



UNIVERSIDAD NACIONAL DE ROSARIO
FACULTAD DE CIENCIAS EXACTAS, INGENIERÍA Y AGRIMENSURA
Licenciatura en Ciencias de la Computación
Arquitectura del Computador

Mini Virtual Machine

Alumnos:

BORRERO, Paula (P-4415/6)
CRESPO, Lisandro (LEGAJO)
MISTA, Agustín (M-6105/1)

Docentes:

RUIZ, Esteban
FEROLDI, Diego
SCANDOLO, Leonardo
BERGERO, Federico

9 de Abril de 2015

1 Introducción

En éste trabajo se pretende explorar los detalles de ejecución e implementación de una máquina abstracta básica, como así también de un lenguaje ensamblador que la misma pueda ejecutar de manera nativa.

Para ello desarrollamos una con las siguientes características:

- Posee 8 registros de 4 bytes. Ellos son: ZERO, siempre vale cero. PC indica qué instrucción es la próxima a ejecutar (de la lista de instrucciones). SP Apunta al tope de la pila. R0,...,R3 son registro de uso general y FLAGS contiene bits de estado (resultado de operaciones, comparaciones, etc).
- Posee una única memoria en la cual se puede escribir y leer, mediante instrucciones MOV o POP/PUSH.
- Su juego de instrucciones es el siguiente:
 - NOP: no realiza ninguna operación.
 - MOV: mueve un valor de/hacia registro/memoria/constante.
 - SW: guarda el valor de src en la dirección de memoria dst.
 - LW: carga en dst el valor de la memoria apuntada por src.
 - PUSH: escribe el argumento src en el tope de la pila, decrementando SP por 4.
 - POP: escribe en el argumento dst el tope de la pila, incrementando el SP por 4.
 - PRINT: imprime por pantalla el entero descripto en el argumento src.
 - READ: lee un entero y lo guarda en el argumento de destino.
 - ADD/SUB: suma/resta los argumentos src y dst y lo guarda en dst.
 - AND/OR/XOR: realiza un bitwise and/or/xor entre los dos operandos y lo guarda en dst.
 - LSH/RSH: realiza un corrimiento de src bits a izquierda/derecha de dst y lo guarda en dst.
 - MUL/DIV: multiplica/divide los argumentos src y dst y lo guarda en dst.
 - CMP: compara src y dst y setea los bits correspondientes en el registro FLAGS.

- JMP: salta a la instrucción número dst de la lista de instrucciones.
- JMPE: salta a la instrucción número dst si la bandera de igualdad está seteada.
- JMPL: salta a la instrucción número dst si la bandera de menor está seteada.
- CALL: guarda el registro contador de programa en el stack y salta al label apuntado por src.
- RET: borra el registro contador de programa del stack y salta a la dirección apuntada por el mismo.
- HLT: termina el programa.

Todos sus operandos son de 4 bytes.

2 Problemas encontrados

1. En un principio nos enfocamos en entender el código que tendríamos que modificar. Primero analizamos los miembros de la estructura `Operand` para entender mejor su función. Así llegamos a la conclusión de que el entero *val* representaba:
 - a) El valor propiamente dicho del operando si éste era de tipo inmediato (el miembro *type* era *IMM*),
 - b) el índice dentro del arreglo de registros en donde se encontraba el operando si era un registro (*REG*),
 - c) el índice / 4 dentro del arreglo que representa a la memoria de la máquina si era un valor de memoria (*MEM*),
 - d) una dirección en caso de que sea una etiqueta (*LABELOP*)

Además, en el caso de que el operando sea un label, en el miembro *lab* de la estructura se guarda una cadena con su nombre.

2. Luego comenzamos a definir cada una de las funciones tratando de tener cuidado sobre que tipos de operadores íbamos a permitir que se pudieran ejecutar. Para ello usamos como ejemplo las funciones de assembler y tratamos de llegar a un consenso entre los integrantes del grupo.
3. También tuvimos que decidir como utilizar las flags. Nos centramos sobre todo en definir bien en que funciones iba a hacerse uso de la flag

equality and zero, since we could have used only zero and, in the case of comparisons, subtract the operands and if the result was 0 then we set the zero flag to indicate equality. We agreed that the comparison (*CMP*) could only set the zero, lower and equality flags and the arithmetic operations the zero flag.¹

4. As we started making functions we made various tests to check that the machine behaved as we pretended, but, before starting the test2 data, it seemed convenient to implement instructions of bit operators as an additional characteristic. For this we needed to modify the files of Bison and Flex (parser and lexer respectively) to add the new commands. The changes in these files we made by searching information on the internet and using as a base lines written for other commands already existing.
5. One time that we implemented the operation *READ*, we noticed that it did not function as expected, since for the virtual machine to read the file that it had to execute, it was redirected to the standard input of the same, which avoided that the keyboard was used as standard input to read a value. It is clear that a virtual machine that cannot read values from the keyboard is not very useful. Unfortunately, the implementation of the parser has flexibility in this sense, its input is obtained from the pointer to its external variable *yyin*, so now our file to execute by the machine is passed as an argument, and only it remains to open it and assign it to *yyin*.
6. We wanted to add also the commands *CALL* and *RET* to our virtual machine. We did this taking as a reference the functions of assembler of the x86_64 architecture. At first the test that we had made to check these functions did not work, but we saw that it was because we had forgotten to include the *CALL* function in the *processLabels* function.

3 Descripción de los casos de prueba

To check the correct functioning of our machine we wrote various complementary tests to the required ones. In the following a

¹Más detalles sobre el uso de flags en el anexo.

breve descripción de los casos de prueba requeridos junto con la de los que consideramos oportuno anadir:

- **Nombre:** test1.asm
Descripción: lee un entero e imprime el valor absoluto de dicho número.
- **Nombre:** test2.asm
Descripción: imprime en pantalla la cantidad de bits de un entero dado.
- **Nombre:** test3.asm
Descripción: dado un arreglo de elementos guardado en la pila, imprime en pantalla la suma de sus elementos.
- **Nombre:** test_read.asm
Descripción: lee un entero, lo imprime en pantalla y repite la operación a menos que se le pase un 0, en cuyo caso termina su ejecución.
- **Nombre:** test_stack.asm
Descripción: hace push de 5 enteros a la pila y luego los extrae mediante pop, se usó para probar las funciones de pila.
- **Nombre:** test_zero_flag.asm
Descripción: ejecuta algunas operaciones aritméticas y de bits, a fin de probar el funcionamiento de la flag *ZERO*
- **Nombre:** test_call_ret.asm
Descripción: llama a una función f mediante *CALL/RET*, se usó para testear el funcionamiento de éstos operadores.

4 Posibles extensiones

1. Anadir flags adicionales como *parity flag*, *carry flag*, *sign flag* y *direction flag*.
2. Agregar instrucciones de manejo de cadenas.
3. Anadir registros, instrucciones y operaciones para números de punto flotante.
4. Definir una convención de llamadas.

5 Conclusión

Podemos concluir que, si bien la máquina virtual que implementamos tiene notables limitaciones, ésta posee una buena escalabilidad en cuanto a sus capacidades, pudiéndose extender de manera relativamente sencilla, dotándola de capacidades más interesantes.

Anexo:

Manual de Comandos