# Procesadores de lenguajes. Práctica 13.

GENERACIÓN DE CÓDIGO OBJETO EN EL COMPILADOR DE TINTO.

# Índice

# 1.1 Código a utilizar

- Las clases asociadas a la generación de código ensamblador en el compilador de Tinto están contenidas en los paquetes **tinto.mips, tinto.mips.instructions** y **tinto.mips.registers.**
- Dentro del paquete **tinto** se ha añadido la clase **TintoCompilerOptions**. Esta clase desarrolla un contenedor para las opciones de compilación que se introducen desde la línea de comandos. La clase TintoCompiler se ha modificado para generar los ficheros en ensamblador, enlazar todos los ficheros formando una única distribución de salida y tratar las opciones de compilación.

# 1.2 La clase tinto.TintoCompiler

- Respecto a prácticas anteriores, la clase **tinto.TintoCompiler** se ha modificado para desarrollar el proceso de compilación completo. El objetivo es compilar el archivo *Main.tinto* y todos los archivos importados desde él para generar el archivo *Application.s* que contendrá el código ensamblador de la aplicación.

- El método *createApplicationFile()* se encarga de crear el archivo *Application.s* y escribir en él un código común a todas las aplicaciones (generado por la clase **tinto.mips.ApplicationAssembler**). El método *createCode()* se ha modificado para generar no solo el código intermedio de cada biblioteca (un objeto **tinto.code.LibraryCodification**) sino para generar a partir de éste el código ensamblador de la biblioteca (un objeto **tinto.mips.LibraryAssembler**) y almacenarlo en un archivo con extensión "*.s*". Si el modo verbose está activo el código intermedio se almacena en un fichero con la extensión ".tic". Antes de analizar una biblioteca se busca si existe el archivo *.s* correspondiente. Si este archivo ya existe no es necesario generar de nuevo el código de la biblioteca (y por tanto no se lanza el método *createCode()* sobre ella). El método *appendFile()* se encarga de leer un archivo de ensamblador de una biblioteca y añadirlo al archivo *Application.s*. El resultado final es un archivo *Application.s* que contiene el código ensamblador de todas las bibliotecas incluidas en la aplicación y el código común que lanza la aplicación.

# 1.3 La clase tinto.TintoCompilerOptions

- La clase **tinto.TintoCompilerOptions** desarrolla un contenedor que almacenaa las opciones de compilado. Estas opciones son las siguientes:
  - **Path**, indica directorio de trabajo.
  - **-o Name**, indica el nombre del fichero de salida (sin la extensión ".s").
  - **-I Path**, añade un directorio de búsqueda de archivos importados.
  - **-v**, indica que se generen los ficheros de código intermedio (por defecto no se generarán).

- El constructor de la clase toma como entrada los argumentos introducidos en la llamada al compilador y calcula a partir de ellos el valor de los diferentes campos internos de la clase. La clase incluye métodos para acceder a los valores de estas opciones. El método *getWorkingDir()* obtiene el directorio de trabajo en el que se almacenarán los archivos generados en el proceso de compilación. Por defecto será el directorio desde el que se ha lanzado el compilador. El método *addIncludeDir()* añade un nuevo directorio a la lista de directorios de búsqueda para los archivos importados. El método *verboseMode()* indica si la opción verbose está activa o no. El método *getOutputName()* obtiene el nombre del fichero de salida. Por defecto este nombre es Application.

# 1.3 La clase tinto.TintoCompilerOptions

- La clase contiene además dos métodos dedicados a la búsqueda de archivos. Esta búsqueda se realiza tanto en el directorio de trabajo como en todos los directorios de búsqueda añadidos mediante opciones -I. El método *getTintoFile()* busca un archivo con extensión ".tinto" en todos estos directorios. El método *getAsmFile()* busca un archivo con extensión ".s" en los directorios confirgurados.

# 1.4 La representación del código ensamblador de MIPS

- El código objeto generado por el compilador de Tinto es código ensamblador del procesador MIPS. Los registros del procesador se encuentran descrito en el paquete **tinto.mips.registers**. Para describir estos registros se han definido las siguientes clases:
  - **Register**: Clase que describe un registro del procesador MIPS. La clase contiene un campo con el código del registro y un método *getName()* que devuelve el nombre asociado a cada registro.
  - **RegisterConstants**: Interfaz que define los códigos de los registros del procesador MIPS.
  - **RegisterSet**: Clase que contiene la definición de todos los registros de MIPS como referencias a objetos *Register*. Cada vez que una instrucción necesite hacer referencia a un registro de MIPS, se utilizan los campos estáticos de esta clase. De esta forma se evita crear cientos de referencias repetidas a los mismos registros.
  - **DispRegister**: Clase auxiliar que permite describir un direccionamiento indirecto, es decir, un desplazamiento sobre un registro.

# 1.4 La representación del código ensamblador de MIPS

- Las instrucciones del ensamblador de MIPS se definen en el paquete **tinto.mips.instructions**. A continuación se describen las clases utilizadas para representar este código ensamblador.
  - **InstructionSet**: Se trata de una interfaz que define códigos asociados a cada instrucción del procesador MIPS. La interfaz incluye códigos para todas las instrucciones de MIPS, aunque solo algunas de ellas son finalmente utilizadas por el compilador de Tinto.
  - **Instruction**: Clase abstracta que describe una instrucción del procesador MIPS. Las diferentes subclases de *Instruction* se diferencian en el modo de direccionamiento de los datos. El procesador MIPS desarrolla un código ensamblador de tres direcciones donde el direccionamiento puede ser inmediato (un valor constante), directo (un registro) o indirecto (un desplazamiento sobre un registro).
  - **RRRInstruction**: Describe una instrucción MIPS con tres direcciones con direccionamiento directo (tres registros). Típicamente permite describir instrucciones de operaciones entre dos registros, almacenando el resultado en el tercer registro (por ejemplo la suma ADDU).

# 1.4 La representación del código ensamblador de MIPS

- Las instrucciones del ensamblador de MIPS se definen en el paquete **tinto.mips.instructions**. A continuación se describen las clases utilizadas para representar este código ensamblador.
  - **RRInstruction**: Describe una instrucción MIPS que requiere dos direcciones con direccionamiento directo (dos registros).
  - **RInstruction**: Describe una instrucción MIPS con una única dirección con direccionamiento directo (un registro).
  - **RRIInstruction**: Describe una instrucción MIPS con tres direcciones con dos direccionamientos directos (dos registros) y uno inmediato (un valor). Por ejemplo, la suma de un valor (ADDIU).
  - **RIInstruction**: Describe una instrucción MIPS con dos direcciones con direccionamiento directo (un registro) e inmediato (un valor). Por ejemplo, la asignación directa de un valor a un registro.
  - **LabelInstruction**: Describe una instrucción MIPS con una dirección correspondiente a una etiqueta. Típicamente permite representar saltos incondicionales.
  - **RLInstruction**: Describe una instrucción MIPS con dos direcciones correspondiente a un registro y una etiqueta. Típicamente permite representar saltos condicionados al valor de un registro.

# 1.4 La representación del código ensamblador de MIPS

- Las instrucciones del ensamblador de MIPS se definen en el paquete **tinto.mips.instructions**. A continuación se describen las clases utilizadas para representar este código ensamblador.
  - **RRLInstruction**: Describe una instrucción MIPS con tres direcciones correspondiente a dos registros y una etiqueta. Típicamente permite representar saltos condicionados a la comparación entre dos registros.
  - **RDRInstruction**: Describe una instrucción MIPS con tres direcciones correspondiente a un registro y un desplazamiento sobre otro registro. Por ejemplo, las instrucciones de acceso a memoria (load y store) son de este tipo.
  - **NInstruction**: Describe una instrucción MIPS que no necesita ninguna dirección.
  - **InstructionFactory**: Clase que contiene un conjunto de métodos estáticos para crear las instrucciones de MIPS. Sólo se han incluido las funciones necesarias para el compilador de Tinto, de manera que existen muchas instrucciones de MIPS sin su correspondiente método en esta clase.

# 1.5 Generación de código ensamblador

- Además de las clases dedicadas a representar el código ensamblador de MIPS (comentadas en el apartado anterior), el paquete **tinto.mips** contiene tres clases dedicadas a generar el código ensamblador: *ApplicationAssembler*, *LibraryAssembler* y *FunctionAssembler*.

- La clase **tinto.mips.ApplicationAssembler** permite generar el código ensamblador común a todas las aplicaciones. Este código consta de una parte dedicada a describir el control de las excepciones (copiado directamente del código ofrecido por la herramienta PC-Spim) y el código de comienzo de la aplicación (etiquetado como *__start*) que contiene un salto a la etiqueta *Main_Main* correspondiente a la ejecución de la función *Main()* de la clase *Main*.

- La clase **tinto.mips.LibraryAssembler** encapsula el código ensamblador generado para un fichero fuente de Tinto. La clase contiene una lista de objetos *FunctionAssembler* que son los que generan el código ensamblador de cada método. El constructor recibe un objeto *LibraryCodification* con la descripción en código intermedio y construye los objetos *FunctionAssembler* a partir de los correspondientes *FunctionCodification* (que contienen el código intermedio de cada función). El método más importante de la clase es *generateFile()* que se encarga de crear el fichero ensamblador (con extensión "*.s*") asociado a la biblioteca.

# 1.5 Generación de código ensamblador

- La clase **tinto.mips.FunctionAssembler** es la encargada realmente de generar el código ensamblador a partir del código intermedio de cada función descrita en Tinto. La clase contiene los campos *label* (etiqueta de comienzo del método), *size* (tamaño en bytes del registro de activación de la función), *callstack* (pila utilizada para almacenar el tamaño la memoria reservada para los argumentos de cada llamada a función) y *list* (que almacena la lista de instrucciones ensamblador asociada a la función). El método más importante de la clase es *createAssembler()* que recibe como entrada la descripción de la función en código intermedio (*FunctionCodification*) y genera la lista de instrucciones en ensamblador. En primer lugar se genera el código de entrada al método (tal y como se describe en la práctica 11). A continuación se recorre cada instrucción en código intermedio y se traduce a código ensamblador. Por último se genera el código de salida del método (tal y como se describe en la práctica 11). El método *createAssembler()* es el encargado de traducir las instucciones en código intermedio a instrucciones en código ensamblador. Cada tipo de instrucción de código intermedio tiene su correspondiente función *translate...()* que genera el código ensamblador asociado.

# 1.6 Traducción de instrucciones a código ensamblador

- El compilador de Tinto almacena todas las variables locales y temporales, así como los argumentos de llamada a las funciones, en una zona de memoria conocida como el registro de activación de la función. Típicamente, una instrucción en código intermedio se traduce en un conjunto de instrucciones en ensamblador que: (1) almacenan los valores de las operandos en registros del procesador; (2) ejecutan la instrucción equivalente en ensamblador almacenando el resultado en otro registro del procesador; y (3) copian el resultado en memoria. Para resolver los pasos (1) y (3) se utilizan los métodos *translateLoadIntValue()*, *translateDelayedLoadIntValue()*, *translateLoadLiteral()* y *translateStoreIntValue()*. A continuación se describen estas funciones:
  - **translateLoadLiteral():** Genera instrucciones en ensamblador para asignar un valor constante (un literal) a un registro. Si el valor se expresa en 16 bits se utiliza una instucción **li**. Si el valor requiere más de 16 bits se utilizan las instrucciones **lui** y **ori** para cargar por separado la parte alta y la parte baja del literal.

# 1.6 Traducción de instrucciones a código ensamblador

- **translateLoadIntValue():** Genera instrucciones en ensamblador para cargar un valor en un registro del procesador. La función tiene en cuenta las distintas opciones de direccionamiento del valor. Si es un direccionamiento inmediato (el valor es un literal) se ejecuta *translateLoadLiteral()*. Si es un direccionamiento directo (el valor se encuentra en un registro del procesador) no se genera ninguna instrucción, pero se devuelve el registro en el que se encuentra el valor en vez del registro indicado como argumento. Si es un direccionamiento indirecto (típicamente un desplazamiento sobre el *frame pointer*) se genera una instrucción **lw**.
- **translateDelayedLoadIntValue():** Es equivalente a la anterior, pero añade una instrucción **nop** detrás de la instrucción **lw**. Esta función es necesaria para adaptarnos al pipeline de MIPS. La instrucción que sigue a **lw** no puede hacer uso del registro en el que se está almacenando el valor por problemas de pipeline. Cuando queremos utilizar un valor justo después de cargarlo de memoria es necesario esperar un ciclo (por medio de una instrucción **nop**).
- **translateStoreIntValue():** Genera las instrucciones para almacenar un valor contenido un registro sobre una variable. En este caso podemos encontrarnos dos tipos de direccionamiento: directo e indirecto. Si la variable está almacenada en un registro, se genera una instrucción **move**. Si la variable está en memoria se genera una instrucción **sw**.

# 1.6 Traducción de instrucciones a código ensamblador

- Una vez comentadas las funciones auxiliares utilizadas en los pasos (1) y (3) explicados anteriormente vamos a explicar como se traduce a ensamblador cada instrucción de código intermedio, asumiendo ya que tanto los operandos como el resultado se almacenan en registros del procesador.
  - **LABEL:** Se traduce como una etiqueta en ensamblador con el mismo identificador.
  - **ASSIGN:** Genera un *load* seguido de un *store*.
  - **ADD:** Utiliza la instrucción en ensamblador **addu** .
  - **SUB:** Utiliza la instrucción en ensamblador **subu**.
  - **MUL:** Para multiplicar dos número enteros se utiliza la instrucción **mult**. Esta instrucción almacena su resultado de 64 bits en dos registros especiales de MIPS llamados **HI** y **LO**. Para mover los 32 bits menos significativos al registro de resultado se utiliza la instrucción **mflo**.
  - **DIV:** La división entera en MIPS se realiza con la instrucción **div**. Esta instrucción almacena el cociente en el registro **LO** y el resto en el registro **HI**. Para traducir la división se añade la instrucción **mflo**.
  - **MOD:** Para calcular el módulo, es decir, el resto de la división entera, se utiliza la instrucción MIPS **div** seguida de **mfhi**.
  - **INV:** Para generar un cambio de signo se calcula la resta entre el 0 y el valor original. El 0 se obtiene del registro *$r0* y la resta se calcula con **subu**.
  - **AND:** La conjunción lógica se calcula con la instrucción **and**.

# 1.6 Traducción de instrucciones a código ensamblador

- Una vez comentadas las funciones auxiliares utilizadas en los pasos (1) y (3) explicados anteriormente vamos a explicar como se traduce a ensamblador cada instrucción de código intermedio, asumiendo ya que tanto los operandos como el resultado se almacenan en registros del procesador.
  - **OR:** La disjunción lógica se calcula con la instrucción **or**.
  - **NOT:** La negación lógica se calcula con la instrucción **slti** y el valor 1. Esta instrucción puede entenderse como "*set less than inmediate*", es decir, devuelve 1 si el valor del registro es menor que el valor indicado o 0 en otro caso.
  - **JMPEQ:** Para realizar un salto condicionado a la igualdad entre dos registros se utiliza la instrucción MIPS **beq**.
  - **JMPNE:** El salto condicionado a la desigualdad entre dos registros se realiza con **bne**.
  - **JMPGT:** Para el salto condicionado "mayor que" se utiliza en primer lugar la comparación **slt**, que genera 1 si el primer registro es menor que el segundo. A continuación se añade un salto **bne** comparando con el registro *$r0*, que realiza el salto si el resultado de la comparación no fue 0.
  - **JMPGE:** Para el salto condicionado "mayor o igual" se utiliza el equivalente "no menor que". Para ello se utiliza la instrucción **slt** seguda de un salto **beq** comaparando con el registro *$r0*.
  - **JMPLT:** Para el salto condicionado "menor que" se utiliza la instrucción **slt** seguida de un salto **bne** comparando con el registro *$r0*.
  - **JMPLE:** El salto condicionado "menor o igual" se genera como "no mayor que" por medio de la instrucción **slt** y el salto condicional **beq** comparando con el registro *$r0*.

# 1.6 Traducción de instrucciones a código ensamblador

- Una vez comentadas las funciones auxiliares utilizadas en los pasos (1) y (3) explicados anteriormente vamos a explicar como se traduce a ensamblador cada instrucción de código intermedio, asumiendo ya que tanto los operandos como el resultado se almacenan en registros del procesador.
  - **JUMP:** El salto incondicional se genera con la instrucción **j**.
  - **JMP1:** Para el salto condicionado a que un valor sea cierto se utiliza **bne** comparando con el registro *$r0*.
  - **PARAM:** Para almacenar un valor como parámetro en la posición *index* se direcciona como *$sp* + *index* y se almacena con **sw** .
  - **PRECALL:** Esta instrucción se utilizaba para avisar de una próxima llamada a función indicando el número de parámetros de la llamada. Esto se traduce con una modificación del registro *$sp* (stack pointer) para reservar espacio para estos parámetros. Como la memoria de pila crece en sentido descendente, la actualización de *$sp* consiste en restarle el espacio indicado.
  - **CALL:** Esta instrucción indica la función a la que se llama y la variable en la que se debe almacenar el resultado de la llamada. Para saltar a la función llamada se ejecuta la instrucción **jal**. A la vuelta de la llamada el resultado debe encontrarse en *$sp - 4*, por lo que el siguiente paso es cargar ese calor en registro (**lw**) y salvarlo en la variable indicada (**sw**). Una vez salvado el valor de retorno se actualiza el registro *$sp* liberando el espacio reservado para los parámetros de la llamada.
  - **RETURN:** La instrucción de retorno se traduce como un almacenamiento del resultado en el registro *$v0* seguido de un salto a la etiqueta de retorno de la función. El código que desarrolla el retorno de la función se encuentra escrito a partir de esa etiqueta.