Trabajo Práctico
Mini VM

Rossi, Martín. Legajo 9901/8

Febrero 2018

Resumen

El programa está compuesto por tres partes, el lexer o analizador léxico (flex), el parser o analizador sintáctico (bison) y el programa que simula la máquina.

Lo primero que se ejecuta es el lexer que toma como entrada el nombre de un archivo, lo lee y lo separa en tokens. Ésto está definido en el archivo *parser.lex*. Un token puede ser una cadena fija, como por ejemplo la palabra *call*, o un conjunto de cadenas, y para ello se usan expresiones regulares, como por ejemplo `"\$"DIGIT` donde `DIGIT` se define como `[0-9]\+`.

Luego se ejecuta el parser, que toma los tokens como entrada y a partir de una gramática libre de contexto forma la estructura de árbol correspondiente a la cadena que define la gramática. En el archivo *parser.y* se definen los tokens que serán la entrada y las reglas, así como un código que se ejecutará cuando se apliquen. En este caso cuando se aplica la regla para el no terminal `input` se guarda en el arreglo `code` una estructura que representa a una instrucción.

Finalmente se ejecuta el programa principal que va leyendo el código y modifica el estado de la máquina virtual, o sea, los registros y la memoria.

Posibles extensiones

- Agregar secciones data y text, e instrucciones para reservar espacio para números o texto.
- Agregar instrucciones más complejas de x86.
- Agregar más registros, y simular la unidad de punto flotante.
- Usar más bits del registro FLAGS, para simular las banderas CARRY, OVERFLOW, ZERO, etc.
- Agregar comentarios al lenguaje de la máquina virtual.

Problemas

1. Encontrar una forma concisa de escribir los casos de cada instrucción, sean memoria, registro o inmediato.
2. Los registros son int y la memoria es char.
3. Al agregar call y ret el programa no necesariamente inicia en la primera instrucción del código.
4. Las instrucciones jump y call no incrementan el program counter.

Soluciones

1. Elegir con un array indexado por el tipo de operando.
2. La memoria se maneja manualmente como arreglo de ints.
3. Empezar por la etiqueta main. Cuando se procesan las etiquetas se guarda la ubicación de main y se empieza por ahí. Si no existe main empieza por el principio del código.
4. Cuando se ejecuta una de esas instrucciones restar 1 al program counter.

Tests

Test 1: Valor absoluto (test1.asm)

Representación en C++:

```
Instruction code[]={
  Instruction(LABEL,Operand("main",LABELOP)),
  Instruction(READ,Operand(REG,R0)),
  Instruction(CMP,Operand(REG,R0),Operand(IMM,0)),
  Instruction(JMPL,Operand("prnt",LABELOP)),
  Instruction(MUL,Operand(IMM,-1),Operand(REG,R0)),
  Instruction(LABEL,Operand("prnt",LABELOP)),
  Instruction(PRINT,Operand(REG,R0)),
  Instruction(HLT)
}
```

Explicación:

El programa lee el número, si es menor a cero lo multiplica por -1, sino salta la instrucción de multiplicación y finalmente se muestra por pantalla.

Test 2: Contador de bits (test2.asm)

Representación en C++:

```
Instruction code[]={
  Instruction(LABEL,Operand("main",LABELOP)),
  Instruction(MOV,Operand(IMM,0),Operand(REG,R0)),
  Instruction(LABEL,Operand("loop",LABELOP)),
  Instruction(MOV,Operand(REG,R1),Operand(REG,R2)),
  Instruction(SUB,Operand(IMM,1),Operand(REG,R2)),
  Instruction(AND,Operand(REG,R2),Operand(REG,R1)),
  Instruction(ADD,Operand(IMM,1),Operand(REG,R0)),
  Instruction(CMP,Operand(REG,R1),Operand(IMM,0)),
  Instruction(JMPL,Operand("loop",LABELOP)),
  Instruction(PRINT,Operand(REG,R0)),
  Instruction(HLT)
}
```

Explicación

Dado un entero `n`, `n&(n-1)` apaga el bit más significativo. El programa lee un número en `r1`, y hace `r1=r1&(r1-1)` hasta que sea 0, agregando 1 en el registro `r0` cada vez. Al final imprime `r0`.

Test 3: Suma de arreglo (test3.asm)

Representación en C++:

```
Instruction code[]={
  Instruction(LABEL,Operand("sum",LABELOP)),
  Instruction(MOV,Operand(IMM,0),Operand(REG,R0)),
  Instruction(ADD,Operand(IMM,4),Operand(REG,SP)),
  Instruction(LW,Operand(REG,SP),Operand(REG,R2)),
  Instruction(ADD,Operand(IMM,4),Operand(REG,SP)),
  Instruction(LW,Operand(REG,SP),Operand(REG,R1)),
  Instruction(SUB,Operand(IMM,8),Operand(REG,SP)),
  Instruction(LABEL,Operand("loop",LABELOP)),
  Instruction(LW,Operand(REG,R1),Operand(REG,R3)),
  Instruction(ADD,Operand(REG,R3),Operand(REG,R0)),
  Instruction(ADD,Operand(IMM,4),Operand(REG,R1)),
  Instruction(SUB,Operand(IMM,1),Operand(REG,R2)),
  Instruction(CMP,Operand(REG,R2),Operand(IMM,0)),
  Instruction(JMPL,Operand("loop",LABELOP)),
  Instruction(PRINT,Operand(REG,R0)),
```

```
  Instruction(RETURN),
  Instruction(LABEL,Operand("main",LABELOP)),
  Instruction(SW,Operand(IMM,4),Operand(IMM,0)),
  Instruction(SW,Operand(IMM,5),Operand(IMM,4)),
  Instruction(SW,Operand(IMM,6),Operand(IMM,8)),
  Instruction(SW,Operand(IMM,7),Operand(IMM,12)),
  Instruction(PUSH,Operand(IMM,0)),
  Instruction(PUSH,Operand(IMM,4)),
  Instruction(CALL,Operand("sum",LABELOP)),
  Instruction(HLT)
}
```

Explicación

Primero se cargan números a la memoria simulando un arreglo, en este caso 4, 5, 6, 7 en las ubicaciones 0, 4, 8, 12. Ésto se hace con la instrucción `SW`. Después se hace push de el inicio del arreglo, 0, el tamaño, 4, y se llama a la función `sum`. `sum` primero pone en 0 a `r0`, y copia los argumentos (arreglo y tamaño) en `r1` y `r2`. A partir de `loop` se suma a `r0` el número apuntado por `r1` y se resta en 1 `r2`. Se repite hasta que `r2` sea 0, quedando en `r0` la suma, que luego muestra por pantalla.