

### 8. Generación y Optimización de Código

#### 8.1. Introducción

- Consideraciones generales
- Arquitectura de la máquina destino

#### 8.2. Selección de instrucciones

- Selección de instrucciones por reescritura de árboles
- Concordancia de patrones mediante análisis sintáctico

#### 8.3. Asignación de registros

- Jerarquía de memorias
- Asignación de registros por coloración de grafos

### ➤ Objetivos de la Generación de Código (GC)

- El código objeto generado debe ser correcto y de gran calidad  
⇒ **Uso juicioso de los recursos de la máquina objeto**
- El problema de generar código objeto óptimo es indecidible  
⇒ **Desarrollo de técnicas que proporcionen código bueno pero no siempre óptimo**

### ➤ Entrada al GC

CI y la información de la TDS + [ BB y sus GDA; GF y la actividad de sus variables ]

### ➤ Salida del GC

- Lenguaje Máquina absoluto
- Lenguaje Máquina relocizable
- Lenguaje Ensamblador

### Lenguaje Máquina absoluto

- las direcciones están codificadas de forma rígida (posición fija de memoria) y se puede ejecutar inmediatamente.
- útil en gestores de interrupciones y controladores.
- la aplicación es sencilla y directa pero inflexible (difícil de recargar)

### Lenguaje Máquina relocizable (módulo objeto)

- todas las ubicaciones (direcciones) se representan por símbolos
- la asignación de las direcciones de memoria se hace en tiempo de enlace y carga
- flexibilidad de la compilación separada

### Lenguaje Ensamblador

- simplificación del proceso por el uso de instrucciones (y macros) y nombres simbólicos
- implica una sobrecarga (proceso adicional de ensamblado y enlazado)

### Años 50

Diseños no homogéneos de los procesadores ⇒ rehacer completamente los programas

### Años 60

IBM propone **CISC** (“Complex Instruction Set Computer”) (IBM/360 en 1964)

- ✓ Produce programas compactos y fáciles de depurar
- ✓ Pocos accesos a memoria
- ✓ Reduce la complejidad del desarrollo de los compiladores y sus costes
- ✗ Instrucciones estructuralmente más complejas y más lentas
- ✗ Dificulta el paralelismo
- ✗ Las instrucciones de longitud variable reducen el rendimiento y dificultan la planificación (“scheduling”)

Ej. **x86** VAX; PDP-11; Motorola 68000, ...; Intel 8086, ...

Años 70

IBM propone RISC (“Reduced Instruction Set Computer”) (IBM/801 en 1975)

- ✓ Conjunto de instrucciones reducido con instrucciones simples y rápidas
- ✓ Instrucciones de longitud fija y de un solo ciclo de reloj que facilitan la planificación
- ✓ Facilita el paralelismo
- ✓ Permite un hardware más simple
- ✗ Incrementa el tamaño del código generado
- ✗ Muchos accesos a memoria y necesidad de memorias rápidas
- ✗ Aumenta la complejidad del desarrollo de compiladores y sus costes

Ej. ARM MIPS; PA-RISC (Hewlett Packard); SPARC (Sun Microsystem); POWER-PC (Apple)

SELECCIÓN DE INSTRUCCIONES

La eficiencia de la Generación de Código depende (sobre todo) de:

- mantener la tubería (“pipeline”) llena;
- uso eficiente de los registros;
- adecuada planificación (“scheduling”) de las instrucciones: selección de instrucciones

Encontrar la apropiada secuencia de instrucciones máquina asociada con cada una de las instrucciones 3-direcciones.

- Solución trivial: uso de plantillas.

Ejemplo	$x = y + z$		// $x, y, z$	asignadas estáticamente
LD	r0,	y	// $r0 = y$	(carga y en el registro r0)
LD	r1,	z	// $r1 = z$	(carga z en el registro r1)
AD	r0,	r1	// $r0 = r0 + r1$	(suma y deja en r0)
ST	x,	r0	// $x = r0$	(almacena r0 en x)

	Arquitectura RISC	Arquitectura CISC
Num. registros	muchos (32 o más)	pocos (6 u 8)
Tipo registros	solo un tipo	tipos diferentes agrupados en clases
Op. aritméticas	solo entre registros	entre registros y memoria
Tipo instrucciones	código 3-direcciones $(r_1 \leftarrow r_2 \oplus r_3)$	código 2-direcciones $(r_1 \leftarrow r_1 \oplus r_2)$
Direccionamiento	Reg-Reg	Reg-Reg; Reg-Mem; Mem-Reg
Long. instrucciones	fija ( $\approx 1$ ciclo de reloj)	variable (varios ciclos de reloj)
Resul. instrucción	un solo resultado	más de un resultado (posibles efectos secundarios)

SELECCIÓN DE INSTRUCCIONES

La solución adecuada debería considerar:

- 1) La complejidad del juego de instrucciones

Ejemplo	$a = a + 1$		
LD	r0,	a	// $r0 = a$
AD	r0,	#1	// $r0 = r0 + 1$
ST	a,	r0	// $a = r0$
IN	a		// $a = a + 1$

- 2) El contexto de aplicación

Ejemplo	$a = b + c$		$d = a + e$
LD	r0,	b	// $r0 = b$
LD	r1,	c	// $r1 = c$
AD	r0,	r1	// $r0 = r0 + r1$
ST	a,	r0	// $a = r0$
LD	r0,	a	// $r0 = a$
LD	r1,	e	// $r1 = e$
AD	r0,	r1	// $r0 = r0 + r1$
ST	d,	r0	// $d = r0$

## SELECCIÓN DE INSTRUCCIONES POR REESCRITURA DE ÁRBOLES

- Código Intermedio representado por árboles semánticos (o GDA)
- Reglas de reescritura (Esquema de traducción de árboles)
  - sustitución (nodo)  $\leftarrow$  plantilla (subárbol) { acción (fragmento de código) }

### ➤ Método:

Dado un árbol de entrada

### Repetir

Aplicar las plantillas de las reglas de reescritura a los diferentes subárboles

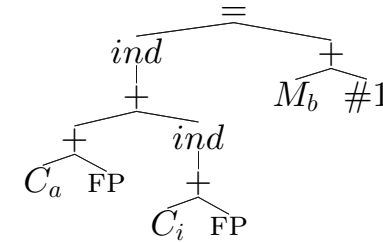
**Si** una plantilla concuerda **entonces**

reemplazar ese subárbol por el nodo de sustitución de la regla

ejecutar la acción asociada

**hasta** el árbol se reduce a un solo nodo o no concuerdan más plantillas

## SELECCIÓN DE INSTRUCCIONES POR REESCRITURA DE ÁRBOLES



Árbol de código intermedio para:  $a[i] = b + 1$

$a, i$  variables locales alojadas en la "pila".  $C_a$  y  $C_i$  desplazamientos relativos al FP

$b$  variable global y  $M_b$  su dirección en memoria.

$ind$  operador de indirección.

## SELECCIÓN DE INSTRUCCIONES POR REESCRITURA DE ÁRBOLES

- $R_i \leftarrow C_a$  { LD  $R_i, C_a$  }
- $R_i \leftarrow M_x$  { LD  $R_i, M_x$  }
- $M \leftarrow M_x \text{ } R_i$  { ST  $M_x, R_i$  }
- $M \leftarrow ind \text{ } R_j$  { ST  $* R_i, R_j$  }
- $R_i \leftarrow ind \text{ } C_a \text{ } R_j$  { LD  $R_i, *(C_a + R_j)$  }
- $R_i \leftarrow R_i \text{ } ind \text{ } C_a \text{ } R_j$  { AD  $R_i, *(C_a + R_j)$  }
- $R_i \leftarrow R_i \text{ } R_j$  { AD  $R_i, R_j$  }
- $R_i \leftarrow R_i \text{ } \#1$  { IN  $R_i$  }

## SELECCIÓN DE INSTRUCCIONES POR REESCRITURA DE ÁRBOLES

- ¿Cómo se hace la comprobación de patrones?
  - ⇒ Extender los algoritmos de comparación de cadenas a árboles, empleando la información contextual
- ¿Cuál es el orden óptimo de comparación de patrones? y ¿Qué hacer si existen más de una comparación posible?
  - ⇒ Mediante algoritmos de Programación Dinámica basados en alguna definición de coste de las instrucciones
- ¿Qué hacer si no existe ninguna comparación posible? ¿Se bloquea el GOC?
  - ⇒ No, se completa mediante soluciones *ad hoc*
- ¿Qué hacer si un subárbol se reescribe indefinidamente?
  - ⇒ No, se detiene mediante soluciones *ad hoc*

## CONCORDANCIA DE PATRONES MEDIANTE ANÁLISIS SINTÁCTICO

- Convertir el árbol semántico en un recorrido prefijo y usar un AS LR(1) para hacer la concordancia de patrones teniendo en cuenta el contexto
- Dado que  $G$  será muy ambigua, se debe considerar una resolución de conflictos que optimice el resultado
- **Ventajas**
  - El AS LR es eficiente y conocido
  - Es muy fácil portarlo a otra máquina
  - La calidad puede incrementarse notablemente incorporando reglas específicas aprovechando las peculiaridades de la máquina
- **Inconvenientes**
  - Implica un orden de evaluación de izquierda-derecha
  - Para árboles con muchos nodos  $G$  puede ser enorme
  - Hay que incorporar técnicas *ad hoc* para que el AS LR no se bloquee (por falta de reglas o por la resolución de conflictos)
  - Hay que incorporar técnicas *ad hoc* para que el AS LR no entre en bucle

## CONCORDANCIA DE PATRONES MEDIANTE ANÁLISIS SINTÁCTICO

$$= ind + + C_a R_{FP} ind + C_i R_{FP} + M_b \#1$$

1) $R_i \leftarrow C_a$	$LD R_i, C_a$	2) $R_i \leftarrow M_x$	$LD R_i, M_x$
3) $M \leftarrow = M_x R_i$	$ST M_x, R_i$	4) $M \leftarrow = ind R_i R_j$	$ST * R_i, R_j$
5) $R_i \leftarrow ind + C_a R_j$	$LD R_i, *(C_a, R_j)$	6) $R_i \leftarrow + R_i ind + C_a R_j$	$AD R_i, *(C_a, R_j)$
7) $R_i \leftarrow + R_i R_j$	$AD R_i, R_j$	8) $R_i \leftarrow + R_i \#1$	$IN R_i$

$$\begin{aligned}
 &= ind + + \underbrace{C_a}_{r1} R_{FP} ind + C_i R_{FP} + M_b \#1 \xRightarrow{r1} \\
 &= ind + + \underbrace{R_0 R_{FP}}_{r7} ind + C_i R_{FP} + M_b \#1 \xRightarrow{r7} \\
 &= ind + \underbrace{R_0 ind + C_i R_{FP}}_{r6} + M_b \#1 \xRightarrow{r6} = ind R_0 + \underbrace{M_b}_{r2} \#1 \xRightarrow{r2} \\
 &= ind R_0 + \underbrace{R_1 \#1}_{r8} \xRightarrow{r8} = \underbrace{ind R_0 R_1}_{r4} \xRightarrow{r4} M
 \end{aligned}$$

## JERARQUÍA DE MEMORIAS

- Administración de la memoria: jerarquía [Scott, 2009]

	Tiempo de acceso	Capacidad
Registros	1 ciclo	~500 Bytes
Memoria caché (L1)	1 - 3 ciclos	~64 KBytes
Memoria caché (L2)	5 - 10 ciclos	1 - 10 MBytes
Memoria principal	~100 ciclos	~10 GBytes
Disco	$10^6 - 10^7$ ciclos	~100 GBytes and up

## JERARQUÍA DE MEMORIAS

### ➤ Restricciones de la jerarquía de memoria

- Los programas se escriben como si sólo hubiera dos tipos de memoria: memoria principal y disco
- El programador es responsable de mover datos de disco a memoria
- El hardware es responsable de mover los datos entre la memoria y la caché
- El compilador es responsable de mover los datos entre la memoria y los registros (que el programador normalmente no ve)
- Las tallas de los registros y la caché crecen muy lentamente: es muy importante manejarlos bien.
- La velocidad del procesador aumenta más rápidamente que la velocidad de la memoria y del disco

## ASIGNACIÓN DE REGISTROS

### > Necesidad/justificación del uso de registros

- > Nuestro estilo de código intermedio utiliza profusamente variables temporales, lo que simplifica la generación y optimización de código, pero complica la traducción final a ensamblador
- > El objetivo consiste en emplear los registros de la máquina para reducir el número de variables temporales reescribiendo el código sin cambiar el comportamiento del programa:
  - almacenando los operandos de las instrucciones
  - almacenando las variables temporales
  - almacenando variables que se calculan en un bloque y se usan en otros (ej. variables de inducción de un bucle)
  - almacenando valores relacionados con la gestión de la memoria en tiempo de ejecución (ej. "frame pointer")

## ASIGNACIÓN DE REGISTROS POR COLORACIÓN DE GRAFOS

### > Asignación de registros: idea básica

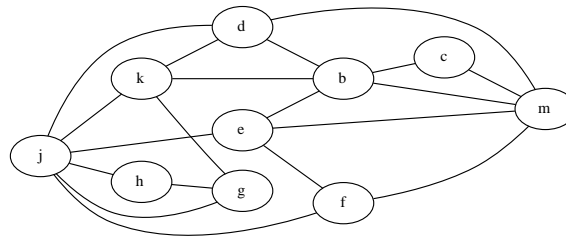
- > Asignar el mayor número de variables temporales al menor número de registros. Además, sería deseable que el origen y destino de las instrucciones "mv" sea el mismo para poder eliminar dicha instrucción
- > Un conjunto de variables temporales puede compartir un único registro físico, con la condición de que a lo sumo una esté activa en cualquier punto del programa

### > Grafo de interferencias

- > Representa las restricciones de dos o más objetos con un tiempo de vida simultáneo y por tanto no puedes compartir el mismo registro
- > Es un grafo no dirigido que se define como:
  - Cada nodo representa a una variable
  - Cada arco  $(a, b)$  representa la imposibilidad de que  $a$  y  $b$  puedan asignarse al mismo registro

## ASIGNACIÓN DE REGISTROS POR COLORACIÓN DE GRAFOS

Activas de entrada	{ j, k }
$g \leftarrow j + 1$	{ j, g, k }
$h \leftarrow k - 1$	{ j, g, h }
$f \leftarrow g * h$	{ f, j }
$e \leftarrow j + 8$	{ e, f, j }
$m \leftarrow j + 16$	{ m, e, f }
$b \leftarrow f + 12$	{ b, m, e }
$c \leftarrow e + 8$	{ b, m, c }
$d \leftarrow c$	{ b, d, m }
$k \leftarrow m + 4$	{ d, k, b }
$j \leftarrow b$	
Activas de salida	{ d, k, j }

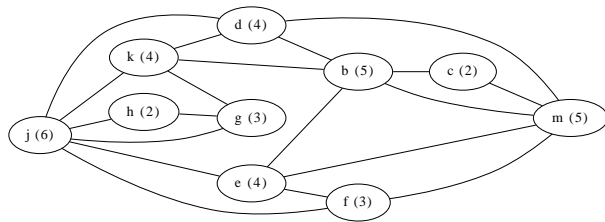


## ASIGNACIÓN DE REGISTROS POR COLORACIÓN DE GRAFOS

### > Coloreado del grafo de interferencias

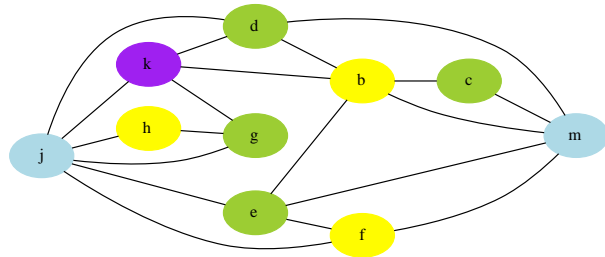
- > Colorear un grafo consiste en asignar un color a los nodos del grafo, de forma que los nodos conectados por una arista tengan diferentes colores
- > Un grafo es  $k$ -coloreable si tiene una coloración con  $k$  colores. En nuestro caso: **colores  $\equiv$  registros**
- > Para un número  $k$  de registros/colores dado, podría no existir un grafo  $k$ -coloreable. Esto implicaría que algunas variables temporales deban estar en memoria
- > Se trata de un problema NP-completo. Propondremos una solución heurística:
  1. Seleccionar un nodo  $t$  con menos de  $k$  vecinos del grafo
  2. Apilar  $t$  en una pila y eliminar  $t$  y sus arcos del grafo
  3. Repetir los pasos 1 y 2 hasta que el grafo no tenga nodos
  4. Desapilar un nodo de la pila y añadirlo al grafo
  5. Asignar al nodo un color diferente del de sus vecinos
  6. Repetir los pasos 4 y 5 hasta que la pila esté vacía

## ASIGNACIÓN DE REGISTROS POR COLORACIÓN DE GRAFOS



$K = 4$

\$ h g c k f j d b e m



## ASIGNACIÓN DE REGISTROS POR COLORACIÓN DE GRAFOS

### > Problema de volcado ("Spilling")

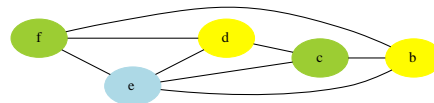
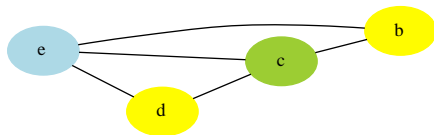
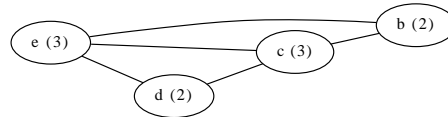
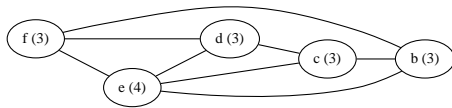
- > ¿Qué pasa si durante la simplificación del paso 1 todos los nodos del grafo tienen  $k$  o más vecinos?
- > Solución heurística:

1. Seleccionar un nodo  $t$
2. Eliminar  $t$  y sus arcos del grafo
3. Volcar  $t$  a memoria
4. Proceder al coloreado del grafo

## ASIGNACIÓN DE REGISTROS POR COLORACIÓN DE GRAFOS

$K = 3$

\$ d b c e



## EJEMPLO(S): GM

```
int M[100];
float func (int x, float y, int z) {
    int N[100]; int j;
    .... // <---- [1]
}
struct{int a; float b; int c;} R;
int main () {
    int i; float M[100]; float k;
    .... // <---- [2]
}
```

La talla de los enteros y lógicos es 2 y la de los reales es 4. Además la talla del segmento de enlaces de control es 8. Mostrad el contenido completo de la TdS en los puntos de control [1] y [2].

Diseñad un ETDS que genere código intermedio para,

```
l → repeat E : l
l → l ; l
l → break
```

La instrucción `repeat` indica que la instrucción que le sigue, `l`, debe ejecutarse un total de `E` veces, donde `E` debe ser una expresión de tipo entero.

La instrucción `break`, si aparece, supone la finalización del bucle.

3. Dado el código intermedio de un bloque básico, aplicad las optimizaciones locales a partir de su GDA. A la salida del bloque solo estarán activa las variables: `A`, `x`.

(100)	$t_1 = 0$	(104)	$t_4 = A[t_3]$	(108)	$k = x$	(112)	$t_{10} = k * 4$
(101)	$k = t_1$	(105)	$t_5 = x * 4$	(109)	$t_8 = k * 1$	(113)	$t_{11} = A[t_{10}]$
(102)	$t_2 = x + k$	(106)	$t_6 = A[t_5]$	(110)	$t_9 = t_8 * 4$	(114)	$x = t_{11}$
(103)	$t_3 = t_2 * 4$	(107)	$t_7 = t_4 + t_6$	(111)	$A[t_9] = t_7$		

4. Dado el siguiente fragmento de código intermedio:

(100)	$i = 5$	(104)	$t_2 = 10$	(108)	$x = x + t_5$	(112)	$t_7 = A[t_6]$
(101)	$j = 20$	(105)	$t_3 = t_2 * 2$	(109)	$j = j - 2$	(113)	$x = x + t_7$
(102)	$x = 0$	(106)	$t_4 = t_1 + t_3$	(110)	$if\ i > 10\ goto\ 116$	(114)	$i = i + 2$
(103)	$t_1 = j * 4$	(107)	$t_5 = B[t_4]$	(111)	$t_6 = i * 2$	(115)	$goto\ 103$

- a) Determinad los bloques básicos que forman el/los bucle/s. Extraed el código invariante. Indicad las variables de inducción y sus ternas asociadas.
- b) Aplicad el algoritmo de reducción de intensidad.
- c) Aplicad el algoritmo de eliminación de variables de inducción.