Universidad Francisco de Vitoria
UFV Madrid

PROCESADORES DE LENGUAJE

Tema 5: Optimización de Código

Grado en Ingeniería Informática
Universidad Francisco de Vitoria

Objetivos del tema

- Objetivos generales
  - Comprender los **principios básicos de la optimización en computación** y los aspectos clave en la mejora de la calidad de los sistemas informáticos
  - Comprender la **misión de la fase de optimización de código** y su localización en la cadena de compilación
  - Conocer los diferentes **tipos de optimización**
  - Conocer y **aplicar algunas de las técnicas** de optimización

Antes de empezar

Tiempo de ejecución en un sistema concreto

Área y recursos necesarios (memoria)

Energía

~~Optimización de Código~~

Mejorar una dimensión es en ocasiones a costa de empeorar otra

Se utilizan algoritmos heurísticos y técnicas que mejoran pero no llegan a la solución óptima

Antes de empezar

~~Optimización de Código~~

Si no es optimización de código, ¿qué es entonces lo que vamos a aprender en este tema?

COMPILACIÓN/SOFTWARE

¿Cómo generar un código objeto tras la fase de análisis del compilador que al ejecutarse en un sistema presente buenas características en TIEMPO, ÁREA y/o ENERGÍA?

¿Cómo impacta el código en estas dimensiones?
¿De qué formas alternativas podemos escribir un código para mejorarlas?

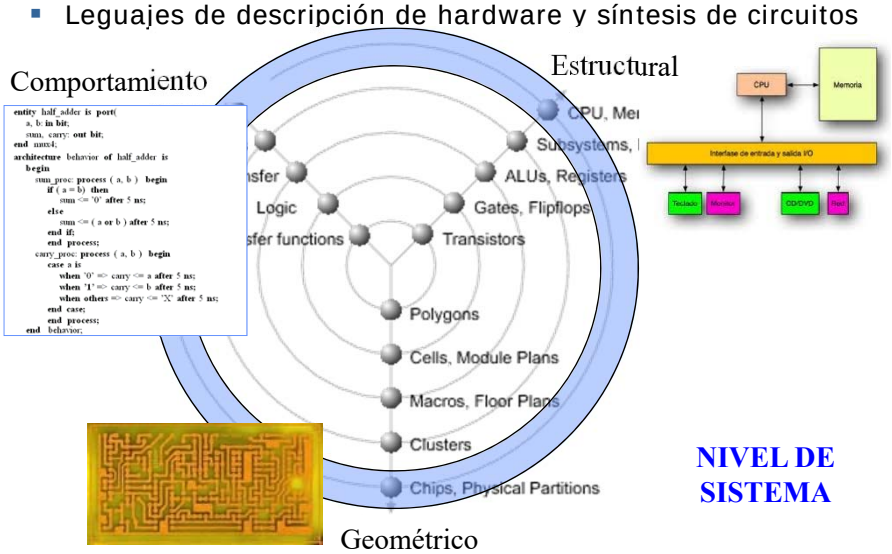
Antes de empezar

DISEÑO/HARDWARE

¿Cómo generar una descripción del sistema que pueda ser sintetizada y dar lugar a un circuito más rápido, que ocupe menos área y/o que consuma menos potencia?

Antes de empezar

- Leguajes de descripción de hardware y síntesis de circuitos

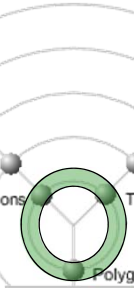


Antes de empezar

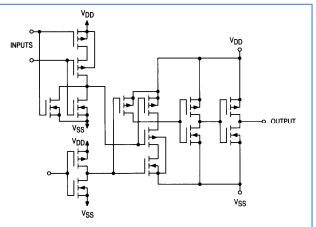
Comportamiento

Estructural

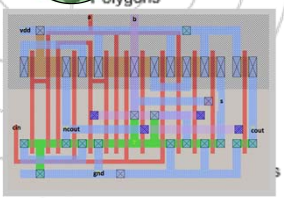
$$\varepsilon_0 \oint \mathbf{E} \cdot d\mathbf{s} = q$$
$$\oint \mathbf{B} \cdot d\mathbf{s} = 0$$
$$\oint \mathbf{B} \cdot d\mathbf{l} = \mu_0 \varepsilon_0 \frac{d\Phi_E}{dt} + \mu_0 i$$
$$\oint \mathbf{E} \cdot d\mathbf{l} = -\frac{d\Phi_E}{dt}$$
Maxwell's Equations



CMOS



Polygons

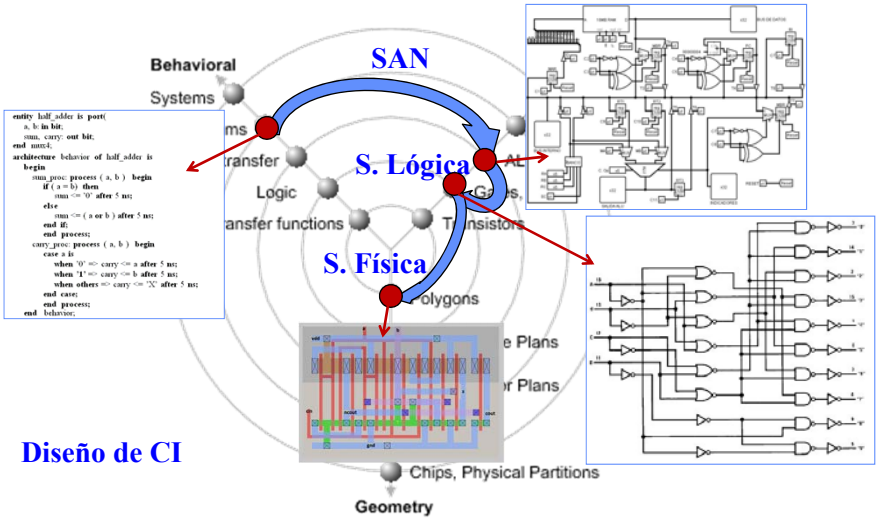


Geométrico

NIVEL Físico-Electrónico

Antes de empezar

- Diseño de un circuito integrado de propósito específico



Diseño de CI

Antes de empezar

- Lenguajes de descripción de hardware (HDL): ejemplos

```
 4
 5 entity signed_adder is
 6   port
 7   (
 8     aclr : in    std_logic;
 9     clk  : in    std_logic;
10     a    : in    std_logic_vector;
11     b    : in    std_logic_vector;
12     q    : out   std_logic_vector
13   );
14 end signed_adder;
15
16 architecture signed_adder_arch of signed_adder is
17   signal q_s : signed(a'high+1 downto 0); -- extra bit wide
18
19 begin -- architecture
20   assert (a'length >= b'length)
21     report "Port A must be the longer vector if different sizes!"
22     severity FAILURE;
23   q <= std_logic_vector(q_s);
24
25   adding_proc:
26   process (aclr, clk)
27     begin
28       if (aclr = '1') then
29         q_s <= (others => '0');
30       elsif rising_edge(clk) then
31         q_s <= ('0'&signed(a)) + ('0'&signed(b));
32       end if; -- clk'd
33     end process;
34
35 end signed_adder_arch;
```

VHDL (VHSIC HDL)

Antes de empezar

- Lenguajes de descripción de hardware (HDL): ejemplos

```
#define SIZE 8
process gcd (xi, yi, rst, ou)
     in port xi[SIZE], yi[SIZE];
     in port rst;
     out port ou[SIZE];
{
     boolean x[SIZE], y[SIZE];

     write ou = 0;
     if ( rst ) <
       x = read(xi);
       y = read(yi);
     >

     if ((x != 0) & (y != 0))
       repeat {
         while (x >= y)
           x = x - y;
         <
           x = y; /* swap x and y */
           y = x;
         >
       } until (y == 0);
     else
       x = 0;
     write ou = x;
}
```

HardwareC

Aspectos básicos en la ejecución del código objeto

Para entender los principios y técnicas de la optimización en el proceso de compilación es necesario primero recordar cómo ejecutan los programas los procesadores

Aspectos básicos de la ejecución del código objeto

- Arquitecturas de destino
  - **Características del repertorio de instrucciones** del procesador: hay seis aspectos básicos
    1. Número de direcciones
    2. Ubicación de los datos
    3. Modos de direccionamiento
    4. Número y tipos de instrucciones
    5. Tamaño y tipos de datos
    6. Formato máquina
  - **Arquitecturas actuales** de propósito general: hay dos enfoques principales, **CISC** (Complex Instruction Set Computer) o **RISC** (Reduced Instruction Set Computer)

Aspectos básicos de la ejecución del código objeto

- CISC vs. RISC

Tecnología CISC

- 2 direcciones
- Instrucciones de formato variable
- Muchas instrucciones y modos de direccionamiento
- Cualquier instrucción puede referenciar a la memoria
- Número reducido de registros
- La complejidad reside en el microprograma

Tecnología RISC

- 3 direcciones
- Instrucciones de formato fijo
- Pocas instrucciones y modos de direccionamiento
- Sólo carga/almacenamiento pueden referenciar a memoria
- Varios conjuntos de registros
- La complejidad reside en el compilador

Aspectos básicos de la ejecución del código objeto

- CISC: ejemplo IA-32

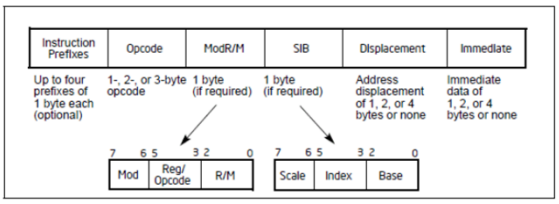
```
xor esi, esi
mov ecx, 6
bucle1:
mov ax, [datos2+esi]
add ax, var
add esi, 2
loop bucle1
```

2 direcciones, reg-mem

64 bits	32 bits	16 bits	8 bits	8 bits
RAX	EAX	AX	AH	AL
RBX	EBX	BX	BH	BL
RCX	ECX	CX	CH	CL
RDX	EDX	DX	DH	DL
RSI	ESI	SI	No disponible	
RDI	EDI	DI	No disponible	
RSP	ESP	SP	No disponible	
RBP	EBP	BP	No disponible	

8 registros

Formato máquina de tamaño variable



Aspectos básicos de la ejecución del código objeto

- RISC: ejemplo MIPS-32 bits

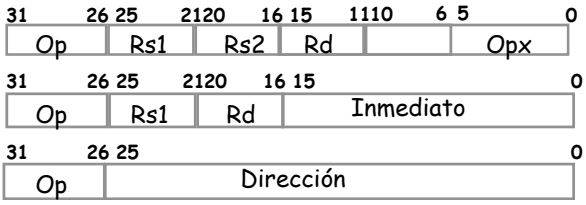
```
lui $s1, 4097
lw  $s3, 0($1)
lw  $s4, 4($1)
add $s5, $s4, $s3
sw  $s5, 8($s1)
mul $s5, $4,$s3
sw  $s5, 12($s1)
```

3 direcciones, reg-reg-reg

Registro	Número	Uso
$zero	0	Constante con valor 0
$at	1	Temporal para el ensamblador
$v0 - $v1	2 – 3	Valores devueltos en funciones
$a0 - $a3	4 – 7	Argumentos en funciones
$t0 - $t7	8 – 15	Temporales
$s0 - $s7	16 – 23	Temporales salvados
$t8 - $t9	24 – 25	Temporales
$k0 - $k1	26 – 27	Reservados para kernel
$gp	28	Puntero global
$sp	29	Puntero de pila
$fp	30	Puntero de marco
$ra	31	Dirección de retorno

32 registros

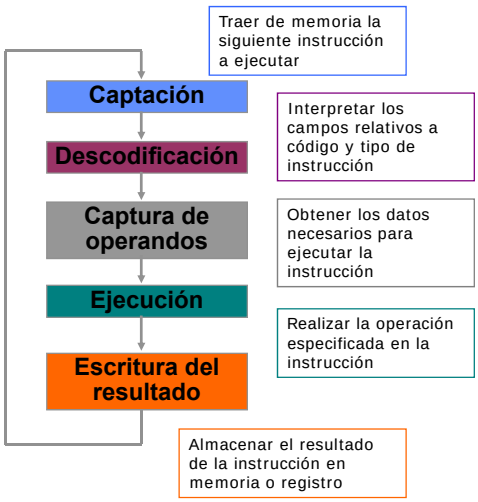
Formato máquina de tamaño fijo



Aspectos básicos de la ejecución del código objeto

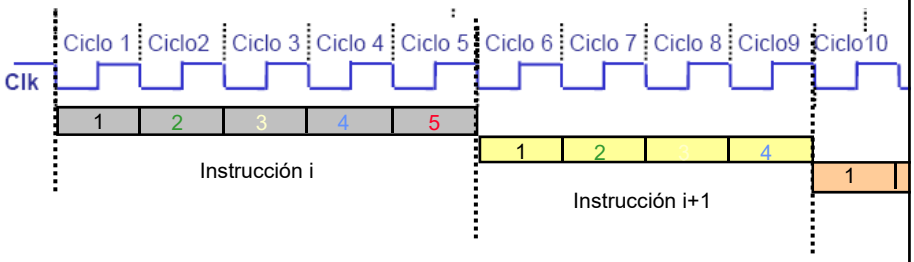
- Fases en la ejecución de instrucciones

- Para ejecutar un programa hay que ejecutar las instrucciones que lo forman, una a una.
- La ejecución de cada instrucción implica comúnmente las tareas reflejadas en la figura de la derecha.
- Estas tareas se suelen realizar en etapas, donde cada etapa, a su vez, se realiza en un ciclo de reloj.
- El número de etapas y la duración de cada ciclo de reloj depende del procesador.



Aspectos básicos de la ejecución del código objeto

- Ejecución multiciclo
  - Cada instrucción requiere varios ciclos de reloj para completarse
  - Tras finalizar una instrucción se ejecuta la siguiente según el orden del programa



Aspectos básicos de la ejecución del código objeto

- **Tiempo de ejecución**

Tiempo de CPU para un programa

$$T_{CPU} = Ciclos\_total\_programa \text{ x } T_{ciclo}$$

$$T_{CPU} = NI \text{ x } CPI \text{ x } T_{ciclo} = NI \text{ x } CPI \text{ x } 1/f$$

- NI = nº de instrucciones del programa
- CPI = nº medio de ciclos por instrucción
- T_{ciclo} = período de reloj del procesador
- f = frecuencia de reloj del procesador

Aspectos básicos de la ejecución del código objeto

- **Tiempo de ejecución: ejemplo**

¿Cuál sería el T_{CPU} para el fragmento de código suponiendo que se ejecuta en un procesador a 40MHz con el repertorio de instrucciones de la tabla?

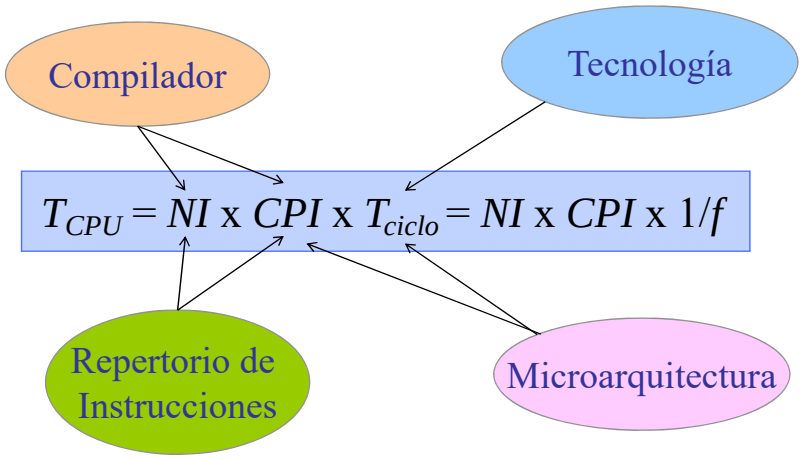
T_{CPU} = 33 ciclos / 40MHz
= 0,825 * 10^{-6} segundos
= 0,825 microsegundos

¿Cuál sería el CPI para este programa?
CPI = 33 ciclos / 7 instruc. = 4,7

INSTRUCCIÓN	ciclos
lui $s1, 4097	4
lw $s3, 0($s1)	5
lw $s4, 4($s1)	5
add $s5,$s4,$s3	4
sw $s5, 8($s1)	5
mul $s5,$s4,s$3	6
sw $s5, 12($s1)	4

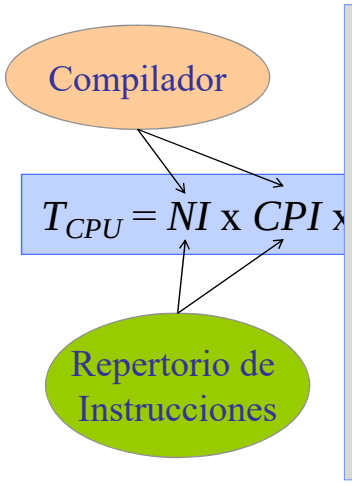
Aspectos básicos de la ejecución del código objeto

- Tiempo de CPU: ¿de qué depende?



Aspectos básicos de la ejecución del código objeto

- Tiempo de CPU: ¿de qué depende?



El compilador es el responsable del código final que se ejecuta, por tanto el NI final y también el CPI dependen de él.

El repertorio de instrucciones del procesador para el que se compila determina qué instrucciones pueden utilizarse y cuál es su CPI.

A veces producir una secuencia de código con menor NI es a costa de utilizar instrucciones más complejas, que pueden requerir más ciclos, por lo tanto, el CPI se incrementaría.

El objetivo del compilador debe ser reducir el producto NIxCPI (ver el siguiente ejemplo).

Aspectos básicos de la ejecución del código objeto

- Tiempo de CPU: el impacto del compilador en el NI y CPI
  - Ejemplo: Consideremos el siguiente repertorio de instrucciones de un determinado procesador.

Tipo de instrucción	Número de ciclos
A	1
B	2
C	3

  - Para una declaración particular de un lenguaje de alto nivel, el compilador tiene dos alternativas:

Alternativas	Número de instrucciones por tipo		
	A	B	C
Código 1	2	1	2
Código 2	4	1	1

Aspectos básicos de la ejecución del código objeto

- Tiempo de CPU: el impacto del compilador en el NI y CPI
  - Ejemplo:

Alternativas	Número de instrucciones por tipo		
	A	B	C
Código 1	2	1	2
Código 2	4	1	1

  - De estas dos alternativas, el Código 2 tiene mayor NI (6 frente a 5 del Código 1). Sin embargo, las instrucciones del tipo A requieren menos ciclos, y eso hace que el CPI del Código 2 sea menor (1,5 frente a 2 del Código 1).
  - La alternativa mejor es la 2, porque el producto NIxCPI es menor (9 frente a 10 que tiene el Código 1).

Aspectos básicos de la ejecución del código objeto

- **Caso práctico 1**. Escribir un programa en un lenguaje de alto nivel que sume dos vectores de 5 componentes. Traducirlo después a lenguaje ensamblador del MIPS R2000.
  - ¿Qué alternativas tenemos a la hora de escribir el código en alto nivel?
  - ¿Qué alternativas tenemos a la hora de traducirlo a ensamblador?
  - ¿Cómo afectan estas alternativas al tamaño del programa? ¿Y al tiempo de ejecución?

OPCIÓN		TAMAÑO			TRÁFICO	INSTRUCCIONES EJECUTADAS	TIEMPO EJECUCIÓN
		Datos	Instruc.	Total			
1	Sin bucle 1	60B	104B	164B	164B	26	$130T_{ciclo}$
2	Sin bucle 2	60B	84B	144B	144B	21	$105T_{ciclo}$
3	Con bucle 1	60B	40B	100B	216B	39	$195T_{ciclo}$

Si comparamos la opción 2 con la 3, tenemos:

- La 3 requiere un **30% aprox. menos de memoria**
- Genera un 35% aprox. más de tráfico con memoria
- Requiere un **75% aprox. más de tiempo** de ejecución

Aspectos básicos de la ejecución del código objeto

- Microarquitecturas de destino (características de la organización de los procesadores)
  - Explotación del paralelismo
    - Procesadores segmentados
    - Procesadores superescalares
    - Procesadores VLIW
    - Procesadores multi-hebra
    - Multiprocesadores
  - Mejora del sistema de memoria
    - Jerarquía de memoria, memoria cache

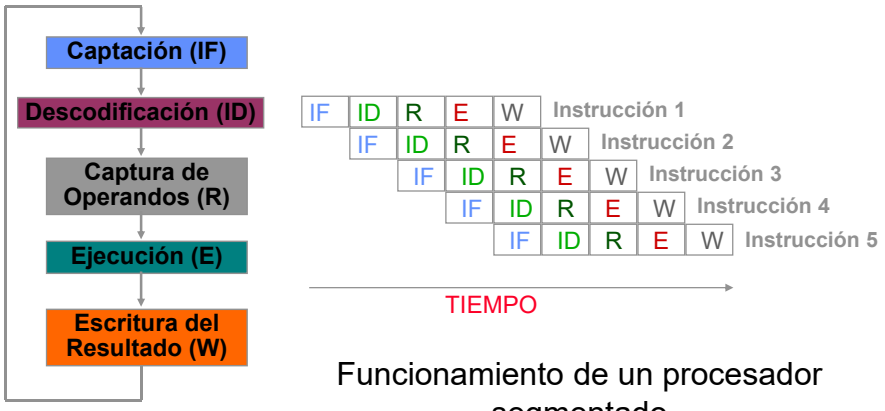
$$T_{CPU} = NI \times CPI \times T_{ciclo}$$

Microarquitectura

Las técnicas que explotan el paralelismo y mejoran el tiempo de acceso a memoria lo que hacen básicamente es reducir el número de ciclos medio por instrucción (CPI) del programa, aunque pueden también afectar a la frecuencia de trabajo del procesador.

Aspectos básicos de la ejecución del código objeto

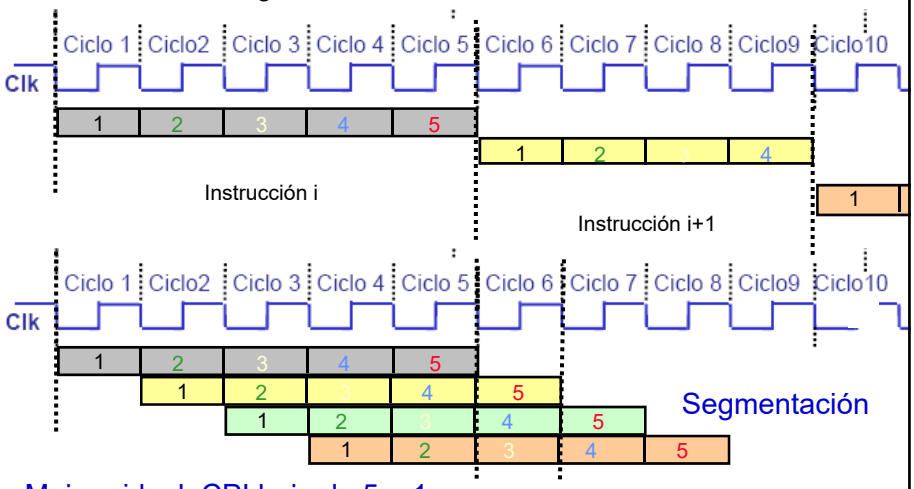
- Mircroarquitecturas destino: procesadores segmentados



Funcionamiento de un procesador segmentado

Aspectos básicos de la ejecución del código objeto

- Multiciclo vs. segmentación



Mejora ideal: CPI baja de 5 a 1

Aspectos básicos de la ejecución del código objeto

- **Microarquitecturas destino:** dificultad del código para aprovechar la máxima ganancia de la segmentación
  - **Riesgo (*hazard*)**: situación que impide que una cierta instrucción pueda ejecutarse correctamente en el ciclo de reloj que le correspondería
  - **Reducen el rendimiento**, haciendo que la ganancia real se aleje de la ganancia ideal que promete la segmentación
  - **Implican** disponer de mecanismos para detectar estas situaciones y, **al menos**, detener la ejecución de la instrucción que entra en conflicto hasta que éste se resuelva (durante uno o varios ciclos). De no ser así, el programa se ejecutaría incorrectamente.

Aspectos básicos de la ejecución del código objeto

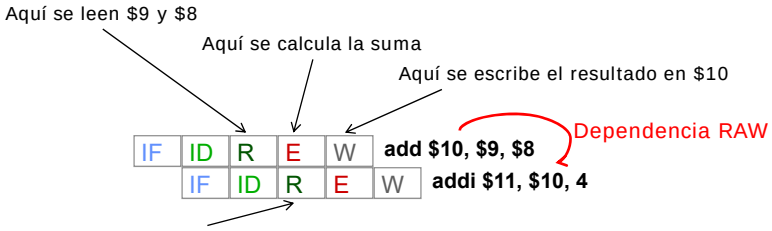
- **Microarquitecturas destino:** dificultad del código para aprovechar la máxima ganancia de la segmentación

Tipos de riesgos derivados del código

- **De datos**: se producen cuando existen dependencias de datos entre dos instrucciones. Pueden ser de 3 tipos:
  - **RAW**: una instrucción depende de los resultados de otra previa cuya ejecución aún no ha concluido
  - **WAW**: una instrucción escribe su resultado en el mismo registro (o dirección de memoria) que otra previa
  - **WAR**: una instrucción escribe su resultado en un registro (o dirección de memeoria) que otra instrucción previa lee
- **De control**: se deben a los saltos, que provocan que no se conozca cuál es la siguiente instrucción que hay que ejecutar hasta que no se resuelve el salto

Aspectos básicos de la ejecución del código objeto

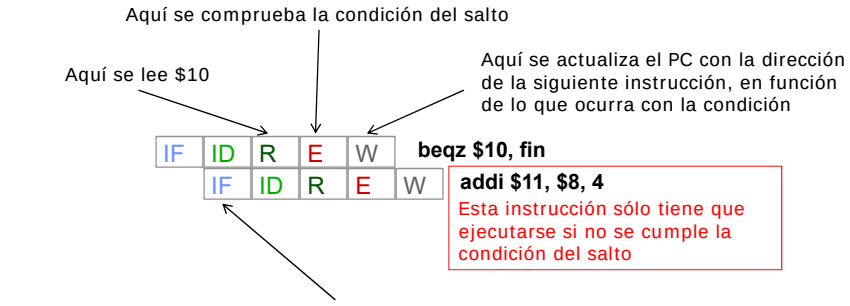
- **Microarquitecturas destino:** dificultad del código para aprovechar la máxima ganancia de la segmentación
  - Impacto de las dependencias de datos RAW en la ejecución sobre un procesador segmentado: consideremos el siguiente ejemplo, donde una instrucción de suma calcula un valor (y lo almacena en $10) que es utilizado por la siguiente instrucción.



Aquí se debe leer $10, pero la anterior instrucción aún no ha calculado y almacenado el valor, por lo hay que detener la ejecución de la instrucción addi hasta que el dato esté disponible, con la consiguiente penalización en T_{CPU}

Aspectos básicos de la ejecución del código objeto

- **Microarquitecturas destino:** dificultad del código para aprovechar la máxima ganancia de la segmentación
  - Impacto de las dependencias de control en la ejecución sobre un procesador segmentado.



Aquí se comienza la ejecución de la siguiente instrucción en el código, pero aún no se sabe cuál es la instrucción que hay que ejecutar

Aspectos básicos de la ejecución del código objeto

Tipos de riesgos: soluciones habituales

- **De datos**:
  - Las penalizaciones por riesgos de tipo RAW se reducen habitualmente mediante una técnica ***hardware*** conocida como **forwarding** (anticipación de operandos). También pueden eliminarse vía ***software***, mediante reordenamiento del código
  - Los riesgos WAW y WAR sólo aparecen si se altera el orden de ejecución de instrucciones (*ejecución fuera de orden*). Se eliminan mediante una técnica denominada **renombrado de registros**, que puede aplicarse tanto estática (vía ***software***) como dinámicamente (vía ***hardware***)
- **De control**:
  - Resolver el salto lo antes posible (averiguar si se toma o no, y calcular la dirección de destino) para minimizar penalizaciones
  - **Salto retardado**: técnica ***software/hardware*** que consigue reducir la penalización en algunos de los saltos de un programa
  - **Predicción de saltos**: técnica que puede aplicarse ***estática*** o ***dinámicamente***. Existen diferentes algoritmos que la implementan

Aspectos básicos de la ejecución del código objeto

- **Caso práctico 2**. Analizar las dependencias de datos de un fragmento de código en ensamblador y estudiar cómo impacta el orden de las instrucciones en el tiempo de ejecución sobre un procesador segmentado.
  - ¿Cómo pueden eliminarse las dependencias WAR y WAW?
  - ¿Qué orden de las instrucciones favorece un mayor aprovechamiento de la segmentación, ofreciendo un mejor T_{CPU}?

Aspectos básicos de la ejecución del código objeto

Optimización del rendimiento del procesador

- Con **segmentación** se reduce el **CPI**, idealmente hasta **1**.
- Para mejorar el rendimiento de un procesador hay que reducir el CPI por debajo de 1, es decir, lograr ejecutar más de una instrucción por ciclo.
- Para conseguir **CPI < 1** hay que
  - **Encontrar** instrucciones en el código que puedan ejecutarse en paralelo
  - Disponer de recursos HW suficientes para poder **realizar el procesamiento simultáneo** de dichas instrucciones

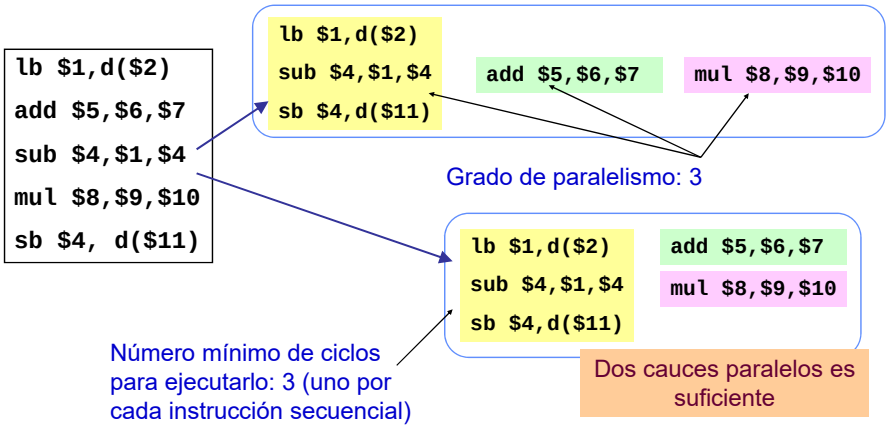
Aspectos básicos de la ejecución del código objeto

Paralelismo entre instrucciones (ILP)

- Capacidad para **poder ejecutar varias instrucciones** de manera simultánea
- Hay paralelismo entre dos o más instrucciones **si no existen entre ellas dependencias** de datos ni de control
- El **grado de paralelismo** en el código es el número máximo de instrucciones que pueden ejecutarse en paralelo y supone un límite al CPI alcanzable
- Sólo es relevante el grado de paralelismo entre **instrucciones próximas** en un fragmento de código

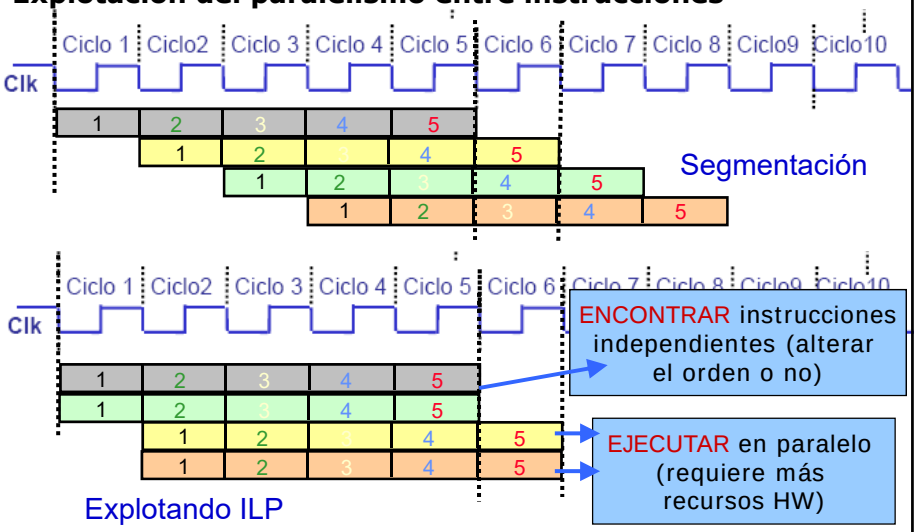
Aspectos básicos de la ejecución del código objeto

- **Paralelismo entre instrucciones (ILP)**: Ejemplo
  - ¿Cuál es el grado de paralelismo en el fragmento de código? ¿Cuál es el número mínimo de ciclos necesarios para su ejecución? ¿Cuántos caminos de ejecución paralelos hacen falta para alcanzar el mínimo?



Aspectos básicos de la ejecución del código objeto

Explotación del paralelismo entre instrucciones



Aspectos básicos de la ejecución del código objeto

Explotación del paralelismo entre instrucciones

- Existen dos enfoques para explotar ILP, que se diferencian fundamentalmente en las técnicas aplicadas para encontrar e incrementar las instrucciones independientes que se ejecutan simultáneamente
  - **Procesadores superescalares = explotación dinámica**
    - Las instrucciones que se procesan en cada ciclo las decide el procesador, respetando el orden en que aparecen en el programa o no (en este caso se dice que hay *ejecución fuera de orden*).
    - Procesa un número variable de instrucciones por ciclo.
  - **Procesadores VLIW** (Very Long Instruction Word) **= explotación estática**
    - Es el compilador quien determina qué instrucciones se van a ejecutar en paralelo.
    - Varias instrucciones se codifican en lenguaje máquina como una única instrucción.
    - Procesa un número fijo de instrucciones por ciclo.

Aspectos básicos de la ejecución del código objeto

Explotación del paralelismo entre instrucciones

- Con independencia del enfoque utilizado (superescalares o VLIW), la ejecución paralela de instrucciones permite **mejorar el T_{CPU}**, reduciendo, idealmente, el CPI por debajo de 1.
- Los **problemas derivados de las dependencias** de datos y de control que vimos para el caso de los procesadores segmentados **se agravan**. Por ejemplo, las dependencias WAW, que en un procesador segmentado no ocasionan riesgos, sí pueden provocarlos cuando dos instrucciones con estas dependencias se ejecutan a la vez.
- El **compilador** adquiere por tanto un rol **más relevante** de cara a la mejora del tiempo de ejecución.

Generación y optimización de código en el proceso de compilación

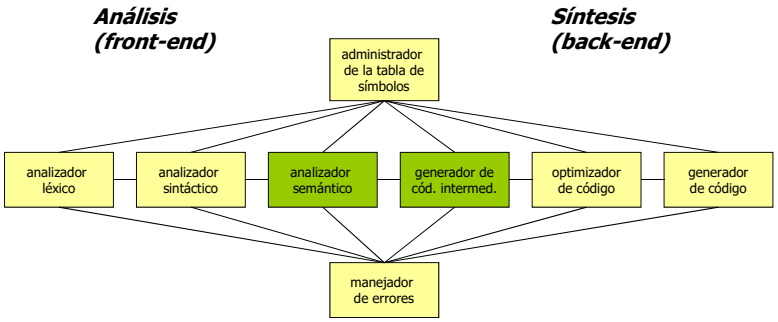
Una vez entendido cómo se ejecutan los programas en el procesador y cuál es el impacto que tiene diferentes características del código (como los bucles, saltos o el orden de las instrucciones) en el tiempo de ejecución, vamos a ver cómo trabaja el compilador para intentar generar un código objeto que ofrezca buenos resultados en memoria requerida y/o tiempo de ejecución.

Generación y optimización del código: Introducción

- El proceso de compilación se desglosa en dos partes: la parte que depende sólo del **lenguaje fuente** (etapa inicial o **front-end**) y la parte que depende sólo del **lenguaje objeto** (etapa final o **back-end**).
- La etapa inicial traduce un programa fuente a una **representación intermedia** a partir de la cual la etapa final genera (y optimiza) el código objeto.

Generación y optimización del código: Introducción

- Fases básicas del proceso de compilación



Generación y optimización del código: Introducción

- La misión del proceso de optimización es mejorar la calidad y eficiencia del código objeto para incrementar la velocidad de ejecución y reducir el espacio ocupado, manteniendo el significado del programa.
- Factores a optimizar:
  - **Velocidad de ejecución**
  - **Necesidades de memoria**
- Las optimizaciones de espacio suelen ser incompatibles con las de velocidad:
  - mejorar la velocidad suele aumentar el espacio requerido y viceversa.
- Cada vez toma más relevancia el consumo de energía y la disipación de **potencia** como factor de rendimiento a mejorar con las técnicas de compilación

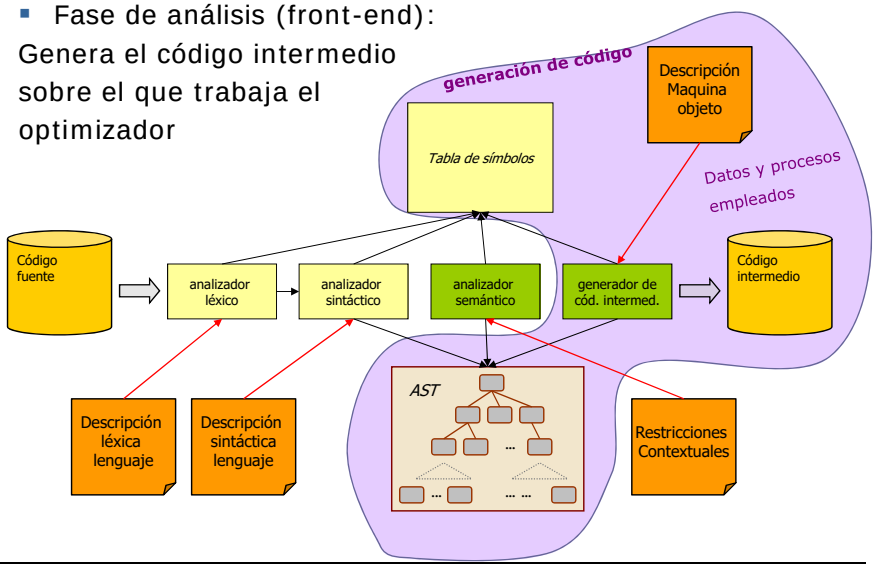
Generación y optimización del código: Introducción

- Estadísticamente se sabe que un programa utiliza el 90% del tiempo de ejecución en el 10% del código, por tanto, las técnicas de optimización:
  - se basan en un análisis de la estructura del programa y del flujo de datos, subdividendo el programa en regiones de optimización.
- El proceso de mejora de código se puede hacer sobre:
  - el **código intermedio** generado tras el análisis semántico
  - el **código objeto** tras o durante la fase de generación de código.

Generación y optimización del código: Introducción

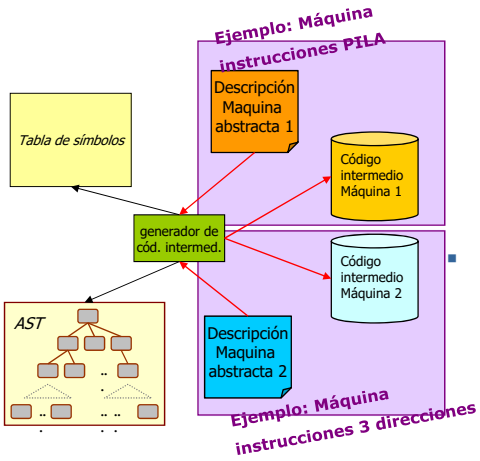
- Fase de análisis (front-end):

Genera el código intermedio sobre el que trabaja el optimizador



Generación y optimización del código: Introducción

- Fase de análisis (front-end)

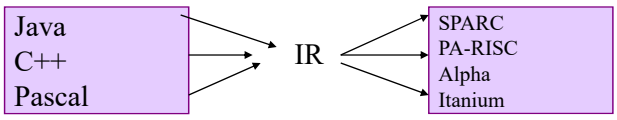


- Una misma representación intermedia de árbol sintáctico y tabla de símbolos se puede traducir a distintos códigos objeto intermedios en función de las características del tipo de máquina abstracta que vaya a evaluar las instrucciones.
- La mayoría de los compiladores transforman el programa fuente en algún tipo de representación intermedia (versión del código fuente independiente del lenguaje de la máquina objeto), para luego convertir esa representación en otra más eficiente o en el código final.

Generación y optimización del código: Introducción

Representaciones intermedias

- Aunque esas representaciones intermedias son 'un paso más', ofrecen ventajas que aconsejan su uso:
  1. **Independizar el front-end** (dependiente del lenguaje**) del back-end** (dependiente de la máquina), con lo que eso supone en cuanto a modularidad y a posibilidades para compilaciones cruzadas (portabilidad).
  2. Mayor nivel de abstracción.
  3. **Posibilidad de realizar optimizaciones**: la mayor parte de las optimizaciones se realizan sobre representaciones intermedias del código.
- Las representaciones intermedias (su tipo, diseño…) afectan tanto la velocidad como la eficiencia del compilador.



Generación y optimización del código: Introducción

Técnicas de optimización de código

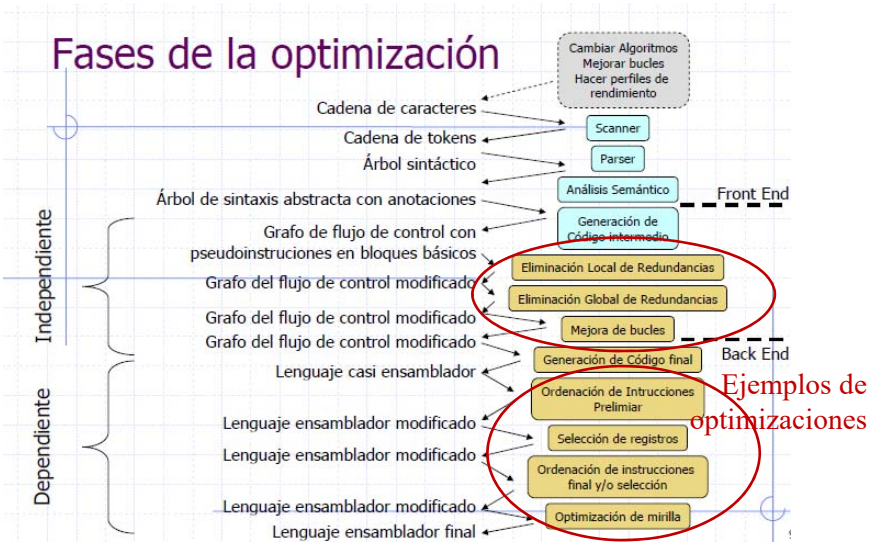
- Muchas y muy variadas.
- La experiencia muestra que existe:
  - un conjunto de técnicas básicas y muy sencillas que llevan a grandes mejoras
  - técnicas muy complejas que al implementar sólo mejoran un pequeño porcentaje del tiempo
- Evaluar/ponderar técnicas que impliquen:
  - mejoras significativas en el código (en promedio acelerar el tiempo de ejecución)
  - bajo incremento en la complejidad del compilador (costo de programación, mayor tiempo de la compilación)

Generación y optimización del código: Introducción

Clasificación de las optimizaciones

- En función de la dependencia de la arquitectura (por la etapa en el proceso de compilación en que se hacen las transformaciones).
  - Optimizaciones **independientes de la máquina**
  - Optimizaciones **dependientes de la máquina**
- Por el área del programa al que se le aplican las transformaciones: es decir, en función del ámbito de aplicación:
  - Optimizaciones **locales**: Aplicadas dentro de un Bloque Básico
    - Sólo estudian las instrucciones del B.B. actual
  - Optimizaciones **globales**: Aplicadas a más de un B.B.
    - Consideran contenido y flujo de datos entre todos o parte de los B.B.
    - Necesidad de recoger información sobre los B.B. y sus interrelaciones, por lo que se usan algoritmos de análisis global de flujo de datos

Generación y optimización del código: Introducción



Generación y optimización del código: Bloques básicos y GFC

Código intermedio de 3 direcciones

- Representación lineal de un árbol sintáctico.
- Las instrucciones utilizan tres direcciones de memoria o registros, dos para los operandos y una para el resultado: Esta RI es una generalización del código ensamblador de una máquina virtual de 3 direcciones.

  resultado = operando1 operador operando2
- Es la representación intermedia es la más extendida.

  Ej. a : = b * (-c)+ b * (-c) :

árbol sintáctico

```
t1 : = - c
t2 : = b * t1
t3 : = - c
t4 : = b * t3
t5 : = t2 + t4
a  : = t5
```

código de 3 direcciones

- Ventajas: adecuado para la mayoría de las máquinas, aspecto 'compacto'

Generación y optimización del código: Bloques básicos y GFC

Bloques básicos

- Un **bloque básico** es una secuencia máxima de instrucciones consecutivas con las siguientes propiedades:
  - El flujo de control sólo puede entrar en el bloque básico a partir de la primera instrucción (es decir, no hay saltos hacia la parte media del bloque)
  - El control saldrá del bloque sin bifurcarse, excepto tal vez en la última instrucción del bloque.
- Las optimizaciones de ámbito local se aplican dentro de cada bloque básico.
- Algoritmo para identificar bloques básicos:
  - Buscar instrucciones "líder":
    - Primera instrucción
    - Instrucción destino de un salto
    - Instrucción siguiente a un salto
  - El bloque básico lo forman las instrucciones entre líderes

Generación y optimización del código: Bloques básicos y GFC

- Grafos de Flujo de Control (GFC)
  - Son grafos dirigidos donde los nodos se corresponden con los bloques básicos identificados y las flechas indican el orden de ejecución entre ellos.
  - Son una representación gráfica del código intermedio que resulta útil para las fases de optimización y generación de código.
- **Caso práctico 3**. Generar el GFC a partir del código intermedio de 3 direcciones correspondiente a un fragmento de código.

Generación y optimización del código: Técnicas de optimización

Optimizaciones independientes de la máquina

- Técnica 1. **Eliminación de subexpresiones comunes**
  - Las expresiones que aparecen calculadas más de una vez pueden calcularse sólo una y reutilizar el resultado.
  - Combinar su aplicación con propiedades de los operadores, como la conmutativa o asociativa.
  - Ejemplo:

```
x = a + b                    x = a + b
…   (a y b no cambian aquí) …
z = a + b                    z = x
```


Generación y optimización del código: Técnicas de optimización

Optimizaciones independientes de la máquina

- Técnica 2. **Eliminación de código muerto**
  - Eliminar aquellas instrucciones que nunca van a llegar a ejecutarse
  - Puede deberse a
    - Código no alcanzable en el grado de flujo de control (bloques básicos que no son destino de ningún salto o bloques que nunca se alcanzan debido a la evaluación de la condición del salto)
      - Ejemplo:

```
if (DEBUG) {
  printf(…);
}
```

    - Asignaciones a variables que no se utilizan posteriormente (la variable deja de "estar viva" a la salida de ese bloque básico)

Generación y optimización del código: Técnicas de optimización

Optimizaciones independientes de la máquina

- Técnica 3. **Simplificación algebraica**
  - Aplicando identidades algebraicas (propiedades y leyes aritméticas) pueden simplificarse o eliminarse algunas instrucciones.
    - Ejemplo: las siguientes instrucciones pueden eliminarse

```
x = x + 0
x = x - 0
x = x *1
x = x/1
```

    - Ejemplo: las siguientes instrucciones pueden simplificarse (reducción de fuerza)

```
x = a * 0      =>    x = 0
x = a ** 2     =>    x = a * a
x = a * 2      =>    x = a + a
x = a * 8      =>    x = a << 3
```


Generación y optimización del código: Técnicas de optimización

Optimizaciones independientes de la máquina

- Técnica 4. **Constant folding**
  - Reemplazar expresiones por su resultado, cuando éste es evaluable en tiempo de compilación.
  - Combinar con la aplicación de propiedades de los operadores, como la conmutativa o asociativa.
  - Ejemplo:

```
x = 2 + 5 + a + b  =>    x = 7 + a + b
x = 2 + c + 10     =>    x = 12 + c
if (5<0) jump L    =>    puede eliminarse
```

Generación y optimización del código: Técnicas de optimización

Optimizaciones independientes de la máquina

- Técnica 5. **Propagación de copias**
  - Ante instrucciones de la forma f=a sustituir todos los usos de f por a (donde a es una constante o una variable).
  - Ejemplo:

```
x = a                    x = a
y = x * 2      =>        y = a * 2
```

(*si x no se necesitase más, podría eliminarse*)

Generación y optimización del código: Técnicas de optimización

Optimizaciones independientes de la máquina

- Ejemplo: aplicar las anteriores transformaciones a este fragmento de código para reducirlo al mínimo número de instrucciones.

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

Generación y optimización del código: Técnicas de optimización

Optimizaciones independientes de la máquina

- **Optimización de bucles**
  - Es una de las principales técnicas globales de optimización
  - Incluye dos tipos de transformaciones:
    - **Factorización de expresiones invariantes**
    - **Variables de inducción y reducción en fuerza**
  - En el caso de que existan bucles anidados, las transformaciones se aplican de dentro hacia fuera (es decir, empezando por el bucle más interno)

Generación y optimización del código: Técnicas de optimización

Optimizaciones independientes de la máquina

- Técnica 6. **Optimización de bucles: factorización de expresiones invariantes (reducción de frecuencia)**
  - *Expresión invariante de un bucle*: expresión cuyo valor es constante durante toda la ejecución del bucle
  - Idea: sacar las expresiones invariantes del cuerpo del bucle (al sacarlas pueden quedar dentro del cuerpo de otro bucle más externo, repetir el proceso)
  - Ejemplo:

```
while (i <= limite – 2) {
…

}
```

```
t = limite – 2;
while (i <= t) {
…
}
```

Generación y optimización del código: Técnicas de optimización

Optimizaciones independientes de la máquina

- Técnica 7. **Optimización de bucles: variables de inducción y reducción de potencia**
  - *Variable de inducción*: variable tal que cada vez que se asigna dentro del cuerpo del bucle, su valor se incrementa en una constante *c*.
  - Las variables de inducción pueden calcularse con un solo incremento (suma o resta) por cada iteración del bucle.
  - La transformación de sustituir una operación más costosa, como la multiplicación, por una más económica, como la suma, se conoce como *reducción de potencia*.

Generación y optimización del código: Técnicas de optimización

Optimizaciones independientes de la máquina

- Técnica 7. **Optimización de bucles: variables de inducción y reducción de potencia**
  - Ejemplo:

```
i= 1;
while (i <= 1000) {
    …
    a = 4*i+3;
    b = 2*i;
    …
    i++;
}
```

```
i = 1;
t1 = 7;
t2 = 2;
while (i <= 1000) {
    …
    a = t1;
    b = t2;
    …
    i++;
    t1= t1+4;
    t2= t2+2;
}
```

Generación y optimización del código: Técnicas de optimización

Optimizaciones dependientes de la máquina

- Uso eficiente de los modos de direccionamiento
- Uso eficiente de las instrucciones del repertorio
- Asignación eficiente de registros a datos, para minimizar accesos a memoria
- Planificación de instrucciones para maximizar las posibilidades de ejecución paralela

Generación y optimización del código: Técnicas de optimización

Optimizaciones dependientes de la máquina (en realidad, son independientes, pero favorecen la aplicación de otras dependientes, como la planificación de instrucciones)

- Técnica 8. **Desenrollado de bucles**
  - Consiste en eliminar algunas iteraciones del bucle, haciendo que las instrucciones del cuerpo se ejecuten de forma consecutiva.
  - La idea es crear bloques básicos más largos, pudiendo aplicar mayor número de optimizaciones locales y reorganizar las instrucciones para sacar mayor aprovechamiento a las técnicas de optimización hardware del procesador (segmentación, paralelismo)

Generación y optimización del código: Técnicas de optimización

Optimizaciones dependientes de la máquina

- Técnica 8. **Desenrollado de bucles**

```
for ( i = 1000 ; i > 0 ; i = i – 1 )
    x[i] = x[i] + s;
```

```
loop:   ld      f0, 0(r1)
        addd    f4, f0, f2
        sd      f4, 0(r1)
        subui   r1, r1, #8
        bne     r1, loop
```

```
loop:   ld      f0, 0(r1)
        ld      f6, –8(r1)
        ld      f10, –16(r1)
        ld      f14, –24(r1)
        ld      f18, –32(r1)
        addd    f4, f0, f2
        addd    f8, f6, f2
        addd    f12, 10, f2
        addd    f16, f14, f2
        addd    f20, f18, f2
        sd      f4, 0(r1)
        sd      f8, –8(r1)
        sd      f12, –16(r1)
        sd      f16, –24(r1)
        sd      f20, –32(r1)
        subui   r1, r1, #40
        bne     r1, loop
```

Existen dependencias RAW entre cada dos instrucciones consecutivas, y además existe una instrucción de salto que controla el final del bucle:

Parece que no se puede aprovechar mucho ILP

El desenrollado pone de manifiesto un mayor paralelismo ILP

Generación y optimización del código: Técnicas de optimización

Optimizaciones dependientes de la máquina

- Técnica 9. **Segmentación software**
  - Consiste en reorganizar los bucles de forma que cada iteración del código transformado contiene instrucciones de diferentes iteraciones del bucle original.
  - Con esta técnica se consigue que las instrucciones que forman parte de un bloque básico tengan menos dependencias de datos entre ellas, mejorando el aprovechamiento de las técnicas de optimización hardware del procesador.

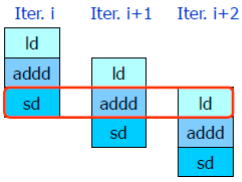
Generación y optimización del código: Técnicas de optimización

Optimizaciones dependientes de la máquina

- Técnica 9. **Segmentación software**

```
for ( i = 1000 ; i > 0 ; i = i – 1 )
    x[i] = x[i] + s;
```

```
loop:   ld      f0, 0(r1)
        addd    f4, f0, f2
        sd      f4, 0(r1)
        subui   r1, r1, #8
        bne     r1, loop
```




```
loop:   sd      f4, 16(r1)
        addd    f4, f0, f2
        ld      f0, 0(r1)
        subui   r1, r1,#8
        bne     r1, loop
```

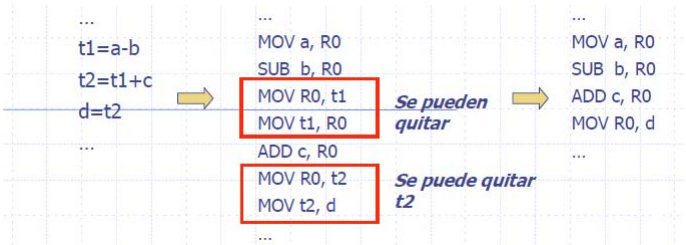
```
Iter. i:    ld      f0, 16(r1)
            addd    f4, f0, f2
            sd      f4, 16(r1)

Iter. i+1:  ld      f0, 8(r1)
            addd    f4, f0, f2
            sd      f4, 8(r1)

Iter. i+2:  ld      f0, 0(r1)
            addd    f4, f0, f2
            sd      f4, 0(r1)
```

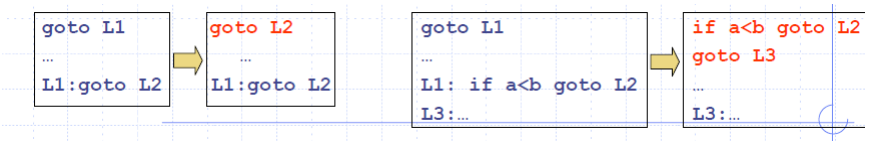

Generación y optimización del código: Técnicas de optimización

- Optimización de "mirilla"
  - Buscar patrones de instrucciones en el código y reemplazarlas por otras más eficientes
  - Puede aplicarse en el código intermedio o en el código objeto
- Técnica 10. **Eliminación de cargas y almacenamientos redundantes**



Generación y optimización del código: Técnicas de optimización

- Optimización de “mirilla”
  - Buscar patrones de instrucciones en el código y reemplazarlas por otras más eficientes
  - Puede aplicarse en el código intermedio o en el código objeto
- Técnica 11. **Mejora del flujo de control**



Generación y optimización del código: Técnicas de optimización

- Observaciones sobre la aplicación de las técnicas
  - En muchas ocasiones, estas técnicas reducen el impacto de la traducción automática (que genera un código mucho menos eficiente que el que podría escribir directamente un programador)
  - La aplicación de unas técnicas a su vez genera posibilidades para la aplicación de otras, por lo que suelen aplicarse de forma iterativa, hasta que no hay transformación posible o hasta que se alcanza un límite de iteraciones (o tiempo)
  - Las transformaciones locales (dentro de los bloques básicos) son las más sencillas y las que más se aplican