TRABAJO FINAL DE ARQUITECTURA DEL COMPUTADOR

MINI-VM

REALIZADO POR

Sebastián Zimmermann
Mauricio Muratori
Bruno Baruffaldi

Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura

09/07/2017

1 Introducción

El objetivo de este trabajo es aprender sobre los detalles de ejecución e implementación de una máquina abstracta. Esta se compone por 8 registros de 4 bytes (zero, pc, sp, flags y 4 registros de uso general). La máquina posee una única memoria y su propio lenguaje para ejecutar programas dentro de esta. Este lenguaje consta inicialmente de 17 instrucciones (NOP, MOV, LW, SW, PUSH, POP, PRINT, READ, ADD, SUB, MUL, DIV, CMP, JMP, JMPE, JMPL y HLT) las cuales trabajan con operandos de 4 bytes.

2 Metodología

Inicialmente nuestro trabajo consistió en comprender el código de la mini-vm brindado junto con las consignas, una vez hecho esto pasamos a implementar las instrucciones mencionadas anteriormente.

- **Manejo de datos y registro de flags**

  Para facilitar la lectura y escritura de los datos dentro de la mini-vm definimos dos funciones (*writer()* y *reader()*) e implementamos macros para modificar los valores del registro flags de una forma sencilla.

- **Instrucciones sin JMP**

  La mayoría de las instrucciones son simples y solo debieron ser implementadas dentro de la función *runIns()*. Para las instrucciones (NOP, MOV, SW, LW, PUSH, POP, PRINT, READ, ADD/SUB, MUL/DIV, CMP, HLT) solamente bastó extender el switch original que nos proporcionaba la función para poder aplicarlas.

- **JMP**

  Para lograr el correcto funcionamiento de dicha instrucción, hubo que modificar la función *run()*, dicha modificación se debió a que originalmente la función solamente verificaba que la instrucción recibida sea distinta de HLT (La cual termina el programa), y de no ser así aumentaba registro PC en uno.
  El problema era que el programa recibía las instrucciones JMP, JMPE o JMPL y aumentaba en uno el registro PC y no iba a la dirección de memoria dada por la etiqueta de la instrucción, incumpliendo con el comportamiento esperado de la instrucción JMP.
  Lo que se hizo fue agregar una condición, la cual se encarga de verificar si la instrucción recibida era JMP y en caso de recibir JMPE o JMPL también verifica que la FLAG correspondiente esté encendida. Luego, se asigna al registro PC la dirección de memoria brindada por el argumento.

- **Características adicionales**

  Como característica adicional decidimos implementar las instrucciones CALL (que saltan a un número de instrucción guardando el PC en la pila) y RET (que salta al número de instrucción guardado en la pila), con una pequeña convención de llamada en la cual no hay registros reservados, el resultado de la función debe ser almacenado en R0 y los argumentos son pasado a través de la pila. Implementar estas instrucciones fue relativamente sencillo, lo más complejo fue agregarlas al lenguaje definido. Para esto fue necesario modificar los archivos parser.lex y paser.y (aquí es donde se define el opcode y los operandos que toma la operación).
  Para probar los códigos nos pareció más práctico modificar los descriptores de archivos del programa para que lea el código de un archivo fuente que es pasado como argumento al programa.
  Para poder traducir el argumento recibido a instrucciones ejecutables por la máquina es necesario tener instalado GNU Bison y Flex.
  GNU bison es un programa generador de analizadores sintácticos de propósito general perteneciente al proyecto GNU disponible para prácticamente todos los sistemas operativos, se usa normalmente acompañado de Flex.
  Flex es un una herramienta que permite generar analizadores léxicos. A partir de un conjunto de expresiones regulares, Flex busca concordancias en un fichero de entrada y ejecuta acciones asociadas a estas expresiones.

3 Funcionamiento

Para ejecutar algún código en la mini-vm este debe ser pasado en un archivo de extensión asm como argumento del programa a través de la terminal. Dentro de la Mini-VM las funciones *yyparse()* y *processLabels()* se encargan de convertir el código en instrucciones ejecutables por la Mini-VM, que son almacenadas en un arreglo code de tipo struct Instruction (una estructura que almacena la instrucción con sus operandos) para poder realizar las operaciones. Una vez hecho esto se imprime el código en pantalla y se llama a la función *run()* para ejecutarlo. La función *run()* consiste en un bucle que ejecuta instrucción por instrucción hasta llegar a la instrucción HTL que denota el fin del programa.

4 Tests

4.1 Test 1

El test1 se basaba en implementar un programa que imprima el valor absoluto de un entero leído.

Output:

```
Running the following code
0: READ %r0
1: CMP %r0,$0
2: JMPL $5
3: PRINT %r0
4: HLT
5: MUL $-1,%r0
6: PRINT %r0
7: HLT
***************
-7
7
```

4.2 Test 2

Consiste en implementar un programa que que cuente los bits de un entero leído y lo imprima por pantalla.

Output:

```
Running the following code
0: READ %r0
1: MOV $1,%r1
2: MOV $32,%r3
3: MOV %r1,%r2
4: MUL $2,%r1
5: AND %r0,%r2
6: CMP $0,%r2
7: JMPE $16
8: CMP $1,%r1
9: JMPL $3
10: MOV %r1,%r2
11: AND %r0,%r2
12: CMP $0,%r2
13: JMPE $18
14: PRINT %r3
15: HLT
16: SUB $1,%r3
```

```
17: JMP $8
18: SUB $1,%r3
19: JMP $14
20: HLT
***************
15
4
```

4.3 Test 3

El test3 se basaba en implementar un programa que sume los enteros de un arreglo dado como argumento. Para ello leemos 6 valores arbitrarios que serán guardados en la pila y utilizado como los argumentos del programa.

Output:

```
Running the following code
0: READ %r0
1: SW %r0,$600
2: READ %r0
3: SW %r0,$601
4: READ %r0
5: SW %r0,$602
6: READ %r0
7: SW %r0,$603
8: READ %r0
9: SW %r0,$604
10: READ %r0
11: SW %r0,$605
12: PUSH $6
13: PUSH $600
14: POP %r0
15: POP %r1
16: MOV $0,%r2
17: LW %r0,%r3
18: ADD $1,%r0
19: ADD %r3,%r2
20: SUB $1,%r1
21: CMP $0,%r1
22: JMPL $17
23: PRINT %r2
24: HLT
***************
0 1 2 3 4 5
15
```

4.4 Test 4

Para corroborar que las instrucciones RET y CALL funcionan adecuadamente decidimos escribir una función que calcule el determinante de una matriz de 2*2.

Output:

```
Running the following code
0: READ %r0
1: PUSH %r0
2: READ %r0
3: PUSH %r0
4: READ %r0
5: PUSH %r0
6: READ %r0
7: PUSH %r0
8: CALL $11
9: PRINT %r0
10: HLT
11: POP %r3
12: POP %r0
13: MOV %r0,%r2
14: POP %r0
15: MOV %r0,%r1
16: POP %r0
17: MUL %r0,%r1
18: POP %r0
19: MUL %r0,%r2
20: SUB %r1,%r2
21: PUSH %r3
22: MOV %r2,%r0
23: RET
***************
1 3
-6 2
20
```

5 Problemas

5.1 Leer y escribir con funciones

Inicialmente nos dimos cuenta que podíamos facilitar la lectura y escritura de los datos dentro de la mini-vm. Para ello definimos dos funciones auxiliares (reader y writer) que permiten acceder y modificar los datos de una forma simple y hace el código más legible.

5.2 Macros de flags

Para poder alterar el registro de flags de manera eficiente y sencilla, definimos tres macros que permiten modificar rápidamente los valores de esta.

5.3 Archivo

Por una cuestión de simplicidad de testeo decidimos pasar el código como un argumento en forma archivo. Para ello utilizamos modificamos la tabla de descriptores de archivos del programa y evitamos modificar las funciones dadas por el enunciado.

6 Extensiones

Si bien esta máquina es relativamente simple existen varias implementaciones que podemos realizar. Una de ellas sería modificar el registro de flags para incluir las banderas de overflow, sing, entre otras. Tambien podriamos agregar al lenguaje instrucciones de manejo de bits o para facilitar el manejo de strings.

Bibliografía

- **http://webdiis.unizar.es/asignaturas/TC/wp/wpcontent/uploads/2011/09/Intro_Flex_Bison.pdf**
- **https://es.wikipedia.org/wiki/GNU_Bison**
- **https://es.wikipedia.org/wiki/Analizador_l%C3%A9xico**