Generación Automática de Código

Máster en Investigación en Ingeniería de Software y Sistemas Informáticos

Trabajo de Investigación
"Generación basada en paradigmas"
UNI-ASM: Lenguaje Ensamblador Universal

Alumno: Carlos Jiménez de Parga Bernal – Quirós

Í N D I C E

1. Resumen

Los conocimientos en informática han avanzado considerablemente. El desarrollo de compiladores, intérpretes y programación a bajo nivel es una actividad frecuente en el mundo del hardware. El auge de la multiplataforma y multidispositivo hace necesaria la programación a bajo nivel en diferentes tipos de microprocesadores y microcontroladores. Los diferentes microprocesadores tienen diferentes arquitecturas y modos de direccionamiento: algunos son CISC, otros RISC. Esto hace un conjunto heterogéneo de modos de programación en ensamblador. Aunque todos los lenguajes ensambladores difieren en el modo de movimiento de operandos, de direccionamiento, número de operandos, nomenclatura, etc.; hay una parte común que se ha utilizado para la generación de un lenguaje ensamblador universal. Por ello, se ha considerado un lenguaje independiente de la plataforma, único y configurable para cada tipo de lenguaje de microprocesador. De esta forma el usuario es libre de programar a un nivel más cómodo sin sacrificar la potencia del lenguaje de bajo nivel sobre el que trabaja. Es decir, se ha seguido un paradigma uno a muchos. Un único lenguaje de alto nivel, muchos lenguajes de destino configurables.

2. Introducción

La industria del hardware está en contacto continuo con microprocesadores y microcontroladores de muy diversa índole. Así por ejemplo, existe el caso de una empresa de éxito en mi comunidad autónoma encargada de realizar rótulos luminosos para establecimientos. Esta empresa está continuamente desarrollando en lenguaje C y ensamblador. Los rótulos luminosos se componen de un microcontrolador o microprocesador de 16 bits (68000 ó 8x52) para el control de los leds.

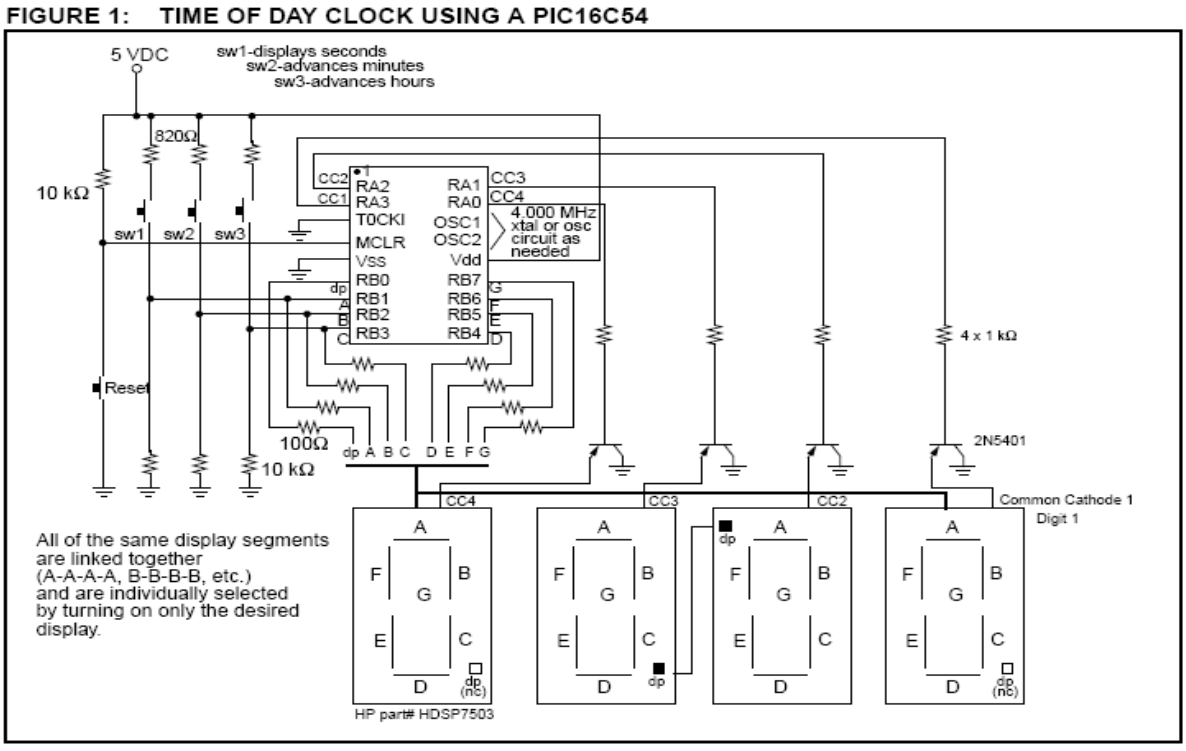


Figura 1. Display con microcontrolador PIC 16C54 [15]

Tal como se muestra en la imagen, la arquitectura de un sistema PIC embebido en un rótulo, se compone de un circuitería muy simple. En núcleo principal de dicha arquitectura es un microcontrolador. En este tipo de sistema es complejo programar y almacenar aplicaciones muy extensas, para ello se hace necesaria la programación en ensamblador, principalmente con dos objetivo: optimizar el rendimiento y reducir el

espacio en memoria.

Ante esta situación, se nos presenta el siguiente problema: ¿qué pasaría si se cambiara el PIC del sistema?, y, ¿cómo almacenar un programa muy reducido para realizar una tarea monótona como encender y apagar una matriz de valores?.

Una solución sería convertir el programa al nuevo PIC o microprocesador. Esto implicaría un coste de tiempo adicional con el potencial riesgo de cometer algún error en la codificación de las instrucciones.

La solución que aquí se propone está basada en la utilización de un lenguaje único o universal que pueda ser configurado para cada tipo de microprocesador o PIC.

Esto se llevaría a cabo mediante un lenguaje híbrido entre el alto y el bajo nivel que englobe todos los tipos de arquitecturas, ya sea CISC o RISC. A modo de ejemplo, si suponemos los siguientes códigos en MIPS y x86_64, podemos apreciar la similitud y convergencia a un mismo lenguaje universal:

MIPS	x86_64

lw $t3, 0($t0)
lw $t2, 4($t0)
add $t1, $t2, $t3
sw $t1, 8($t0)
	mov eax, [rsp + 0]
	add eax, [rsp +8]
	mov [rsp + 16], eax

Tabla 1. Dos ensambladores totalmente diferentes

Lenguaje universal MIPS	Lenguaje universal x86_64
data: x: integer y: integer z: integer code: z = x + y	data: x: integer y: integer z: integer code: z = x + y

Tabla 2. Convergencia al lenguaje universal

Tal y como se observa en las tablas, el lenguaje MIPS es un ensamblador RISC y por ello utiliza instrucciones de suma de tres operandos. Sin embargo, en la arquitectura x86_64 difiere un poco con respecto a su homóloga en MIPS, utilizando menos instrucciones. Obviamente, la arquitectura RISC se ejecuta más rápido e implicando menos CPI, aunque tenga más instrucciones.

La consecuencia final es que ambos códigos pueden unificarse en un mismo lenguaje ensamblador para realizar exactamente la misma operación matemática.

Obviamente con este enfoque se resuelven los dos problemas anteriormente planteados. Si necesitáramos un cambio en el microcontrolador o microprocesador únicamente tendríamos que crear un fichero de configuración para el nuevo chip sin tener que cambiar el código de alto nivel. Por otro lado se mantendrían los aspectos de optimización de rendimiento y optimización de espacio de memoria.

En consecuencia los objetivos que pretende esta aproximación se resumen en los siguientes apartados:

1. Sistema multiplataforma.
2. Optimización de rendimiento y espacio.
3. Mayor abstracción.
4. Configuración personalizada de dicha abstracción.
5. Aumento de la legibilidad.
6. Modularidad.
7. Facilidad de mantenimiento y pruebas.
8. Fácil reversibilidad del código ensamblador de destino.

3. El problema

3.1 Descripción del problema

Actualmente la industria del hardware se encuentra rodeada de gran variedad de sistemas microprocesador y microcontrolador para el desarrollo de sistemas embebidos, periféricos, electrodomésticos, sistemas Hi-Fi, automóviles, etc. En todos ellos es frecuente encontrar sistemas hardware que utilizan un pequeño microprocesador con una memoria y un sistema de E/S.

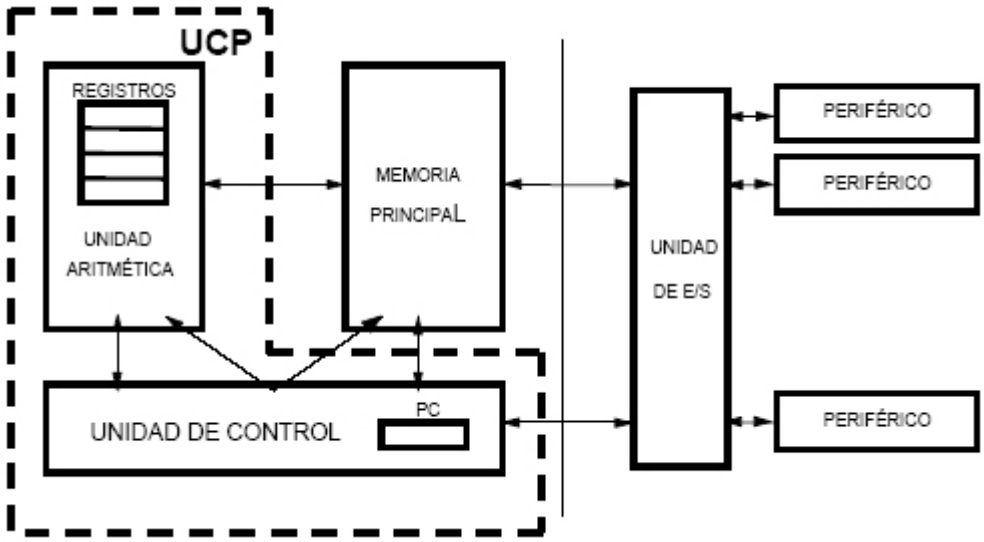


Figura 2. Sistema básico planteado

En la imagen se puede apreciar la estructura de un sistema de computador basado en PIC o microprocesador. Estas arquitecturas son comunes hallarlas en sistemas móviles, PDAs, Tablets, sistemas embebidos, electrodomésticos, sistemas críticos, etc.

Como se comentó anteriormente nos encontramos con dos problemas fundamentales:

- Qué hacer si el chip cambia o es necesaria la implementación en varias

plataformas.

- Cómo conseguir código legible y optimizado para el bajo nivel.

Para estas dos preguntas surge la solución de utilizar un lenguaje único y más elevado con respecto al ensamblador pero sin obviar muchos detalles del mismo. Con esto se consigue una mayor abstracción sin sacrificar el rendimiento. Por otro lado obtenemos beneficios como la multiplataforma, la legibilidad, la modularidad, la facilidad de prueba y mantenimiento.

También se puede considerar útil para los diseñadores de compiladores, ya que como se verá más adelante, la estructura del lenguaje UNI-ASM está inspirada en el diseño de estos.

3.2 Restricciones

Obviamente el sistema tiene algunas restricciones.

- Es necesaria la configuración para cada ensamblador de destino.
- Actualmente sólo está implementada la versión para un único ámbito.
- No es tan versátil como un lenguaje de alto nivel como podría ser C/C++.
- Como se verá más adelante es necesario utilizar marcas (“tags”) para las instrucciones que dependen del tipo de dato del operando: inmediato, byte, word, etc.
- No está implementada la opción de procedimiento y función, aunque sería muy fácil el cambio.
- No optimiza el código ensamblador final.
- El sistema no está probado exhaustivamente.

3.3 Panorama de las soluciones ya existentes

En el mercado existen soluciones para la programación a bajo nivel. Si descontamos los propios ensambladores, la solución más cercana a la programación para varias CPU es el uso de ***gcc*** por medio del uso de:

```
gcc -O2 -S -c foo.c
```

este comando creará un fichero foo.s con el código ensamblador para la plataforma seleccionada.

Otra opción comercial para la programación a bajo nivel es la utilización de ***LabView*** que permite la programación visual de dispositivos con el uso de una interfaz gŕafica muy amigable.

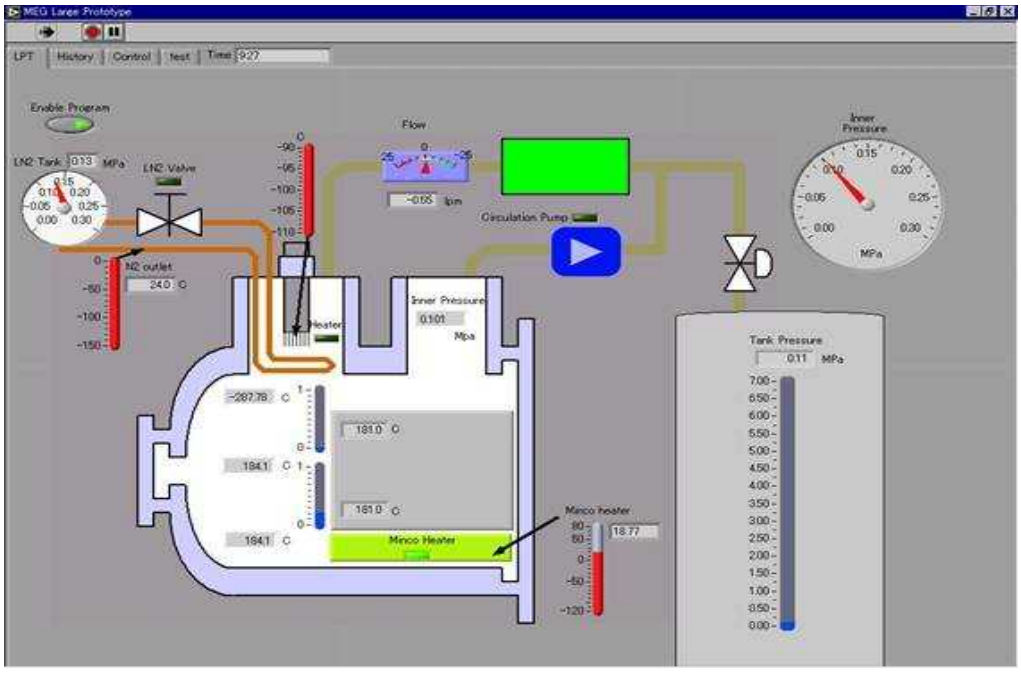


Figura 3. Ejemplo de interfaz gráfica creada con LabView

4. Evaluación de soluciones

Para la evaluación de los resultados se ha procedido a implementar cuatro algoritmos básicos tanto en el compilador ***g++*** como con el lenguaje UNI-ASM. Para las pruebas de las ejecuciones se ha utilizado un ordenador portátil Mac Book Pro con las siguientes especificaciones:

CPU	Memoria	Disco Duro	Sistema Operativo	Núcleo
Intel Core i5 M520 @ 2.40GhZ	3.7 GB	74.7 GB	Ubuntu 10.04 (Lucid)	Núcleo Linux 2.6.32 -38-generic

Tabla 3. Especificaciones del sistema de prueba

Para la realización de los benchmarks se han implementado los siguientes cuatro algoritmos:

Suma atómica (1)

```
int x = 10 + 10
print x
```

Bucle (2)

```
for i = 1 to 10000
  print i
```

Multiplicación (3)

```
for  i = 0 to 10
  for j = 0 to 10
        print i, j,  i * j
```

Acumula (4)

```
t1 = 1
t2 = t1 + 2
t3 = t1 + t2 + 3
t4 = t1 + t2 + t3 + 4
t5 = t1 + t2 + t3 + t4 + 5
t6 = t1 + t2 + t3 + t4 + t5 + 6
t7 = t1 + t2 + t3 + t4 + t5 + t6 + 7
t8 = t1 + t2 + t3 + t4 + t5 + t6 + t7 + 8
t9 = t1 + t2 + t3 + t4 + t5 + t6 + t7 + t8 + 9
t10 = t1 + t2 + t3 + t4 + t5 + t6 + t7 + t8 + t9 + 10

print t10
```

Posteriormente se evaluarán dichos algoritmos utilizando las versiones de gcc y de UNI-ASM y comparando los tiempos de ejecución en microsegundos. Se diseñarán histogramas estadísticos para visualizar con claridad las diferencias de tiempo entre versiones. Por último se realizará una comprobación de mejora del rendimiento en tanto por ciento entre la versión gcc y la versión UNI-ASM.

5. Descripción detallada de la solución

5.1 Un enfoque MDA

Siguiendo el esquema de la Generación Basada en Paradigmas[1] me he fundamentado en una filosofía de generación automática de código basada en modelos (MDA/MDD).

MDA (Model Dirven Architecture) es un paradigma de resolución de sistemas basados en modelos. Un modelo se plantea en términos de cualquier representación o abstracción para resolver problemas dentro de un ámbito específico de negocio. En el caso del trabajo de investigación que nos atañe, dicho modelo de alto nivel es textual, e incluye información léxica, sintáctica y semántica. Esta representación se le denomina más comúnmente DSL (Domain Specific Language) y es donde se ubica el lenguaje UNI-ASM.

MDA se sustenta en tres niveles fundamentales que en el presente trabajo de investigación se concretan en el siguiente gráfico:

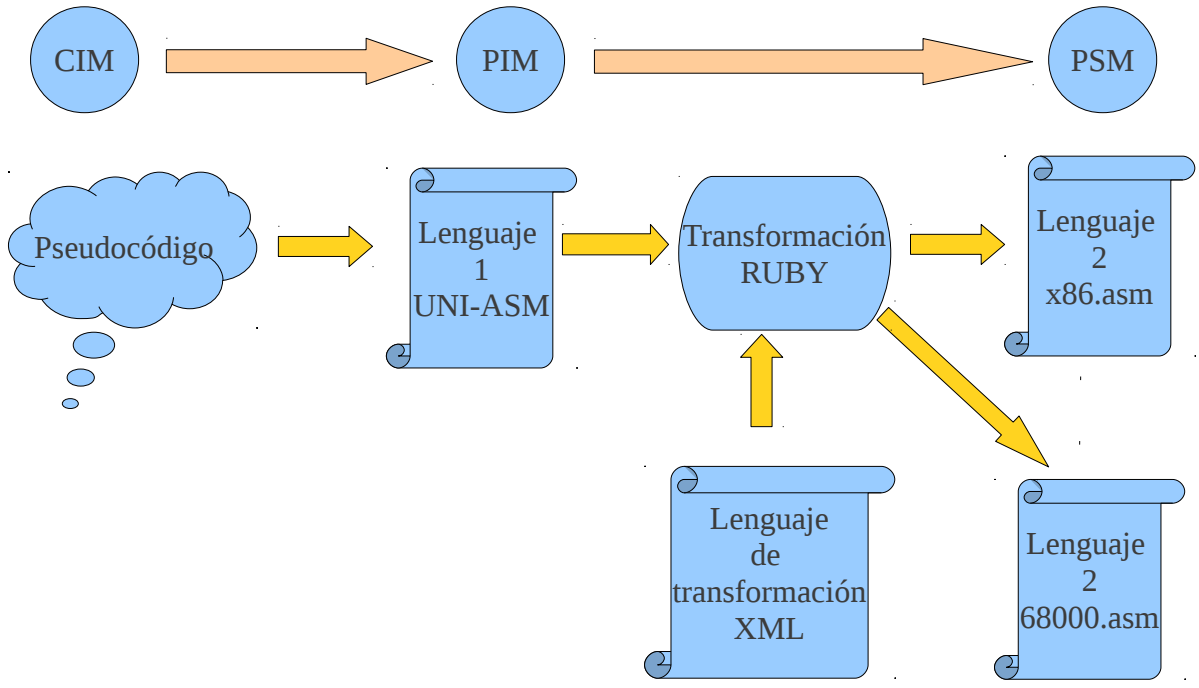


Figura 6. Esquema de transformación MDA

1 Título del enunciado del trabajo de investigación de la asignatura de Generación Automática de Código

Como se observa en la figura 6, se comienza esbozando un modelo CIM (Modelo Independiente de la Computación) que será transferido y editado en un sistema informático en el lenguaje UNI-ASM, que representa, en este caso, el PIM (Modelo Independiente de la plataforma). Por último, previa escritura y configuración del lenguaje de transformación se procederá a generar el PSM (Modelo específico de la plataforma), modelo textual que representa el lenguaje ensamblador específico del microprocesador o PIC del sistema.

El PIM es un modelo textual, representado por el lenguaje UNI-ASM y que por tanto omite toda referencia a la plataforma (modos de direccionamientos, movimiento de operandos, códigos de operación, etc.) y se enfoca principalmente al concepto algorítmico y en los objetivos enunciados en el apartado 2 de este estudio. Posteriormente, y previa configuración de un fichero XML con las características del microprocesador y de su lenguaje ensamblador, podrá transformarse mediante el uso de la aplicación Ruby en el lenguaje máquina de destino, representado este último como el PSM.

5.2 Inspirado en los compiladores

Para el desarrollo del lenguaje UNI-ASM y con la finalidad de diseñar un lenguaje a medio camino entre el alto y bajo nivel, cómodo e inteligible, vi oportuno la estructura del código de tres dirección de la generación de código intermedio de la fase de compilación.

Para explicar estos conceptos se ilustra la fase de diseño de un compilador comercial:

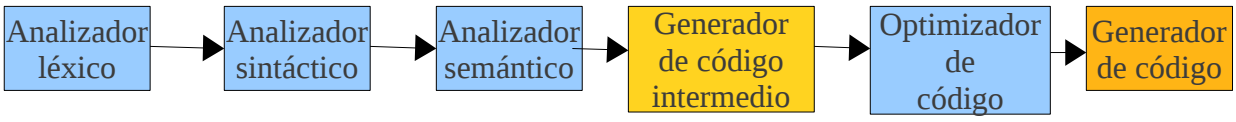


Figura 7. Esquema de las fases compilación (en amarillo y naranja las partes utilizadas en la investigación)

El código intermedio sirve para transcribir las anotaciones realizadas por medio de atributos sintetizados y heredados en el árbol semántico[2] con la finalidad de generar código objeto de destino.

De esta manera si tuviéramos el siguiente grafo de dependencia de un árbol semántico:

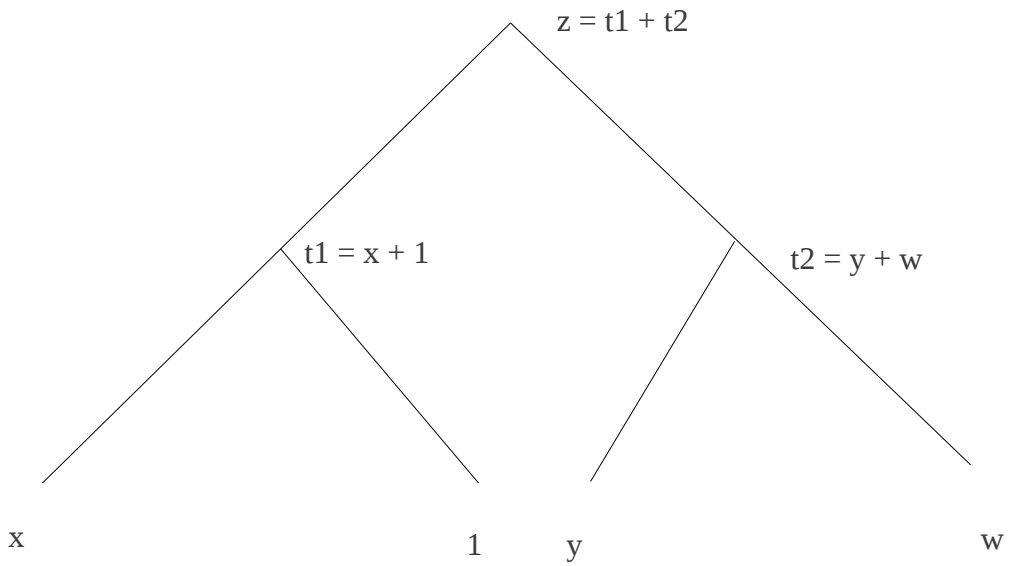


Figura 8. Árbol con anotaciones para la expresión z = x + 1 + y + w

éste se convertirá posteriormente en las siguientes instrucciones de código intermedio:

t1 = x + 1
t2 = y + w
z = t1 + t2

Obsérvese que el código de tres direcciones generado únicamente contiene 3 operandos: operando resultado, primer operando y segundo operando.

La parte interesante viene cuando este código de tres direcciones se debe transformar en código objeto o ensamblador. Estamos ya en la fase de generación de código. Para llevar a cabo esta funcionalidad requerimos del uso de las cuádruplas o cuartetos.

2 Implementado en la fase de análisis semántico

De esta forma el código anterior quedaría transformado en la siguiente lista de cuádruplas:

(add, t1, x, 1)
(add, t2, y, w)
(add, z, t1, t2)

donde el primer elemento de la tupla es el código de operación y el resto, el operando resultado y los operandos.

La pregunta a continuación es ¿cómo se obtiene el valor de x, y y w? Para ello disponemos de una estructura de datos especial en el diseño de un compilador. Dicha estructura de datos se la denomina *Tabla de Símbolos*.

La *Tabla de Símbolos* tiene la siguiente estructura:

Identificador	Ámbito	Tipo	Línea
x	**Main**	**integer**	**1**
y	**Main**	**integer**	**1**
z	**Main**	**integer**	**3**
w	**Main**	**integer**	**1**

Figura 9. Ejemplo de Tabla de símbolos

Las variables t1 y t2 son los llamados *temporales* y que sirven para albergar operaciones intermedias. Estas variables son autogeneradas durante el proceso de análisis semántico.

Utilizando la *Tabla de Símbolos* anteriormente descrita y una *Tabla de Temporales* (muy parecida a la *Tabla de Símbolos*) se procede a reservar memoria en la pila por medio del uso de un *Registro de Activación*.

Para entender el concepto de funcionamiento de un *Registro de Activación* y cómo se reserva memoria para las variables y temporales, suponemos el siguiente modelo de memoria en la pila del sistema:

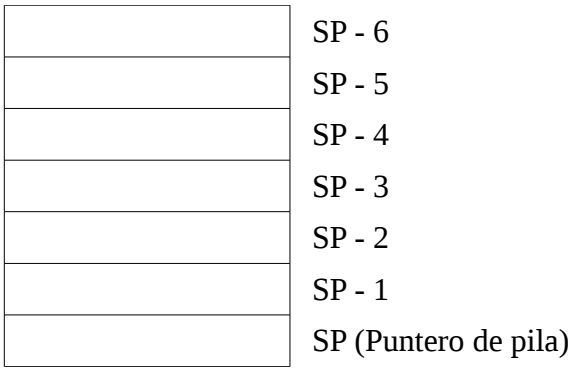


Figura 10. Esquema de la pila en memoria antes de la asignación

Suponemos que SP es el puntero de pila y que apunta a un dirección de memoria. También consideramos que la dirección de movimiento del puntero SP es decreciente. Como necesitamos reservar memoria para cuatro variable y dos temporales, únicamente deberíamos desplazar seis posiciones de memoria. Por lo tanto el resultado de la asignación quedaría como:

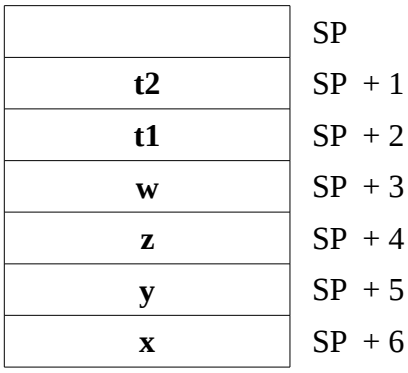


Figura 11. Esquema de la pila en memoria después de la asignación

Ahora, consideremos el caso de inicialización de una variable:

```
x = 5
```

Tan sólo deberíamos generar el código ensamblador para:

```
mov [sp + 6], 5
```

De igual forma para la generación de código de la lista de cuádruplas que nos atañe únicamente debemos crear el siguiente código ensamblador:

```
(add, t1, x, 1)
(add, t2, y, w)
(add, z, t1, t2)

mov eax, [sp + 6]
add eax, 1
mov [sp + 2], eax

mov eax, [sp + 5]
mov ebx, [sp + 3]
add eax, ebx
mov [sp +1], eax

mov eax, [sp + 2]
mov ebx, [sp + 1]
add eax, ebx
mov [sp + 4], eax
```

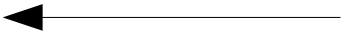
5.3 Descripción funcional de la solución aportada

Supongamos el siguiente código en UNI-ASM:

```
(- Ejemplo de prueba del lenguaje UNI-ASM desarrollado
para la asignatura de Generación Automática de Código
del Máster en Investigación en Ingeniería de Software y Sistemas Informáticos -)
' Desarrollado por alumno: Carlos Jiménez de Parga
' Versión x86_64 (64 bits NASM)

data:

      t1: integer
      t2: integer
      t3: integer
      i:  integer
      j:  integer
      x:  integer
      y:  integer
      s1 = "FIN DEL PROCEDIMIENTO"
code:

      t1 = 0
      t2 = 1

      jump eti1

label mult:
      i = 0
      x = 0
label bucle:
      i = i + 1
      x = x + y

      t3 = i == j
      if_false t3 goto bucle
      jump eti2

label eti1:

      t1 = t1 + 1
      writei t1
      writes " X "
      writei t2
      y = t1
      j = t2
      jump mult
label eti2:
      writes " = "
      writei x
      writes "  "
      t3 = t1 == 10
      if_false t3 goto eti1

      t1 = 0

      t2 = t2 + 1
```

Zona de declaración de variables

Zona de sentencias de código



Código de tres direcciones

```
t3 = t2 == 11

if_false t3 goto eti1

writes s1

halt
```

Considerando las explicaciones del apartado anterior sobre compiladores, podremos deducir que cualquier código de tres direcciones se puede definir con la siguiente estructura XML:

```
<component id = "assign" num = "3">
            <item type = "var" cid = "1"/>
            <item type = "=" cid = "2"/>
            <item type = "varnum" cid = "3"/>
</component>
<component id = "add" num = "5">
            <item type = "var" cid = "1"/>
            <item type = "=" cid = "2"/>
            <item type = "varnum" cid = "3"/>
            <item type = "+" cid = "4"/>
            <item type = "varnum" cid = "5"/>
</component>
```

Si nos fijamos en la estructura de la asignación podemos observar que:

1. El antecedente de la asignación es una variable, por consiguiente definiremos un tipo reservado en UNI-ASM para definir que el componente tiene un ítem *variable*.
2. De igual forma que el caso anterior, la asignación tiene un elemento constante (no definido en el lenguaje UNI-ASM), "=", que es constante y representa el símbolo de la igualdad.
3. Por último, el tercer ítem del componente asignación puede ser bien una variable o bien una constante literal numérica.

Gracias a la estructura proporcionada por el tag *component* es posible definir

expresiones y sentencias en el lenguaje UNI-ASM.

Otra vez mirando el ejemplo, la suma (add) se resolvería de igual manera que se ha planteado para el caso de la asignación

El fichero XML de transformación, que en este caso está definido para el procesador Intel IA64 bits, es leído por la aplicación Ruby para la gestión de las clases internas que posteriormente se explicarán.

Con el fin de poder declarar variables, existe una sección en el lenguaje llamada *data* donde se podrán crear variables y especificarles un tipo de datos. La forma de crear variables también debe ser especificada en el fichero XML de la misma forma como se hizo para la asignación y la suma, es decir mediante el tag *component*. De esta manera la definición de datos viene proporcionada por el siguiente código:

```
<typeconfig>
      <type id = "byte" bytes = "1"/>
      <type id = "word" bytes = "2"/>
      <type id = "string" bytes = "1"/>
      <type id = "integer" bytes = "8"/>
</typeconfig>

<language>
      <comment-line start="'" />
      <comment-block start = "\(-" end = "-\)" />
      <component id = "type" num = "3" isData = "true">
            <item type = "var" cid = "1"/>
            <item type = ":" cid = "2"/>
            <item type = "type" cid = "3"/>
      </component>
      <component id = "string" num = "3" isData = "true">
            <item type = "var" cid = "1"/>
            <item type = "=" cid = "2"/>
            <item type = "string" cid = "3"/>
      </component>
```

En la sección typeconfig se procede a definir todos los tipos de datos utilizados, indicando los bytes de memoria que ocupan. En consecuencia, dentro de la sección *language* se definen los componentes que representarán la regla gramatical de declaración de variable.

Otra consideración a tener en cuenta es la definición de cadena alfanumérica. Ésta se define mediante la palabra clave "*string*" en el id de *component*. Ahí se especificará cómo se declara y se asigna a una variable una cadena.

Por último, para distinguir los componentes de código de los componentes de declaración de datos, habrá que recurrir al atributo XML ***isData*** que indica que estos componentes de definición del lenguaje son de declaración de datos, y que por consiguiente, y sólo en caso de que se escriban en la zona de código, se generará un error de sintaxis por el compilador de UNI-ASM.

Una vez explicadas las nociones básicas sobre UNI-ASM es hora de revelar la estructura del fichero de transformación. El fichero de transformación es un fichero de texto plano en formato XML que contiene las siguiente estructura básica que posteriormente se desarrollará en detalle:

```
<uniconfig>
      <typeconfig>
            <type id = "" bytes = ""/>

      </typeconfig>

      <asmconfig>
            <basictype bytes = ""/>
            <stack>
                  <addregister>

                  </addregister>
                  <restoreregister>
                  </restoreregister>
            </stack>
            <inmediate value = ""/>
            <reference value = ""/>
      </asmconfig>

      <language>
            <comment-line start="'" />
            <comment-block start = "" end = "" />
            <component id = "" num = "”>
                  <item type = "" cid = "1"/>
            </component>
            <assembler id = "">
            </assembler>
      </language>
</uniconfig>
```

La estructura básica está compuesta de tres partes fundamentales, ésta son:

- **typeconfig:** zona de definición de tipos de datos
- **amsconfig:** zona de especificación de requisitos del lenguaje ensamblador de

destino.

- **language**: donde se definen los componentes tanto de declaración de variables como de sentencias de código.

Nos centraremos ahora en la sección de *language*. Esta sección contiene el tag *component* en el cual deben especificarse los siguientes elementos y atributos:

- **id:** identificador de componente, tal como: add, assign, etc.
- **num:** número de ítem dentro del componente
- **item:**
    - **type:** tipo del item. Puede ser:
        - **var:** únicamente una variable
        - **varnum:** variable alfanumérica o constante literal numérica
        - **num:** únicamente un número
    - **cid:** identificador del ítem (debe ser unívoco).

Dentro de la sección *language* y con la finalidad de generar código se define la sección *assembler*, con la siguiente estructura:

- **id:** debe ser el mismo identificador que se asignó la "id" de su respectivo *component*.

Una vez definido el tag assembler se puede proceder a la escritura del fragmento de código en ensamblador específico de dicha máquina para el componente en cuestión.

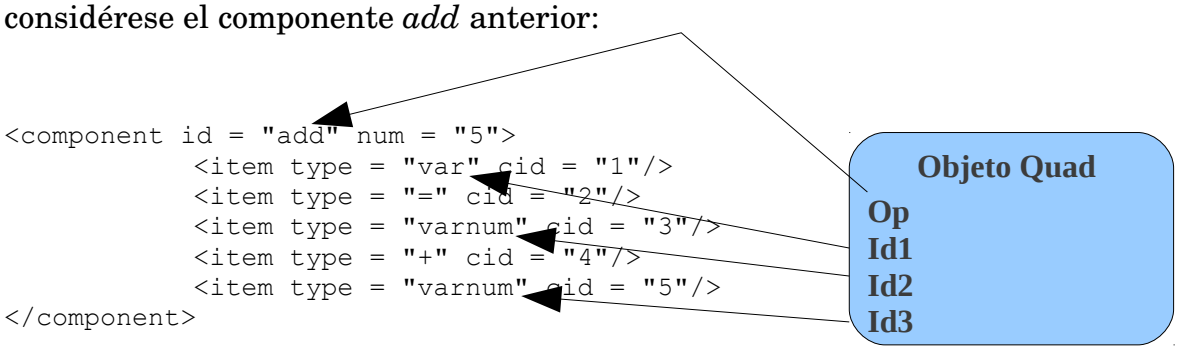
Para ilustrar un ejemplo, supongamos el siguiente componente:

```
<component id = "add" num = "5">
                <item type = "var" cid = "1"/>
                <item type = "=" cid = "2"/>
                <item type = "varnum" cid = "3"/>
                <item type = "+" cid = "4"/>
                <item type = "varnum" cid = "5"/>
</component>
```

le corresponderá la siguiente definición en relación a su CPU:

```
<assembler id = "add">
      mov eax, id2
      add eax, id3
      mov id1, eax
</assembler>
```

Como se puede observar en el fragmento de código se especifican la instrucción *mov* del ensamblador NASM con el registro de 32 bits EAX y con el identificador id2. Para entender mejor la dinámica de los identificadores, es necesario tener presente los conceptos de cuádrupla. El lenguaje UNI-ASM transforma los elementos de los tags componente – exactamente los elementos variables, números y etiquetas – en objetos Q*uadruple*[3] que representan un cuarteto de código intermedio. Estos objetos Q*uadruple* constan de un atributo por cada operando. De esta forma la clase viene representada por la tupla quadruple(op, id1, id2, id3). A modo de ejemplo considérese el componente *add* anterior:



3 Este objeto se describirá con mayor precisión en el próximo apartado

Utilizando estos identificadores del objeto Quadruple se consigue una equivalencia entre los item de los componentes y el orden del identificador. Los valores de los identificadores pueden ser o bien valores numéricos constantes o bien identificadores almacenados en la Tabla de Símbolos o de Temporales.

Si suponemos que id1 es la variable (t1), id2 la variable (t2) e id3 una constante numérica literal, el código objetivo final generado sería:

```
mov eax, [sp + 8]
add eax, [sp + 16]
mov [sp + 0], eax
```

Los desplazamientos 8 y 16 es porque estamos trabajando con posiciones de memoria de 64 bits (qwords de x86_64), aunque luego es convertido a un registro de 32 bits (EAX).

Por último queda comentar la sección *asmconfig*. Esta sección es la responsable de la configuración de la máquina o CPU de destino. En esta estructura del lenguaje de transformación XML indicaremos, entre otros elementos:

- **basictpye:** Se indica el peso del elemento básico de medida en el lenguaje ensamblador. Por ejemplo en el caso de la versión x86_64 el peso de medida básico de la CPU es 1 byte. Por lo tanto cuando en la sección *typeconfig* indicamos que el tipo *qword* tiene 8 bytes, el compilador UNI-ASM trabajará haciendo la división entre el tipo *qword* / *basictype*. Sin embargo en la configuración del ensamblador experimental Winens2001.xml el tipo básico son 2 bytes, puesto que todas las posiciones y los registros de su arquitectura son *word* (2 bytes).
- **addregister:** Establece el código para la disposición del registro de activación.
- **restoreregister:** Establece el código para la recuperación de la pila.

- **inmediate**: Define cómo se especifica el valor inmediato en el lenguaje ensamblador de destino. Por ejemplo: #4, etc.
- **reference value:** Define, de igual modo que en el caso anterior, el formato de acceso por direccionamiento relativo. Es decir los tipos de acceso a memoria para variables. Por ejemplo:
  - En Winens2001: mov .r1, **#5[.ix]**
  - En x86_64: mov eax, **[sp + 0]**
  - En MIPS32: sw $t1, **20($t9)**
  - En 68000: move.w d3, **12(a0)**

5.4 Palabras reservadas del lenguaje de representación

En XML:

uniconfig: indica comienzo de configuración.

typeconfig: indica comienzo de configuración de tipos.

asmconfig: indica comienzo de configuración CPU.

basictype: tipo básico.

stack: parte concerniente a la gestión de la pila del sistema.

addregister: código de establecimiento registro de activación.

restoreregister: código de restablecimiento del registro de activación.

inmediate: especificación del modo de direccionamiento inmediato.

reference: especificación del modo de direccionamiento relativo.

language: comienzo de sección de declaración de componentes.

comment-line: definición de comentario de una sola línea.

comment-block: definición de comentario de bloque.

component: define un componente. Cuando id = “type” o ”string” se están definiendo cómo se especifican los tipos de datos y la cadena alfanumérica

respectivamente.

item: item de un componente.

assembler: define el código ensamblador de destino.

En UNI-ASM:

data: sección de datos.

code: sección de código.

Por lo demás el lenguaje UNI-ASM queda abierto para configuración personalizada. Es decir, el usuario además de poder crear una especificación XML para una CPU de destino, es libre de crear las instrucciones de tres direcciones que desee para ampliar el lenguaje.

5.5 Detalles del prototipo realizado

El prototipo ha sido desarrollado enteramente en Ruby con el fichero de lenguaje de transformación implementado en XML. A continuación se muestra el diagrama de clases del prototipo propuesto:

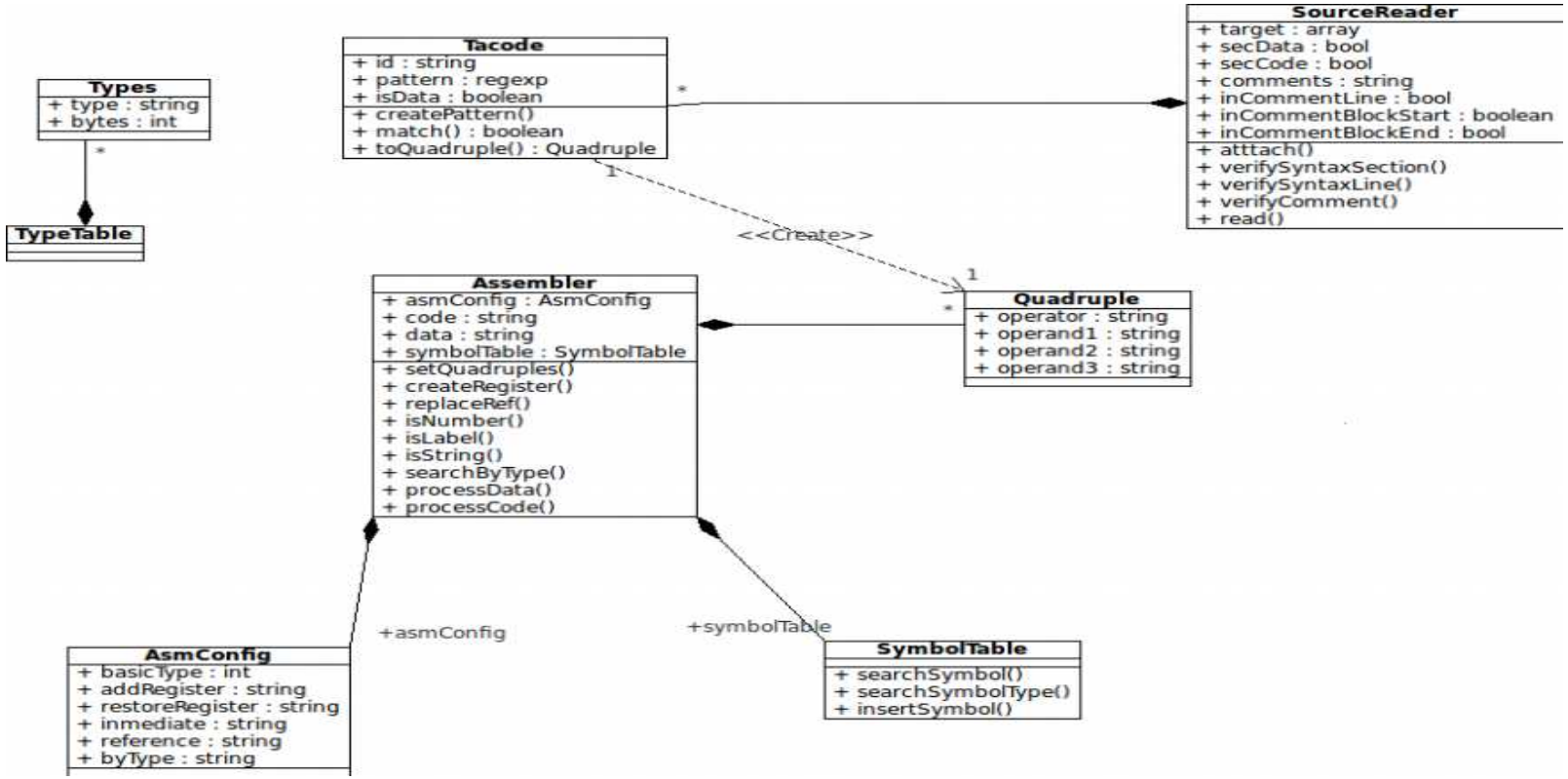


Figura 12. Diagrama de clases del compilador UNI-ASM

En la figura 12 se aprecian las 8 clases principales de las que consta el compilador. La clase **SourceReader**, que es la responsable de leer el código fuente en lenguaje UNI-ASM y comparar con las estructuras **Tacode** que representan cada uno de los componentes de los que consta el lenguaje. Cada clase Tacode contiene un patrón en expresión regular que representa el componente en cuestión. *Para cada línea del lenguaje UNI-ASM que coincide con un patrón regular definido dentro de una clase Tacode se generará un objeto Quadruple.*

La clases **Types** y **TypeTable** son las encargadas de mantener una lista de tipos en el sistema. La clase **Assembler** representa la segunda fase de la compilación puesto que será la encargada de generar el código objeto definitivo, basándose en la información de las cuádruplas generadas por la clase Tacode. Junto con la información proporcionada por la clase **SymbolTable**, que representa una lista de con los identificadores utilizados en el programa, y la clase **AsmConfig**, encargada de suplir información sobre las características principales del lenguaje ensamblador de destino, la clase Assembler iterará en una lista de objetos **Quadruple** e irá generando la sección de declaración de datos así como la secuencia de instrucciones y etiquetas que representa el lenguaje ensamblador de destino.

En relación a la vista dinámica del sistema, el funcionamiento es muy similar al explicado en los apartados anteriores. En primer lugar se observa la clase que representa el programa principal en Ruby. Esta clase será la responsable de leer el fichero XML en formato de texto y de crear la Tabla de Tipos representada por la clase **TypeTable**. Posteriormente se vuelve a leer de nuevo el fichero XML y se generan los objetos **Tacode** que serán adjuntados al objeto Observer siguiendo un patrón de diseño GoF. Esta clase Observer no es más que la clase **SourceReader** cuya finalidad es comparar cada línea de código del lenguaje UNI-ASM con los patrones expresados en la lista de objetos Tacode.

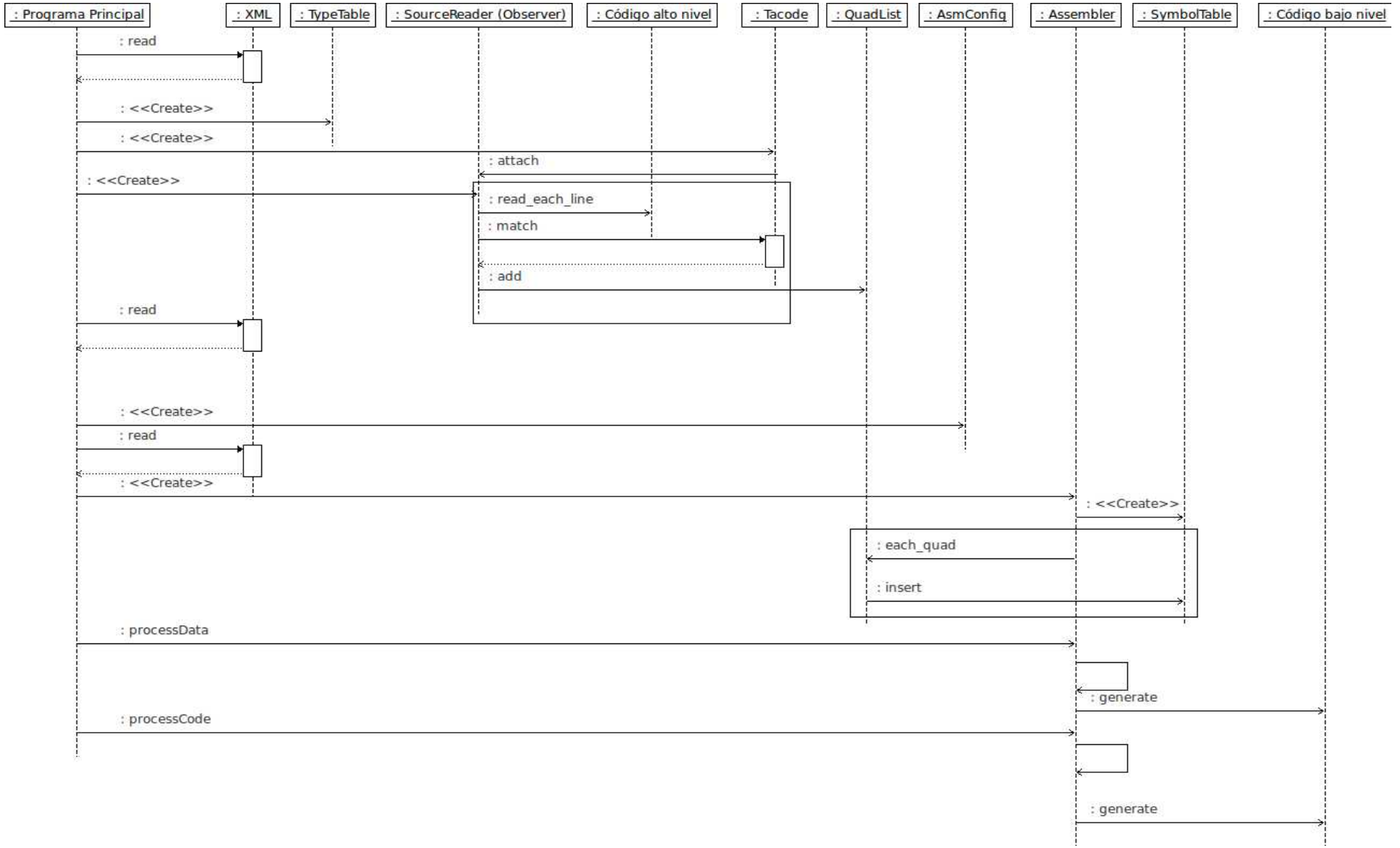


Figura 13. Diagrama de interacción del funcionamiento del proceso de traducción de UNI-ASM

Por cada objeto Tacode que haga la función match *true* se añadirá al la lista de objetos **Quadruple** cuya finalidad es representar la esencia de la instrucción de alto nivel. Posteriormente se volverá a leer el fichero XML para la creación del objeto **AsmConfig**. En la segunda fase del proceso de traducción se instanciará la clase **Assembler** que iterará por la lista de objeto Quadruple para crear la Tabla de Símbolos y generar la sección de datos y de código del código objetivo final.

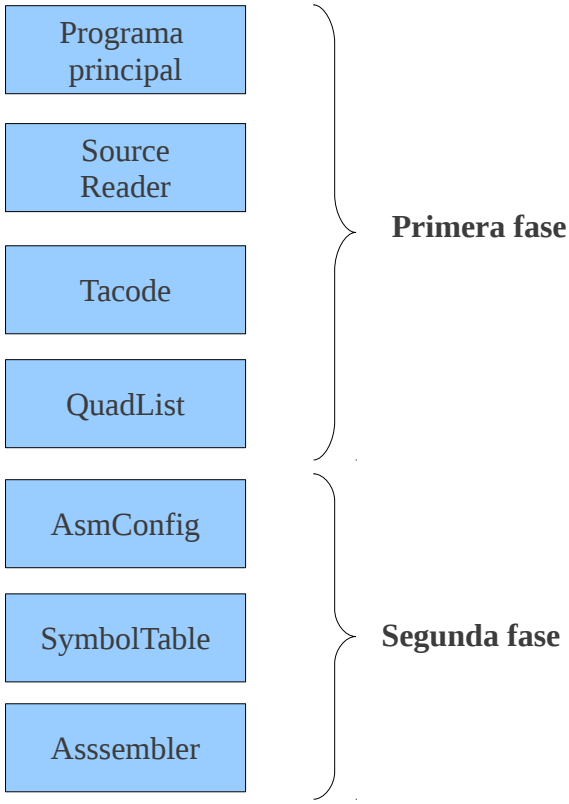


Figura 14. Clases principales involucradas en las dos fases de la traducción

La última fase del compilador añade unos ficheros de *template* en Ruby para que el usuario pueda extender la configuración de su ensamblador. Este fichero de *template* tiene dos partes vitales: <%=data%> y <%= code %>. En general la estructura para un fichero de *template* se muestra en el código x86_64 siguiente:

```
; Template file for x86_64 assembler model

      SECTION .data
fmti: db "%i", 10, 0
fmts: db "%s", 10, 0

      <%= data %>

      SECTION .text
      extern printf
      global main, _start

main:
_start:

<%= code %>

      mov ebx, 0
      mov eax, 1
      int 0x80
```

En la sección de *data* es donde se ubicará el código ensamblador para la declaración de datos y *code* es donde se ubicará el código ejecutable proveniente de la compilación de UNI-ASM. Este fichero es fundamental para el correcto funcionamiento del compilador. La ventaja que tiene el mismo es poder ampliar bajo demanda los requerimientos del programador, consiguiendo de esta forma mayor granularidad y un diseño a medida (*ad-hoc*).

5.6 Casos de uso de la pruebas de demostración

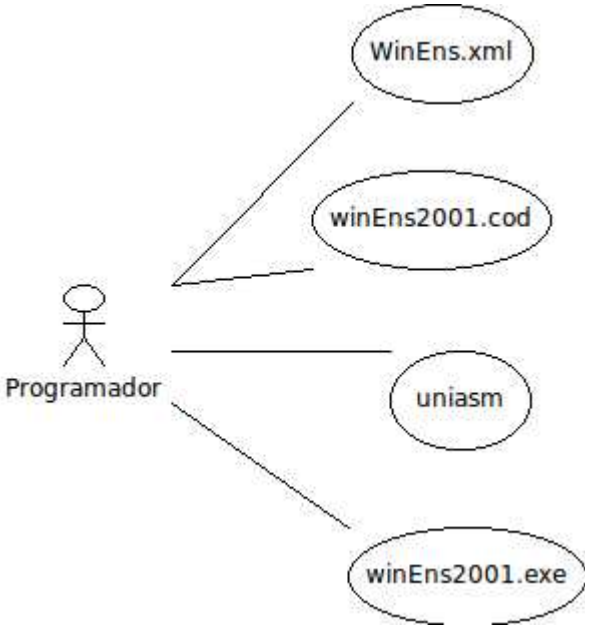


Figura 15. Casos de uso en el ensamblador WinEns2001

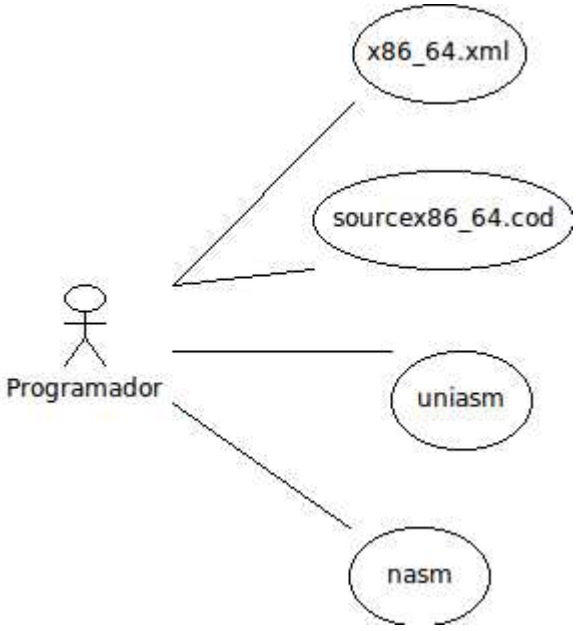


Figura 16. Casos de uso en el ensamblador x86_64

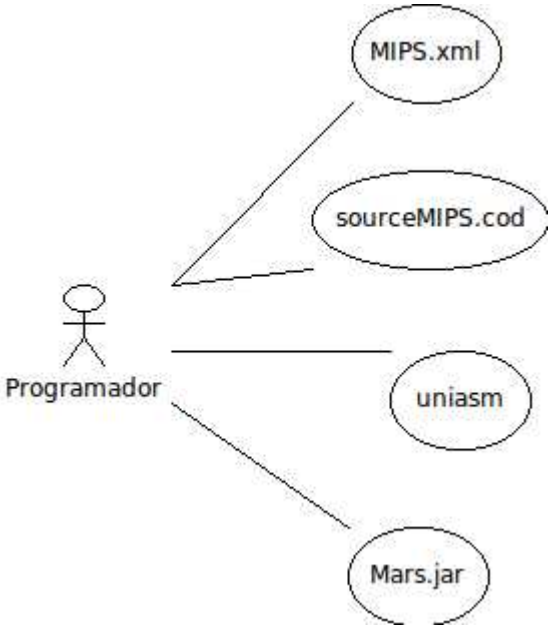


Figura 17. Casos de uso en el ensamblador MIPS32

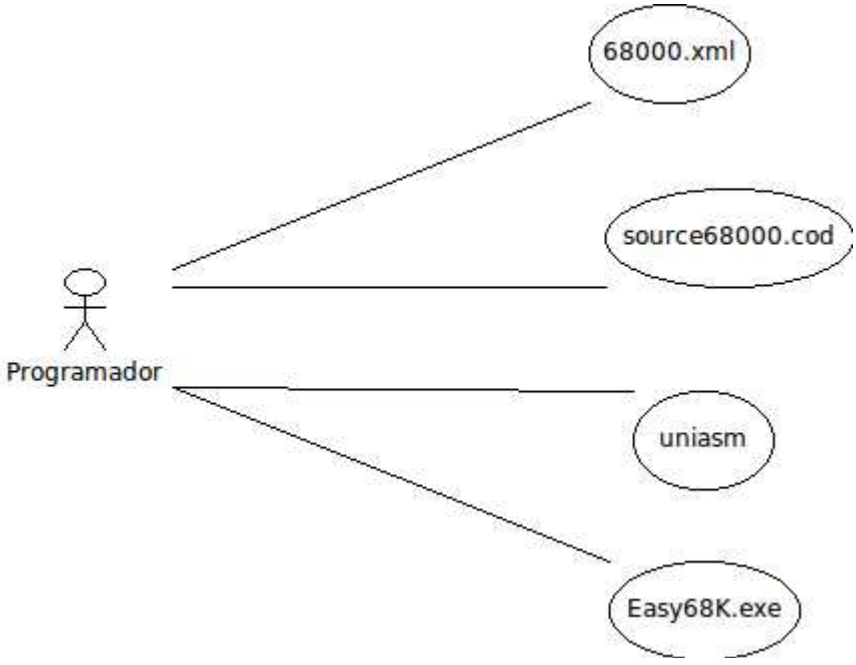


Figura 18. Casos de uso en el ensamblador Motorola 68000

6. Evaluación y comparación de la solución

Al ejecutar los algoritmos sobre la especificación hardware descrita en la **sección 4** se obtienen los siguientes resultados[4]:

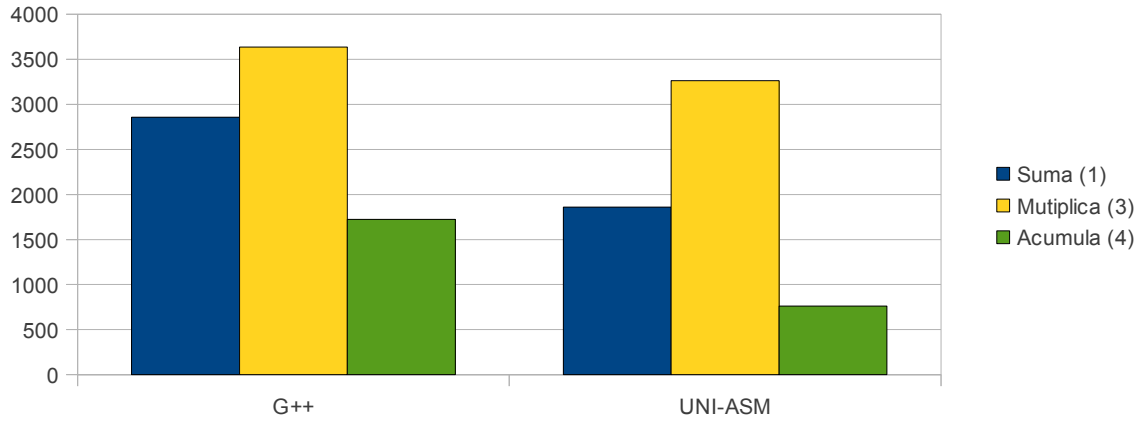


Figura 4. Tiempos de la primera ejecución

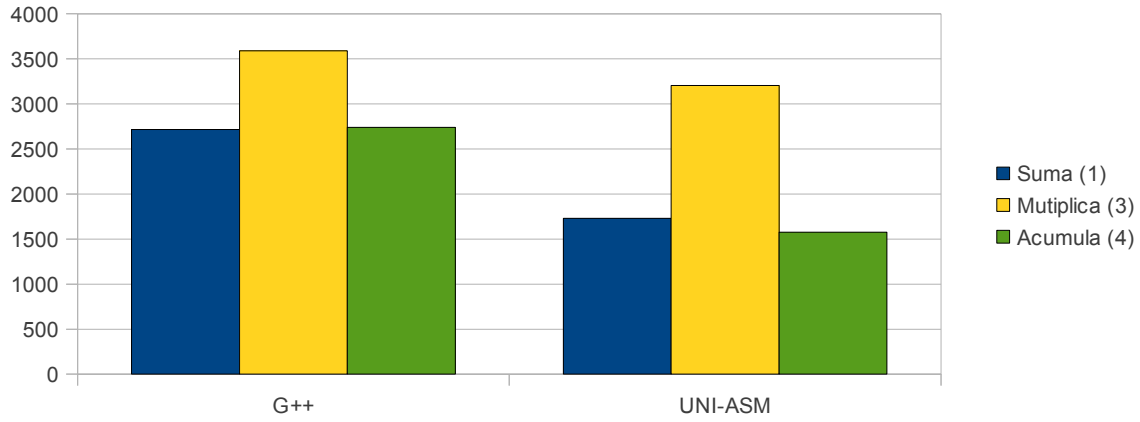


Figura 4. Tiempos de la segunda ejecución

4 **Importante: Se ha compilado con las optimizaciones 2 y 3 de Gcc y en 64 bits**
g++ -o programa programa.cpp -m64 -O2
g++ -o programa programa.cpp -m64 -O3

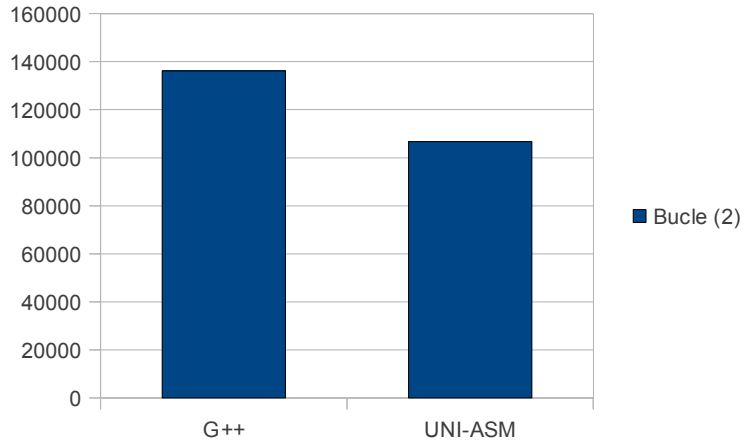


Figura 5. Tiempos para la primera ejecución de Bucle (2)

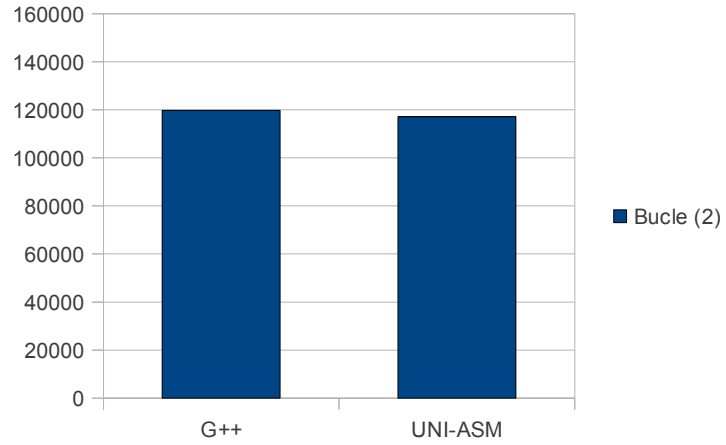


Figura 5. Tiempos para la segunda ejecución de Bucle (2)

Para concluir este apartado de evaluación de las prestaciones del lenguaje UNI-ASM se incluirá a continuación las mejoras porcentuales de rendimiento conseguido. La fórmula utilizada para la deducción de la mejora ha sido:

$$mejora = \frac{(tiempo\,versión\,Gcc - tiempo\,versión\,UNI - ASM)}{tiempo\,versión\,Gcc} * 100$$

De esta forma se obtiene los siguientes resultados con la herramienta de benchmark *test.cpp* que se incluye dentro de los directorios de código fuente:

Mejora %	Suma g++	Suma2 UNI-ASM
37.069883 %	2819 microsegundos	1774 microsegundos

Mejora %	Multi g++	Multi2 UNI-ASM
12.784013 %	3653 microsegundos	3186 microsegundos

Mejora %	Bucle g++	Bucle UNI-ASM
6.889984 %	124427 microsegundos	115854 microsegundos

Mejora %	Acumula g++	Acumula UNI-ASM
38.814601 %	2767 microsegundos	1693 microsegundos

Tabla 4. Resultados obtenidos de la evaluación

Es evidente el aumento de rendimiento en todos los casos evaluados. Las mejoras oscilan entre el 7% y el 39% si se redondea al alza. Los mejores rendimientos se obtienen en los casos *Suma* y *Acumula* con casi un 39% de mejora. Únicamente en el caso del bucle de 10000 iteraciones el rendimiento decrece considerablemente hasta converger con su homólogo en g+, lo que indica que ambos códigos ensambladores son muy similares.

7. Conclusiones y trabajos futuros

El presente trabajo es únicamente una muestra del alcance y la potencia a la cual puede llegar un lenguaje compilado y orientado a varias plataformas hardware al mismo tiempo. El lenguaje UNI-ASM se considera solamente como un prototipo de orientación para las especificaciones y las tareas clave de dicha lógica de compilación.

Como consecuencia, el lenguaje contiene muchas lagunas y huecos incompletos que deben completarse en trabajos futuros. Faltaría por tanto, unos ficheros de configuración más completos, una especificación completa y adaptativa de los modos de direccionamiento, así como de una fase adicional de optimización de código. Aún sin la fase de optimización de código los resultados obtenidos (según las pruebas realizadas en el anterior apartado) son bastante satisfactorias y prometedoras.

Gracias a este trabajo de investigación proporcionado por la asignatura de Generación Automática de Código del Máster en investigación en Ingeniería de Software y Sistemas Informáticos e impartida por el profesor D. Ismael Abad Cardiel, he aprendido a discurrir en cuanto a la importancia de la innovación y su aplicabilidad en el mundo real, que son desde mi punto de vista las piezas clave para una investigación de calidad y excelencia.

8. Manual de usuario para las pruebas

Se ha implementado el algoritmo de la multiplicación para varias plataformas Winens2001, x86_64, MIPS y 68000. Para[5] todas estas plataformas se han definido ficheros de configuración XML para el lenguaje de transformación, se ha procedido a escribirlo en el lenguaje UNI-ASM y se ha procedido a compilador y ejecutarlos. Para la correcta prueba en cada una de las siguientes plataformas se recomienda seguir las siguientes instrucciones:

8.1 Plataforma Winens2001

1. Se ha creado un fichero ***winens.sh*** para compilar el fichero de prueba.
2. Ejecutar **Winens.exe** que se encuentra dentro de recursos.
3. Una vez en ejecución, pulsar

   Archivo->Abrir y ensamblar->sourceWinEns.asm

4. Probar

8.2 Plataforma x86_64

1. Ejecutar directamente **x86_64.sh** desde el shell. Este fichero compilará y llamará automáticamente al ensamblador NASM. Por último también ejecutará el fichero final.
2. **Importante:** Instalar NASM de los repositorio de Linux. En mi caso he utilizado el **NASM 2.07**

8.3 Plataforma MIPS

1. Ejecutar el fichero **mips.sh** desde el shell.
2. Abrir el programa **Mars_4_2.jar** del directorio de recursos.
3. Una vez ejecutado, pulsar en **File->Open->SourceMIPS.asm**
4. Pulsar en Run->Assembler

5 Ver casos de uso (sección 5.6)

5. Pulsar el botón de Play

8.4 Plataforma 68000

1. Ejecutar el fichero **68000.sh**
2. Instalar el programa **SetupEASy68K.exe** incluido dentro de recursos.
3. Ejecutar **Easy68K**[6]
4. Pulsar en **File->Open File->(*.*)->source68000.asm**
5. Pulsar el **Play** y luego **execute**
6. Volver a pulsar **Play**

Nota importante: La versión de Ruby utilizada para la programación del trabajo de investigación es:

```
ruby 1.8.7 (2010-01-10 patchlevel 249) [x86_64-linux]
```

6 Preferiblemente con Wine

9. Bibliografía

[1] David Garlan and Mary Show. *An Introduction to Software Architecture.* School of Computer Science. CMU-CS-94-166. January 1994.

[2] I. Abad Cardiel. *Trabajos de Investigación. Generación Basada en Paradigmas.* ISSI-UNED. 2012.

[3] Jack Herrignton. *Code Generation in Action*. Manning 2003.

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Desing Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley 1995.

[5] Stephen J. Mellor, Kendall Scott, Axel Uhl, Dirk Weise. *MDA Distilled*. Addison-Wesley. 2004.

[6] John Levine. *Flex & Bison*. O'Reilly. 2009.

[7] *ENS2001 Manual de usuario v1.1.* 2003.

[8] José Mª Angulo, Enrique M. Funke. M*icroprocesadores Avanzados: 386 y 486. Introducción al PENTIUM y PENTIUM PRO*. Paraninfo. 1997.

[9] David A. Patterson, John L. Hennesy. *Organización y Diseño de Computadores: La interfaz hardware/software*. McGrawHill 1994.

[10] David A. Patterson, John L. Hennesy. A*rquitectura de computadores: un enfoque cuantitativo.* McGrawHill. 1993.

[11] Motorola. *M68000 8/16/32 bit microprocessors. User's Manual*. Motorola 1994.

[12] Kenneth C. Louden. *Construcción de Compiladores*. Thomson 1997.

[13] Perdita Stevens, Rob Pooley. *Utilización de UML en Ingeniería del Software con Objetos y Componentes.* Pearson Addison - Wesley.

[14] Randal E. Bryant , David R. O'Hallaron. *x86-64 Machine-Level Programmin*g . September 9, 2005 .

[15] Dan Matthews, *A Clock Design Using the PIC16C54 for LED Displays and Switch Inputs, Microchip Technology Inc.*

10. Referencias Web

[1] x86registers

http://www.eecg.toronto.edu/~amza/www.mindsec.com/files/x86regs.html

[2] Ruby Language Reference Manual

http://web.njit.edu/all_topics/Prog_Lang_Docs/html/ruby/

[3] Winens2001

http://usuarios.multimania.es/ens2001/

[4] NASM

http://www.nasm.us/

[5] Mars Assembler and Runtime Simulator

http://courses.missouristate.edu/KenVollmar/MARS/

[6] Easy68K

http://www.easy68k.com/

[7] Easy68K examples

http://www.easy68k.com/easy68kexamples.htm

11. Herramientas software

- Sistema Operativo Ubuntu Linux 10.04 LTS
- Editor de textos: OpenOffice 3.2
- Editor UML: Umbrello
- Editor de texto: Gedit
- Lenguaje de programación: Ruby[7]
- Verificador XML: Mozilla Firefox

7 Ver página 39

A. ANEXO1: Código fuente del compilador UNI-ASM

```
#####################################################################
####                                                             #####
####                                                             #####
####   Trabajo de Investigación GAC                              #####
####   UNI-ASM: Lenguaje Ensamblador Universal                   #####
####   Alumno: Carlos Jiménez de Parga Bernal - Quirós           #####
####                                                             #####
####   ---------------------------------------------------       #####
####   MASTER INVESTIGACION EN INGENIERIA DE SOFTWARE Y           #####
####   SISTEMAS INFORMATICOS - ETSII - UNED                      #####
####                                                             #####
####                                                             #####
#####################################################################


# Inclusión de librerías
require 'rexml/document'
require 'erb'

# Clase para unir dos expresiones regulares
class Regexp
      def +(re)
            Regexp.new self.to_s + re.to_s
      end
end

# Clase para albergar contenidos de la tabla de tipos
class Types
      def initialize(type, bytes)
            @type  = type
            @bytes = bytes
      end
end

# Clase para almacenar información sobre los comentarios
class Comments
      def initialize(cls, cbs, cbe)
            @lineStart = cls
            @blockStart = cbs
            @blockEnd = cbe
      end

      attr_reader:lineStart
      attr_reader:blockStart
      attr_reader:blockEnd
```

```ruby
end

# Información para almacenar instrucciones por tipo de datos específico

class ByTypeComponent
	def initialize(instruction, type)
		@instruction = instruction
		@type = type
	end
	attr_reader:instruction
	attr_reader:type
end

# Clase para el código de tres direcciones
class Tacode
	def initialize(id, isData)
		@id = id # Identificación
		@pattern = /\s*/ # Patrón
		@isData = (isData=="true") ? true : false # Si código tres direcciones es de declaración de variables
	end

	# Establece componentes
	def setComponents(components)
		@components = components
	end

	# Crea patrón
	def createPattern(typeTable)

		@components.each { |compo| # Lee sus componentes
			case compo.type
				when "num" # número
					@pattern+= /(\d+)\s*/
				when "var" # variable
					@pattern+= /([a-z][a-z0-9]*)\s*/
				when "string" #string
					@pattern+= /(\".*\"|\'.*\')/
				when "lab" # label
					@pattern+= /([a-z][a-z0-9]*)\s*/
				when "varnum" # variable o número
					@pattern+= /(\w+)\s*/
				when "type" # tipo de datos
					i = 0
					typpat = ""
					typeTable.each { |k,type|
						typpat+= "#{k}"
						typpat+= "|" if i != typeTable.length - 1
						i = i + 1
```

```
                        }

                        @pattern += /(#{typpat})/
                    else  # Inserta el operador en el patrón
                        @pattern+=  "\s*\\#{compo.type}\s*"
                end
            }
            @pattern+= /\s*$/ # Fin de patrón

        end

        # Detecta coincidencias
        def match(line)
            line.match(@pattern)
        end


        # Crea objeto cuádrupla
        def toQuadruple(line)
            operand1 = ""
            operand2 = ""
            operand3 = ""
            if (line =~ pattern) # Almacena los operandos
                operand1 = $1 if $1
                operand2 = $2 if $2
                operand3 = $3 if $3
            end
            Quadruple.new(@id, operand1, operand2, operand3) # @id es el operador
        end

        attr_reader:id
        attr_reader:isData
        attr_reader:pattern

end

# Clase para albergar los componentes del código en tres direcciones
class TacodeComponent
        def initialize(type, cid)
            @type = type
            @cid  = cid
        end

        attr_accessor:type
        attr_accessor:cid
end

# Clase cuádrupla
```

```ruby
class Quadruple
	def initialize(operator, operand1, operand2, operand3)
		@operator = operator
		@operand1 = operand1
		@operand2 = operand2
		@operand3 = operand3
	end

	attr_reader:operator
	attr_reader:operand1
	attr_reader:operand2
	attr_reader:operand3

end

# Clase para leer y manejar el fichero de código fuente
class SourceReader
	def initialize(source, comments)
		@target = [] # Lista de objetos Tacode
		@secData = false # Flag para comienzo de datos
		@secCode = false # Flag para comienzo de código
		@comments = comments
		@inCommentLine = false
		@inCommentBlockStart = false
		@inCommentBlockEnd = false
		begin # Lee fichero de código fuente
			fh = File.open(source)
			@contents = fh.read
			fh.close
		rescue => error
			print "Error reading source file: #{error}\n"
			exit -1
		end
	end

	# Adjunta objeto Tacode a la lista. Patrón Observer
	def attach(tacode)
		@target.push tacode
	end

	# Detecta comienzo de etiqueta de definición de datos y de código
	def verifySyntaxSection(line)
		if (line.match(/\s*data:\s*$/))
			@secData = true
			@secCode = false
			print "Find data section\n"
			return true
		elsif (line.match(/\s*code:\s*$/))
```

```
            @secCode = true
            @secData = false
            print "Find code section\n"
            return true
      end
end

# Verifica linea para indagar si está en la zona correcta
def verifySyntaxLine(isData)
      if (@secData && !isData)
            print "Syntax error. Code in data section.\n"
            return true
      end
      if (@secCode && isData)
            print "Syntax error. Data in code section.\n"
            return true
      end
      return false
end

def verifyComment(line)
      if (line =~ /#{@comments.lineStart}/) then
            @inCommentLine = true
      end

      if (line =~ /#{@comments.blockStart}/) then
            @inCommentBlockStart = true
      end

      if (line =~ /#{@comments.blockEnd}/) then
            @inCommentBlockEnd = true
      end
end


# Lee fichero de código fuente
def read(quadList)
      contLine = 1 # Contador de líneas
      contErrors = 0 # Contador de errores
      fatalError = false # Flag de error fatal
      @contents.each_line { |line| # Lee fichero línea a línea
            verifyComment(line)
            if (@inCommentLine) then
                  @inCommentLine = false
                  contLine = contLine + 1 # Incrementa contador de líneas
                  next
            end
```

```
	if (@inCommentBlockEnd) then
		@inCommentBlockStart = false
		@inCommentBlockEnd = false
		contLine = contLine + 1 # Incrementa contador de líneas
		next
	end

	if (@inCommentBlockStart) then
		contLine = contLine + 1 # Incrementa contador de líneas
		next
	end


	if verifySyntaxSection(line) then # Verifica si está en bloque correcto
		contLine = contLine + 1
		next
	end
	error = true

	print line
	@target.each { |trg| # Para cada objeto Tacode

		if trg.match(line) then # Si coincide con línea
			print "La linea #{line} coincide con #{trg.id}\n"
			if verifySyntaxLine(trg.isData) then # Verifica línea en bloque correcto
				contErrors = contErrors + 1
				fatalError = true
			end
			quadList.push trg.toQuadruple(line) # Añade cuádrupla
			error = false
			break
		end

		}

	if (error && line.strip! != "") # Imprime error
			print "Syntax error in line num: #{contLine}. Pattern does not match.\n"
			fatalError = true
			contErrors = contErrors + 1
	end
	contLine = contLine + 1 # Incrementa contador de líneas

}

if (fatalError) then # Imprime error fatal y termina ensamblaje
	print "Assembling errors: #{contErrors}\. Finish."
	exit -1
end
```

```ruby
	end

end


# Clase para almacenar información de la CPU concreta
class AsmConfig
	def initialize(basicType, addRegister, restoreRegister, inmediate, reference, byType)
		@basicType		= basicType		# Tipo básico de la CPU
		@addRegister		= addRegister	# Sentencias para actualizar el registro de activación
		@restoreRegister	= restoreRegister # Sentencias para reestablecer el registro de activación
		@inmediate		= inmediate		# Indicador del modo de direccionamiento inmmediato de esa CPU
		@reference		= reference		# Indicador del modo de direccionamiento relativo de esa CPU
		@byType			= byType		# Indicador de la instrucción por tipo de datos
	end

	attr_reader:basicType
	attr_reader:addRegister
	attr_reader:restoreRegister
	attr_reader:inmediate
	attr_reader:reference
	attr_reader:byType
end


# Entrada de la tabla de símbolos
class SymbolEntry
	def initialize(sym, type, shift)
		@sym	= sym # Símbolo
		@type	= type # Tipo
		@shift	= shift # Desplazamiento dentro del RA
	end

	attr_reader:sym
	attr_reader:shift
	attr_reader:type

end

# Clase tabla de símbolos

class SymbolTable

	# Constructor
	def initialize
		@symbolTable = Hash.new
	end
```

```ruby
	# Busca símbolo  en la tabla de símbolos
	def searchSymbol(symbol)
		@symbolTable["main"].each { |s|
			if (s.sym == symbol) then
				return s
			end
		}

		return nil
	end


	# Busca tipo de dato de un símbolo en la tabla de símbolos
	def searchSymbolType(type)
		@symbolTable["main"].each { |s|
			if (s.type == type) then
				return s
			end
		}

		return nil
	end

	# Inserta símbolo en la tabla de símbolos
	def insertSymbol(sym, type, gaps)
		if @symbolTable["main"] == nil then @symbolTable["main"] = Array.new end
		@symbolTable["main"].push SymbolEntry.new(sym, type, gaps)
	end

end

# Clase para ensamblaje

class Assembler
	def initialize(asmConfig)
		@asmConfig = asmConfig
		@code = "" # Parte de código generado a fichero
		@data = "" # Parte de datos generados a fichero
		@symbolTable = SymbolTable.new # Tabla de símbolos
	end

	attr_reader:symbolTable

	# Establece referencia a cuádruplas
	def setQuadruples(listQuadruples)
		@listQuadruples = listQuadruples
	end
```

```ruby
# Establece tabulación
def setTabulation(tabs)
      tabs.times { @code += "\t" }
end

# Crea registro de activación según las sentencias de declaración de datos especificadas en las cuádruplas
def createRegister(typeTable)
      gaps = 0
      basicType = 0
      @listQuadruples.each {|quad|
            if quad.operator == "type" then
                  @symbolTable.insertSymbol(quad.operand1, quad.operand2, gaps)
                  basicType = typeTable[quad.operand2].to_i / @asmConfig.basicType.to_i
                  gaps = gaps + basicType # Desplazamiento total dentro del RA
            elsif quad.operator == "label" then
                  @symbolTable.insertSymbol(quad.operand1, "label", 0)
            elsif quad.operator == "string" then
                  @symbolTable.insertSymbol(quad.operand1, "string", 0)
            end

      }
      @code = @code + @asmConfig.addRegister.gsub(/\[reg\]/, gaps.to_s)
end

# Reemplaza direccionamiento relativo cuando se inserta en código e inserta en snippet
def replaceRef(asm, operand, id)
      if ((sym = @symbolTable.searchSymbol(operand)) != nil && sym.type!="label" && sym.type != "string") then
            if ((ref = @asmConfig.reference)!=nil) then
                  refAux = ref.gsub(/\[shf\]/,sym.shift.to_s)
                  asm.gsub!(id, refAux)
            end
            return true
      else return false
      end
end

# Detecta si el operando es un número e inserta en snippet
def isNumber(asm, operand, id)
      if (operand =~ /^\d+$/) then
            inm = @asmConfig.inmediate
            asm.gsub!(id, "#{inm}#{operand}")
            return true
      else return false
      end
end

# Detecta si el operando es una etiqueta e inserta en snippet
def isLabel(asm, operand, id)
```

```
            if ((sym = @symbolTable.searchSymbol(operand)) != nil && sym.type == "label") then
                        asm.gsub!(id, operand)
                        return true
            else return false
            end
      end

      # Detecta si el operando de la cuádrupla es un string e inserta en snippet
      def isString(asm, operand, id)
            if ((sym = @symbolTable.searchSymbol(operand)) != nil && sym.type == "string") then
                        asm.gsub!(id, operand)
                        return true
            elsif (sym = @symbolTable.searchSymbolType(operand)) then
                        asm.gsub!(id, sym.sym)
                        return true
            else return false
            end
      end

      # Busca instrucciones específicas para un determinado tipo
      def searchByType(asm, quad)
            asm_dup = asm.dup
            @asmConfig.byType.each {|k, bt|      # Lista los tipos de instrucciones
                  asm_dup.each_line { |al| # Para cada línea del snippet generado
                        if al =~ /^\s*\[(#{k})\]\s*/ then  # Si se utiliza ese tipo de instrucción entonces
                                    bt.each { |it|
                                          ss = nil
                                          case al     # El tipo de instrucción viene definido por el tipo de sus operandos
                                                when /id1/
                                                      ss = @symbolTable.searchSymbol(quad.operand1)
                                                when /id2/
                                                      ss = @symbolTable.searchSymbol(quad.operand2)
                                                when /id3/
                                                      ss = @symbolTable.searchSymbol(quad.operand3)


                                          end
                                          # Si el tipo de instrucción coincide con alguno de los operandos del snippet
entonces
                                          if (ss == nil || it.type == ss.type) then
                                                asm.gsub!(/\[#{k}\]/," #{it.instruction} ")
                                                break
                                           end

                                    }

                        end
```

```
            }
      }
      end

      # Procesa la sección de datos del ensamblador
      def processData(asmCode, typeTable)
            # Crea registro de activación
            createRegister(typeTable)
            autoGenerateIDstr = 0   # Autogenerador de strings
            @listQuadruples.each {|quad|

                  if (quad.operator == "string") then # Operador string. Sustituye la definición de string en datos

                        asm = asmCode[quad.operator].dup
                        asm.gsub!(/id1/, quad.operand1)
                        asm.gsub!(/\[string\]/, quad.operand2)
                        @data += asm
                  elsif  (quad.operand1  =~  /(\'.*\'|\".*\")/)  then  #  String  autogenerador.  Típicamente  como  cadena
constante literal, tipo write("...")
                        rep = $1
                        autoGenerateIDstr = autoGenerateIDstr + 1
                        asm = asmCode["string"].dup
                        asm.gsub!(/id1/,"strID" + autoGenerateIDstr.to_s)
                        asm.gsub!(/\[string\]/, quad.operand1)
                        @symbolTable.insertSymbol("strID" + autoGenerateIDstr.to_s , rep, 0)


                        @data += asm
                  end
            }
            return  @data
      end

      # Procesa la sección de código del ensamblador
      def processCode(asmCode, typeTable)
            fatalError = false
            contErrors = 0
            @listQuadruples.each {|quad|
                  if (quad.operator != "type" && quad.operator != "string") then   # Tipo de operadores diferentes a los
datos
                        asm = asmCode[quad.operator].dup
                        searchByType(asm, quad) ##########################  Reemplaza  tipos  de  operandos  en  snippets
################################
                        if !(replaceRef(asm, quad.operand1, /id1/) || isNumber(asm, quad.operand1, /id1/) || isLabel(asm,
quad.operand1, /id1/) || isString(asm, quad.operand1, /id1/) || (quad.operand1 =="")) then
                              print "Semantic error: variable #{quad.operand1} not declared\n"
                              fatalError = true
                              contErrors = contErrors + 1
```

```
                        end
                        if  !(replaceRef(asm,  quad.operand2,  /id2/)  ||    isNumber(asm,  quad.operand2,  /id2/)  ||
isLabel(asm, quad.operand2, /id2/) || isString(asm, quad.operand2, /id2/) || (quad.operand2 =="")) then
                              print "Semantic error: variable #{quad.operand2} not declared\n"
                              fatalError = true
                              contErrors = contErrors + 1
                        end
                        if !(replaceRef(asm, quad.operand3, /id3/) || isNumber(asm, quad.operand3, /id3/) || isLabel(asm,
quad.operand3, /id3/) || isString(asm, quad.operand3, /id3/) || (quad.operand3 =="")) then
                              print "Semantic error: variable #{quad.operand3} not declared\n"
                              fatalError = true
                              contErrors = contErrors + 1
                        end
                        @code += asm
                  end
            }
            # Error fatal en compilación
            if (fatalError) then
                  print "Assembling errors: #{contErrors}\. Finish."
                  exit -1
            end
            return  @code + @asmConfig.restoreRegister
      end



end

########################################### PROGRAMA PRINCIPAL ###################################3


# Comprueba parámetros correctos
unless ARGV[0] && ARGV[1] && ARGV[2]
      print "usage: uniasm asmconfig.xml model_file template_file\n"
      exit
end

# Abre fichero de configuración del ensamblador
doc = REXML::Document.new(File.open(ARGV[0]).read)

# Crea tabla de tipos
typeTable = Hash.new

# Lee tipos y los almacena en tabla
doc.root.each_element("typeconfig/type") { |type|
            typeTable[type.attributes["id"]] = type.attributes["bytes"]
}
```

```
# Lectura de definición de comentarios

cls = doc.root.elements["language/comment-line"].attributes["start"]
cbs = doc.root.elements["language/comment-block"].attributes["start"]
cbe = doc.root.elements["language/comment-block"].attributes["end"]

comments = Comments.new(cls, cbs, cbe)

# Lee código fuente del ensamblador
sourceReader = SourceReader.new(ARGV[1], comments)

# Lee configuración del ensamblador
doc.root.each_element("language/component") { |compo|
            # Crea objeto tres direcciones
            tacode = Tacode.new(compo.attributes["id"], compo.attributes["isData"])
            # Crea array de componentes
            components = []
            # Lee los componentes del código de tres direcciones
            compo.each_element("item") { |it|
                  components.push TacodeComponent.new(it.attributes["type"],it.attributes["cid"])
            }

            tacode.setComponents(components)
            tacode.createPattern(typeTable)
            # Adjunta el objeto de tres direcciones
            sourceReader.attach(tacode)
      }


# Lista de cuádruplas
listQuadruples = []

# Procesa código fuente
sourceReader.read(listQuadruples)

# Lectura de parámetros de la CPU específica. Ver clase AsmConfig.

basicType = doc.root.elements["asmconfig/basictype"].attributes["bytes"]
addRegister = doc.root.elements["asmconfig/stack/addregister"].text
restoreRegister = doc.root.elements["asmconfig/stack/restoreregister"].text
inmediate = doc.root.elements["asmconfig/inmediate"].attributes["value"]
reference = doc.root.elements["asmconfig/reference"].attributes["value"]


# Estructura Hash para almacenar instrucciones dependientes de tipos

byType = Hash.new
```

```
# Lectura de instrucciones dependientes de tipos

doc.root.each_element("asmconfig/bytype") { |bt|
      bt.each_element("inst") { |ins|
            byType[bt.attributes["id"]] = Array.new() if byType[bt.attributes["id"]] == nil
            btc = ByTypeComponent.new(ins.attributes["id"], ins.attributes["type"])
            byType[bt.attributes["id"]].push btc
      }
}


asmConfig = AsmConfig.new(basicType, addRegister, restoreRegister, inmediate, reference, byType)

# Creación de los snippets

asmCode = Hash.new

doc.root.each_element("language/assembler") { |asm|
            asmCode[asm.attributes["id"]] = asm.text
      }


print " GENERANDO SOLUCION...\n"

# Creación del ensamblador
assembler = Assembler.new(asmConfig)
assembler.setQuadruples(listQuadruples)

# Procesa la parte de datos
data = assembler.processData(asmCode, typeTable)
# Procesa la parte de código
code = assembler.processCode(asmCode, typeTable)

# Genera salida. Siempre con extensión asm
fh = File.open(ARGV[1].gsub(/.cod/,".asm"),"w")
erb = ERB.new(File.open(ARGV[2]).read)

fh.print erb.result(binding)

print "### IMPRIMIENDO CUADRUPLAS ###\n"

p listQuadruples

print "### IMPRIMIENDO TABLA de SIMBOLOS ###\n"

p assembler.symbolTable
```

```
print " OK. \n"
```

A. ANEXO2: Código fuente del programa Multiplica.cod en UNI-ASM

```
(- Ejemplo de prueba del lenguaje UNI-ASM desarrollado
para la asignatura de Generación Automática de Código
del Máster en Investigación en Ingeniería de Software y Sistemas Informáticos -)
' Desarrollado por alumno: Carlos Jiménez de Parga
' Versión x86_64 (64 bits NASM)


data:

      t1: integer
      t2: integer
      t3: integer
      i:  integer
      j:  integer
      x:  integer
      y:  integer
      s1 = "FIN DEL PROCEDIMIENTO"
code:

      t1 = 0
      t2 = 1

      jump eti1

label mult:
      i = 0
      x = 0
label bucle:
      i = i + 1
      x = x + y

      t3 = i == j
      if_false t3 goto bucle
      jump eti2

label eti1:

      t1 = t1 + 1
      writei t1
      writes " X "
      writei t2
      y = t1
      j = t2
      jump mult
label eti2:
```

```
writes " = "
writei x
writes "  "
t3 = t1 == 10
if_false t3 goto eti1

t1 = 0

t2 = t2 + 1

t3 = t2 == 11

if_false t3 goto eti1

writes s1

halt
```