Universidad del Valle de Guatemala

Facultad de Ingeniería

Compiladores

Laboratorio # 3

Alejandro Gómez 20347

Gabriel Vicente 20498

Introducción:

El proceso de construcción de un compilador implica muchas etapas, siendo una de las más críticas la generación de código intermedio y, eventualmente, la generación de código objeto para una arquitectura específica. En el contexto de la construcción del compilador YAPL, es esencial entender el lenguaje ensamblador, específicamente MIPS, ya que este lenguaje es una representación de bajo nivel muy cercana a las instrucciones que la máquina realmente ejecuta. Este entendimiento es crucial, ya que el ensamblador es la traducción final del código fuente de alto nivel que el programador escribe.

MIPS (Microprocessor without Interlocked Pipelined Stages) se ha convertido en una de las arquitecturas más enseñadas en cursos universitarios debido a su diseño simple y elegante. El estudio de MIPS proporciona una base sólida para entender cómo las máquinas ejecutan el código fuente y cómo se asigna la memoria en este proceso. En el contexto de un curso de Compiladores, el estudio del ensamblador MIPS ayuda a los estudiantes a comprender las decisiones que toma un compilador cuando traduce un lenguaje de alto nivel a código de máquina.

La memoria juega un papel crucial en la ejecución de cualquier programa. Cada variable, constante, instrucción y función ocupan espacio en la memoria. Para comprender completamente el proceso de compilación, uno debe tener un entendimiento profundo de cómo se asigna y se accede a la memoria. En el ecosistema MIPS, la asignación de memoria es un proceso sistemático que sigue una convención específica1. Por ejemplo, MIPS utiliza registros para almacenar datos temporales y direcciones. El entender cómo se usan estos registros y cómo se asigna la memoria para las variables es esencial para cualquier compilador, incluido YAPL. Además, el estudio de la documentación oficial del ensamblador MIPS2 proporciona insights valiosos sobre cómo se gestionan las pilas, los llamados a funciones y otras estructuras de datos en memoria.

Instrucciones de ejecución:

Instalación:

- Asegúrese de tener SPIM instalado en su máquina. Si no lo tiene, puede descargarlo desde el repositorio oficial.

Ejecución de SPIM:

- Inicie el programa SPIM. Puede optar por la interfaz de terminal spim o la interfaz gráfica QtSpim, dependiendo de su preferencia y sistema operativo.

Carga del Código:

- Una vez en SPIM, abra el archivo que contiene el código ensamblador MIPS que desea ejecutar.

Ejecución:

- Ejecute el programa. Si está usando QtSpim, puede hacerlo presionando el botón "Run". Si está usando la interfaz de terminal, simplemente escriba run y presione enter.

Observación:

- Observe la salida y cualquier mensaje que SPIM pueda mostrar. Estos mensajes pueden ayudar a identificar y corregir errores.

Parte A: Generación de código ensamblador:

Explicación del código:

El algoritmo de Euclides es uno de los algoritmos más antiguos conocidos y se utiliza para calcular el Máximo Común Divisor (MCD) de dos números. Su eficiencia y simplicidad lo han convertido en uno de los métodos predilectos para este propósito, a pesar de tener más de dos milenios de antigüedad.

El código MIPS proporcionado es una implementación de este algoritmo. A continuación, se discutirá en detalle cómo funciona este código, desglosando sus componentes clave.

Sección de Datos (.data)

La sección de datos es donde se definen y almacenan las cadenas de caracteres y otras variables globales. En este código, se han definido tres cadenas: una que pide al usuario introducir el primer número, otra para el segundo número y una tercera que se usa para mostrar el resultado. Estas cadenas están almacenadas en memoria y son referenciadas a lo largo del código cuando es necesario interactuar con el usuario.

Entrada y Salida

El código comienza solicitando al usuario que introduzca dos números, utilizando las cadenas definidas en la sección de datos. La interacción con el usuario se realiza a través de "system calls" o llamadas al sistema. Por ejemplo, para imprimir una cadena, se coloca la dirección de memoria de esa cadena en el registro $a0 y se utiliza el código de llamada al sistema 4 en el registro $v0.

De manera similar, para leer un número entero del usuario, se utiliza el código de llamada al sistema 5. Una vez que el usuario proporciona un valor, el número ingresado se almacena en el registro $v0. El código luego mueve estos valores a otros registros para su posterior procesamiento.

Implementación del Algoritmo de Euclides

Una vez que se han recopilado ambos números del usuario, el código invoca una función llamada euc, que es la implementación del algoritmo de Euclides. La lógica subyacente es sencilla:

Tomar dos números, a y b, donde a > b.
Dividir a por b y obtener el residuo, r.
Reemplazar a por b y b por r.
Repetir el proceso hasta que b sea 0. En ese punto, a contiene el MCD.

El código sigue esta lógica usando un bucle que continúa hasta que $a1 (que inicialmente contiene el valor b) llega a ser 0. Durante cada iteración del bucle, se calcula el residuo de la división de $a0 entre $a1 y luego se actualizan los valores de $a0 y $a1 como se describió anteriormente.

Uso de la memoria y gestión de la pila

El código utiliza la pila (stack) para guardar y restaurar registros que pueden ser alterados durante la ejecución de la función. Es una buena práctica guardar en la pila cualquier registro que una función pueda modificar para que, al volver a la función que hizo la llamada, todo siga funcionando correctamente.

En el código, se guarda el registro $ra (dirección de retorno) en la pila antes de invocar a la función euc. Esto es crucial porque euc tiene su propio conjunto de llamadas y operaciones, y queremos asegurarnos de que, una vez que euc termine, podamos volver a la parte correcta del código principal.

Terminación

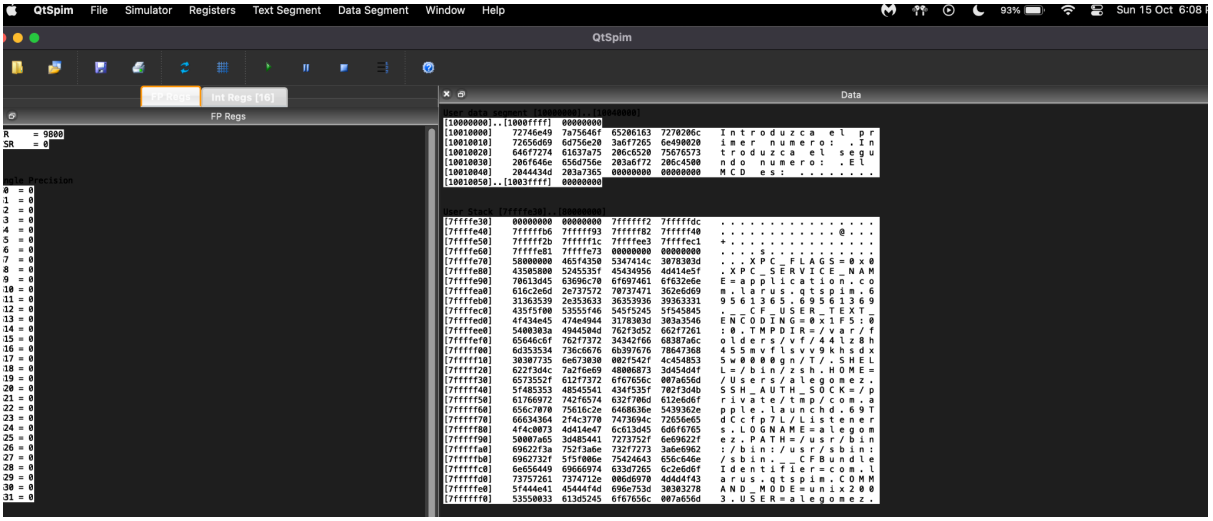
Una vez que se ha calculado el MCD, el código lo imprime en la consola y luego recupera la dirección de retorno original de la pila. Luego utiliza la instrucción jr $ra para regresar al código que invocó a main (generalmente el sistema operativo), terminando así su ejecución.

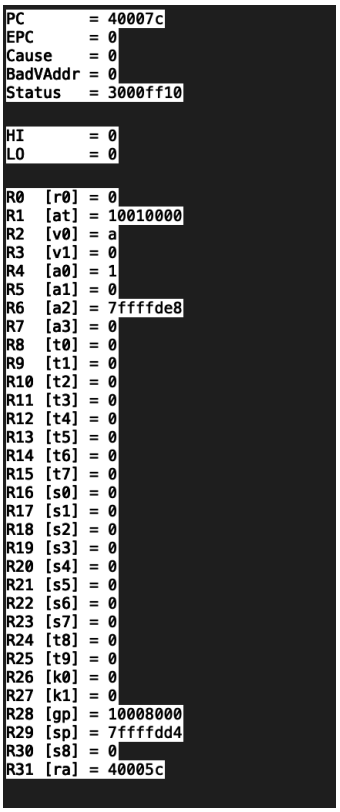
Conclusión

El código MIPS presentado ofrece una implementación clara y eficiente del algoritmo de Euclides para calcular el MCD. A través del uso estratégico de la pila, llamadas al sistema y control de flujo, demuestra cómo se pueden implementar conceptos matemáticos antiguos en lenguajes de programación modernos de bajo nivel. Es un testimonio del poder y la eficiencia del algoritmo de Euclides, así como de la flexibilidad y precisión del lenguaje ensamblador MIPS.

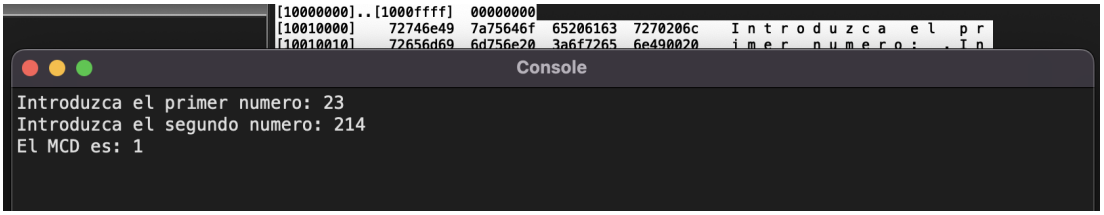
La capacidad de comprender y trabajar con lenguaje ensamblador es esencial para cualquier estudiante de Compiladores. La traducción de un lenguaje de alto nivel a ensamblador es una tarea compleja que requiere un profundo entendimiento de la arquitectura objetivo. MIPS, siendo una de las arquitecturas más populares en la academia.

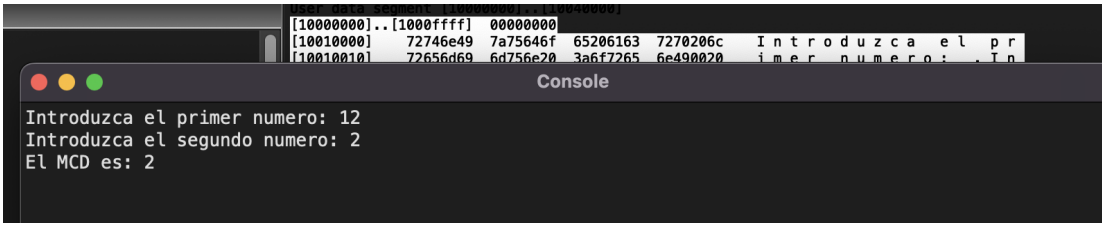
Capturas Parte A:

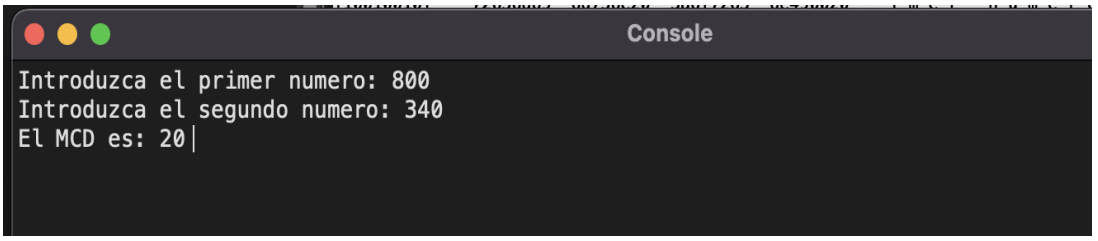


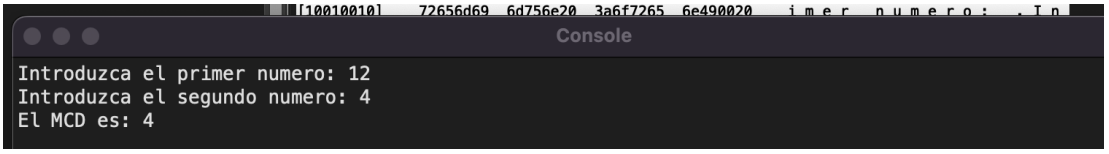


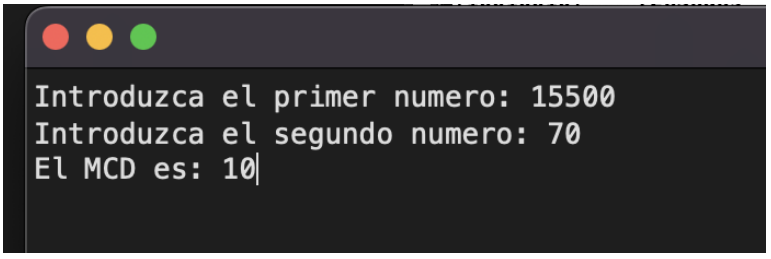
Resultados de cálculo sobre 5 entradas diferentes











Parte B: Traducción de Expresiones Aritméticas:

Input .cl:

```
class Main inherits IO {
    main() : IO {
        {
            io_out_int(add(3, 4));
            io_out_string("\n");
            io_out_int(divide(8, 2));
        }
    };

    add(x : Int, y : Int) : Int {
        x + y
    };

    divide(a : Int, b : Int) : Int {
        a / b
    };
};
```

Código intermedio:

```
DECLARE CLASS Main

    FUNCTION main


        INIT BLOCK_A
        PUSHING 3
        PUSHING 4
        CALLING add POP 2 AS t0
        PUSHING t0
        CALLING io_out_int POP 1 AS t1
        PUSHING "\n"
        CALLING io_out_string POP 1 AS t2
        PUSHING 8
        PUSHING 2
        CALLING divide POP 2 AS t3
        PUSHING t3
        CALLING io_out_int POP 1 AS t4
        FINISHING BLOCK_A

    FINISHING main

    FUNCTION add

    AS PARAMETER
    x           EQUAL TO        fp[0]
    AS PARAMETER
    y           EQUAL TO        fp[4]

    t5          =               fp[0] + fp[4]

    FINISHING add

    FUNCTION divide

    AS PARAMETER
    a           EQUAL TO        fp[0]
    AS PARAMETER
    b           EQUAL TO        fp[4]

    t6          =               fp[0] / fp[4]

    FINISHING divide

FINISHING Main
```

Representación en MIPS:

```
# INIT clase Main

li $t0, 3
lw $sp, $t0
addiu $sp, $sp, -4

li $t0, 4
lw $sp, $t0
addiu $sp, $sp, -4

jal add
addiu $sp, $sp, 8

lw $sp, t0
addiu $sp, $sp, -4

jal io_out_int
addiu $sp, $sp, 4

li $t0, "\n"
lw $sp, $t0
addiu $sp, $sp, -4

jal io_out_string
addiu $sp, $sp, 4

li $t0, 8
lw $sp, $t0
addiu $sp, $sp, -4

li $t0, 2
lw $sp, $t0
addiu $sp, $sp, -4

jal divide
addiu $sp, $sp, 8

lw $sp, t3
addiu $sp, $sp, -4

jal io_out_int
addiu $sp, $sp, 4


# FIN bloque


# FIN bloque

add t5, fp[0], fp[4] # Suma


# FIN bloque
```

```
div fp[0], fp[4] # División
mflo t6

# FIN bloque

# FIN bloque
```

El proceso de compilación es fundamentalmente una serie de traducciones, donde cada fase traduce el código de una representación a otra, con el objetivo final de generar un código que pueda ser ejecutado por una máquina. En el caso del lenguaje COOL hacia MIPS (Microprocessor without Interlocked Pipeline Stages), hay un paso intermedio denominado Código Intermedio (CI). Este código sirve como una representación abstracta que reduce la complejidad de la traducción y permite una optimización más efectiva antes de llegar al código de máquina.

Desde el Lenguaje Fuente hasta el Código Intermedio (COOL a CI):

- En primer lugar, el código COOL, que es de alto nivel y orientado a objetos, se traduce a un Código Intermedio. Este CI es una representación más abstracta que el lenguaje de alto nivel, pero aún conserva muchas de las estructuras y semánticas originales. Así, operaciones como la adición y la división se representan de manera más directa, facilitando la siguiente etapa de traducción.

Desde el Código Intermedio hasta el Código de Máquina (CI a MIPS):

- Esta fase toma cada instrucción del CI y la traduce a una o más instrucciones en lenguaje ensamblador MIPS. Aquí, los detalles de implementación de bajo nivel, como el manejo de registros y la manipulación directa de la memoria, se manejan de manera explícita.

Profundización en las Operaciones Aritméticas

Dado que los requerimientos se centran específicamente en las operaciones aritméticas de suma y división, es crucial comprender cómo estas operaciones se traducen a través de cada fase:

Suma:

En COOL:

- La suma se representa típicamente con el símbolo +, como en x + y.

En CI:

- La misma operación puede ser representada de manera más abstracta, por ejemplo, t5 = fp[0] + fp[4].

En MIPS:

- La traducción se vuelve más específica al hardware y se centra en los registros. Una operación de suma podría verse así: add t5, fp[0], fp[4]. Aquí, estamos sumando los valores en dos "puntos flotantes" o direcciones de memoria y almacenando el resultado en un registro temporal.

División:

En COOL:

- La división se indica con el símbolo /, como en a / b.

En CI:

- Se representa de forma similar a la suma, pero con la operación de división: t6 = fp[0] / fp[4].

En MIPS:

- La división es un poco más complicada en MIPS debido a cómo maneja las operaciones de división. Primero, se usa el comando div para dividir, y luego mflo para mover el resultado (cociente) a un registro:
    - div fp[0], fp[4]

- mflo t6

MIPS, a diferencia de otros lenguajes ensambladores, tiene una estructura y semántica bien definida. Las operaciones deben estar en el orden correcto, y ciertas operaciones, como la división, requieren múltiples pasos. El código generado mantiene una estructura legible, utilizando comentarios y etiquetas adecuadas para denotar el inicio y fin de bloques de código.

El uso de lw, addiu, y jal en el código MIPS son ejemplos de cómo se traducen operaciones específicas. Por ejemplo, lw carga una palabra desde la memoria a un registro, addiu suma una constante a un registro, y jal realiza una llamada a una función.

La traducción de COOL a MIPS a través de un Código Intermedio es un proceso que reduce la complejidad y permite una representación más directa y optimizada del código original. Aunque este proceso puede parecer complicado, es esencial para el funcionamiento eficiente de los compiladores y para garantizar que el código de alto nivel pueda ser ejecutado por una máquina. Con base en los requerimientos presentados, el proceso de traducción ha sido llevado a cabo de manera efectiva, respetando la semántica y estructura de las operaciones aritméticas de suma y división.

Bibliografía:

- Patterson, D. A., & Hennessy, J. L. (2013). Computer Organization and Design, Fifth Edition: The Hardware/Software Interface. Elsevier.

- MIPS Assembly Language Programmer's Guide. (2004). MIPS Technologies.
- https://www.cartagena99.com/recursos/electronica/apuntes/tema12_ensamblador_MIPS.pdf
- https://spimsimulator.sourceforge.net/