imprimo el resultado  (-3).
hlt
```

6. Posibles extensiones

Algunas ideas para extender esta VM son:

- Dar la opción de usar subregistros y manipular enteros de 16 y 8 bits, agregando sufijos para las operaciones.
- Que todas las operaciones modifiquen RFLAGS, no sólo CMP.
- Mejorar el parser (ignorar espacios en blanco, aceptar comentarios)
- Pedir una etiqueta main obligatoria, y comenzar a ejecutar desde allí.
- Agregar instrucciones de manejo de arreglos (Las que vimos en x8664)
- Agregar manejo de valores en punto flotante.
- Guardar el código en memoria.

Referencias

[1] `http://aquamentus.com/flex_bison.html`