



Universidad Nacional de Río Cuarto

Facultad de Ciencias Exactas Físico Químicas y Naturales

Departamento de Computación

Taller de Diseño de Software

Proyecto Cuarta entrega

Campregher Bruno
Machuca Franco
Natali Valentino

Generador Código Objeto

1. Introducción

En esta etapa del compilador se realiza la traducción del código intermedio generado previamente a código en lenguaje ensamblador.

El generador de código assembly toma como entrada la secuencia de instrucciones intermedias y produce como salida un archivo `.s` con las instrucciones necesarias para que el programa pueda ser posteriormente ensamblado y ejecutado.

Durante esta traducción se generan las instrucciones de bajo nivel que representan operaciones aritméticas, lógicas, de comparación, saltos y manejo de funciones.

Además, se administra el uso de registros y del stack para conservar el contexto de ejecución, así como la correcta asignación de direcciones de memoria a variables locales y globales.

2. Descripción del proceso general

El generador de código objeto funciona recorriendo secuencialmente la lista de instrucciones intermedias. Para cada instrucción, se analiza el operador (op) y se genera la instrucción de ensamblador correspondiente según el tipo de operación: aritmética, lógica, comparación, salto, asignación o llamada a función.

El flujo general de traducción de una instrucción puede resumirse en tres etapas principales:

1. **Carga de operandos:**

Cada operando se prepara para la operación en un registro. Si el operando es un literal, se carga directamente con una instrucción como `movl $valor, %eax`. Para variables locales, se accede mediante su desplazamiento (offset) relativo al registro base de la pila (`%rbp`). Las variables globales se acceden usando direcciones relativas al registro de instrucción (`%rip`). Esta etapa asegura que todos los datos necesarios estén disponibles en registros para la operación.

2. **Ejecución de la operación:**

Una vez cargados los operandos, se aplica la operación correspondiente. Operaciones binarias como suma, resta o multiplicación se realizan mediante la función `aplicar_operacion_binaria`, que emite instrucciones de ensamblador específicas (`addl`, `subl`, `imull`, etc.) sobre los registros apropiados.

3. **Almacenamiento del resultado:**

Finalmente, el resultado de la operación se almacena en la variable destino utilizando la función `guardar_desde_registro`. Dependiendo del alcance de la variable, el resultado se escribe en memoria local (stack frame) o en memoria global, garantizando la correcta persistencia de los valores generados durante la ejecución.

Este proceso se repite para cada instrucción de la lista, construyendo de manera secuencial el código assembly completo que representa la ejecución del programa original.

3. Componentes del generador

El módulo `asm.c` constituye el núcleo del generador de código objeto y contiene las funciones responsables de traducir las instrucciones intermedias a código ensamblador. Sus principales componentes son:

- `cargar_a_registro`: Carga el valor de un símbolo (literal, variable o etiqueta) en un registro, preparando los datos para su manipulación.
- `guardar_desde_registro`: Almacena el contenido de un registro en la variable destino, ya sea local o global.
- `aplicar_operacion_binaria`: Aplica una operación binaria entre el registro `%eax` y un segundo operando, generando las instrucciones correspondientes en ensamblador.
- `generar_asm`: Función principal que recorre la lista de instrucciones intermedias y emite el código assembly para cada operación, llamada a función, salto o asignación.
- `operadores`: Genera instrucciones para operaciones aritméticas, lógicas y de comparación.
- `mov`: Genera instrucciones para asignaciones, manejando variables locales y globales según corresponda.
- `params_decl`: Asigna offsets a los parámetros de función y genera las instrucciones necesarias para almacenarlos en el stack.
- `calcular_offset_var`: Calcula el desplazamiento (offset) de una variable local dentro del stack frame, asegurando un acceso correcto durante la ejecución de la función.

Este conjunto de funciones permite que el generador construya de manera sistemática el código ensamblador completo, respetando la ubicación de variables, el uso de registros y el flujo de ejecución del programa.

4. Manejo de saltos y condicionales

Los operadores `IF_FALSE` y `JMPC` generan comparaciones y saltos condicionales.

Ejemplo:

```
if (arg1->flag == LITERAL)
```

```
cargar_a_registro(out_s, arg1, "%eax");  
fprintf(out_s, "\tcml\t$0, %%eax\n");  
fprintf(out_s, "\tje\t%s\n", etiqueta);
```

Esto compara el valor con cero y salta a la etiqueta si la condición es falsa.

El operador `JMP` genera un salto incondicional (`jmp etiqueta`).

Las etiquetas (TAG) marcan posiciones de código a las cuales se puede saltar.

5. Gestión de memoria: variables globales y locales

El generador de código objeto diferencia entre **variables globales** y **variables locales**, y las maneja de manera separada para garantizar un acceso correcto y eficiente durante la ejecución del programa.

Variables globales:

Se declaran en la sección de datos (`.data`) y se marcan como símbolos globales con `.globl`. Cada variable recibe un valor inicial (`.long`) y tiene un espacio fijo en memoria, accesible desde cualquier función. Por ejemplo:

```
.globl  vglobal  
.data  
vglobal:  
.long  5  
.text
```

Variables locales:

Se almacenan en el stack frame de cada función con offsets relativos a `%rbp`. Cada variable tiene un offset único calculado por `calcular_offset_var`. Se accede a ellas cargando o guardando valores en memoria mediante registros. Por ejemplo:

```
movl    $10, -4(%rbp)    # int x = 10;
```

6. Alineación del stack y cumplimiento del ABI

Antes de reservar espacio para las variables locales, el generador ajusta el tamaño total del stack al múltiplo más cercano de 16 bytes, mediante la expresión:

```
int total_alloc = ((OFFSET + 15) & ~15);
```

Esta operación garantiza que el puntero de pila (%rsp) esté alineado a 16 bytes, tal como exige el ABI System V AMD64, que es la convención estándar usada en sistemas Linux.

Dicha alineación es necesaria para que las llamadas a funciones y el acceso a memoria sean eficientes y seguros.

7. Manejo de funciones

Cuando se detecta un `START_FUN`, se genera el prólogo estándar de función:

```
pushq %rbp
movq %rsp, %rbp
subq $N, %rsp
```

Donde `N` es el espacio necesario para variables locales y parámetros.

Los parámetros se asignan a offsets dentro del stack en `params_decl`, siguiendo la convención:

- Los primeros 6 parámetros se pasan por registros (`%edi`, `%esi`, `%edx`, `%ecx`, `%r8d`, `%r9d`).
- Los restantes se pasan por stack.

El epílogo de función (`END_FUN`) restaura el stack y retorna:

```
leave
ret
```

8. Pruebas y verificación

Se desarrollaron nuevos conjuntos de prueba específicos para la etapa de generación de código assembler. Los tests se encuentran en la carpeta `test/assembler` y abarcan expresiones aritméticas, estructuras de control anidadas, llamadas a funciones (incluyendo funciones externas) y funciones con comportamiento recursivo. Estos permitieron verificar la correcta traducción del código intermedio a instrucciones assembler válidas..

9. Ejecución del compilador

El archivo principal del proyecto (`main.c`) se reutiliza de la entrega anterior. Se incorpora la etapa de generación de código assembly, que se ejecuta únicamente después de que el programa ha completado la generación de código intermedio y las etapas previas. Esto garantiza que el código assembly se genere a partir de una representación del programa previamente validada y correcta.

Uso del programa (actualización)

- `-target s:` genera el código ensamblador (`.s`) a partir del código intermedio, produciendo un archivo listo para ser ensamblado por herramientas como gcc.