IV. GENERACIÓN DE CÓDIGO

IV.1. El Código Intermedio

Habitualmente los compiladores generan un código intermedio entre el código fuente y el lenguaje máquina propiamente dicho. Este código puede considerarse como el lenguaje de una máquina abstracta que el diseñador del compilador define. El código intermedio sólo requiere para ser considerado como tal, el que pueda ser traducido fácilmente al código de la maquina real.

La definición de este código intermedio se hace de forma que tenga las siguientes ventajas:

? Evitar las limitaciones propias de una maquina real en cuanto a número de registros, alineación de datos, etc...
? Facilitar la optimización de código al ser mas flexible en cuanto a direccionamiento que el código de la máquina real.
? Facilitar la portabilidad del compilador. En general, cuando se requiere que un compilador de un lenguaje funcione sobre maquinas diferentes, la alternativa a construir un compilador distinto para cada maquina consiste en traducir el código objeto a un código intermedio, independiente de cualquier máquina y a continuación escribir un traductor de código intermedio a código máquina para cada una de las máquinas que se desee. A la primera parte de este proceso se le denomina parte frontal del compilador (“*front end* ”), mientras que la segunda es la parte posterior del compilador (“*back end* “). Evidentemente la ventaja de este proceso es que la parte posterior es mas sencilla de realizar que la parte frontal y, por consiguiente, el trabajo total a realizar es menor.

El inconveniente principal que puede achacarse a la generación de código intermedio es que puede producir programas menos eficientes al no aprovechar algunas de las instrucciones especificas de la máquina objeto. Este inconveniente puede soslayarse si existe una adecuada optimización de código máquina propiamente dicho.

IV. 2. Tipos y representaciones del código intermedio

Existen muchas formas de código intermedio, de hecho, el diseñador del compilador puede definir la máquina abstracta que considere mas adecuada al lenguaje fuente o a la clase de máquinas a las que va destinado Las representaciones que más se emplean son

? Árboles semánticos y grafos acíclicos dirigidos
? Código de tres direcciones. Cuartetos, Tercetos o Tercetos indirectos.

Existen algunas máquinas abstractas o lenguajes intermedios ya clásicos en generación de código que se han definido para la compilación de lenguajes concretos, como el código P, que se utiliza en la compilación de Pascal, o la máquina de Warren para la compilación del lenguaje Prolog.

Árboles semánticos y grafos acíclicos dirigidos

Una forma de representar el código generado por un compilador es mediante una estructura de tipos arborescente, a la que se denomina árbol semántico a fin de diferenciarlo del árbol sintáctico o árbol del análisis sintáctico que representa la estructura gramatical.

Así, por ejemplo, para la representación de la sentencia: `a:=b*-c+b*-c,` se tiene:

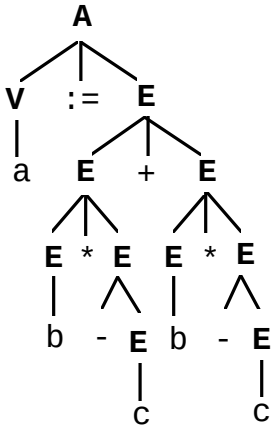


Fig 1a. Árbol sintáctico

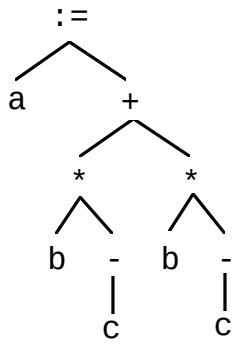


Fig 1b. Árbol semántico

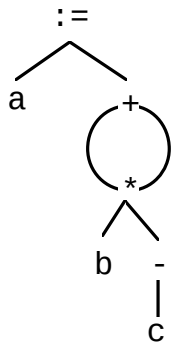


Fig 1c. Grafo acíclico dirigido

En el árbol semántico los nodos son ocupados por operadores y sus operandos se obtienen evaluando las operaciones de sus nodos descendientes.

Pueden usarse grafos acíclicos dirigidos (DAG) para obtener una representación condensada de los árboles semánticos. Cuando se encuentran subestructuras idénticas dentro de un mismo árbol basta usar múltiples referencias a la misma subestructura, sin necesidad de duplicar la subestructura repetida. Esta representación consigue por si misma una mejora considerable en el tamaño y la eficiencia del código generado.

Código de tres direcciones

La representación mediante código de tres direcciones se basa en un conjunto de instrucciones capaces de manejar un máximo de tres direcciones de memoria. En general dos de estas direcciones corresponden a los argumentos y la tercera al resultado. Una instrucción de tres direcciones por ejemplo puede consistir en sumar el contenido de dos direcciones y situar el resultado en una tercera dirección, o saltar a una determinada dirección si el valor contenido en una dirección es mayor que el contenido en otra, etc.

instrucción	arg_1	arg_2	resultado
+:=	x	y	z
if.>.goto	a	b	L

Fig 2. Ejemplos de código de tres direcciones

Usando un adecuado conjunto de instrucciones de este tipo, cualquier sentencia o conjunto de sentencias del código fuente, puede traducirse por una secuencia de instrucciones de tres direcciones. Por ejemplo, la sentencia `a:=b*(c+d)` puede traducirse como la secuencia de instrucciones de tres direcciones:

	instr.	arg_1	arg_2	rest.
(100)	+:=	c	d	t_1
(101)	*:=	b	t_1	a

En donde las letras minúsculas representan las direcciones de memoria asociadas a cada una de las variables del programa. y t_1 representa una dirección de memoria temporal (auxiliar) que permite realizar los cálculos. Para mayor comodidad se escribe el código auxiliar con la notación infija:

(100) $t_1 := c + d$
(101) $a := b * t_1$

Esta representación es equivalente a la representación mediante árbol semántico o mediante grafo acíclico dirigido que se ha visto anteriormente, siempre que se limite la ramificación del árbol a un máximo de dos descendientes. Para obtener la representación en código de tres direcciones basta con recorrer el árbol ascendentemente de izquierda a derecha y generar una variable temporal para cada nodo intermedio del árbol, resumiendo mediante ella la estructura descendiente. Así, por ejemplo, a partir de los grafos de la figura 1b y 1c, se obtienen las siguientes secuencias de código intermedio:

(100) $t_1 := - c$
(101) $t_2 := b * t_1$
(102) $t_3 := - c$
(103) $t_4 := b * t_3$
(104) $t_5 := t_2 * t_4$
(105) $a := t_5$

Fig 3a. Código correspondiente a la fig. 1b

(100) $t_1 := - c$
(101) $t_2 := b * t_1$
(102) $t_3 := t_2 * t_2$
(103) $a := t_3$

Fig 3b. Código correspondiente a la fig. 1c

El conjunto de instrucciones de tres direcciones del código intermedio debe definirse adecuadamente según el lenguaje a compilar, y teniendo en cuenta que cada instrucción del mismo debe convertirse fácilmente en una o varias instrucciones del código máquina. No cabe por tanto definir un lenguaje intermedio universal, pero a título de ejemplo se va a estudiar el tipo de instrucciones que pueden usarse:

- Instrucciones de asignación:

directa:	a **:=** b
de un operando:	a **:=** op_{unario} b
de dos operandos:	a **:=** b $op_{binario}$ c

- Saltos en ejecución:

incondicionales:	**goto** L
condicionales:	**if** a op_{cond} b **then goto** L
etiquetas:	**label** L

- Definición de parámetros y llamada a procedimientos.

definición de parámetros	**param** X
llamada:	**call** p, n

Por lo general se usan conjuntamente varias instrucciones de tres direcciones para realizar la llamada a un procedimiento o función. Primero se definen las variables que van a usarse como parámetros y luego se transfiere el control mediante una instrucción de llamada, indicando la dirección del procedimiento y el número de parámetros.

- Operaciones de direccionamiento indirecto:

en el argumento:	x **:=** y **[**i**]**
en el resultado:	x **[**i**]** := y

- Operaciones con punteros:

asignación de dirección:	x **:= &** y
asignación de valor:	x **:= *** y
asignación indirecta:	***** x **:=** y

La representación que se ha utilizado hasta el momento se lama de *cuartetos*, ya que utiliza cuatro campos para almacenar cada instrucción. Existe otra técnica que permite representar el código de tres direcciones mediante solo tres campos. Para ello se mantiene la secuencia de instrucciones de código intermedio en un vector. Ya que toda instrucción lleva aparejada una dirección como resultado, cuando sea necesario aludir a la dirección del resultado se hará mediante el número correspondiente a la posición del vector en la que se calculó. Por tanto, esta representación sólo almacena los campos correspondientes a la instrucción y a los dos argumentos y se llama de *tercetos*. Así, representando mediante tercetos el código del ejemplo anterior se obtiene:

	instr.	**arg1**	**arg2**
(100)	--:=	c	
(101)	*:=	b	(100)
(102)	--:=	c	
(103)	*:=	b	(102)
(104)	+:=	(101)	(103)
(105)	:=	a	(104)

El uso de tercetos tiene una evidente ventaja sobre los cuartetos, ya que no es necesario generar un sinfín de variables temporales. Además es más compacta y por tanto ocupa menos espacio en memoria. Por contra, la representación mediante tercetos es mas complicada y puede dificultar la optimización de código, ya que cualquier variación en la secuencia de instrucciones obliga a un reposicionamiento de las referencias. Este problema puede solventarse usando la representación de *tercetos indirectos*, que consiste en utilizar un vector auxiliar para almacenar la secuencia de instrucciones. Por ejemplo, el código anterior se representaría mediante dos vectores, el primero de ellos contiene la secuencia de instrucciones y el segundo las instrucciones:

referencia	**dirección**
(1)	(100)
(2)	(101)
(3)	(102)
(4)	(103)
(5)	(104)
(6)	(105)

	instr.	**arg1**	**arg2**
(100)	--:=	c	
(101)	*:=	b	(100)
(102)	--:=	c	
(103)	*:=	b	(102)
(104)	+:=	(101)	(103)
(105)	:=	a	(104)

De esta forma, para eliminar el código duplicado en la fase de optimización basta con eliminar estas instrucciones de la tabla de referencias y modificar solamente la instrucción que se optimiza:

referencia	**dirección**
(1)	(100)
(2)	(101)
(3)	(104)
(4)	(105)

	instr.	**arg1**	**arg2**
(100)	--:=	c	
(101)	*:=	b	(100)
(104)	+:=	(101)	(101)
(105)	:=	a	(104)

IV.3. Construcción de un generador de código intermedio simple.

(Este epígrafe se desarrolla mediante prácticas consistentes en la realización de un traductor de un lenguaje de alto nivel - PL96 - a código de tres direcciones)

Generación de código para expresiones aritméticas
Implementación de la tabla de símbolos
Implementación del sistema de tipos
Generación de código para expresiones condicionales
Generación de código para instrucciones de control de flujo

IV.4. Bloques básicos y diagramas de flujo.

Los bloques básicos son trozos de código intermedio en los que el flujo de control es lineal, es decir, trozos de código cuyas instrucciones se ejecutan una detrás de otra sin saltos intermedios. Los bloques básicos tienen por tanto una única instrucción de comienzo de bloque y una única instrucción de salida del bloque.

Por ejemplo, la siguiente secuencia de instrucciones de tres direcciones forma un bloque básico.

```
(100) t1 := - c
(101) t2 := b * t1
(102) t3 := - c
(103) t4 := b * t3
(104) t5 := t2 * t4
(105) a  := t5
(106) if (a>0) goto (712)
```

Para identificar los bloques básicos que componen una secuencia de instrucciones en código maquina puede seguirse el siguiente algoritmo:

1. Buscar los puntos de entrada, que corresponden a:
   - la primera instrucción de código maquina.
   - cualquier instrucción situada en la dirección a la que se refiere una instrucción de salto condicional o incondicional. .
   - cualquier instrucción consecutiva a una instrucción de salto condicional o incondicional.
2. Para cada punto de entrada construir un bloque básico que vaya desde el punto de entrada hasta el siguiente punto de entrada.
3. Los arcos del *diagrama de flujo* se obtienen mediante las reglas:
   - si el bloque básico termina en un salto incondicional, entonces incluir un arco hasta la dirección señalada.
   - si el bloque básico termina en un salto condicional, entonces incluir un arco hasta la dirección señalada y otro al siguiente bloque básico.

Una instrucción de tres direcciones del tipo **`x := y+z`**, se dice que *define* a **`x`** y que *hace referencia a* **`y`** (y también a **`z`**).

Una variable se dice que está *activa* en un determinado punto de la secuencia de instrucciones de tres direcciones si después de haber sido definida se hace referencia a ella en cualquier otro punto del programa que se ejecute posteriormente, (en el mismo bloque básico o en cualquier otro). Es decir, una variable está *activa* desde que nace con una definición, hasta que muere con el último uso que de ella hace cualquier instrucción del programa. Por el contrario, una variable esta *inactiva*, desde la última vez que se usa hasta que se *define* nuevamente.

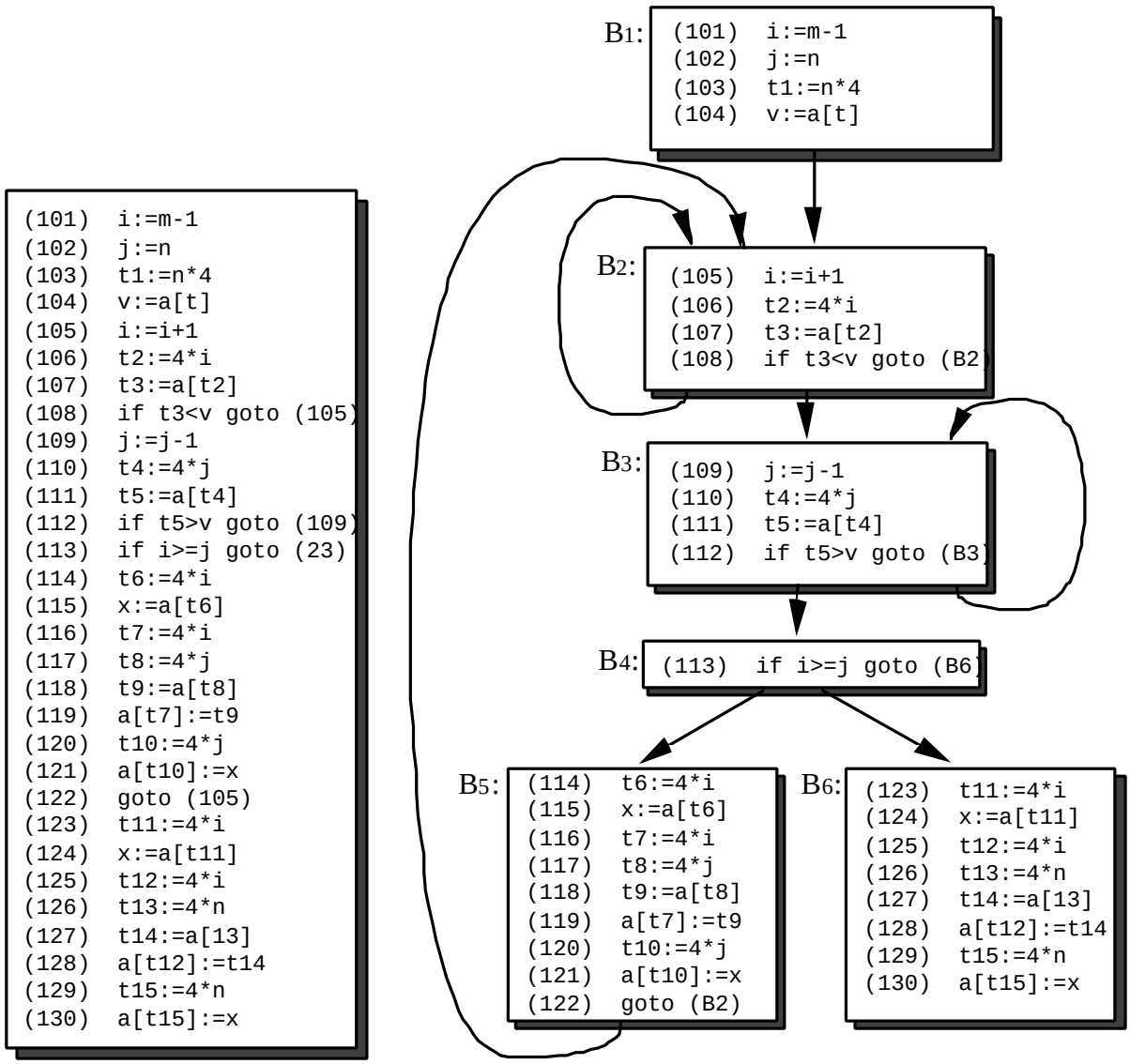
El concepto de actividad e inactividad de las variables tiene importancia para la asignación de registros que se efectúa durante la generación de código máquina.

Ejemplo: Sea la función en lenguaje C:

```
void quicksort(m,n)
int m,n;
{
    int i,j;
    int v,x;
    if (n<=m) return;
    i=m-1; j=n; v=a[n];
    while (1) {
        do i=i+1; while (a[i]<v);
        do j=j-1; while (a[j]>v);
        if (i>=j) break;
      }
    x=a[i]; a[i]=a[n]; a[j]=x;
    quicksort(m,j);  quicksort(i+1,n);
}
```

El trozo de código fuente anterior da lugar a la siguiente secuencia de instrucciones de código intermedio que aparece a la izquierda. A partir de éste se localizan los bloques básicos y se obtiene el diagrama de flujo que aparece a su derecha:

```
(101)  i:=m-1
(102)  j:=n
(103)  t1:=n*4
(104)  v:=a[t]
(105)  i:=i+1
(106)  t2:=4*i
(107)  t3:=a[t2]
(108)  if t3<v goto (105)
(109)  j:=j-1
(110)  t4:=4*j
(111)  t5:=a[t4]
(112)  if t5>v goto (109)
(113)  if i>=j goto (23)
(114)  t6:=4*i
(115)  x:=a[t6]
(116)  t7:=4*i
(117)  t8:=4*j
(118)  t9:=a[t8]
(119)  a[t7]:=t9
(120)  t10:=4*j
(121)  a[t10]:=x
(122)  goto (105)
(123)  t11:=4*i
(124)  x:=a[t11]
(125)  t12:=4*i
(126)  t13:=4*n
(127)  t14:=a[13]
(128)  a[t12]:=t14
(129)  t15:=4*n
(130)  a[t15]:=x
```



IV. 5. La máquina destino y el conjunto de instrucciones

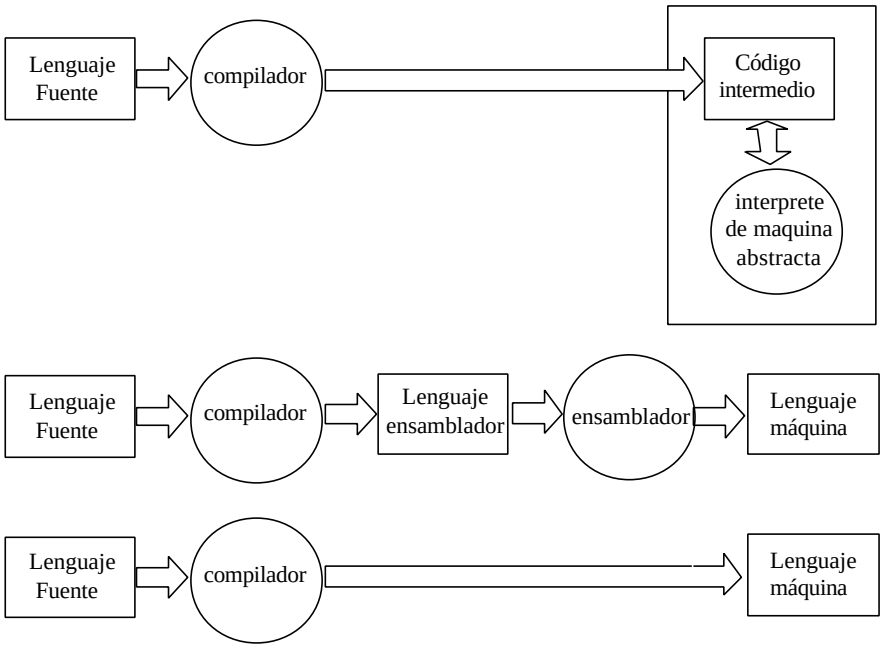
Generación de código máquina

Una vez generado el código intermedio de tres direcciones, y tras la fase de optimización de código intermedio, si la hubiera, se genera el código objeto correspondiente a la máquina sobre la que va a correr el programa compilado.

Hay gran variedad de opciones en este punto. Una de ellas consiste en implementar una máquina abstracta capaz de interpretar el código intermedio directamente. En este caso la tarea del compilador termina tras la generación y optimización del código intermedio, pero es necesario construir el interprete de la máquina abstracta, que junto con el programa objeto forma el programa ejecutable. Este proceso híbrido entre compilación e interpretación es frecuente en compiladores de bajo coste, en aquellos en los que se busca una gran portabilidad o para lenguajes fuente que por su naturaleza son mas fáciles de interpretar que de compilar, como es el caso de Prolog.

La siguiente opción consiste en generar código en lenguaje ensamblador de la maquina destino, y utilizar cualquiera de los ensambladores existentes para generar verdadero código máquina. Esta opción tiene la ventaja de su facilidad de implementación, ya que salvo en el conjunto de instrucciones y en el uso de un número indeterminado de registros, el código intermedio es muy similar a este código ensamblador. Además, de esta forma el programa obtenido puede enlazarse con otros módulos obtenidos por compilación separada con este mismo u otros compiladores. Por contra, esta traducción exige un paso más para la obtención de un programa ejecutable.

Por último, se puede optar por generar código máquina propiamente dicho, entendiendo como tal la secuencia de instrucciones y datos codificados que serán cargados en una porción de la memoria para su ejecución directa por el microprocesador. El código así generado se dice que es reubicable, si puede su ejecución puede realizarse independientemente de la posición exacta que ocupe en memoria. Para ello, los compiladores suelen generar un código máquina reubicable en un “fichero objeto”. Este enfoque permite la compilación separada de diversos módulos del programa, que dan lugar cada uno de ellos a un fichero objeto. Igualmente pueden existir módulos precompilados en ficheros objeto especiales a los que habitualmente se les llama “librerías”, que contienen el código máquina reubicable correspondiente a funciones o procedimientos predefinidos. Los ficheros objeto y las librerías deben ser finalmente enlazados para formar un único programa ejecutable. Esta tarea la lleva a cabo un programa llamado “montador” (del inglés “link”).



Principales alternativas de generación de código

La máquina destino

Actualmente existen dos principales tipos de arquitecturas de máquinas desde el punto de vista del conjunto de instrucciones, que corresponden a dos concepciones diferentes:

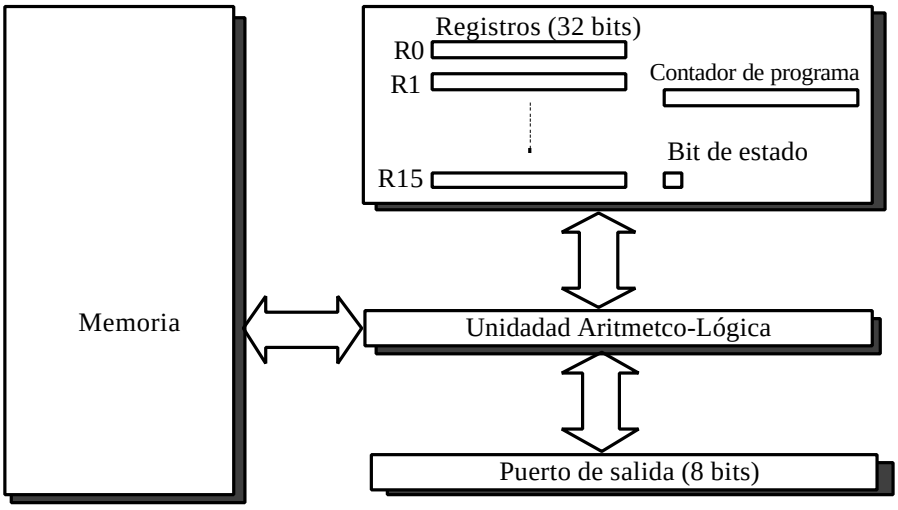
- Arquitecturas CISC: Las iniciales corresponden al ingles “Complex Instruction Set Computer”, se basan en la idea de dotar al microprocesador con un gran número de instrucciones del código maquina de manera que existan instrucciones potentes, es decir, que sea posible realizar tareas “complejas” en una sola instrucción.

- Arquitecturas RISC: Las iniciales corresponden al ingles “Reduced Instruction Set Computer”, como se nombre indica, se basan en restringir el número de instrucciones del microprocesador a un conjunto “reducido” de instrucciones especializadas en tareas simples. Estas arquitecturas por lo general se caracterizan por tener un gran número de registros e instrucciones capaces de operar entre ellos. Esta sencillez de diseño se traduce en una ventaja en lo que se refiere a la velocidad de ejecución.

Desde el punto de vista de la compilación, y a diferencia de lo que podría pensarse en un principio, las arquitecturas RISC son mejores. Las instrucciones complejas no siempre se adecuan exactamente a la semántica del lenguaje a compilar, y por tanto no pueden usarse directamente, lo que complica innecesariamente el proceso. Por el contrario en las arquitecturas RISC es mas fácil la selección de la instrucción o grupo de instrucciones adecuadas, y sobre todo, tras una etapa de optimización que realice un buen aprovechamiento de los registros, es posible obtener un código rápido y sin redundancias innecesarias.

El conjunto de instrucciones de código máquina

Cada procesador tiene su propio conjunto de instrucciones máquina, por lo que en esta sección no es fácil definir unos tópicos generales del conjunto de instrucciones. A continuación, se presenta un ejemplo descrito por J.P.Bennett del conjunto de instrucciones de una máquina abstracta con arquitectura RISC a la que denomina VAM:



Esquema de la maquina abstracta VAM

El esquema de esta maquina aparece en la figura anterior. Consta de 16 registros de 32 bits, un contador de programa y un bit de estado (que se activa siempre que el resultado de una operación aritmética es cero, o cuando se transfiere un cero en una instrucción de copia).El registro $\mathbf{R_0}$ está permanentemente a cero.(para permitir el uso de direccionamiento absoluto) La máquina también dispone de un puerto de

entrada/salida de 8 bits, y una memoria principal (con 32 bits es posible direccionar hasta un máximo de 4 Gbytes) Cada instrucción consiste en un código de operación, un byte que indica los registros a los que está referida, y 4 bytes para contener el desplazamiento ("offset") en forma de complemento a 2, en caso necesario. El conjunto de instrucciones es el siguiente:

(1)	**ADD** R_X,R_Y	Suma el contenido del registro R_X, con el contenido del registro R_Y y almacena el resultado en R_Y
(1)	**SUB** R_X,R_Y	Resta del contenido del registro R_X, el contenido del registro R_Y y almacena el resultado en R_Y
(5)	**MUL** R_X,R_Y	Multiplica el contenido del registro R_X, por el contenido del registro R_Y y almacena el resultado en R_Y
(10)	**DIV** R_X,R_Y	Divide el contenido del registro R_X, por el contenido del registro R_Y y almacena el resultado en R_Y
(2)	**STI** R_X,offset(R_Y)	Almacena el contenido de R_X en la dirección de memoria (contenido de R_Y+offset)
(2)	**LDI** offset(R_X),R_Y	Almacena en R_Y el contenido de la dirección de memoria (contenido de R_X+offset)
(2)	**LDA** offset(R_X),R_Y	Almacena en R_Y la dirección de memoria (contenido de R_X+offset)
(1)	**LDR** R_X,R_Y	Almacena en R_Y el contenido de R_X
(2)	**BRA** offset	Salta hasta la dirección (dirección actual +offset)
(2)	**BZE** offset	Salta hasta la dirección (dirección actual +offset) si el bit de estado está activo.
(2)	**BNZ** offset	Salta hasta la dirección (dirección actual +offset) si el bit de estado no está activo.
(10)	**TRAP**	Manda al puerto de salida el byte menos significativo del registro R_{15}.
(1)	**HALT**	Termina la ejecución
(1)	**NOP**	No hacer nada.

El coste de operaciones aparece entre paréntesis a la izquierda y es proporcional al tiempo que requiere su ejecución

IV.6. Algoritmos de Generación de código máquina.

Uno de los principales problemas que deben afrontarse al generar código maquina a partir de código de tres direcciones consiste en determinar que registros deben utilizarse para cada instrucción. En maquinas en los que el número de registros es pequeño, y algunos de estos registros están especializados, este problema se hace especialmente crítico. En cualquier caso, dado que el número de registros es finito, se debe establecer una estrategia que nos permita aprovechar al máximo los recursos del sistema, de modo que se reduzcan las operaciones de transferencia entre los registros y la memoria principal del programa.

Existen diversas técnicas que permiten optimizar el uso de los registros, de las que se estudiarán sólo algunas. En general, se basan en el computo del ámbito de actividad de las diversas variables que intervienen en el código de tres direcciones.

La generación de código se realiza de forma independiente para cada uno de los bloques básicos que componen la traducción del programa en código de tres direcciones. Dada una secuencia de instrucciones que componen un bloque básico, la primera tarea a realizar es el cómputo del ámbito de actividad de las variables que en él intervienen. Esto se lleva a cabo mediante la siguiente estructura de datos:

- **Tabla de estado de las variables en cada instrucción:** A cada instrucción de tres direcciones *i*, (por ejemplo, del tipo **a:=b+c**), se le asocia una estructura que permiten conocer la siguiente instrucción *j* en

la que se utilizan cada una las variables que intervienen en la instrucción, (**a**,**b** y **c)** y si cada una de ellas están o no activas. (Un registro para cada instrucción).

- **Tabla auxiliar de actividad de las variables:** Para cada variable que interviene en el bloque básico, se emplea una estructura auxiliar que contiene dos campos que se va actualizando conforme se ejecuta el algoritmo que calcula la primera. El primero de estos campos es un puntero a la dirección de la instrucción en la que se usa próximamente la variable, el segundo, si la variable está o no activa en un instante dado. Puesto que es necesario un registro para cada variable, es lógico que en general se utilice la propia *tabla de símbolos* del compilador como estructura auxiliar, incluyendo en ella estos campos.

El algoritmo se basa en recorrer hacia atrás la secuencia de instrucciones que componen el bloque básico:

1. Inicialmente se almacenan en la tabla de símbolos el estado de las variables al final del bloque básico, siguiendo las reglas:

   1.1. Toda variable temporal está inactiva al final del bloque básico. Es decir, en general se puede suponer que las variables temporales creadas en un bloque básico no se emplean más que en éste. Hay algunas excepciones a esta reglas que por el momento no seconsideran.
   1.2 Toda variable no-temporal está activa al final del bloque básico, es decir, se usa en alguna otra parte del programa, (y por tanto su valor no es desechable). Esta es una presunción que debe hacerse a menos que se haya realizado un adecuado análisis del flujo de datos del código de tres direcciones. Este análisis se estudia en el capitulo próximo.

   Toda esta información se asocia a la última instrucción. Además, para cada una de las variables .se incluye en la tabla de símbolos

2. A partir de la última instrucción y hasta alcanzar la primera, para cada instrucción *i* de tres direcciones (por ejemplo, del tipo **a:=b+c**), se determina si las variables que intervienen en ella están o no activas en ese punto, haciendo lo siguiente:

   2.1 Almacenar en el registro asociado a la instrucción *i* , el contenido de la tabla de símbolos en ese instante.
   2.2 Marcar **a** como inactiva en la tabla de símbolos, e indicar que no tiene uso próximo.
   2.3 Marcar en la tabla de símbolos las variables **b** y **c** como activas, indicando como próximo uso la instrucción *i*.

Una vez calculado el ámbito de actividad de las variables mediante el procedimiento anterior, se puede comenzar a generar código máquina para los cada uno de los bloques básicos, siguiendo la secuencia de instrucciones hacia en orden creciente. Durante esta etapa es cuando se decide la utilización de los registros para contener una u ora variable, así pues, para gestionar adecuadamente estas asignaciones, es necesario disponer de dos estructuras de datos:

- **Tabla de contenido de los registros:** En cada instrucción del código objeto, los registros de la maquina pueden estar libres o almacenar el contenido de una variable, (o más de una si sus valores son iguales). Por tanto, .para cada registro de la máquina debe almacenarse la lista de variables cuyos valores están almacenados en dicho registro.

- **Tabla de localización de las variables:** Cuando el programa se ejecute, los valores de las variables serán transferidos desde las posiciones de memoria que les correspondan a los registros de la máquina y viceversa, con el fin de realizar ciertas operaciones. Por tanto, en un instante dado de la ejecución del programa, el valor actual de una variable puede encontrarse en la memoria, en los registros o en ambas posiciones, dependiendo de las transferencias y operaciones que se hayan realizado anteriormente. Por consiguiente, es necesario disponer de una tabla que en cada instante, para cada variable, indique la lista de posiciones de memoria en la que se encuentra su valor actual. Como en el caso anterior puede usarse un campo de propia *tabla de símbolos* del compilador para contener esta información.

La selección de los registros adecuados para contener las variables es uno de los factores mas importantes en la generación de código máquina. En el proceso de generación de código máquina debe

evitarse en la medida de lo posible el trasiego innecesario de información entre la memoria y los registros. Para obtener un código que haga un buen aprovechamiento de los mismos es menester invocar a procedimientos de análisis del flujo de datos y a estrategias globales que utilicen los registros teniendo en cuenta todo el programa. Una simplificación de esta tarea consiste en utilizar estrategias locales de asignación, de manera que una vez fragmentado el código en bloques básicos, la asignación de registros solo se realice teniendo en cuenta las necesidades del bloque.

Estrategias locales de asignación de registros.

La generación de código máquina a partir del código intermedio se realiza secuencialmente, desde la primera hasta la última de las instrucciones del bloque básico. Si la máquina destino es similar a la máquina VAM que se ha descrito anteriormente, el algoritmo de generación de código para una instrucción de tres direcciones (por ejemplo, del tipo **a:=b-c**) podría ser el siguiente:

1. Determinar la posición $\mathbf{R_a}$ en donde pueda situarse el resultado de la operación. (En muchas máquinas solo se permite el uso de registros para el resultado). Esta tarea puede llevarse a cabo con el siguiente algoritmo:

   1.1. Si **c** está situada en un registro y no está activa después de esta instrucción, puede usarse el registro $\mathbf{R_c}$, Toda la información necesaria en este punto está disponible en la tabla de actividad de las variables asociada a la instrucción y en la tabla de localización de las variables. Al final de este paso debe actualizarse la tabla de localización de variables para indicar que **c** ya no está en $\mathbf{R_c}$

   1.2. Si el paso anterior falla, debe buscarse en la tabla de localización de registros si hay algunos vacío, y devolver esta posición.

   1.3. Si el paso anterior también falla, debe desocuparse alguno de los registros usados y devolver esta posición. Entre todos los registros se elegirán:

      1.3.1. Los que contengan un menor número de variables, y si son varios, de entre estos:.
      1.3.2. Aquellos que contengan una variable que esté simultáneamente almacenada en la memoria o en otros registros, y en caso de que esto no sea posible:
      1.3.3. Aquellos que contengan una variable **v** con un mayor ámbito de actividad a partir de la instrucción actual es decir, cuyo próximo uso sea más lejano. En este último caso hay que generar una instrucción máquina adicional para salvar el contenido del registro $\mathbf{R_v}$ en la dirección de memoria asignada a la variable **v**. (Para la máquina VAM esta instrucción sería: **STI** $\mathbf{R_v}$**,v(**$\mathbf{R_0}$**)**)

   Por supuesto, estos criterios pueden variar o adaptarse de una a otra máquina, según el conjunto de instrucciones que se utilice. Tras esta selección debe actualizarse la tabla de localización de la variable o variables que contenga el registro seleccionado.

2. Una vez determinada la posición $\mathbf{R_a}$ del resultado, se consulta la tabla de localización de las variables para saber donde está almacenada la variable **c**.(el segundo operando se sobreescribe al ejecutar la instrucción en la máquina VAM)

   2.1 Si **c** está en el registro $\mathbf{R_a}$, no hay que hacer nada.

   2.2. Si **c** está en otro registro $\mathbf{R_c}$?$\mathbf{R_a}$, transferir su valor al registro $\mathbf{R_a}$ mediante la instrucción de copia entre registros correspondiente. (Para la máquina VAM esta instrucción sería: **STR** $\mathbf{R_c}$**,**$\mathbf{R_a}$)

   2.3. Si **c** está solamente en memoria, copiar en el registro $\mathbf{R_a}$ su valor mediante la instrucción de carga correspondiente.(Para la máquina VAM esta instrucción sería: **LDI c(**$\mathbf{R_0}$**),** $\mathbf{R_a}$)

Dado que el registro $\mathbf{R_a}$, contendrá finalmente el resultado, no es necesario en este punto realizar la actualización de la tabla de localización de variables, ni la de contenido de los registros.

3. Determinar donde está situada la variable **b**, (el primer operando no se sobreescribe en la máquina VAM, pero debe estar situado en un registro).

   3.1. Si **b** está situada en un registro $\mathbf{R_b}$, no hacer nada.

   3.2. Si **b** está almacenada solamente en memoria, determinar un registro $\mathbf{R_b}$? $\mathbf{R_a}$, al que poder transferir la variable **b**, y generar la instrucción de carga correspondiente. Este punto se realiza de forma análoga a los puntos 1.2. y 1.3.

4. Generar la instrucción correspondiente a la operación usando como primer argumento el registro $\mathbf{R_a}$, y como segundo el registro $\mathbf{R_b}$. A continuación hay que actualizar la tabla de contenido del registro $\mathbf{R_a}$ para indicar que contiene solamente a la variable **a**, y la tabla de localización de la variable **a**, para indicar que está almacenada solamente en el registro $\mathbf{R_a}$ (Para la máquina VAM esta instrucción sería: **SUB** $\mathbf{R_b,R_a}$)

5. Si tras esta instrucción la variable **b** pasa a estar inactiva, (no tiene ningún próximo uso), modificar la tabla de contenido de los registros para indicar que **b** no está almacenada en $\mathbf{R_b}$, considerando de esta forma que el registro $\mathbf{R_b}$ está vacío para otros posibles usos.

Casos particulares de este esquema son las instrucciones de operador unario **a:= -b** en los que solo son aplicables los puntos 1.,2. y 4., o bien las instrucciones de copia del tipo **a:=b**, en las que la única operación que hay que realizar consiste en modificar adecuadamente la tabla de localización de la variable **a** y en su caso, la tabla de contenido de los registros.

Otras instrucciones de tres direcciones requieren algoritmos similares de selección de registros, por ejemplo, la instrucción **a:=b[i]**, supuesto que **b** es una matriz que se almacena en memoria estática, debe traducirse en la máquina VAM por la instrucción **LDI** $\mathbf{b(R_i),R_a}$, que requiere que **i** se sitúe en un registro $\mathbf{R_i}$, obteniendo el resultado en otro registro $\mathbf{R_a}$. Para ello se debe seleccionar previamente el registro adecuado para situar el resultado; y en su caso, también el que se use para el argumento.

Una vez terminado el bloque básico es importante salvar todos los registros que contengan variables activas a sus respectivas posiciones de memoria, ya que puede producirse el salto a una dirección correspondiente a otro procedimiento o a cualquier otra parte del programa que de antemano no se sabe que uso hará de los registros. Tras esta operación la tabla de contenido de los registros y la de localización de variables pueden vaciarse.

Estrategias globales de asignación de registros.

En general, los bloque básicos tienden a ser pequeños, y por tanto las estrategias locales de asignación de registros no suelen dar un buen resultado. Especialmente en máquinas con gran cantidad de registros, las estrategias locales no suelen proporcionar un buen aprovechamiento de este recurso. Existen varias estrategias globales de asignación entre las que sólo se presentan algunas:

Estrategia de asignación de registros basada en el mapa de colores

1. A partir del código de tres direcciones se genera código máquina considerando que hay un número infinito de registros virtuales, es decir, al generar código se supone que siempre hay un registro vacío disponible. En esta etapa se resuelven todos los demás problemas de traducción que pudiera haber y se asignan los recursos de la máquina a aquellas instrucciones que los necesiten de forma inequívoca, registros con funciones especiales, etc.

2. Sobre el código obtenido en el punto anterior, se construye el grafo de interferencia entre registros. Este grafo tiene como nodos los registros virtuales que se utilizaron en el punto anterior y sus arcos indican que los nodos que conectan corresponden a registros que se "interfieren", es decir uno de ellos contiene una variable activa en el momento en el que el otro se usa para definir otra variable. Esta información puede obtenerse fácilmente recorriendo el programa secuencialmente gracias a la tabla de estado de las variables en cada instrucción.

3. Una vez obtenido el grafo, se intenta colorear de manera que a dos nodos adyacentes no se les asigne el mismo color, usando una paleta con N colores, siendo N el número de registros reales de la máquina. Este problema no siempre tiene solución, pero puede intentarse mediante el siguiente algoritmo:

   3.1 Se toma un nodo **k** con menos de N vecinos y se borra del grafo junto con todos los arcos que pasan por él, almacenándolo en una lista. Si el grafo que queda se puede colorear con N colores, siempre se podrá incluir en él el nodo **k**, (ya que tenia menos de N vecinos habrá un color disponible).
   3.2. Se repite el proceso anterior hasta que el grafo se quede vacío o bien todos los nodos tengan N o más vecinos.
   3.3. Si el grafo se queda vacío, para colorear el grafo se van introduciendo los nodos en el orden inverso en el que se eliminaron.
   3.4. Si no se puede quitar ningún nodo es que todos los que quedan tienen mas de N vecinos, y por tanto el grafo no puede colorearse con N colores.

4. Si el punto anterior fracasa, se toma uno de los nodos que quedan con mas de N vecinos y se genera el código necesario para almacenar el contenido del registro en memoria y recargarlo con otra variable, modificando adecuadamente el grafo de interferencias y volviendo a repetir para éste el paso anterior. El criterio de selección de nodo que se modifica en este punto se hace de manera que la operación de carga sea menos costosa desde el punto de vista global del programa, es decir, se procura no modificar aquellos registros que intervienen en las instrucciones correspondientes a los bucles mas profundos del código, y que previsiblemente se ejecutan un mayor número de veces.

5. Cada color representa un registro real, por tanto el código máquina generado se reescribe reemplazando los registros virtuales por el registro real asociado al color correspondiente.