
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 (C-6165/4)
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

equals y en cuales de zero, ya que podríamos haber usado solo zero y, en el caso de las comparaciones, hacer una resta entre los operandos y si la misma tenía como resultado 0 entonces prendíamos la flag zero para indicar la igualdad. Concordamos en que la comparación (*CMP*) podía encender solo las flags lower y equals y las operaciones aritméticas y de bits la flag zero.

4. A medida que íbamos haciendo funciones hacíamos varios tests para comprobar que se la máquina se comportaba como pretendíamos pero, antes de empezar el test2 dado, nos pareció conveniente implementar instrucciones de operadores de bits como característica adicional. Para esto necesitábamos modificar los archivos de Bison y Flex (analizador sintáctico y analizador léxico respectivamente) para anadir los nuevos comandos. Lo cambios en estos archivos los realizamos buscando información en Internet y usando como base líneas escritas para otros comandos ya existentes.

5. Una vez que implementamos la operación *READ*, notamos que no funcionaba como era esperado, ya que para que la máquina virtual leyera el archivo que debía ejecutar, éste era pasado redirigiéndolo a la entrada estándar de la misma, lo cual evitaba que se use el teclado como entrada estándar para leer un valor. Está claro que una máquina virtual que no pueda leer valores por teclado no resulta muy útil. Afortunadamente, la implementación del parser tiene flexibilidad en éste sentido, su entrada es obtenida desde lo apuntado por su variable externa *yyin*, por lo que ahora nuestro archivo a ejecutar por la máquina es pasado como argumento, y solo hace falta abrirlo y asignarlo a *yyin*.

6. Quisimos agregar también los comandos *CALL* y *RET* a nuestra máquina virtual. Lo hicimos tomando como referencia las funciones de assembler de la arquitectura x86_64. Al principio no andaba el test que habíamos hecho para probar estas funciones pero después vimos que era porque nos habíamos olvidado de incluir a la función *CALL* en la función *processLabels*.

# 3 Descripción de los casos de prueba

Para comprobar el correcto funcionamiento de nuestra máquina escribimos varias pruebas complementarias a las requeridas. A continuación se da una breve descripción de los casos de prueba requeridos junto con la de los que consideramos oportuno añadir:

- **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. Añadir flags adicionales como *parity flag*, *carry flag*, *sign flag* y *direction flag*.

2. Agregar instrucciones de manejo de cadenas.

3. Añadir 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.

BORRERO, Paula | CRESPO, Lisandro | MISTA, Agustín