Arquitectura i Configuracions Informàtiques

Tema 2. El microprocesador

Davide Careglio

Temario

- Tema 1. Introducción
- Tema 2. El microprocesador
- Tema 3. Memoria
- Tema 4. Dispositivos de E/S y buses
- Tema 5. DataCenters y modelos de comunicación

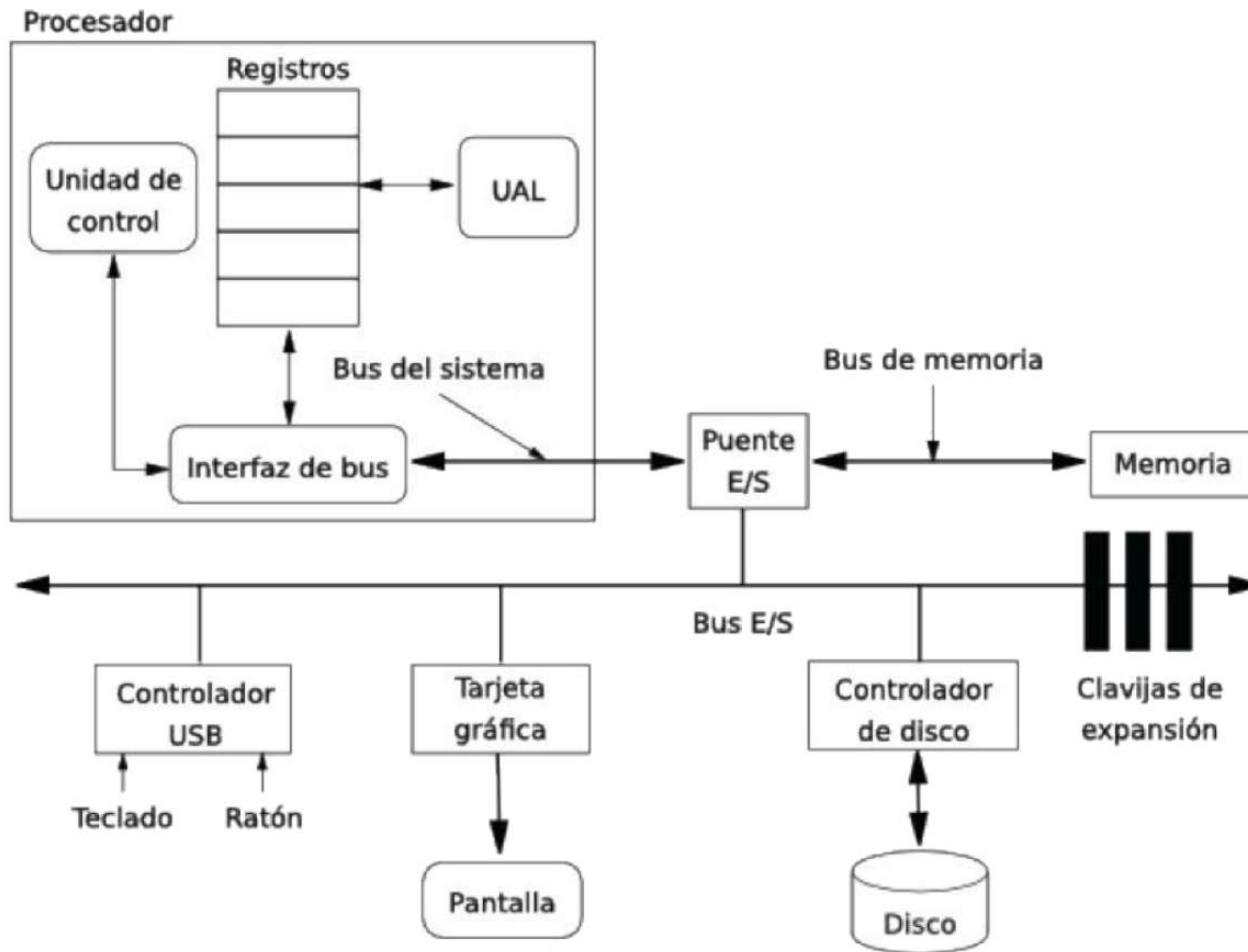
Temario

- Tema 1. Introducción
- Tema 2. El microprocesador
  - Introducción
  - Instruction Set Architecture
  - Arquitectura interna
  - Paralelismo
  - Ejercicios
- Tema 3. Memoria
- Tema 4. Dispositivos de E/S y buses
- Tema 5. DataCenters y modelos de comunicación

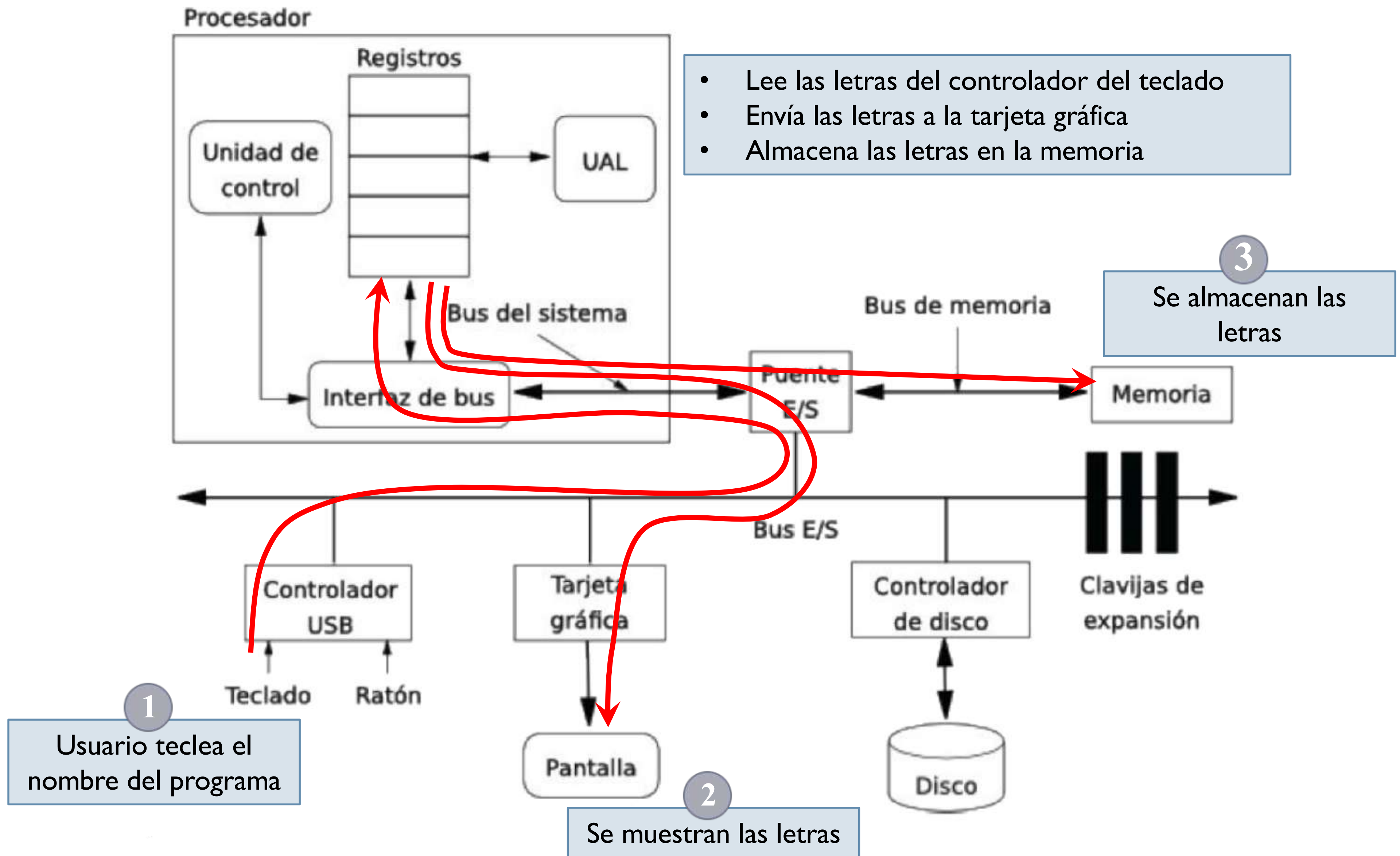
2.1 – Introducción
Ejecución de un programa

- Supongamos que se abre un terminal y se ejecuta el programa "HelloWorld"
- Los pasos para su ejecución son
  1) Usuario teclea el nombre del programa
  2) Se obtiene el programa del disco duro y se almacena en la memoria
  3) El procesador ejecuta una tras otras las instrucciones del programa
  4) Se borra el programa y sus datos de la memoria

2.1 – Introducción
Ejecución de un programa

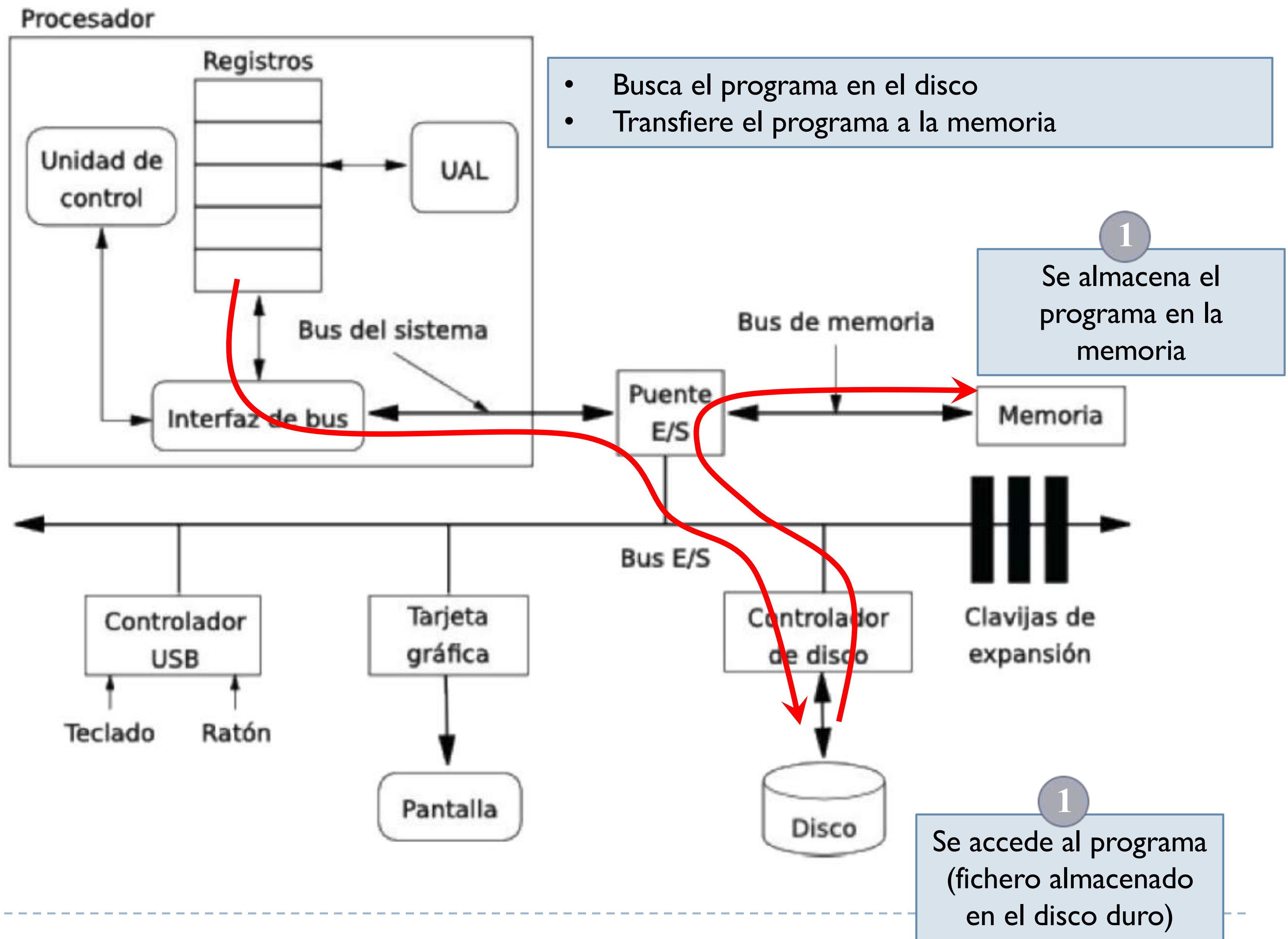


2.1 – Introducción
Ejecución de un programa - Paso 1



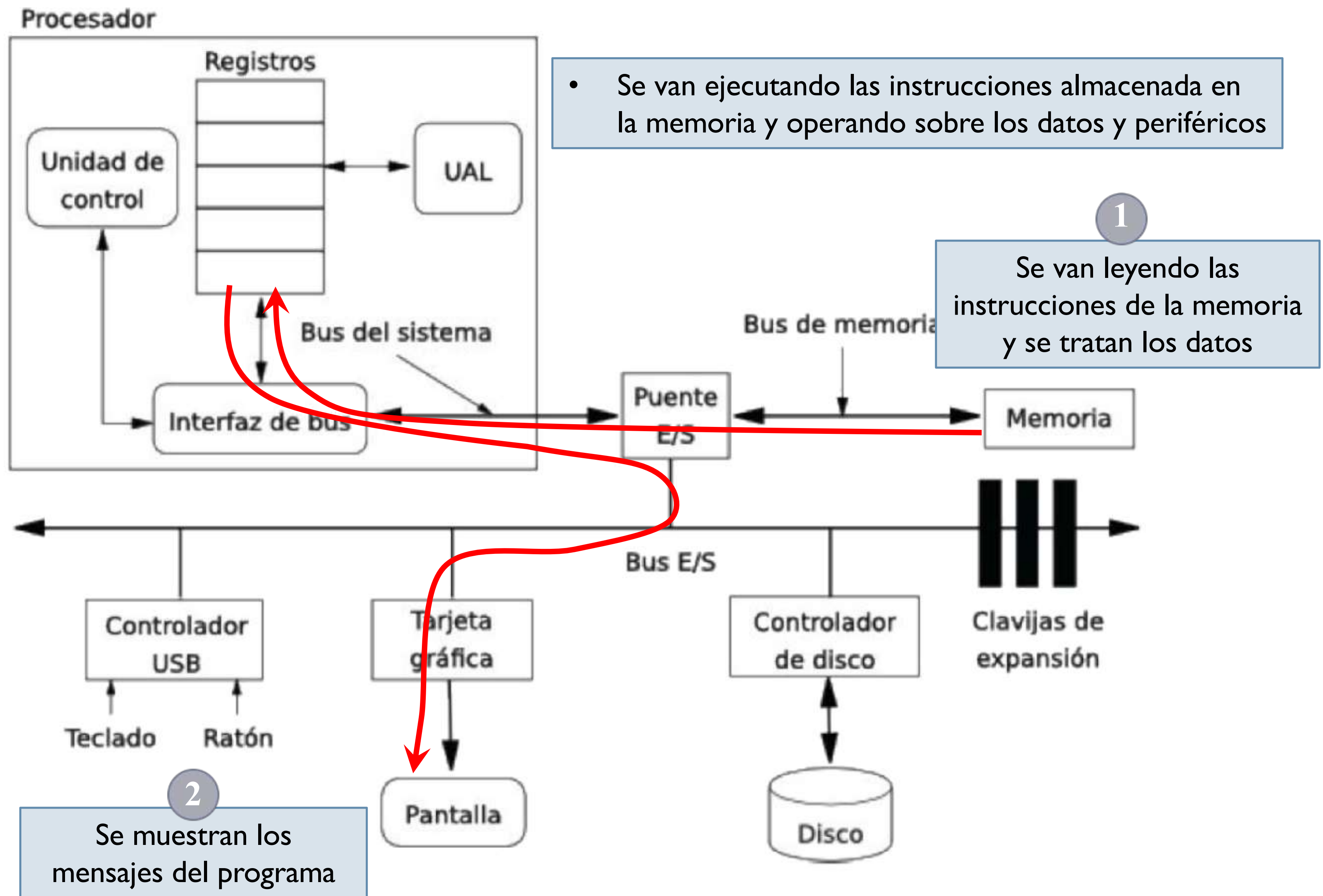
2.1 – Introducción

Ejecución de un programa - Paso 2



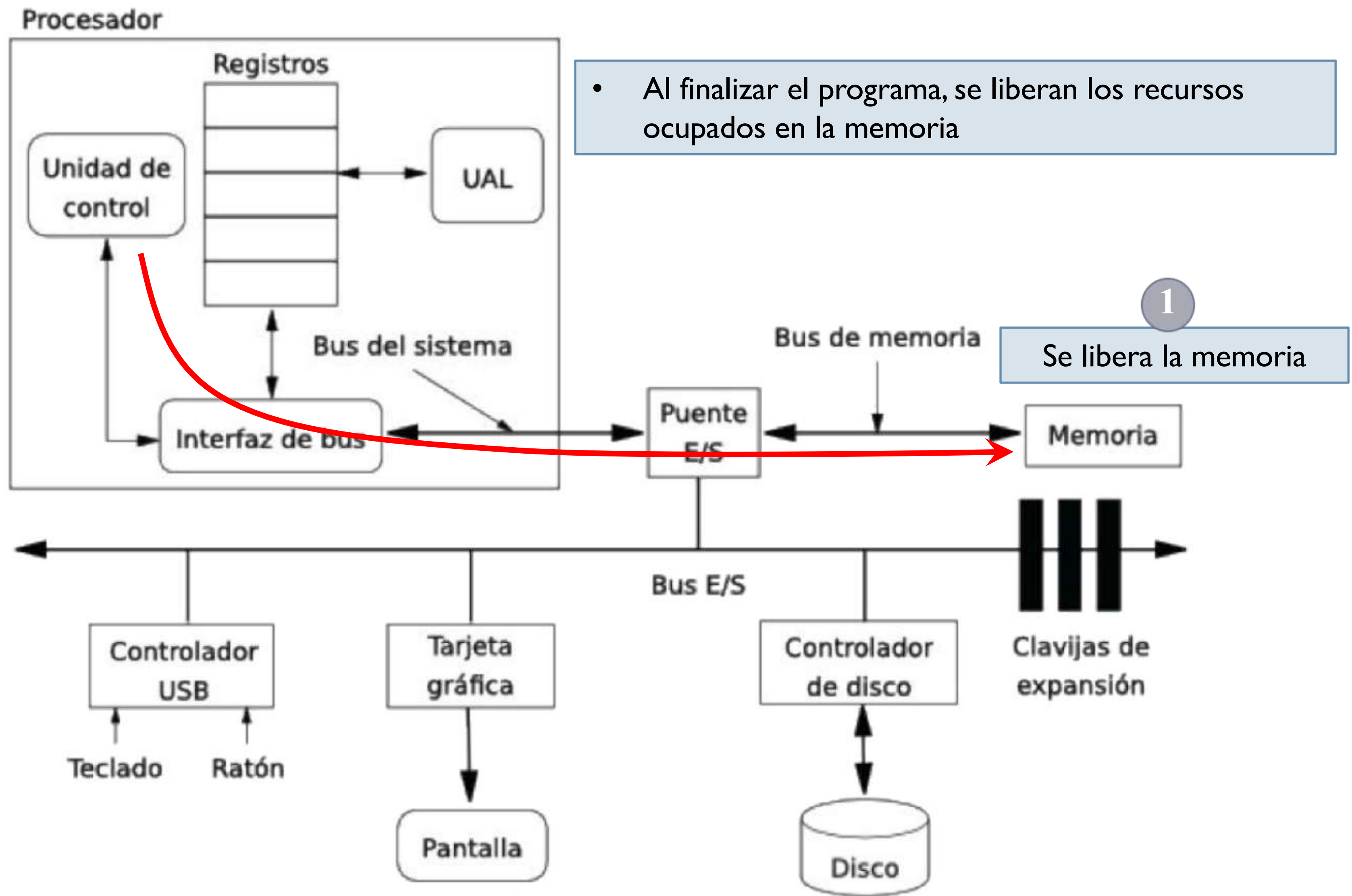
2.1 – Introducción

Ejecución de un programa - Paso 3



2.1 – Introducción

Ejecución de un programa - Paso 4



- Al finalizar el programa, se liberan los recursos ocupados en la memoria

2.1 – Introducción

Alguna noción de rendimiento

- Latency (tiempo de ejecución): tiempo necesario para acabar una tarea
- Throughput (rendimiento): número de tareas por unidad de tiempo
- Conceptos diferentes, a veces contradictorios
  - Se explota el paralelismo para mejorar el throughput

2.1 – Introducción

Alguna noción de rendimiento

- Latency (tiempo de ejecución): tiempo necesario para acabar una tarea
- Throughput (rendimiento): número de tareas por unidad de tiempo
- Conceptos diferentes, a veces contradictorios
  - Se explota el paralelismo para mejorar el throughput

- Ejemplo
- Se quieren mover personas 10 km
  - Coche: capacidad = 5, velocidad = 60 km/h
  - Bus: capacidad = 60, velocidad = 20 km/h

  - Latency → coche = __ min, bus = __ min

2.1 – Introducción
Alguna noción de rendimiento

- Latency (tiempo de ejecución): tiempo necesario para acabar una tarea
- Throughput (rendimiento): número de tareas por unidad de tiempo
- Conceptos diferentes, a veces contradictorios
  - Se explota el paralelismo para mejorar el throughput

- Ejemplo
- Se quieren mover personas 10 km
  - Coche: capacidad = 5, velocidad = 60 km/h
  - Bus: capacidad = 60, velocidad = 20 km/h

  - Latency → coche = 10 min, bus = 30 min
  - Throughput → coche = __ (personas/h), bus = __ (personas/h)

2.1 – Introducción

Alguna noción de rendimiento

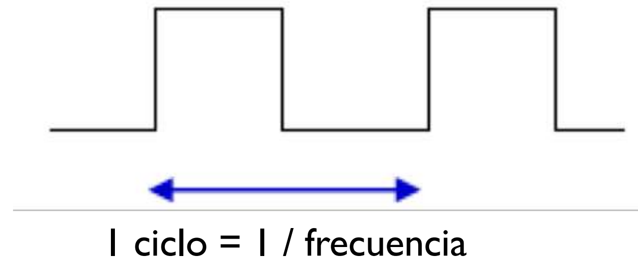
- Latency (tiempo de ejecución): tiempo necesario para acabar una tarea
- Throughput (rendimiento): número de tareas por unidad de tiempo
- Conceptos diferentes, a veces contradictorios
  - Se explota el paralelismo para mejorar el throughput

- Ejemplo
- Se quieren mover personas 10 km
  - Coche: capacidad = 5, velocidad = 60 km/h
  - Bus: capacidad = 60, velocidad = 20 km/h

  - Latency → coche = 10 min, bus = 30 min
  - Throughput → coche = 15 (personas/h), bus = 60 (personas/h)

2.1 – Introducción
Alguna noción de rendimiento

- Frecuencia
  - El reloj marca el "ritmo" con el que se ejecutan instrucciones



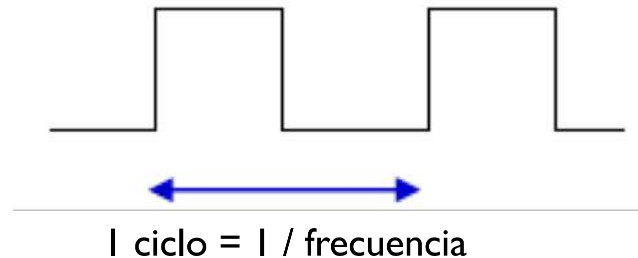
1 ciclo = 1 / frecuencia

  - 1 Hercio (Hz) = 1 ciclo por segundo
  - 1 GHz = 10^9 ciclos por segundo

  - Cuantos segundos para 1 ciclo?

2.1 – Introducción
Alguna noción de rendimiento

- Frecuencia
  - El reloj marca el "ritmo" con el que se ejecutan instrucciones



1 ciclo = 1 / frecuencia

  - 1 Hercio (Hz) = 1 ciclo por segundo
  - 1 GHz = 10^9 ciclos por segundo

  - Cuantos segundos para 1 ciclo?
  - Ciclo = 1 / frecuencia = 1 / 10^9 = 1 ns (1 nanosegundo es 10^{-9} segundos)

2.1 – Introducción
Alguna noción de rendimiento

- Tiempo de CPU
  - Es el tiempo necesario para ejecutar un programa
  - Se determina como

$$\text{Tiempo de CPU} = \frac{\text{Ciclos de reloj de CPU para ejecutar el programa}}{\text{Frecuencia}}$$

2.1 – Introducción

Alguna noción de rendimiento

- CPI (Clock cycles per instruction)
  - Es un valor medio que indica el número medio de ciclos para ejecutar una instrucción
  - Depende de varios factores
    - Diferentes instrucciones tardan diferentes números de ciclos para ejecutarse
    - Una adición puede tardar 1 ciclo y una división 10
    - De la frecuencia con la que se ejecuta cada tipo de instrucción
  - A veces se usa el reciproco IPC (Instruccions per clock cycle)
- Ejemplo
  - Un programa ejecuta un igual numero de operaciones con enteros, reales y de acceso a la memoria
  - CPI por tipo de instrucción
    - Entero = 1
    - Real = 3
    - Acceso memoria = 2
  - CPI = ?

2.1 – Introducción
Alguna noción de rendimiento

- CPI (Clock cycles per instruction)
  - Es un valor medio que indica el número medio de ciclos para ejecutar una instrucción
  - Depende de varios factores
    - Diferentes instrucciones tardan diferentes números de ciclos para ejecutarse
    - Una adición puede tardar 1 ciclo y una división 10
    - De la frecuencia con la que se ejecuta cada tipo de instrucción
  - A veces se usa el reciproco IPC (Instruccions per clock cycle)
- Ejemplo
  - Un programa ejecuta un igual numero de operaciones con enteros, reales y de acceso a la memoria
  - CPI por tipo de instrucción
    - Entero = 1
    - Real = 3
    - Acceso memoria = 2
  - CPI = 1 * 33% + 3 * 33% + 2 * 33% = 2

- CPI (Clock cycles per instruction)
  - Formula genérica para calcular una media pesada

$$\mathrm{CPI}_{\mathrm{average}} = \frac{\left(\sum_{i=1}^{n} \mathrm{CPI}_i \times NI_i \right)}{NI}$$

2.1 – Introducción
Alguna noción de rendimiento

- CPI (Clock cycles per instruction)
  - Ejemplo

Escritura	10	1 ciclo
Lectura	20	5 ciclos
ALU	50	1 ciclo
Salto	20	2 ciclos

CPI = ?

$$CPI_{\text{average}} = \frac{\left(\sum_{i=1}^{n} CPI_i \times NI_i\right)}{NI}$$

2.1 – Introducción

Alguna noción de rendimiento

- CPI (Clock cycles per instruction)
  - Ejemplo

Escritura	10	1 ciclo
Lectura	20	5 ciclos
ALU	50	1 ciclo
Salto	20	2 ciclos

$$CPI_{average} = \frac{\left(\sum_{i=1}^{n} CPI_i \times NI_i\right)}{NI}$$

CPI = (1 * 10 + 5 * 20 + 1 * 50 + 2 * 20) / (10+20+50+20) = 2

2.1 – Introducción

Alguna noción de rendimiento

- CPI (Clock cycles per instruction)
  - Ejemplo

Escritura	10	1 ciclo
Lectura	20	5 ciclos
ALU	50	1 ciclo
Salto	20	2 ciclos

$$CPI_{average} = \frac{\left(\sum_{i=1}^{n} CPI_i \times NI_i\right)}{NI}$$

CPI = (1 * 10 + 5 * 20 + 1 * 50 + 2 * 20) / (10+20+50+20) = 2

Y el IPC = ?

2.1 – Introducción
Alguna noción de rendimiento

- CPI (Clock cycles per instruction)
  - Ejemplo

Escritura	10	1 ciclo
Lectura	20	5 ciclos
ALU	50	1 ciclo
Salto	20	2 ciclos

$$CPI_{average} = \frac{\left(\sum_{i=1}^{n} CPI_i \times NI_i\right)}{NI}$$

CPI = (1 * 10 + 5 * 20 + 1 * 50 + 2 * 20) / (10+20+50+20) = 2

Y el IPC = 1 / CPI = 0,5

2.1 – Introducción
Alguna noción de rendimiento

- Desde la perspectiva de la CPU

  - Latency: segundos / instrucción

    → segundos / instrucción = (ciclos / instrucción) * (segundos / ciclo)

    = CPI / Frecuencia

  - Throughput: instrucciones / segundo

    → instrucciones / segundo = (instrucciones / ciclo) * (ciclos / segundo)

    = Frecuencia / CPI

2.1 – Introducción

Alguna noción de rendimiento

- Desde la perspectiva de la CPU
  - Latency: CPI / Frecuencia
  - Throughput: Frecuencia / CPI

- Ejemplo
  - ¿qué procesador comprarías?
  - Procesador A: CPI = 2, frecuencia = 5 GHz
  - Procesador B: CPI = 1, frecuencia = 3 GHz

2.1 – Introducción

Alguna noción de rendimiento

- Desde la perspectiva de la CPU
  - Latency: CPI / Frecuencia
  - Throughput: Frecuencia / CPI

- Ejemplo
  - ¿qué procesador comprarías?
  - Procesador A: CPI = 2, frecuencia = 5 GHz
  - Procesador B: CPI = 1, frecuencia = 3 GHz

  - $Latency_A = 2 / 5\ 10^9 = 0{,}4\ 10^{-9}$ segundos/instrucción
  - $Latency_B = 1 / 3\ 10^9 = 0{,}33\ 10^{-9}$ segundos/instrucción

2.1 – Introducción
Alguna noción de rendimiento

- Desde la perspectiva de la CPU
  - Latency: CPI / Frecuencia
  - Throughput: Frecuencia / CPI

- Ejemplo
  - ¿qué procesador comprarías?
  - Procesador A: CPI = 2, frecuencia = 5 GHz
  - Procesador B: CPI = 1, frecuencia = 3 GHz

  - $Latency_A = 2 / 5\ 10^9 = 0{,}4\ 10^{-9}$ segundos/instrucción
  - $Latency_B = 1 / 3\ 10^9 = 0{,}33\ 10^{-9}$ segundos/instrucción

  - $Throughtput_A = 5\ 10^9 / 2 = 2{,}5\ 10^9$ instrucciones/segundo
  - $Throughput_B = 3 / 1\ 10^9 = 3\ 10^9$ instrucciones/segundo

2.1 – Introducción
Alguna noción de rendimiento

- Ejecución de un programa
  - 100.000 instrucciones
  - 1 GHz
  - CPI = 2

- Tiempo de ejecución?

2.1 – Introducción
Alguna noción de rendimiento

- Ejecución de un programa
  - 100.000 instrucciones
  - 1 GHz
  - CPI = 2

- Tiempo de ejecución?

(instrucciones/programa) * (ciclos/instrucción) * (segundos/ciclo)

= 100.000 * 2 * (1 / 10^9) = 0,2 ms

2.1 – Introducción
Alguna noción de rendimiento

- MIPS (Millions instructions per second)

$$MIPS = \frac{\text{Número de instrucciones}}{\text{Tiempo de ejecución} \times 10^6}$$

Sabiendo que

Tiempo de ejecución = Número de instrucciones * Latency

= Número de instrucciones * CPI / Frecuencia

$$MIPS = \frac{\text{Frecuencia}}{CPI \times 10^6}$$

2.1 – Introducción
Alguna noción de rendimiento

- MIPS (Millions instructions per second)
  - Depende mucho de cómo se implementan las instrucciones
  - Los MIPS medidos varían entre programas ejecutados en un mismo ordenador
  - Pueden llevar a resultados engañosos donde un procesador pueda tener menos MIPS pero ser más rápido en ejecutar un programa que otro

  - Ejemplo
  - Un programa contiene 200 millones de instrucciones
    - Procesador A (125 MHz) ejecuta el programa en 10 s
    - Procesador B (300 MHz) ejecuta el programa en 5 s

$MIPS_A$ =

$MIPS_B$ =

CPI_A =

CPI_B =

2.1 – Introducción
Alguna noción de rendimiento

- MIPS (Millions instructions per second)
  - Depende mucho de cómo se implementan las instrucciones
  - Los MIPS medidos varían entre programas ejecutados en un mismo ordenador
  - Pueden llevar a resultados engañosos donde un procesador pueda tener menos MIPS pero ser más rápido en ejecutar un programa que otro

  - Ejemplo
  - Un programa contiene 200 millones de instrucciones
    - Procesador A (125 MHz) ejecuta el programa en 10 s
    - Procesador B (300 MHz) ejecuta el programa en 5 s

$MIPS_A$ = Instrucciones / (Tiempo de CPU x 10^6) = (200 x 10^6) / (10 x 10^6) = 20

$MIPS_B$ =

CPI_A =

CPI_B =

2.1 – Introducción

Alguna noción de rendimiento

- MIPS (Millions instructions per second)
  - Depende mucho de cómo se implementan las instrucciones
  - Los MIPS medidos varían entre programas ejecutados en un mismo ordenador
  - Pueden llevar a resultados engañosos donde un procesador pueda tener menos MIPS pero ser más rápido en ejecutar un programa que otro

  - Ejemplo
  - Un programa contiene 200 millones de instrucciones
    - Procesador A (125 MHz) ejecuta el programa en 10 s
    - Procesador B (300 MHz) ejecuta el programa en 5 s

$MIPS_A$ = Instrucciones / (Tiempo de CPU x 10^6) = (200 x 10^6) / (10 x 10^6) = 20

$MIPS_B$ = Instrucciones / (Tiempo de CPU x 10^6) = (200 x 10^6) / (5 x 10^6) = 40

CPI_A =

CPI_B =

2.1 – Introducción

Alguna noción de rendimiento

- MIPS (Millions instructions per second)
  - Depende mucho de cómo se implementan las instrucciones
  - Los MIPS medidos varían entre programas ejecutados en un mismo ordenador
  - Pueden llevar a resultados engañosos donde un procesador pueda tener menos MIPS pero ser más rápido en ejecutar un programa que otro

  - Ejemplo
  - Un programa contiene 200 millones de instrucciones
    - Procesador A (125 MHz) ejecuta el programa en 10 s
    - Procesador B (300 MHz) ejecuta el programa en 5 s

$MIPS_A$ = Instrucciones / (Tiempo de CPU x 10^6) = (200 x 10^6) / (10 x 10^6) = 20

$MIPS_B$ = Instrucciones / (Tiempo de CPU x 10^6) = (200 x 10^6) / (5 x 10^6) = 40

CPI_A = Ciclos de reloj / Instrucciones = 10 x 125 x 10^6 / (200 x 10^6) = 6,25

CPI_B = Ciclos de reloj / Instrucciones = 5 x 300 x 10^6 / (200 x 10^6) = 7,5

2.1 – Introducción
Alguna noción de rendimiento

- MFLOPS (Millions of floating-point operations per second)
  - Más usado porque basado en operaciones y no en instrucciones
  - De todas maneras, las instrucciones en coma flotante varia mucho de una arquitectura a otra
  - También varia mucho en una misma arquitectura según la operación que se quiera hacer

  - Se pueden usar los MFLOPS normalizado que consideran la complejidad de las operaciones
    - Suma, resta, multiplicación, comparación: cuestan poco → 1 operación normalizada
    - División, raíz cuadrada= costosas → 4 operaciones normalizadas
    - Trigonométricas = muy costosas → 8 operaciones normalizadas

2.1 – Introducción

Alguna noción de rendimiento

- MFLOPS (Millions of floating-point operations per second)
  - Ejemplo
  - Un ordenador tarda 94 segundos en ejecutar un programa
  - Este programa contiene
    - 109.970.178 operaciones en coma flotante, de estas
    - 15.682.333 son divisiones
    - El resto son operaciones poco costosas

$$MFLOPS = 109.970.178 / (94 \times 10^6) = 1,2$$

$$MFLOPS_{norm} = ((109.970.178 - 15.682.333) + 15.682.333 \times 4) / (94 \times 10^6) = 1,7$$

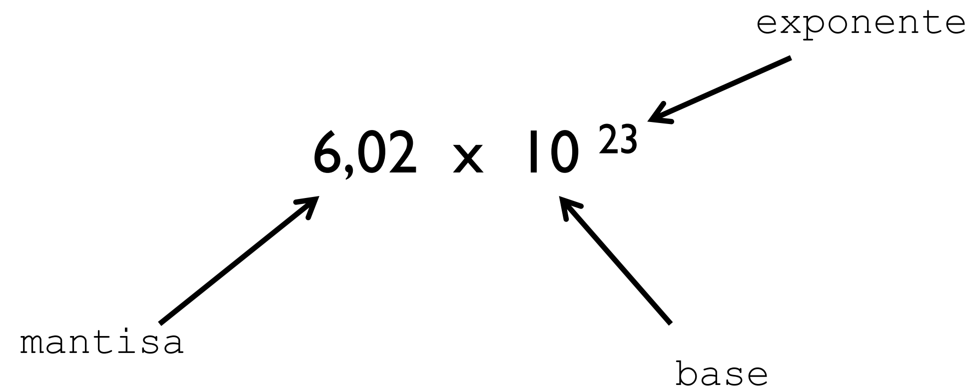
2.1 – Introducción
Floating point?

- Los ordenadores tratan datos en binario
- Estos datos ocupan un número de bits limitado a N (32, 64, 128 bits)
- Que se puede representar en N bits
  - Enteros con y sin signo
    - de 0 a $2^{N}-1$
    - de -2^{N-1} a $2^{N-1}-1$

  - Y que paso con los demás números
    - Muy grandes como 9.349.398.989.787.762.244.859.087.678
    - Muy pequeños como 0.00000000000000000000045691
    - Racionales como 2/3
    - Iracionales como $\sqrt{2}$
    - Transcendentes como e

2.1 – Introducción
Floating point?

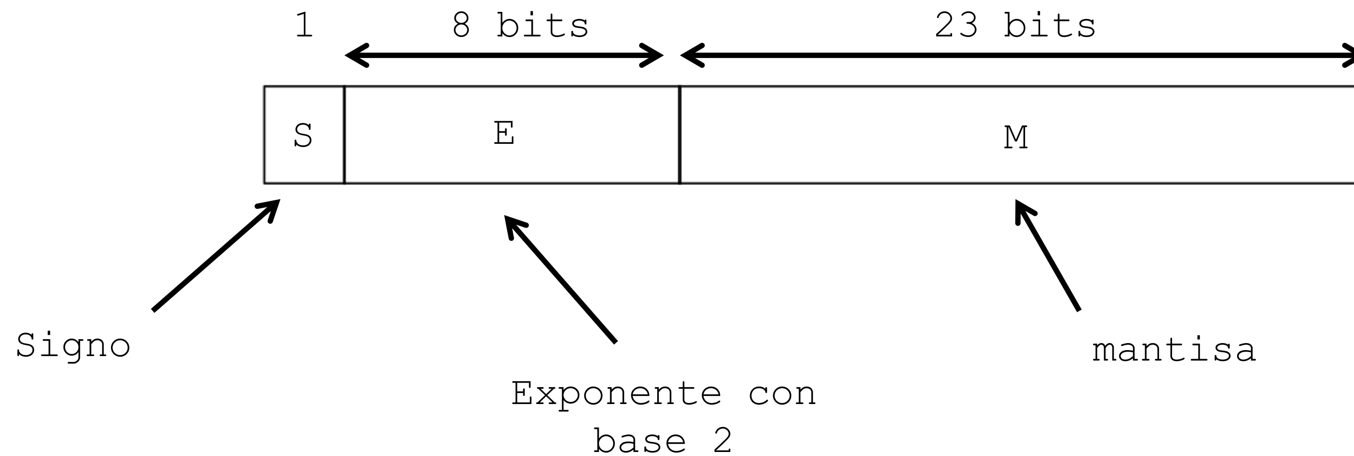
- Se usa la notación científica



2.1 – Introducción
Floating point?

- Por ejemplo la representación con 32 bits según el estándar IEEE 754 es

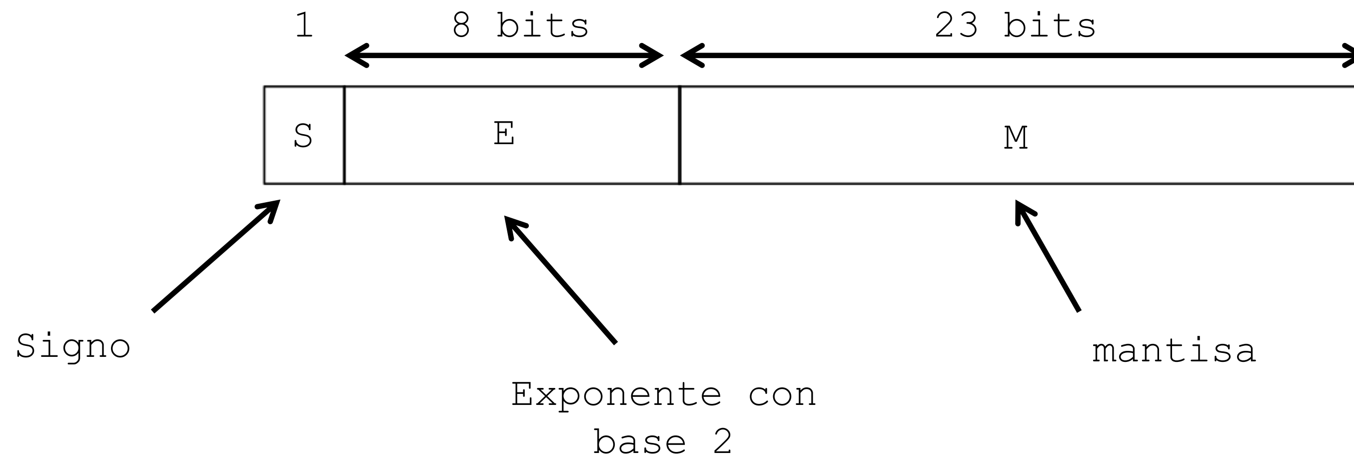
$$(-1)^S \times 1.M \times 2^{E-127}$$



2.1 – Introducción
Floating point?

- Por ejemplo la representación con 32 bits según el estándar IEEE 754 es

$$(-1)^S \times 1.M \times 2^{E-127}$$

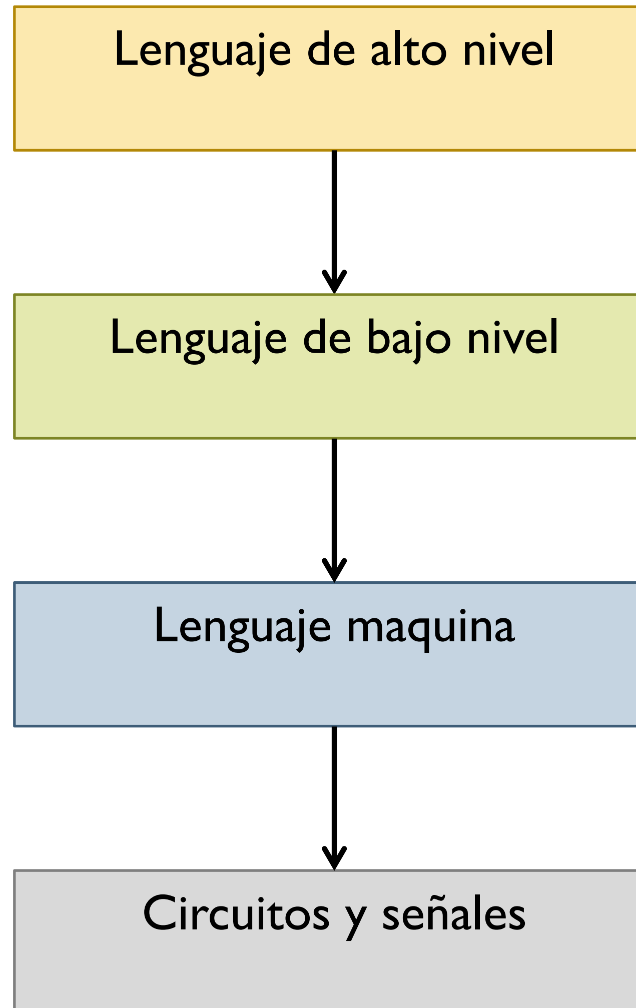


- Número más pequeño: $1.2\ 10^{-38}$
- Número más grande: $3.4\ 10^{38}$

Temario

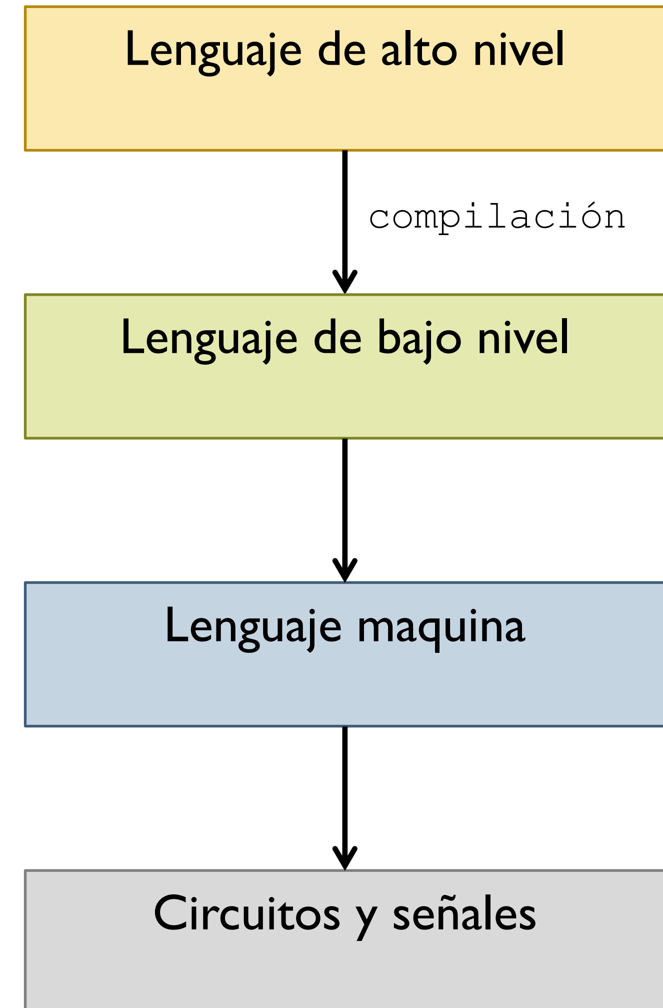
- Tema 1. Introducción
- Tema 2. El microprocesador
  - Introducción
  - Instruction Set Architecture
  - Arquitectura interna
  - Paralelismo
  - Ejercicios
- Tema 3. Memoria
- Tema 4. Dispositivos de E/S y buses
- Tema 5. DataCenters y modelos de comunicación

2.2 – Instruction Set Architecture
Niveles de los lenguajes



2.2 – Instruction Set Architecture
Compilación

- Un programa escrito en un lenguaje de alto nivel
  - c, c++, java, c#, python, etc.
  - Control estructurado y jerárquico: bucles, funciones, condiciones
  - Datos estructurados y jerárquicos: escalares, vectores, punteros, estructuras
  - Independiente del entorno



- Compilador
  - Traduce el programa en lenguaje de bajo nivel (assembly)
  - Análisis y traducción directa del lenguaje de alto nivel
  - Un compilador también suele optimizar el código
  - El compilador es a su vez un programa
    - ¿Quien compila el compilador?

2.2 – Instruction Set Architecture
Bootstrapping

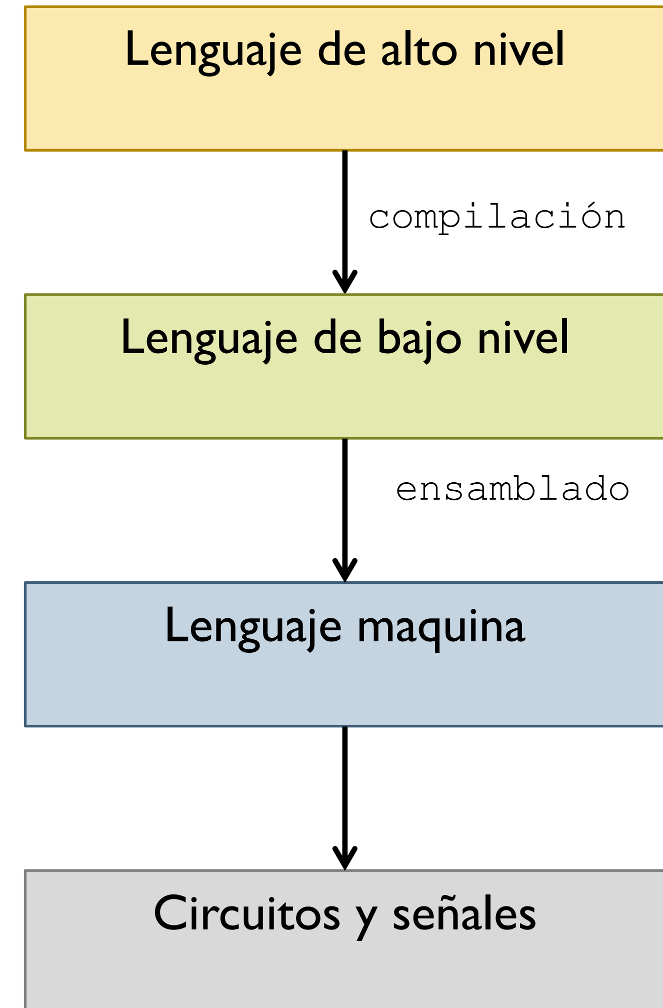
- Método con el cual se crea el compilador de un lenguaje usando el mismo lenguaje u otro ya existente
- Es decir, para tener un compilador en lenguaje X
  - Escribiendo el compilador en un lenguaje Y ya existente
    p.e. el primer compilador Pascal (1968) fue escrito en Fortran
  - Escribiendo directamente el compilador en lenguaje de bajo nivel
    p.e. Fortran (1961)
  - Escribiendo un interprete del lenguaje X de forma que cada instrucción se traduzca al lenguaje de bajo nivel paso a paso (compilación incremental)
    p.e. Lisp (1962)

2.2 – Instruction Set Architecture
Ensamblador

- Lenguaje de bajo nivel
  - Assembly
  - Representación entendible para las personas
  - Depende del entorno (OS y arquitectura)
  - Definido como Instruction Set Architecture (ISA)

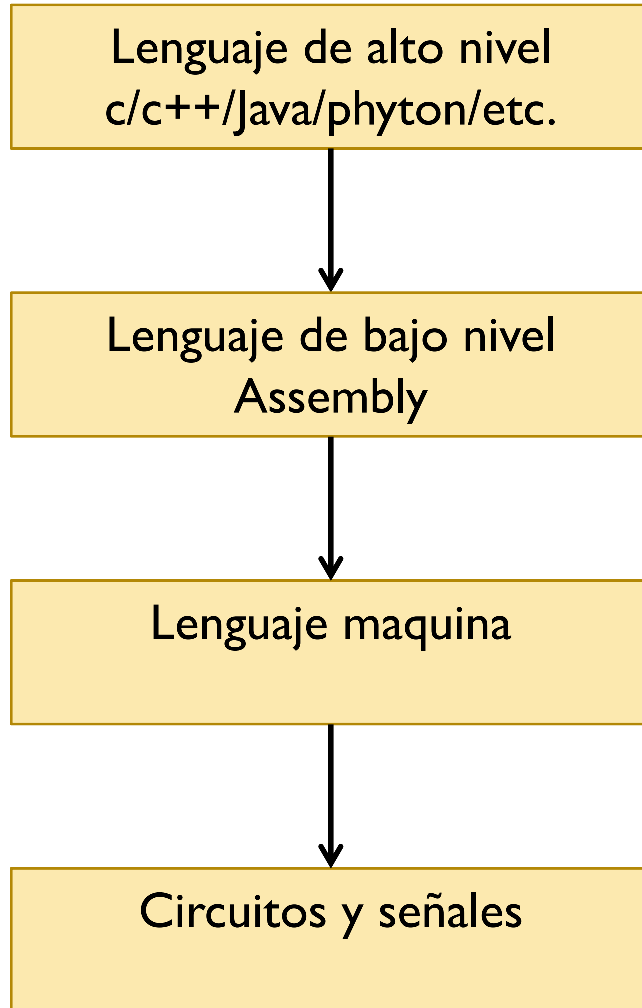
- Lenguaje maquina
  - Representación entendible para la maquina
  - Secuencia de 0s y 1s

- Assembler
  - Traductor del assembly al lenguaje maquina



2.2 – Instruction Set Architecture

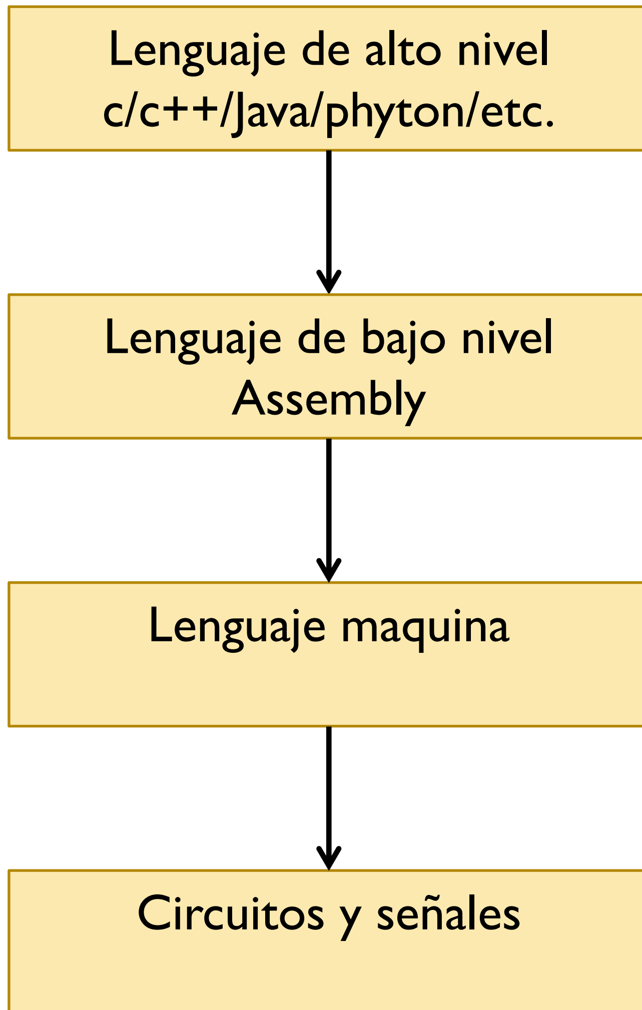
Ejemplo




```
int array[100];
int sum;
void array_sum() {
    for (int i=0; i<100;i++) {
        sum += array[i];
    }
}
```

2.2 – Instruction Set Architecture

Ejemplo

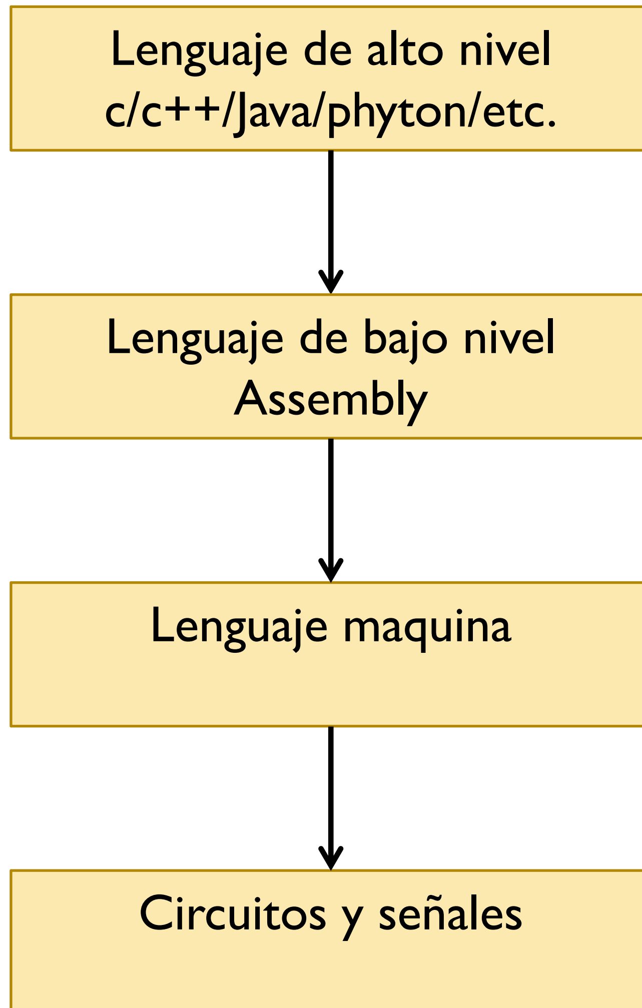



```
array_sum():
        movl    sum(%rip), %edx
        movl    $array, %eax
.L2:
        addl    (%rax), %edx
        addq    $4, %rax
        cmpq    $array+400, %rax
        jne     .L2
        movl    %edx, sum(%rip)
        ret
sum:
        .zero   4
array:
        .zero   400
```

http://gcc.godbolt.org

2.2 – Instruction Set Architecture

Ejemplo




```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
1100 1101 1100 0001 0011 0001 0000 1100
0110 0101 0001 0010 0010 0000 1100 1101
1001 1000 0011 0110 0101 1010 1001 1111
0001 0101 0111 1101 0100 0110 0101 0011
```

2.2 – Instruction Set Architecture

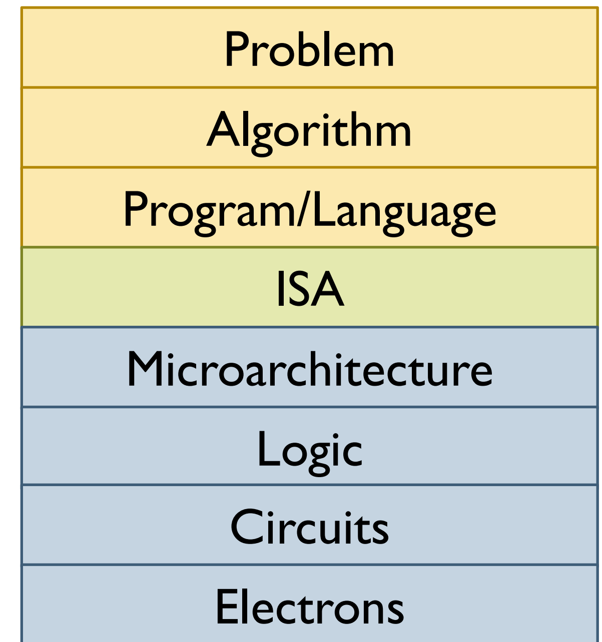
ISA vs. Microarquitectura

- Instruction Set Architecture
  - Especifica como el programador ve las instrucciones
  - Las instrucciones se ejecutan de forma secuencias
  - Se ejecuta la instrucción apuntada por el Instruction Pointer (or Counter)
  - La mayoría de las ISA usan este modelo: x86, ARM, MIPS, Sparc, Alpha, etc.

- Microarquitectura
  - Como realmente los componentes ejecutan las instrucciones
  - La ejecución de estas instrucciones varia mucho entre modelos
    - Segmentación (Pipelining): Intel 80486
    - Múltiples instrucciones al mismo tiempo: Intel Pentium
    - Ejecución fuera de orden: Intel Pentium Pro
  - Son transparentes al programador

2.2 – Instruction Set Architecture
ISA vs. Microarquitectura

- Instruction Set Architecture
  - Es la interfaz entre software y hardware
  - Es lo que el software asume y lo que el hardware proporciona
- Microarquitectura
  - Implementación específica de una ISA
  - No visible desde el software
- Arquitectura
  - ISA + microarquitectura
- Similitud con el coche
  - Interfaz usuario: pedal del acelerador (similar en todos los coche)
  - Implementación: como efectivamente el acelerador comanda el coche para acelerar es propio de cada fabricante y modelo



2.2 – Instruction Set Architecture

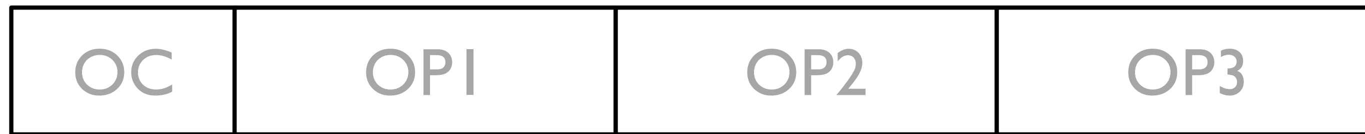
Tipos de instrucciones

- Movimiento de datos
  - Load: origen es la memoria y el destino es un registro
  - Store: origen es un registro y el destino es la memoria
  - Otros (CISC): entre memoria y memoria, entre un dato y memoria, y un dato y un registro (no todas las ISA)
- Operaciones aritméticas o lógicas
  - Add, sub, shift, etc.: se ejecutan en la ALU
- Saltos
  - Br, Loc, Brz, Jge: saltos condicionales o incondicionales

2.2 – Instruction Set Architecture

Tipos de instrucciones

- Formato instrucción
  - Opcode: el código de la operación a ejecutarse
  - Operands: el dato y la dirección que hay que usar en la operación



- Operación
  - add r0, r1, r2
- Operandos
  - add r0, r1, r2
- Donde poner el resultado
  - add r0, r1, r2
- Próxima operación (Instruction Pointer, IP)
  - Pasa a la siguiente operación en la memoria excepto si hay un salto (branch)
  - br endloop

2.2 – Instruction Set Architecture
Algunos ejemplos

- Instrucciones de movimiento

mov A, B	mueve 16 bits de la dirección A de memoria a la de B	VAX11
lwz R3, A	mueve 32 bits de la dirección A de memoria al registro R3	PPC601
li $3, 455	carga el número 455 en el registro 3	MIPS
mov R4, dout	mueve 16 bits del registro R4 al puerto de salida dout	DEC PDP11
IN, AL, KBD	Carga un byte del puerto de entrada KBD en el acumulador	Pentium
LFA.L (A0), A2	Carga la dirección apuntada por A0 en A2	M68000

2.2 – Instruction Set Architecture

Algunos ejemplos

- Operaciones

mulf A, B, C	multiplica los valores en coma flotante en las direcciones A y B y guarda el resultado en C	VAX11
nabs r3, r1	guarda el valor absoluto de r1 en r3	PPC601
ori $2, $1, 255	operación de OR lógico entre el registro $1 y 255 y guarda el resultado en el registro $2	MIPS
DEC R2	decrementa el valor de 16 bits guardado en el registro R2	DEC PDP11
SHL AX, 4	desplaza el valor de 16 bits guardado en el registro AX de 4 bits	Intel 8086

2.2 – Instruction Set Architecture
Algunos ejemplos

- Como se opera con matrices?

2.2 – Instruction Set Architecture
Algunos ejemplos

- Por defecto la siguiente instrucción apuntada por IP

  IP = IP + sizeof(instrucción actual)

- Excepto si la instrucción actual es un salto

BLSS A, Tgt	salta a la dirección Tgt si el bit menos significativo del valor guardado en la dirección A de memoria es 1	VAX11
bun r2	salta a la dirección R2 si el resultado de la anterior operación en coma flotante no ha sido un número valido	PPC601
beq $2, $1, 32	salta a la dirección (IP + 4 + 32) si el contenido de $1 y $2 son iguales	MIPS
SOB R4, Loop	decrementa R4 y salta a Loop si R4 es diferente de 0	DEC PDP11
JCXZ Addr	salta a Addr si el contenido del registro CX es 0	Intel 8086

2.2 – Instruction Set Architecture
Ejercicio

- Se ejecuta este programa
- En un Intel i3 a 1,80 GHz (x86-64, gcc 9.2)
- Se quiere determinar el tiempo de ejecución
- Notas:

```
int sum = 0;
void array_sum() {
   for (int i=0; i<10;i++)
      sum += i;
}
```

http://gcc.godbolt.org

```
edx, eax, rax son registros
rsp y rbp son registros usados para una zona de memoria llamada stack
•  push: mueve valor registro al stack
•  pop: coge primer valor del stack y cópialo en el registro
DWORD PTR [x] es un acceso a la memoria que contiene la variable x
jg y jmp son saltos (jg -> más grande que, jmp -> salto incondicional)
```

```
mov (move, copy)
•  registro -> registro    1 ciclo
•  memoria -> registro     3 ciclos
•  registro -> memoria     2 ciclos
•  valor -> memoria        2 ciclos
cmp (compare)
•  registro, valor         1 ciclo
•  memoria, valor          2 ciclos
jump (salto)               2 ciclos
```

```
add
•  registro, registro      1 ciclo
•  memoria, registro       3 ciclos
•  registro, memoria       2 ciclos
•  valor, registro         1 ciclo
•  valor, memoria          3 ciclos
push/pop                   1 ciclo
ret (return)               1 ciclo
nop (no operation)         0 ciclos
```

2.2 – Instruction Set Architecture

Ejercicio

Intel® 64 and IA-32 Architectures Optimization Reference Manual

2.2 – Instruction Set Architecture
Recordatorio

- El tiempo de ejecución de un programa es producto de tres términos:

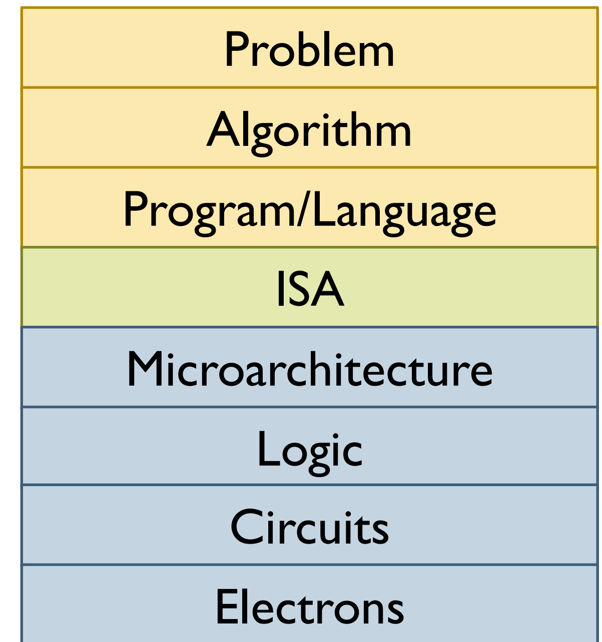
$$Te = \frac{\text{Ciclos de reloj de CPU para ejecutar el programa}}{\text{Frecuencia}}$$

o también

$$Te = \text{Instrucciones totales} \times \text{CPI} / \text{Frecuencia}$$

2.2 – Instruction Set Architecture
ISA vs. Microarquitectura

- Microarquitectura cambia y evoluciona mucho mas rápidamente que una ISA
  - Pocas ISA (x86, ARM, Alpha, Sparc, PowerPC, MIPS)
  - ¿Por qué?



2.2 – Instruction Set Architecture
Compatibilidad ISA

- Para casi todas las ISA se mantiene la compatibilidad
  - IBM 360/370 (la primera familia de ISA)
  - Otro ejemplo: Intel x86
- Retro compatibilidad
  - Nuevo procesadores soportan programas antiguos
  - Difícil cuando se quiere eliminar una funcionalidad
  - Hay que emular esta funcionalidad en software/OS
- Compatibilidad hacía arriba
  - Viejos procesadores soportan programas nuevos
  - Añadir un CPU ID para que el programa sepa adaptarse al procesador
  - Emular las nuevas instrucciones con alguna actualización de firmware/OS

2.2 – Instruction Set Architecture
Compatibilidad ISA

- **Binary-translation:** Trasforma el código del programa en ISA X en otro código en ISA Y
  - Traducción estática: traduce todo el programa ejecutable de un ISA a otro
    - p.e. juego StarCraft de x86 a ARM (2014)
    - juego Super Mario Bros de NES a x86 (2013)
    - todos los ejecutables de GameBoy a lenguaje c (2004)
  - Traducción dinámica (Just In Time (JIT) compilation): traducción de bloques de un ejecutable de un ISA a otro durante su ejecución
    - 1994 Apple pasó de Motorola 680x0 a IBM PowerPC, para mantener la compatibilidad de sus programas creó un traductor dinámico (Mac 68K emulator)
    - 2006 Apple pasó de IBM PowerPC a Intel x86 y creó Rosetta
    - 2003 Intel crea el IA-32 Execution Layer para ejecutar aplicaciones x86 en el Itanium
    - Típicamente se obtiene un decremento del rendimiento

2.2 – Instruction Set Architecture
Compatibilidad ISA

- **Virtual ISA:** Se virtualiza la ISA real con otra virtual
  - Mas para portabilidad ya que la ISA virtual se podría "instalar" en cualquier equipo y sería siempre idéntica independientemente del procesador
    - Nvidia PTX: el compilador nvcc convierte el lenguaje CUDA (similar a c) en PTX (un lenguaje pseudo-assembly). El controlador grafico compila luego el PTX en lenguaje maquina para su ejecución
    - Java Bytecodes: los programas en Java se ejecutan en una maquina virtual Java que usa una ISA virtual.

2.2 – Instruction Set Architecture

Compatibilidad ISA

- "**Last resort**"
- Meter en la placa un procesador adicional para que se ejecute la ISA de aquel procesador
- Como hacía la PlayStation 2 para ejecutar juegos de la PlayStation 1?

    procesador PlayStation 2: MIPS R5900 a 299 MHz

    + MIPS R3000A a 33.8688 MHz (procesador Play 1)

    + traducción dínamica de la ISA

2.2 – Instruction Set Architecture
CISC ISA

- CISC (Complex Instruction Set Computing)
  - Repertorio de instrucciones elevado. El procesador dispone de muchas instrucciones lo que facilita el trabajo de los programadores.
  - El juego de instrucciones proporcionadas por el hardware es muy grande y el código máquina se asemeja a los lenguajes de alto nivel.
    - Compiladores simples
    - Programas generados son muy compactos. Ocupan poca memoria.
  - El procesador incluye más funciones potentes y especializadas, más registros de propósito general, más circuitería.
  - Menos espacio de memoria necesario a causa de la reducción del número de instrucciones necesarias. Ventaja si memorias lentas.
  - La tendencia CISC se rompe cuando aparecen las memorias cachés y es más rápida la obtención de una instrucción. También cuando las nuevas instrucciones incorporan funcionalidades demasiado rebuscadas que no son aprovechadas por el compilador, sino que complican el procesador y lo hacen más lento y costoso.

2.2 – Instruction Set Architecture
RISC ISA

- RISC (Reduced Instruction Set Computing)
  - Repertorio de instrucciones más limitado. Se eliminan las instrucciones más complejas.
    - Mayor complejidad de programación (Combinación de instrucciones simples).
  - El juego de instrucciones proporcionadas por el hardware es simple. Todas las instrucciones son (casi) del mismo número de ciclos de reloj.
  - Los compiladores son más sofisticados.
  - Los programas generados son complejos y mucho más largos. Ocupan mucha memoria.
  - Sólo son capaces de ejecutar instrucciones simples. Chips más simples y baratos.
  - Pueden ejecutar a frecuencias más altas de reloj y mayor velocidad de ejecución.
  - El rendimiento depende de la eficiencia del compilador. Tiempo de desarrollo de software más elevado que en CISC.

2.2 – Instruction Set Architecture
CISC vs RISC

- RISC (Reduced Instruction Set Computing)
  - Termino creado en los 80s por Patterson
  - Instrucciones simples
  - Menor CPI
  - Mayor instrucciones/programa
  - Puede que mayor frecuencia (microarquitectura más rápida ya que instrucciones simples)
  - Ejemplos: PowerPC, ARM Sparc, Alpha
- CISC (Complex Instruction Set Computing)
  - No existía este termino antes de crear RISC
  - Menor número de instrucciones/programa
  - Mayor número de CPI y menor frecuencia
  - Más fácil la programación a bajo nivel
  - Ejemplos: x86, VAX, Motorola 68000
- Recordar: Tiempo de ejecución = instrucciones/programa * CPI / Frecuencia

2.2 – Instruction Set Architecture

CISC vs RISC - ejemplo

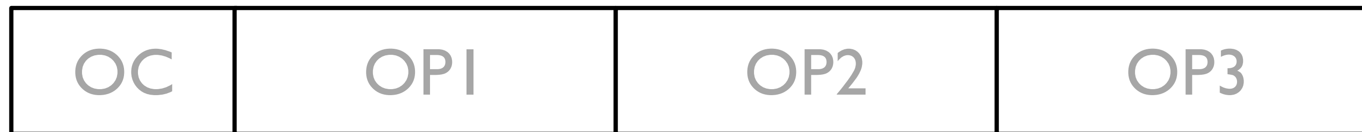
- RISC
  - Las instrucciones de acceso a la memoria se limitan a cargar y guardar
  - Operaciones aritméticas o lógicas se hacen a parte
- CISC
  - Se pueden hacer operaciones directamente en un registro o en memoria
  - Ejemplo x86:

    ```
    addl 100, 4(%eax)
    ```

    1) Carga el valor de la dirección [4 + %eax] en el acumulador
    2) Añade 100 a este valor
    3) Guarda el resultado en la misma dirección
  - En RISC se necesitan las 4 instrucciones por separado

2.2 – Instruction Set Architecture
CISC vs RISC

- Recordatorio
- Formato instrucción
  - Opcode: el código de la operación a ejecutarse
  - Operands: el dato y la dirección que hay que usar en la operación



2.2 – Instruction Set Architecture
CISC vs RISC

- CISC VAX: 1977
  - Instrucciones de longitud variable: 1-321 bytes
  - 14 registros de uso general, PC, stack, condición
  - Tamaño de los datos: 8, 16, 32, 64, 128 bit
  - Centenares de instrucciones especiales

- CISC Intel x86: 1974
  - "Difficult to explain and impossible to love"
  - Instrucciones de longitud variable: 1-16 bytes
  - 8 registros de uso general, condición, accumulator (enteros), stack
  - Tamaño de los datos: 8, 16, 32, 64 bit
  - Instrucciones especiales para multimedia (single instruction, multiple data),

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 *ib*	ADD AL, *imm8*	I	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	I	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	I	Valid	Valid	Add *imm32* to EAX.
REX.W + 05 *id*	ADD RAX, *imm32*	I	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to RAX.
80 /0 *ib*	ADD *r/m8, imm8*	MI	Valid	Valid	Add *imm8* to *r/m8.*
REX + 80 /0 *ib*	ADD *r/m8**, *imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*
81 /0 *iw*	ADD *r/m16, imm16*	MI	Valid	Valid	Add *imm16* to *r/m16.*
81 /0 *id*	ADD *r/m32, imm32*	MI	Valid	Valid	Add *imm32* to *r/m32.*
REX.W + 81 /0 *id*	ADD *r/m64, imm32*	MI	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to *r/m64.*
83 /0 *ib*	ADD *r/m16, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m16.*
83 /0 *ib*	ADD *r/m32, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m32.*
REX.W + 83 /0 *ib*	ADD *r/m64, imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*
00 */r*	ADD *r/m8, r8*	MR	Valid	Valid	Add *r8* to *r/m8.*
REX + 00 */r*	ADD *r/m8**, *r8**	MR	Valid	N.E.	Add *r8* to *r/m8.*
01 */r*	ADD *r/m16, r16*	MR	Valid	Valid	Add *r16* to *r/m16.*
01 */r*	ADD *r/m32, r32*	MR	Valid	Valid	Add r32 to *r/m32.*
REX.W + 01 */r*	ADD *r/m64, r64*	MR	Valid	N.E.	Add r64 to *r/m64.*
02 */r*	ADD *r8, r/m8*	RM	Valid	Valid	Add *r/m8* to *r8.*
REX + 02 */r*	ADD *r8**, *r/m8**	RM	Valid	N.E.	Add *r/m8* to *r8.*
03 */r*	ADD *r16, r/m16*	RM	Valid	Valid	Add *r/m16* to *r16.*
03 */r*	ADD *r32, r/m32*	RM	Valid	Valid	Add *r/m32* to *r32.*
REX.W + 03 */r*	ADD *r64, r/m64*	RM	Valid	N.E.	Add *r/m64* to *r64.*

NOTES:

*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

2.2 – Instruction Set Architecture
CISC vs RISC

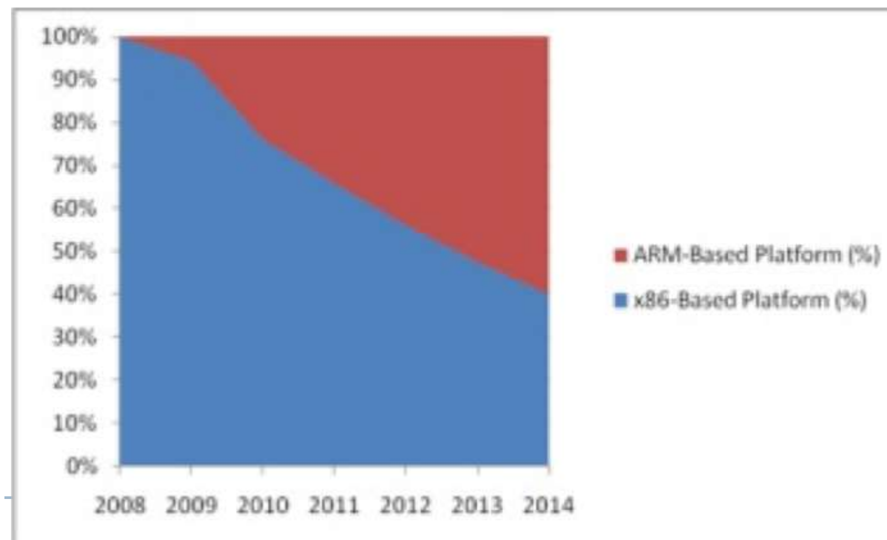
- RISC tienen ISA muy similar: MIPS, Sparc, ARM, PowerPC
  - Instrucciones de longitud fija: 32 bits
  - 32 registros de enteros, 32 registros para coma flotante
  - Pocos modos para direccionar lectura y escritura en memoria
  - Ejecución de una instrucción en 1 ciclo

2.2 – Instruction Set Architecture
CISC vs RISC

- Desktop/Servers: x86
  - x86 fue la primera arquitectura a 16-bit (2 años antes que el resto)
  - IBM puso x86 en sus PCs…
  - El resto es inercia, ley de Moore, y retorno económico
    - x86 tiene un ISA complejo y rápido pero …
    - Ya que Intel es el que vende más…
    - Tiene más dinero y puede contratar los mejores ingenieros…
    - Que le permite mantener un rendimiento competitivo…
    - Y si el rendimiento es competitivo, la compatibilidad gana…
    - Por lo tanto x86 es el ganador en entorno sobremesa/servidores
  - AMD ha añadido presión a Intel y ganó la batalla para la primera versión a 64-bit x86

2.2 – Instruction Set Architecture
CISC vs RISC

- Embedded: ARM
  - Acorn RISC Machine → Advanced RISC Machine (ARM)
  - Primer chip ARM: mitad de los 80s
  - Más unidades vendidas por año que Intel
  - Para sistema low-power, embedido y móviles
  - ARM no vende chips, vende licencia de su ISA y microarquitectura
  - Muchos fabricantes venden chips ARM
    - Apple, Freescale, Philips, Qualcomm, STM, Samsung, Sharp Texas Instruments, etc.



2.2 – Instruction Set Architecture

Intel x86 modernos: RISC inside

- En el 1993, Intel quiso implementar la ejecución fuera de orden en sus Pentium Pro
  - Demasiado complicado hacerlo en su ISA CISC
  - Inviable cambiar ISA por compatibilidad
- Solución?
- Traduce su ISA CISC en RISC a nivel de microoperaciones en hardware
  - De cara al programador sigue siendo la misma ISA x86
  - Pero internamente se implementan las instrucciones como si fueran RISC
  - La manera con la cual se traduce la ISA CISC a nivel de microoperaciones tipo RISC es propietario y se dezconoce

2.2 – Instruction Set Architecture

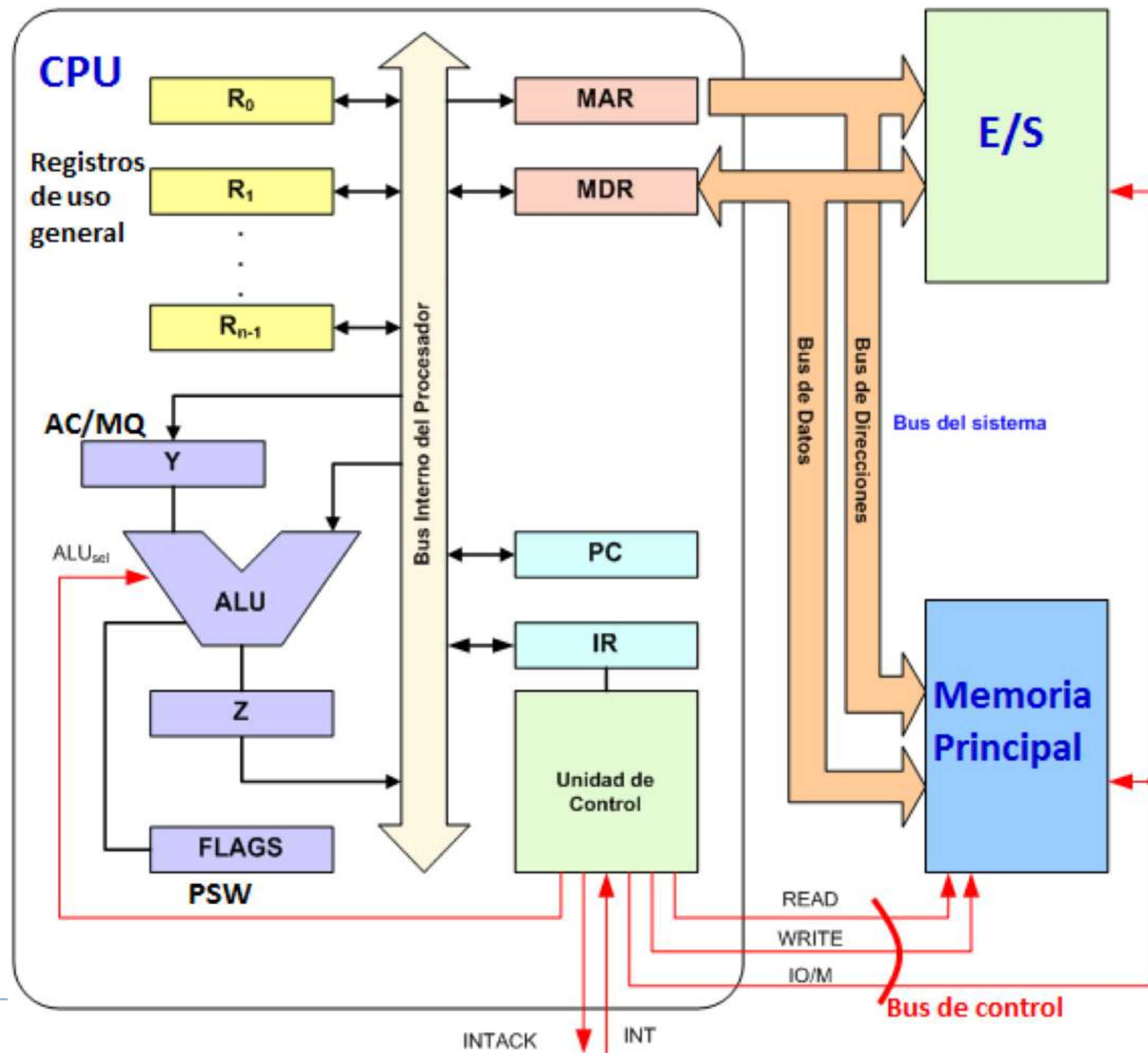
Una paréntesis

- Cuando se habla de procesador a N bits (32 o 64)
  - Significa que cada programa puede direccionar (usar) hasta 2^N bytes
  - Alternativo (erróneo): tamaño de las operaciones aritméticas
- Evolución x86
  - 4 bits 4004
  - 8 bits 8008
  - 16 bits 8086
  - 24 bits 80286
  - 32 bits 80386
  - 64 bits AMD Opteron e Intel Pentium 4

Temario

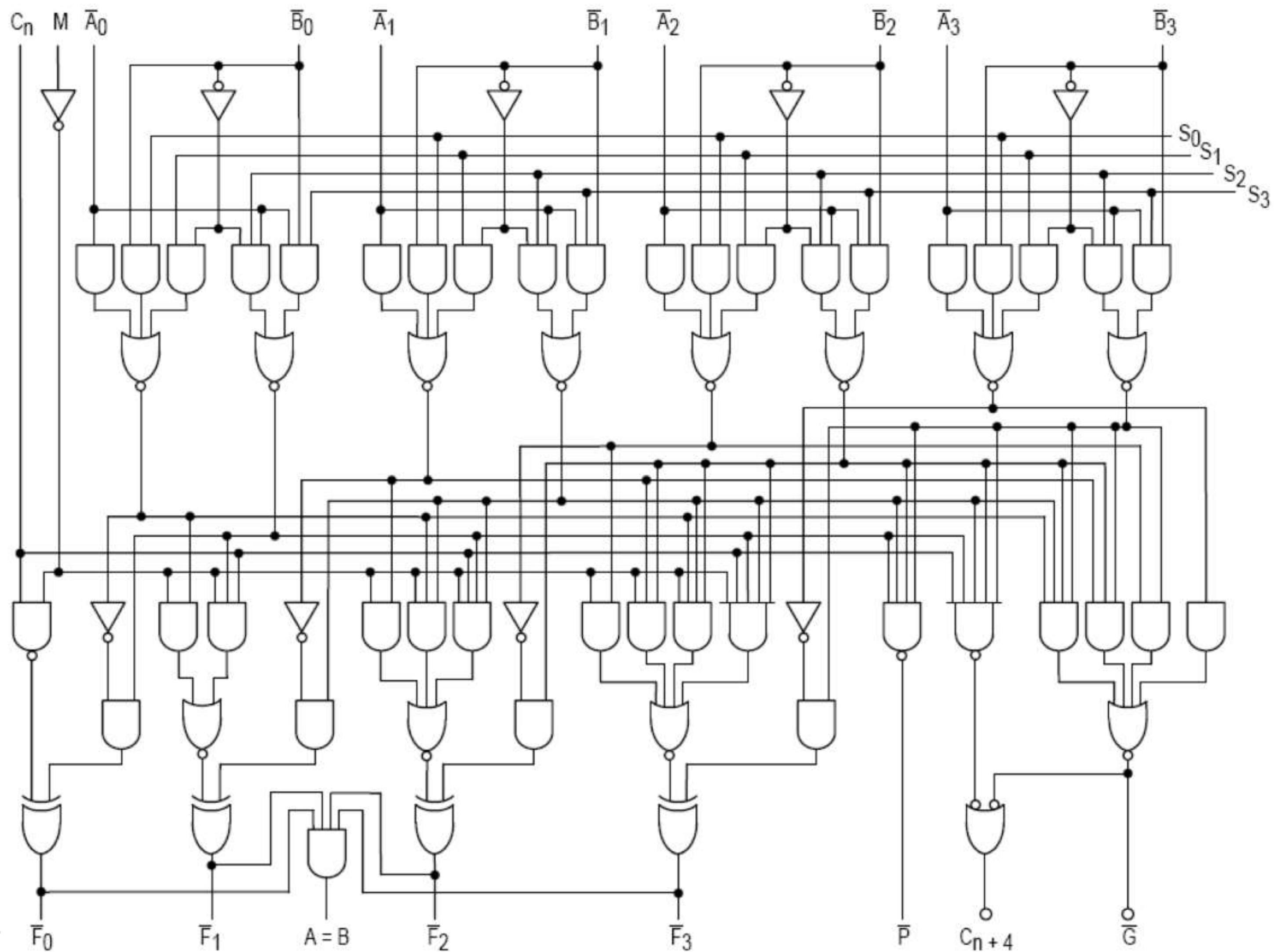
- Tema 1. Introducción
- Tema 2. El microprocesador
  - Introducción
  - Instruction Set Architecture
  - Arquitectura interna
  - Paralelismo
  - Ejercicios
- Tema 3. Memoria
- Tema 4. Dispositivos de E/S y buses
- Tema 5. DataCenters y modelos de comunicación

2.3 - Arquitectura interna



2.3 - Arquitectura interna

- Simple ALU de 4 bits para operaciones con números enteros



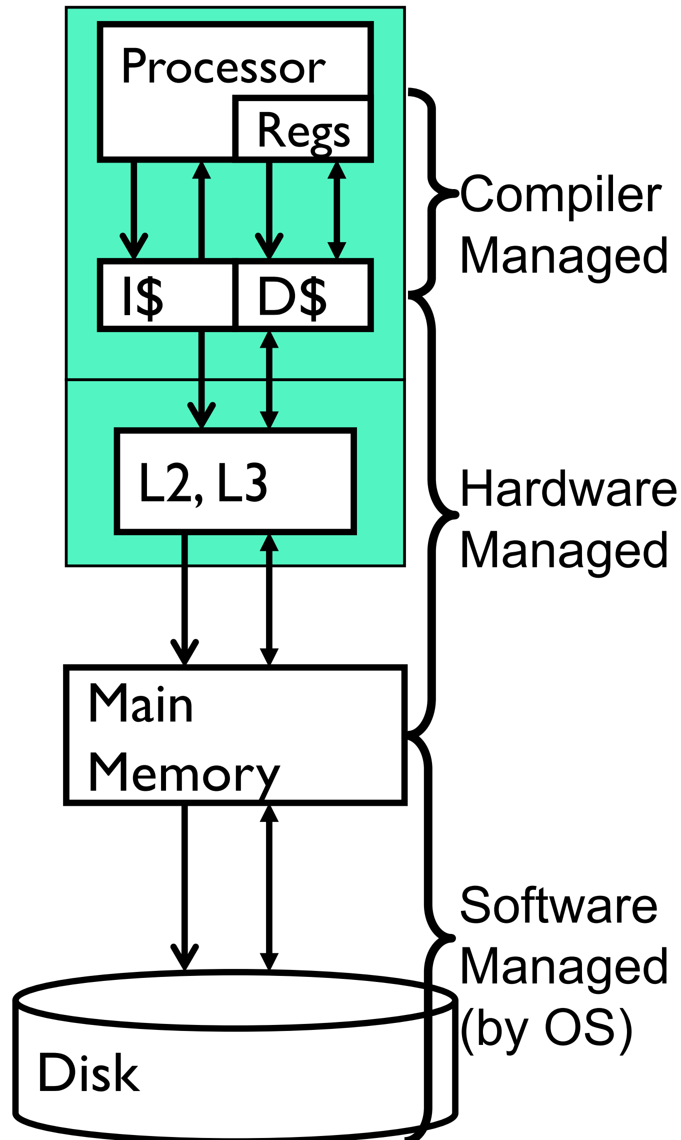
2.3 - Arquitectura interna

- Las operaciones de un procesador dependen mucho de la memoria
- Un procesador a 3GHz puede ejecutar una suma en 0.33 ns
- Una memoria convencional tiene un tiempo de lectura/escritura de 30 ns
- 100 veces más lento que un procesador

FSB: Front Side Bus

- Por eso se ha definido una jerarquía de memoria (veremos detalles en Tema 3)
  - Según la necesidad del procesador, se usa un nivel u otro

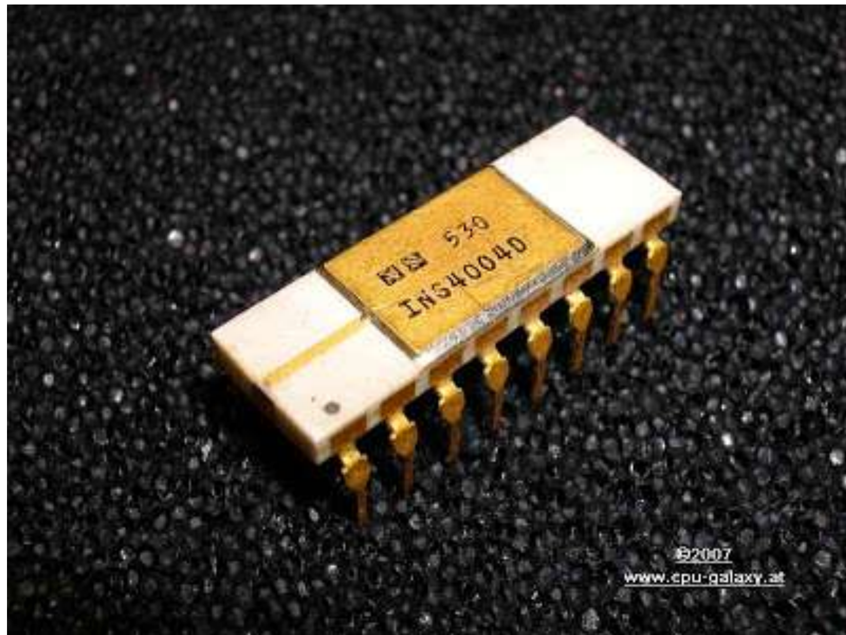
2.3 - Arquitectura interna

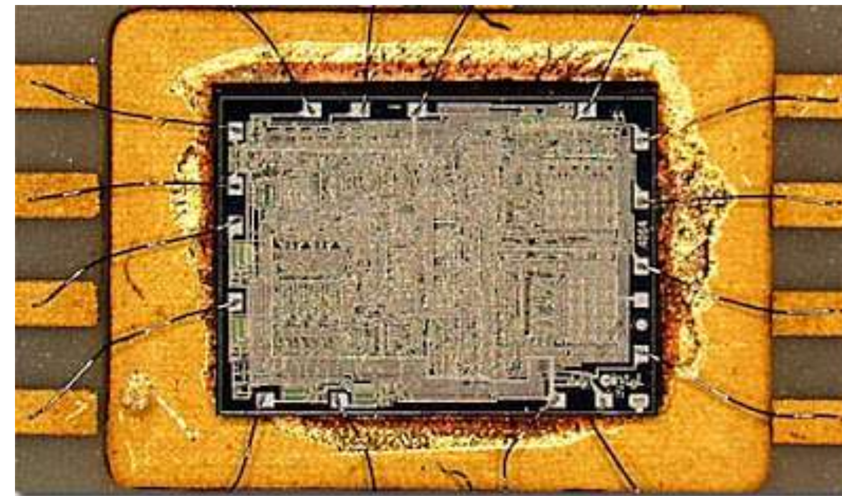


- L0: registros
- L1: Cache de nivel 1 (o primaria)
  - Integrada en el propio procesador
  - Funciona a la misma velocidad del procesador
  - Pocos kbytes
- L2 y L3: Cache de nivel 2 y 3
  - Inicialmente conectadas al procesador a través de un bus trasero
  - Actualmente integradas en el chip del procesador
  - Más lentas que la L1
  - L2 más próxima al procesador y más rápida que L3
  - Algunos Mbytes, L3 con algo más de capacidad que L2
- L4: Cache de nivel 4
  - En algunos procesadores, pero suelen estar en otro chip separado

2.3 - Arquitectura interna

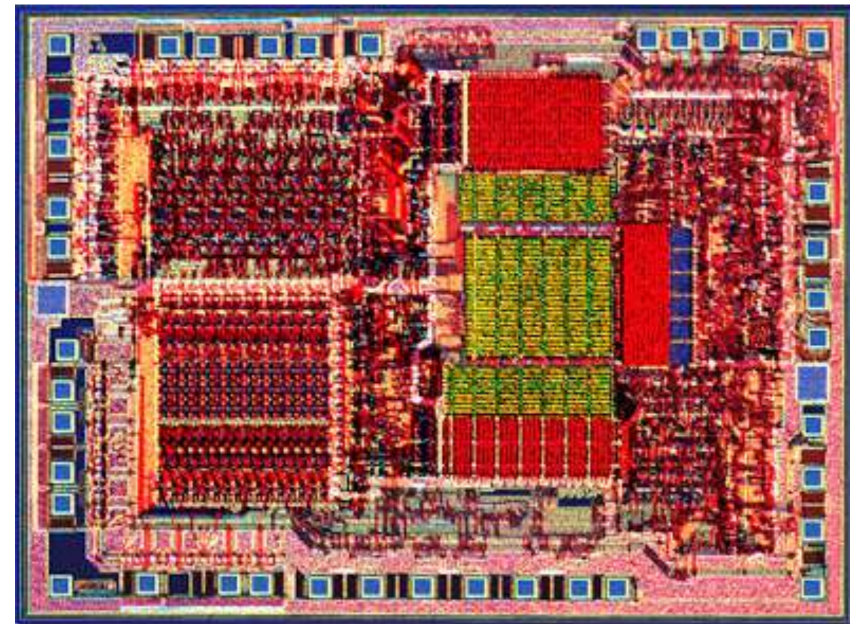
Intel 4004





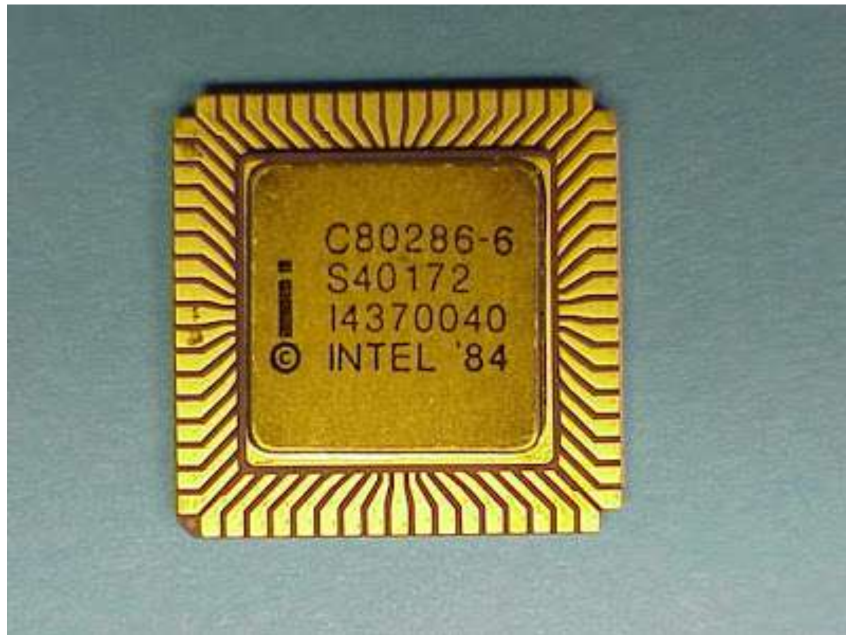
Intel 4004

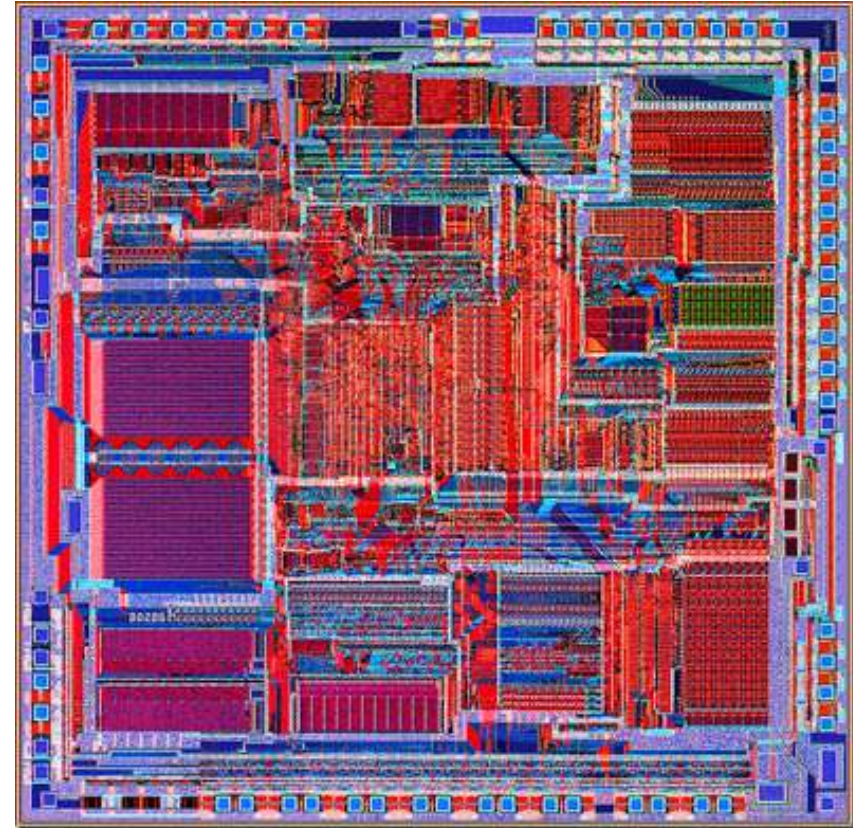
2.3 - Arquitectura interna
Intel 8085



Intel 8085

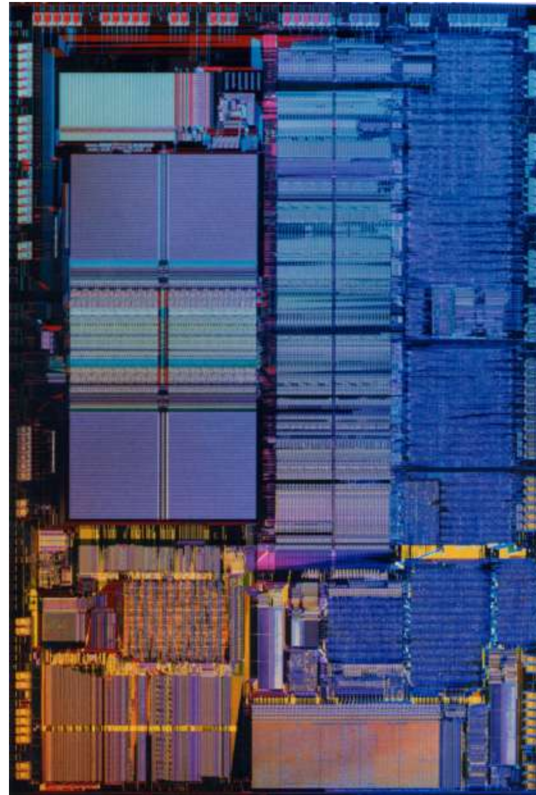
2.3 - Arquitectura interna
Intel 80286





Intel 80286

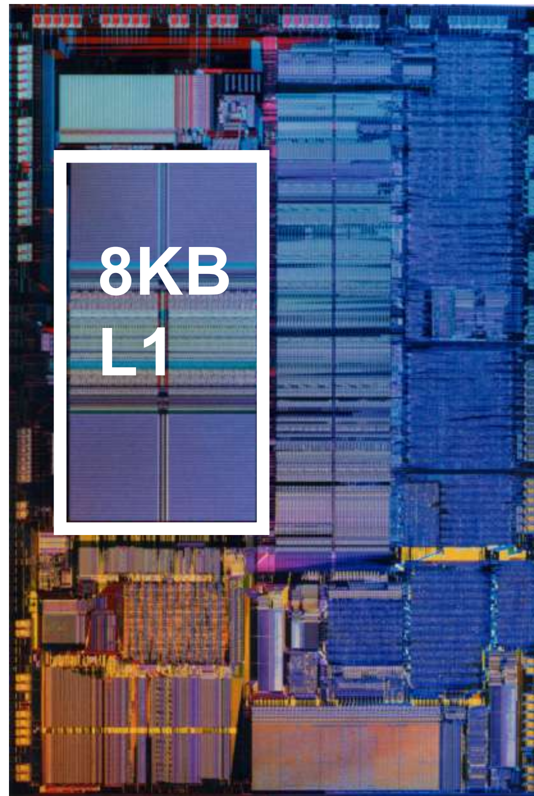
2.3 - Arquitectura interna
Intel 80486



Intel 80486

2.3 - Arquitectura interna

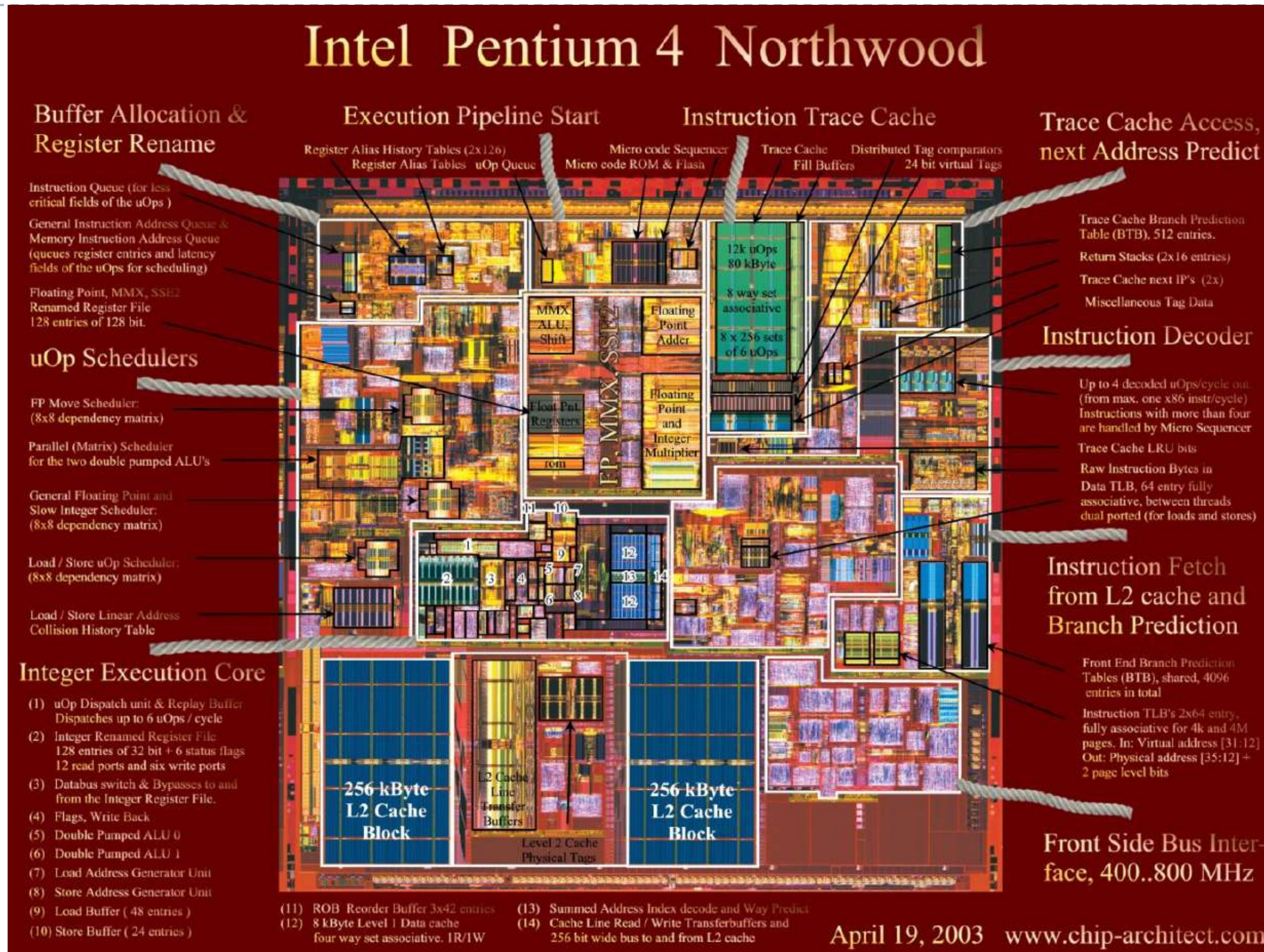
Intel 80486



Intel 80486

2.3 - Arquitectura interna

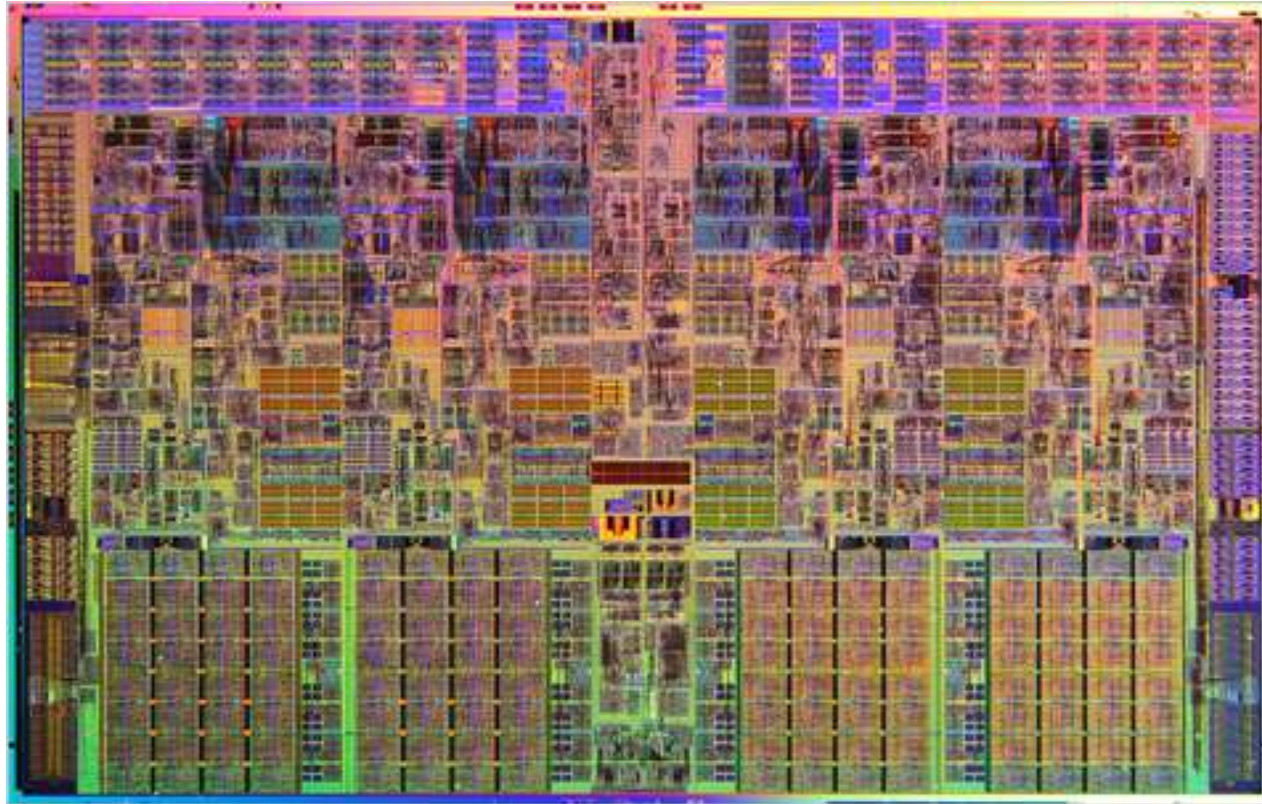
Intel Pentium 4



Intel Pentium 4

2.3 - Arquitectura interna

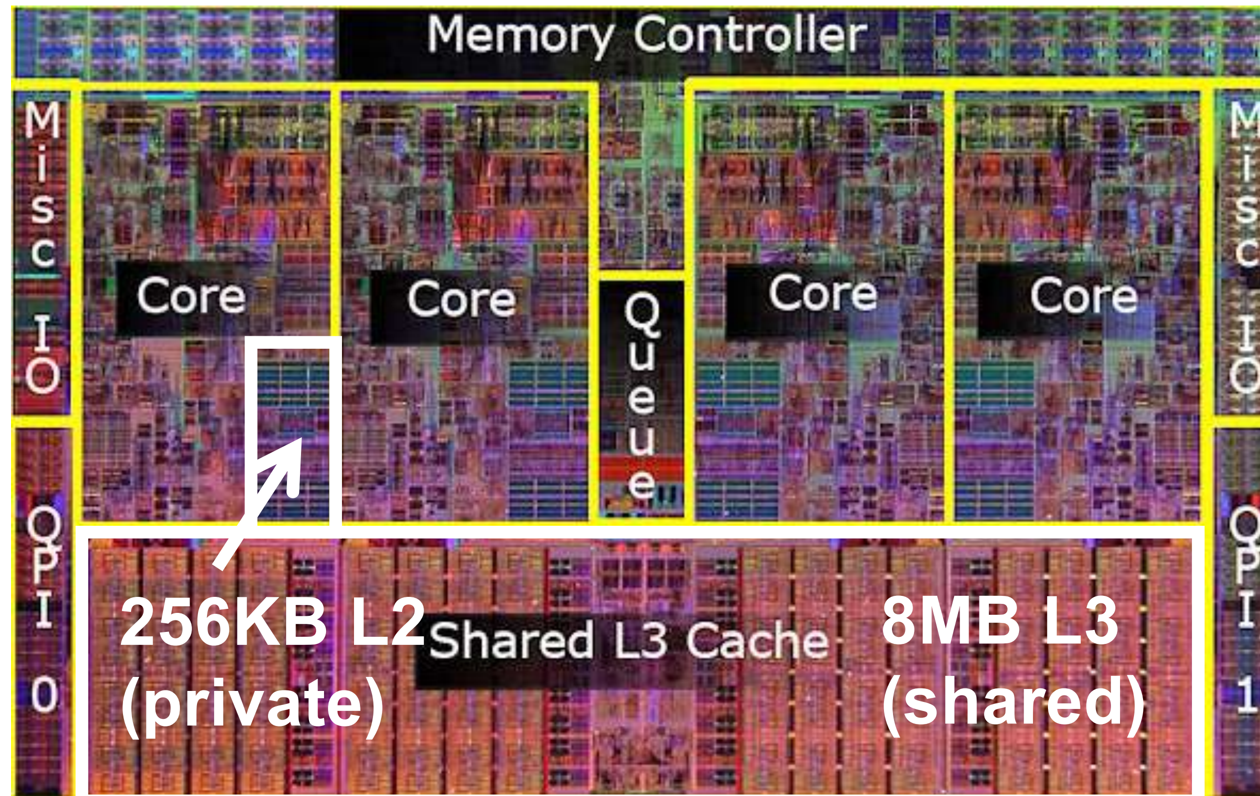
Intel Core i7 (quad core)



Intel Core i7 (quad core)

- Entre el 30 y 70% del área de un chip es típicamente cache

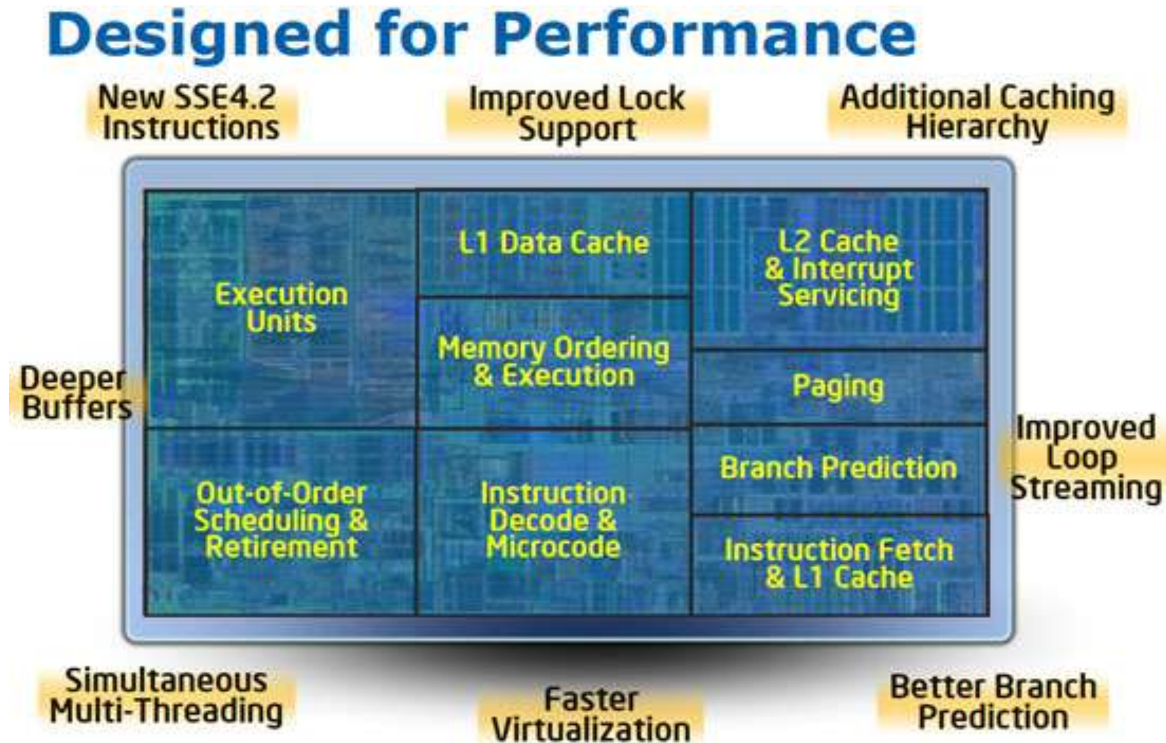
2.3 - Arquitectura interna
Intel Core i7 (quad core)



Intel Core i7 (quad core)

- Entre el 30 y 70% del área de un chip es típicamente cache

2.3 - Arquitectura interna
Intel Core i7 (quad core)

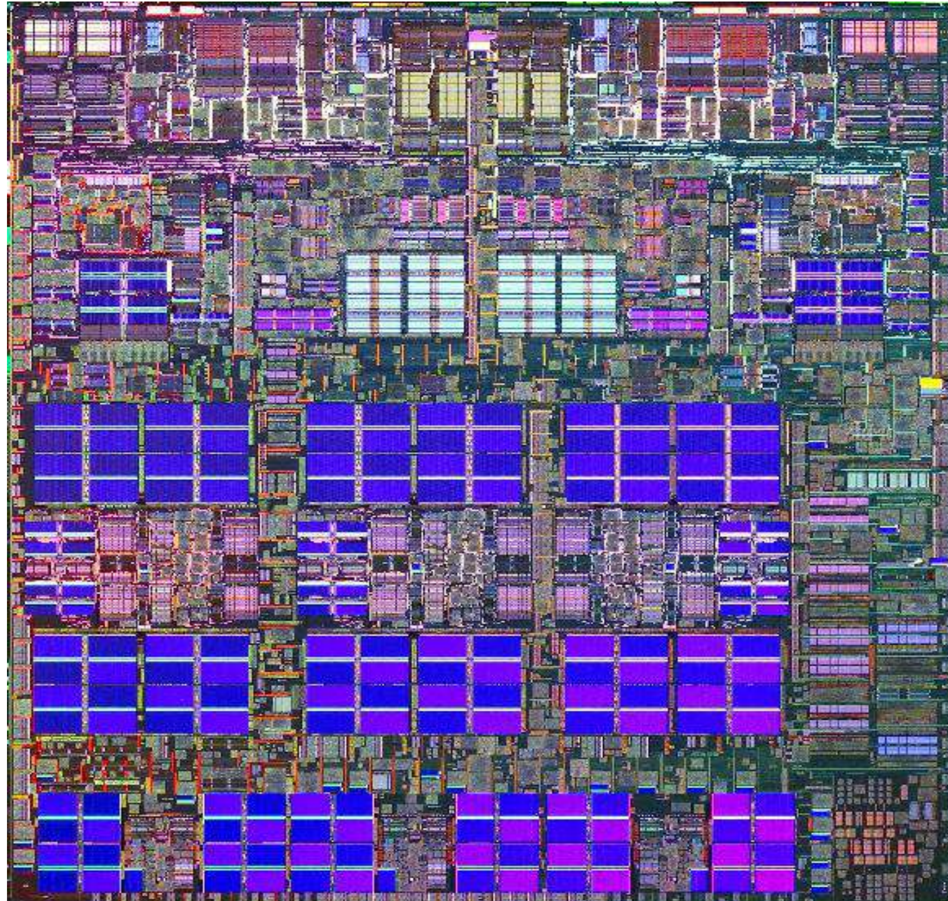


Intel Core i7 (quad core)

- Entre el 30 y 70% del área de un chip es típicamente cache

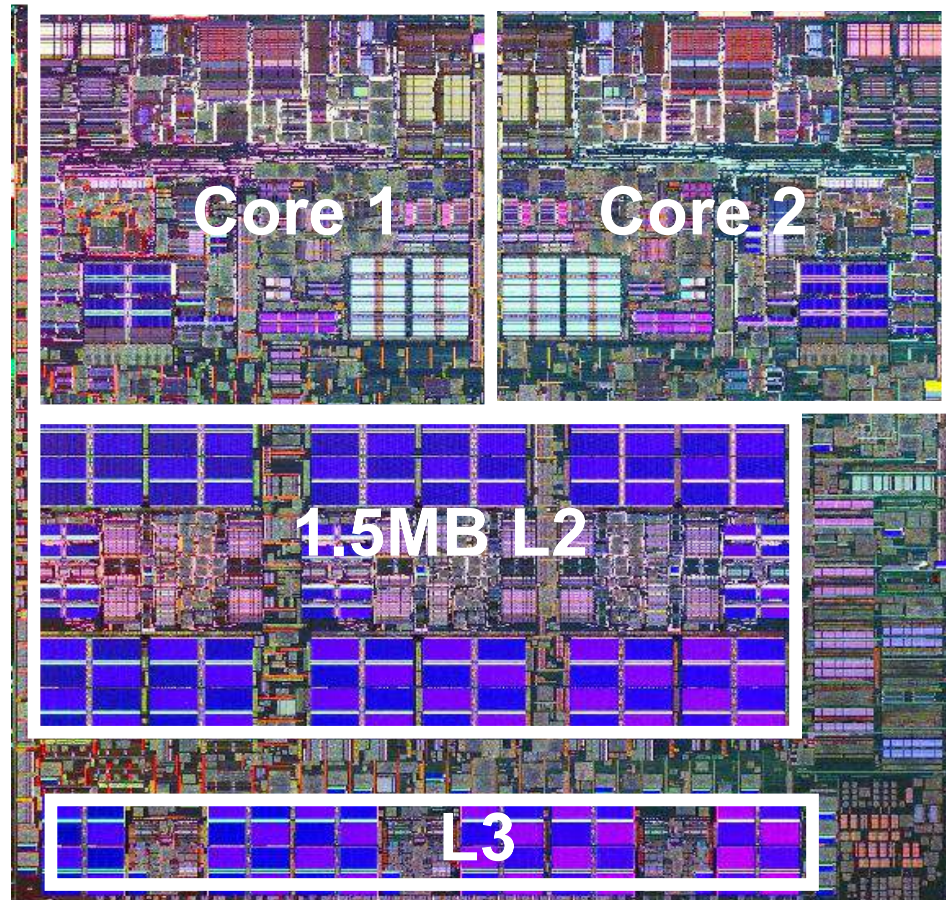
2.3 - Arquitectura interna

Power5 IBM



2.3 - Arquitectura interna

Power5 IBM



2.3 - Arquitectura interna

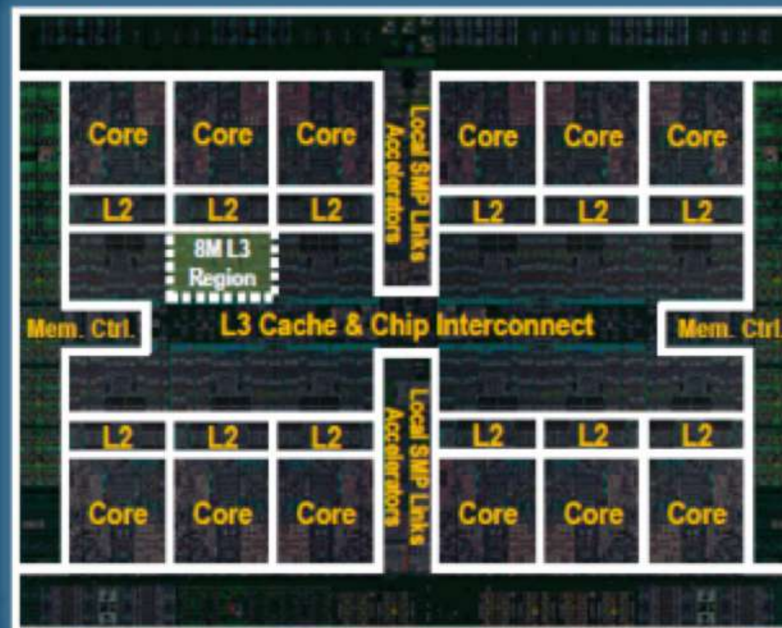
Power8 IBM

POWER8 Processor

Technology
- 22nm SOI, eDRAM, 15 ML 650mm2

Cores
- 12 cores (SMT8)
- 8 dispatch, 10 issue, 16 exec pipe
- 2X internal data flows/queues
- Enhanced prefetching
- 64K data cache, 32K instruction cache

Accelerators
- Crypto & memory expansion
- Transactional Memory
- VMM assist
- Data Move / VM Mobility



Caches
- 512 KB SRAM L2 / core
- 96 MB eDRAM shared L3
- Up to 128 MB eDRAM L4 (off-chip)

Memory
- Up to 230 GB/s sustained bandwidth

Bus Interfaces
- Durable open memory attach interface
- Integrated PCIe Gen3
- SMP Interconnect
- CAPI (Coherent Accelerator Processor Interface)

Energy Management
- On-chip Power Management Micro-controller
- Integrated Per-core VRM
- Critical Path Monitors

© 2013 International Business Machines Corporation

Temario

- Tema 1. Introducción
- Tema 2. El microprocesador
  - Introducción
  - Instruction Set Architecture
  - Arquitectura interna
  - Paralelismo
  - Ejercicios
- Tema 3. Memoria
- Tema 4. Dispositivos de E/S y buses
- Tema 5. DataCenters y modelos de comunicación

2.4 - Paralelismo

- ¿Cómo se puede mejorar las prestaciones de un procesador sin cambiar o mejorar su frecuencia o incluido su arquitectura?
  - Parte de los recursos hardware están desocupados durante las diferentes fases del ciclo de instrucción
- Aumentar el paralelismo a nivel de instrucción:
  - Segmentación (pipelining)
  - Supersegmentación
  - Superescalado
  - Procesadores VLIW
  - Todas implican la ejecución de varias instrucciones con un única CPU.
- Aumentar el paralelismo a nivel de procesador: multinúcleo
  - Disponer de múltiples núcleos ejecutando simultáneamente varias instrucciones.

2.4 - Paralelismo

- El tiempo de ejecución de un programa es producto de tres términos:

$$Te = Inst \times CPI \times Tc$$

donde

Inst: instrucciones/programa

CPI: número medio de ciclos por instrucción

Tc: tiempo de ciclo de reloj

2.4 - Paralelismo

- El tiempo de ejecución de un programa es producto de tres términos:

$$Te = Inst \times CPI \times Tc$$

- Para reducir el tiempo de ejecución Te
  - Segmentación: segmentar el ciclo de instrucción en k etapas y ejecutar las diferentes etapas en paralelo

- Reducir el tiempo de ciclo Tc, sin reducir el tiempo total de cada segmento
  - Procesadores supersegmentados

- Reducir el número medio de ciclos de reloj por instrucción CPI
  - Aumentar el número de instrucciones lanzadas a ejecución en un ciclo.
  - Procesadores superescalares

- Reducir el número de instrucciones ejecutadas Inst
  - Aumentar el trabajo que realiza cada instrucción máquina.
  - Procesadores VLIW

2.4 – Paralelismo
Segmentación

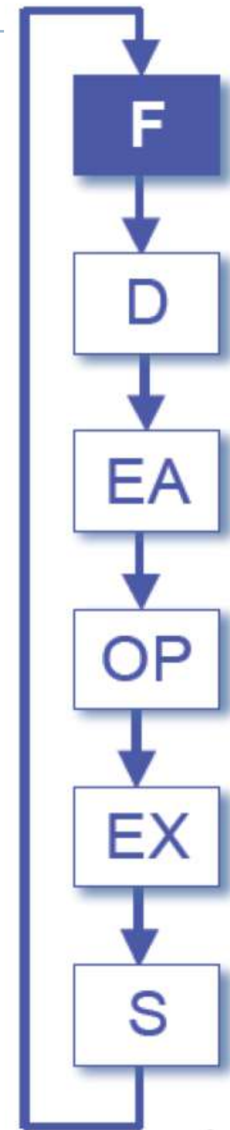
- Un procesador segmentado divide el proceso de ejecución de una instrucción en k etapas de similar duración con el objetivo final de procesar k instrucciones simultáneamente, estando cada una en una etapa diferentes de su ejecución
  - Divide la ejecución de una instrucción en etapas (stages)
  - Cada etapa utiliza diferentes recursos
  - Permite solapar la ejecución de distintas instrucciones que se hallan en diferentes etapas
  - Permite comenzar una instrucción en cada ciclo de reloj
  - En régimen permanente, finaliza una instrucción en cada ciclo de reloj

- Primera versión en el Z1 (1939) y Z3 (1941)
- Usado en ILLIAC II e IBM 7030 (1962)
- Difusión en los 70 y 80
- Hoy en día se usa en prácticamente todas las arquitecturas

2.4 – Paralelismo

Segmentación de 6 etapas

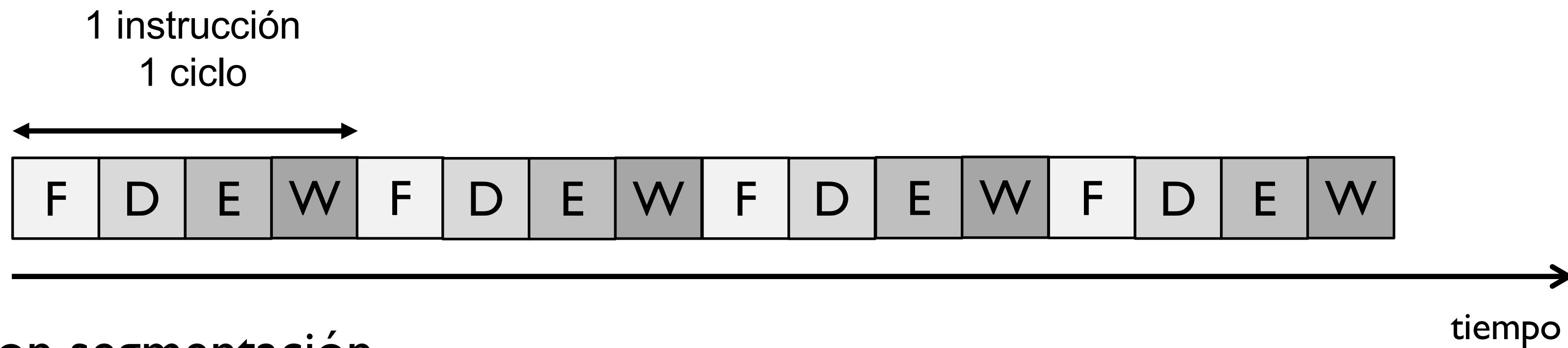
- **Captar instrucción** o **Fetch de instrucción**: el procesador lee una instrucción de la memoria. Las instrucciones y los datos deben ser cargados desde la memoria a los registros de la CPU.
- **Interpretar instrucción**: la instrucción se decodifica para determinar qué acción es necesaria.
- **Evaluar la direcciones**: para instrucciones que requieren acceder a memoria, calcular la dirección (diferentes tipos de direccionamiento).
- **Captar datos o Fetch de datos**: la ejecución de una instrucción puede exigir leer datos de la memoria o de un módulo I/O.
- **Procesar datos**: en la ejecución se puede exigir llevar a cabo alguna operación aritmética o lógica con los datos.
- **Escribir datos**: los resultados de la ejecución pueden exigir escribir datos en la memoria o en un módulo I/O.



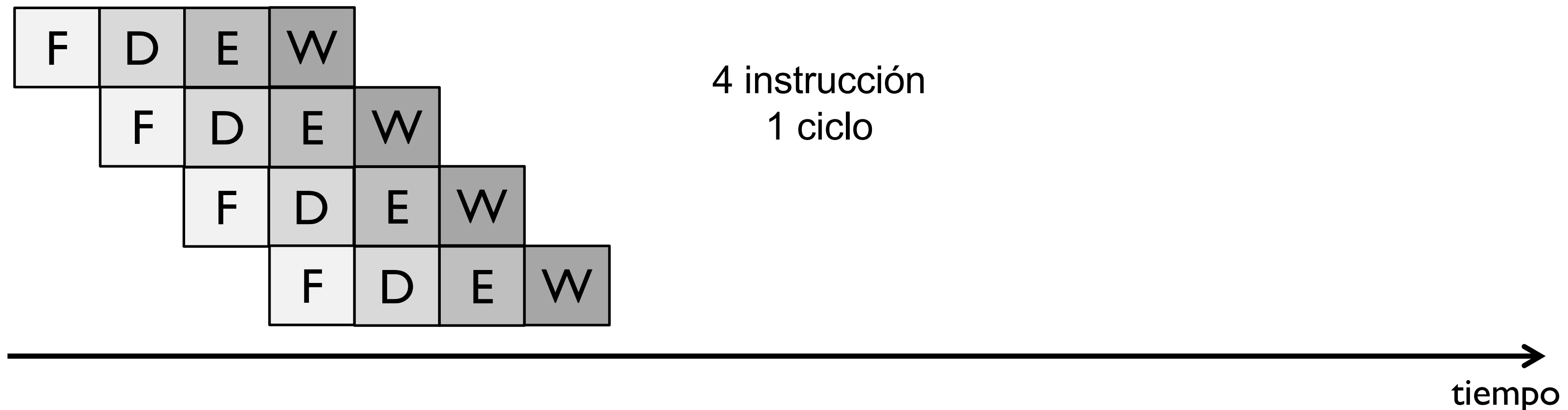
- Cada instrucción no ejecuta necesariamente las 6 etapas
  Depende de la instrucción misma

2.4 – Paralelismo
Segmentación

- Ejemplo de un programa que solo necesita 4 etapas por instrucción
  - F: Fetch, D: Decode, E: Execution, W: Write
- Sin segmentación



- Con segmentación

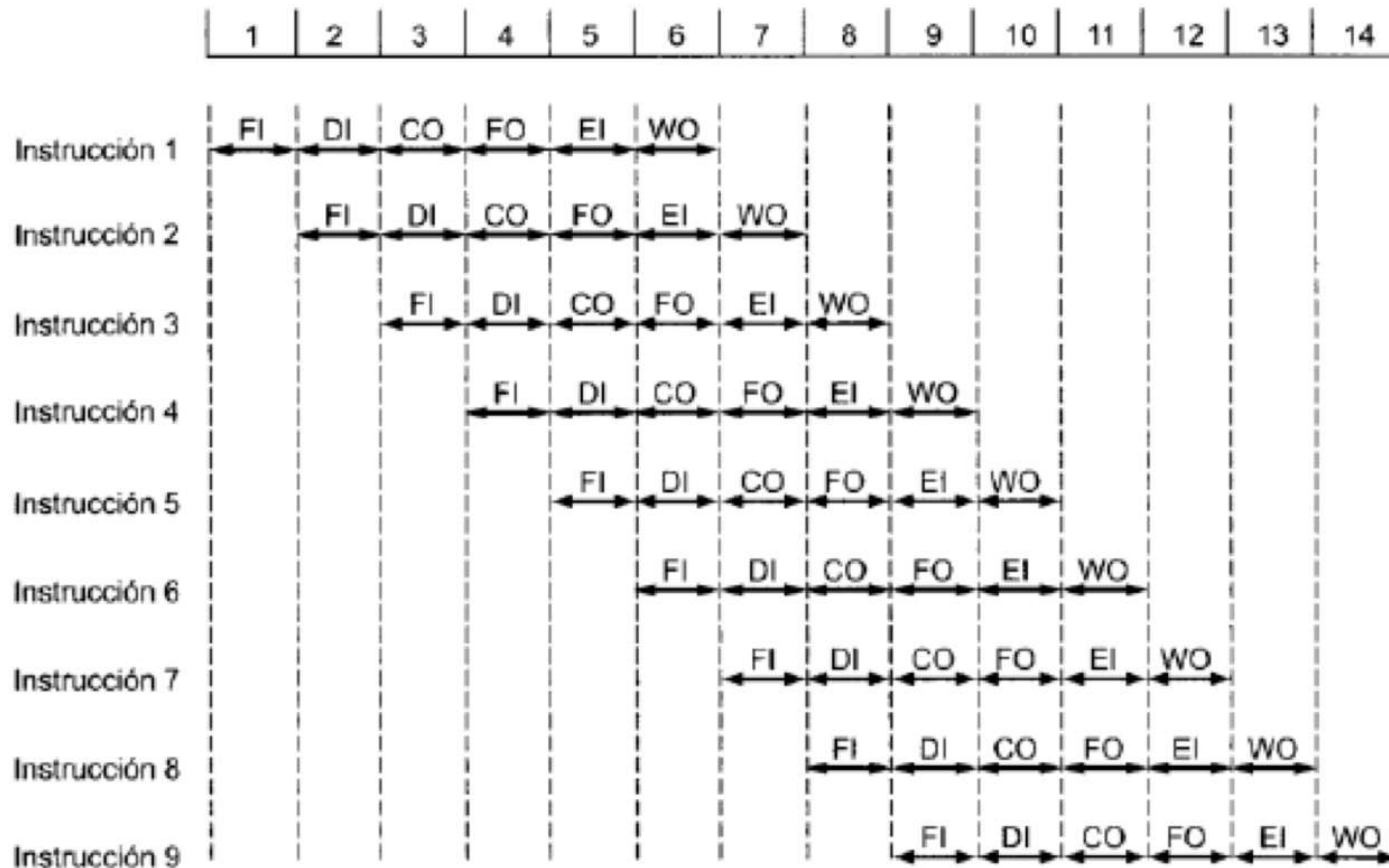


2.4 – Paralelismo
Segmentación ideal

- Repetición de instrucciones idénticas
  - La misma instrucción se repite con diferentes entradas
- Repetición de instrucciones independientes
  - No dependencias entre las instrucciones
- Uniformemente divisibles en etapas de la misma duración que no compartan recursos hardware
  - Etapas lo más equilibradas posibles, con tiempo de ejecución similar
  - Cuantas más etapas, éstas son más sencillas y se pueden implementar con un número menor de transistores.
- En el caso ideal, la velocidad de ejecución de una instrucción se multiplica por el número de etapas en que está segmentado.

2.4 – Paralelismo

Segmentación ideal con 6 etapas



Segmentación ideal

- El tiempo de ejecución de un programa es producto de tres términos:

$$Te = Inst \times CPI \times Tc / k$$

donde

Inst: instrucciones/programa

CPI: número medio de ciclos por instrucción

Tc: tiempo de ciclo de reloj

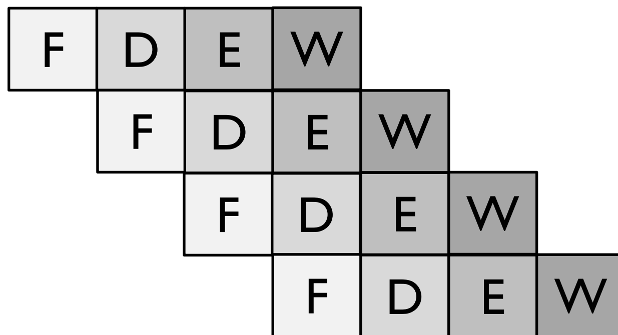
k: número de etapas

2.4 – Paralelismo
Supersegmentación

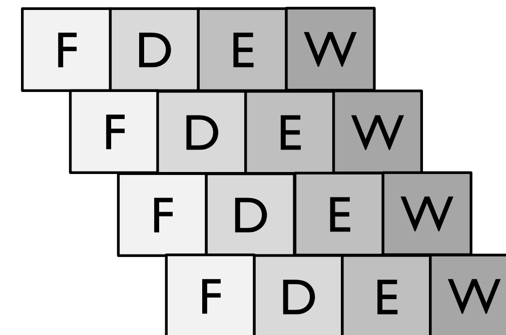
- Cada una de las etapas básicas es a su vez dividida en n tapas aún más simples.
- El número total de etapas de cada ciclo de instrucción es por lo tanto kn
- Donde

  n es el grado de supersegmentación (número de etapas en las que se divide cada etapa)

  k el número de etapas del procesador segmentado.
- Por ejemplo, con k=4 y n=2:



Segmentación



Supersegmentación

2.4 – Paralelismo
Supersegmentación ideal

- El tiempo de ejecución de un programa es producto de tres términos:

$$Te = Inst \times CPI \times Tc / (kn)$$

donde

Inst: instrucciones/programa

CPI: número medio de ciclos por instrucción

Tc: tiempo de ciclo de reloj

k: número de etapas

n: grado de supersegmentación

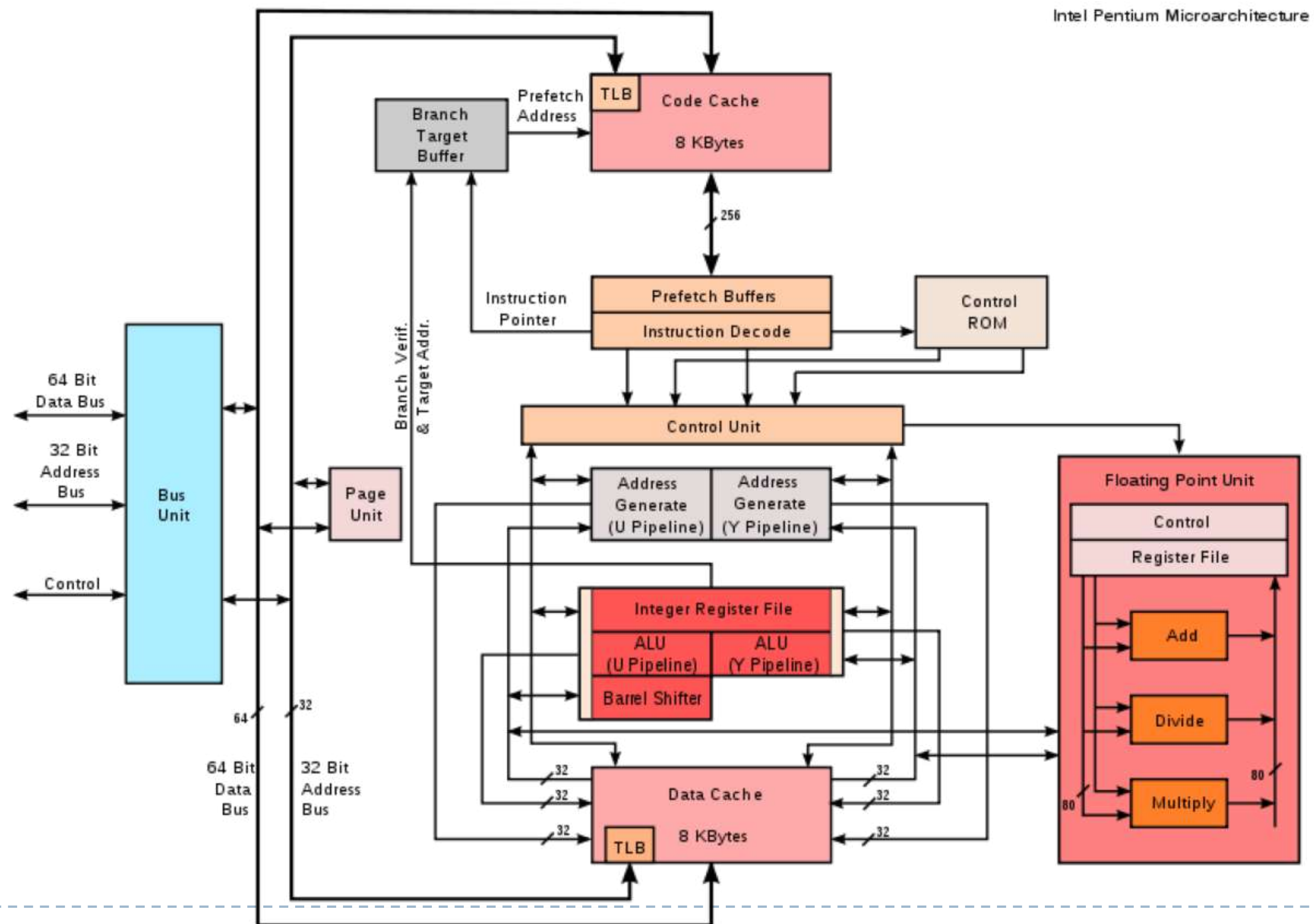
2.4 – Paralelismo
Superescalaridad

- Un procesador superescalar de factor m es aquel que replica m veces la circuitería de alguna de sus unidades funcionales para poder ejecutar m instrucciones a la vez
  - Una única unidad de procesado pero diferentes unidad ejecutivas:
  - Varias unidades de control secuencial
  - Registros duplicados
  - ALU (y FPU) duplicadas
  - Etc.

- Primera implementación en el Cray CDC 6600 (1966)
- Motorola 88100 (1988), Intel i960CA (1989), AMD 29000 (1990) fueron los primeros procesadores comerciales superescalares
- Intel Pentium (1993) el primer superescalar x86
- PowerPC 970 con 4 ALUs, 2 FPU, y 2 SIMD

2.4 – Paralelismo
Superescalaridad

- Intel Pentium (1993) el primer superescalar x86



2.4 – Paralelismo
Superescalaridad

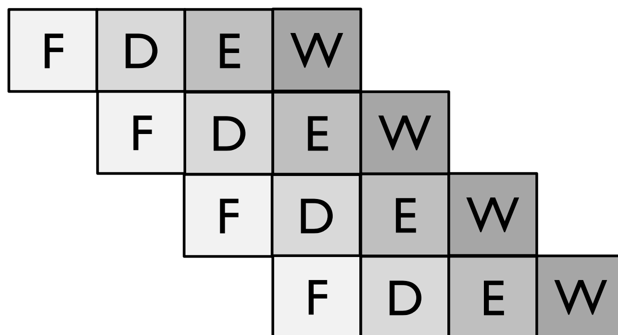
- Terminología

- SISD: Single Instruction, Single Data
  - Un ordenador con una única unidad funcional que realiza una única operación sobre un dato guardado
- SIMD: Single Instruction, Multiple Data
  - Un ordenador con múltiples unidades funcionales que realizar la misma operación sobre múltiples datos
  - Hay múltiples operaciones computacionales a la vez pero una única instrucción
- MISD: Multiple Instruction, Single Data
  - Un ordenador con múltiples unidades funcionales que realiza múltiples operaciones a la vez en un mismo dato
- MISM: Multiple Instruction, Multiple Data
  - Un ordenador con múltiples unidades funcionales que realiza múltiples operaciones a la vez sobre múltiples datos diferentes

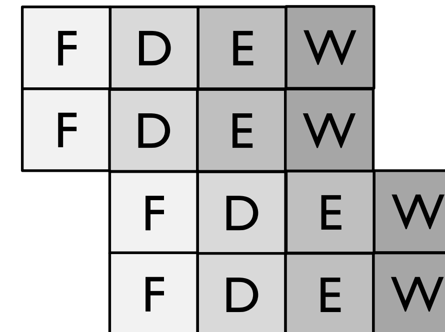
2.4 – Paralelismo

Superescalaridad

- Se buscan instrucciones seguidas no relacionadas y se ejecutan en paralelo
- Se ejecutan multiples instrucciones por ciclo



Segmentación de 4 etapas



Superescalaridad con m=2

2.4 – Paralelismo

Superescalaridad ideal

- El tiempo de ejecución de un programa es producto de tres términos:

$$Te = Inst \times CPI / m \times Tc$$

donde

Inst: instrucciones/programa

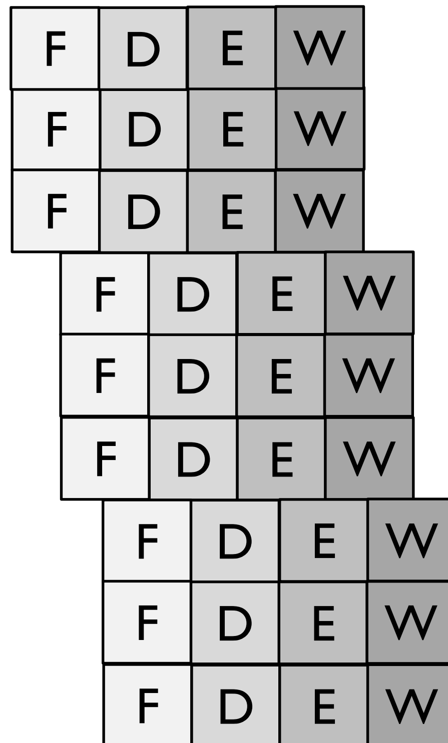
CPI: número medio de ciclos por instrucción

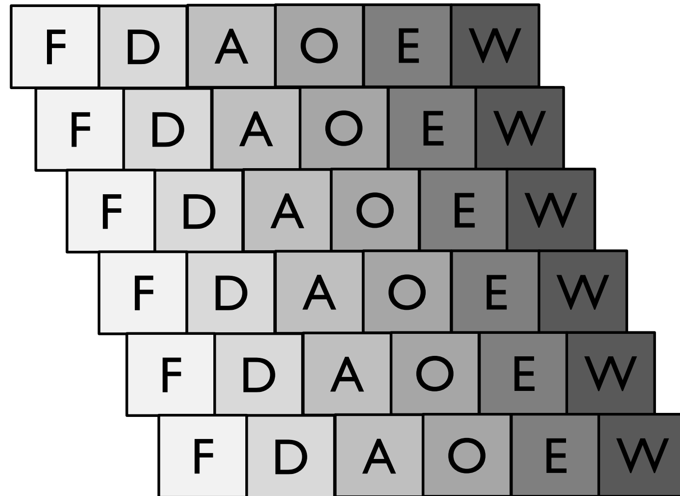
Tc: tiempo de ciclo de reloj

m: grado de superescalaridad

- k: número de etapas
- n: grado de supersegmentación
- m: grado de superescalaridad

Instrucciones por ciclo: k n m





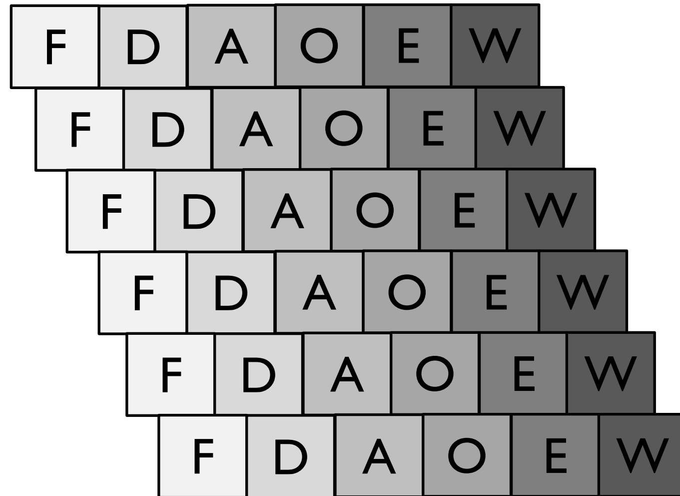
k =

n =

m =

2.4 – Paralelismo

Segmentación, supersegmentación y superescalaridad



k = 6

n = 3

m = 1

2.4 – Paralelismo

Segmentación, supersegmentación y superescalaridad

F	D	A	O	W			
F	D	A	O	W			
	F	D	A	O	W		
	F	D	A	O	W		
		F	D	A	O	W	
		F	D	A	O	W	

k =

n =

m =

2.4 – Paralelismo

Segmentación, supersegmentación y superescalaridad

F	D	A	O	W		
F	D	A	O	W		
	F	D	A	O	W	
	F	D	A	O	W	
		F	D	A	O	W
		F	D	A	O	W

$k = 5$

$n = 1$

$m = 2$

2.4 – Paralelismo

Segmentación, supersegmentación y superescalaridad

F D A O W
F D A O W
F D A O W
F D A O W
F D A O W
F D A O W
F D A O W
F D A O W
F D A O W
F D A O W
F D A O W
F D A O W

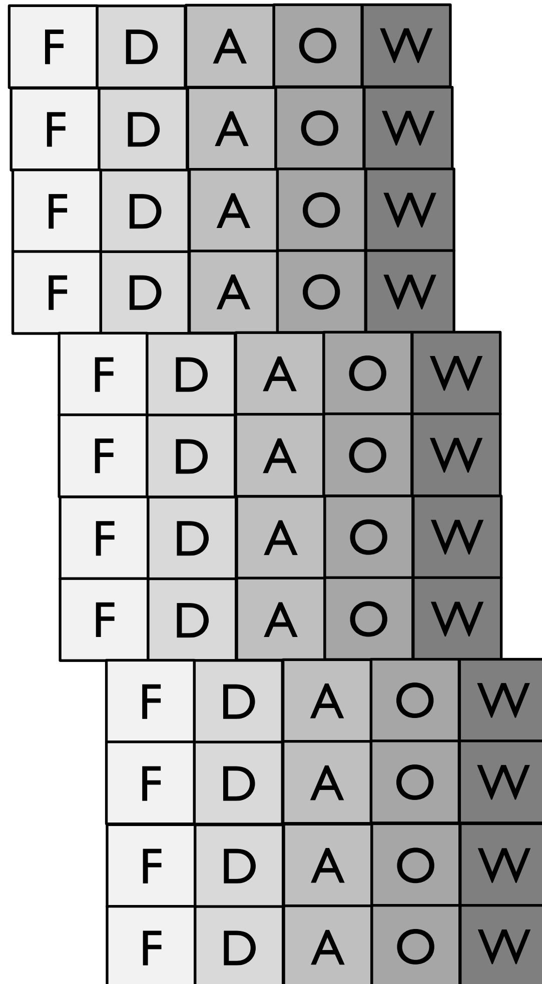
k =

n =

m =

2.4 – Paralelismo

Segmentación, supersegmentación y superescalaridad



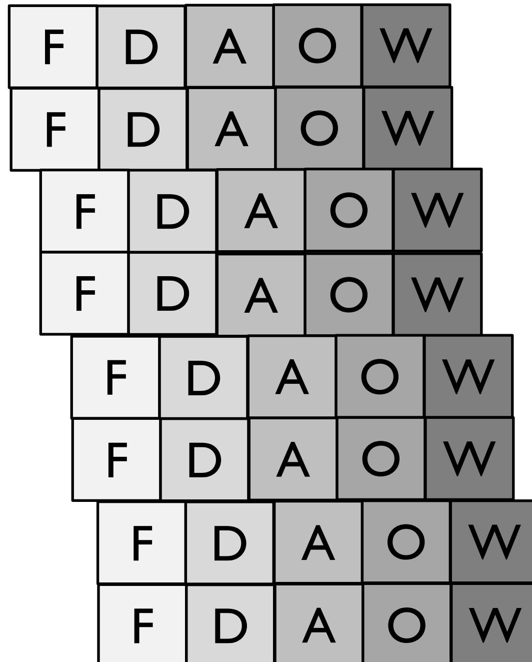
k = 5

n = 2

m = 4

2.4 – Paralelismo

Segmentación, supersegmentación y superescalaridad



k =

n =

m =

2.4 – Paralelismo

Problemas (hazards)

- No todas las instrucciones necesitan todas las etapas
  - Se fuerza a todas las instrucciones por pasar por todas las etapas
  - Fragmentación externa (alguna etapa se queda sin hacer nada)
- No todas las etapas necesitan el mismo tiempo para ejecutarse
  - Se fuerzan todas las etapas a durar lo mismo
  - Fragmentación interna (alguna etapa es demasiado rápida y se queda sin hacer nada durante un tiempo)
- Las operaciones no son independientes entre ellas
  - Se necesitan resolver las dependencias entre instrucciones diferentes para que la segmentación opere correctamente
  - Las ejecución paralela se puede quedar parada
  - Aumentando el paralelismo, aumenta el número de instrucciones que se pueden ejecutar al mismo tiempo, aumenta la probabilidad de dependencia entre ellas

2.4 – Paralelismo
Problemas (hazards)

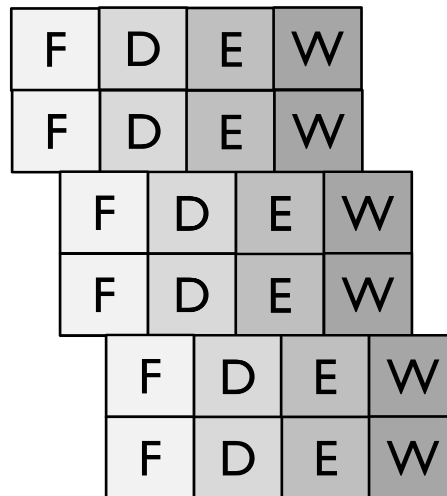
- Las operaciones no son independientes entre ellas

Read-after-write (RAW)	r3 = r1 + r2 r5 = r3 – r4	Se ejecutan al mismo tiempo La segunda lee el valor de r3 antes que la primera escriba su nuevo valor
Write-after-read (WAR)	r3 = r1 + r2 r1 = r4 – r5	La segunda puede que escriba el nuevo valor de r1 antes que la primera acabe de hacer la operación
Write-after-write (WAW)	r3 = r1 + r2 r6 = r6 – r3 r3 = r4 – r5	La tercera escribe en el registro r3 antes que la primera

2.4 – Paralelismo

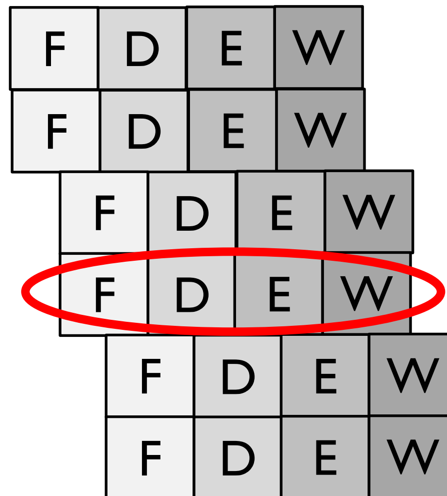
Problemas (hazards)

r1 = load(0x52fe)
r2 = r3 + r4
r5 = r5 * 2
write(0xab43) = r5
r3 = r2 + r5
write(0xab44) = r3



2.4 – Paralelismo
Problemas (hazards)

r1 = load(0x52fe)
r2 = r3 + r4
r5 = r5 * 2
write(0xab43) = r5
r3 = r2 + r5
write(0xab44) = r3

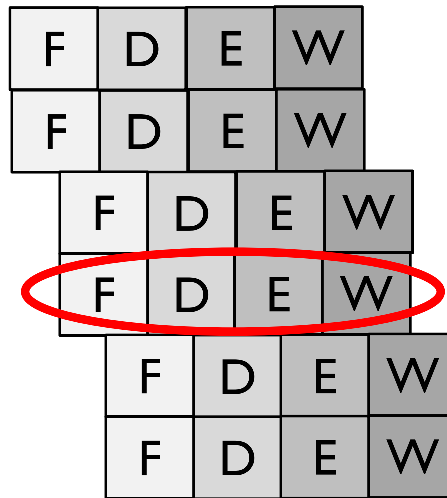


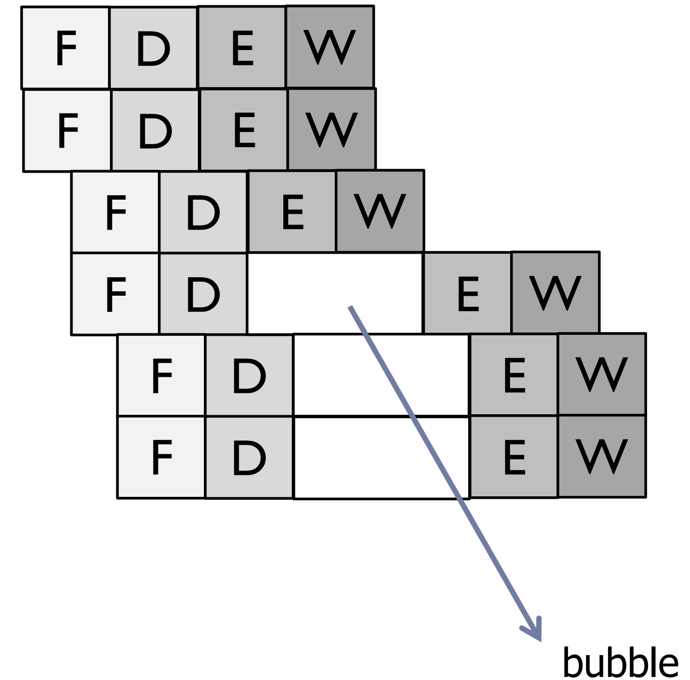
No se puede ejecutar en paralelo con la anterior

2.4 – Paralelismo

Problemas (hazards)

```
r1 = load( 0x52fe )
r2 = r3 + r4
r5 = r5 * 2
write( 0xab43 ) = r5
r3 = r2 + r5
write( 0xab44 ) = r3
```





2.4 – Paralelismo
Problemas (hazards)

- WAR y WAW son fáciles de resolver con una simple regla
  - Se escribe en un mismo registro en una sola etapa y en el orden del programa
  - Se usan registros diferentes (renaming)
- RAW más difícil
  - Detectarlo y esperar que el nuevo valor del registro esté disponible
  - Detectarlo y sacar el valor necesario de una etapa previa (por ejemplo de la ejecución en lugar de esperar el write)
  - Predecir el valor del registro y ejecutar la instrucción igualmente
    - verificar luego si la predicción ha sido correcta y si no lo es recalcular todo
    - muchos mecanismos basado en análisis del pasado, profiling, etc.
  - Multihilo (multithread) de manera que no se están ejecutando múltiples instrucciones al mismo tiempo del mismo hilo
  - Detectarlo y ejecutar la siguiente instrucción que no tiene dependencia con las que se están ya ejecutando (out of order execution)
- Scheduler/Dispacher es el elemento que decide cuando y que instrucciones ejecutar

2.4 – Paralelismo
Problemas (hazards)

- In-order execution

r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5

2.4 – Paralelismo
Problemas (hazards)

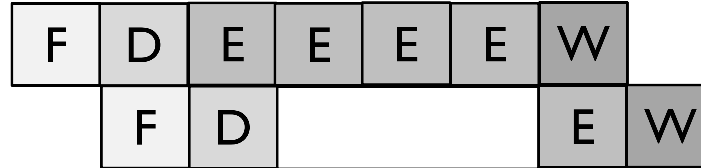
- In-order execution

r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5

2.4 – Paralelismo
Problemas (hazards)

- In-order execution

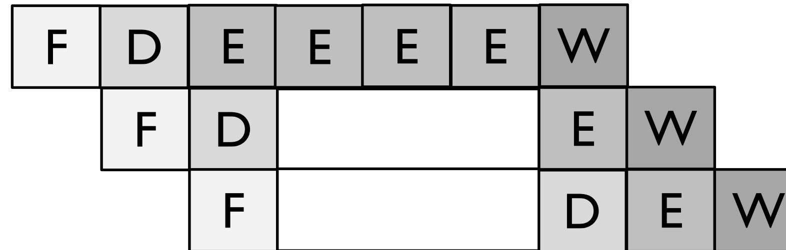
r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5



2.4 – Paralelismo
Problemas (hazards)

- In-order execution

r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5

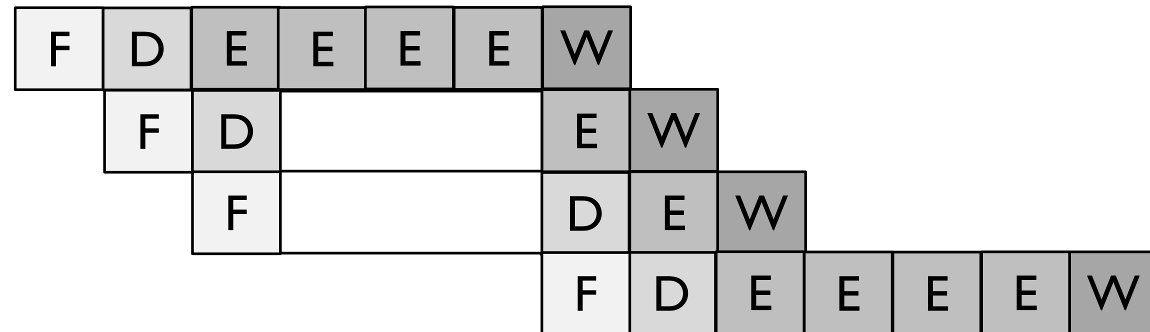


2.4 – Paralelismo

Problemas (hazards)

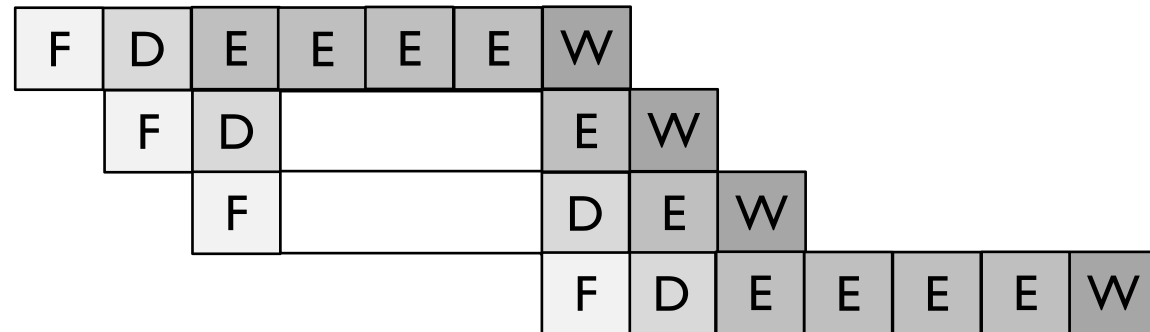
- In-order execution

r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5



- In-order execution

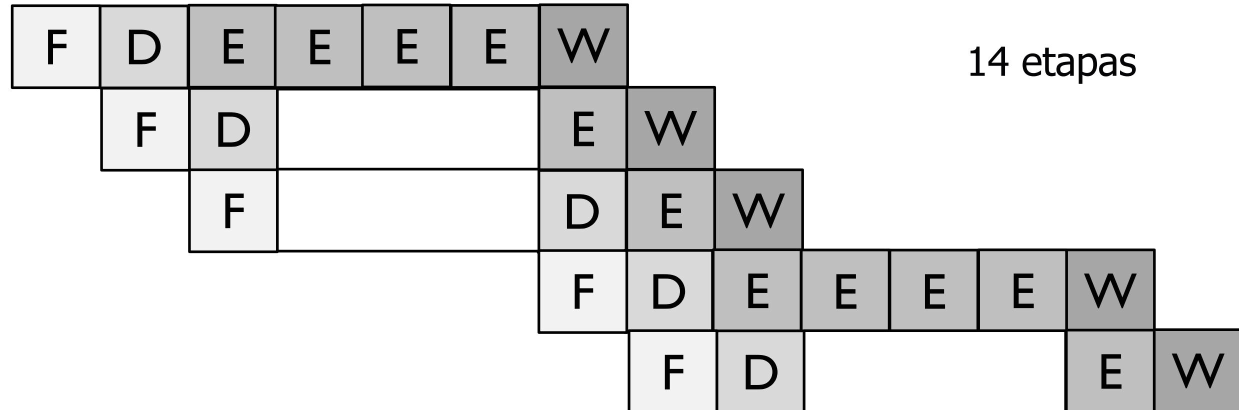
r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5



2.4 – Paralelismo
Problemas (hazards)

- In-order execution

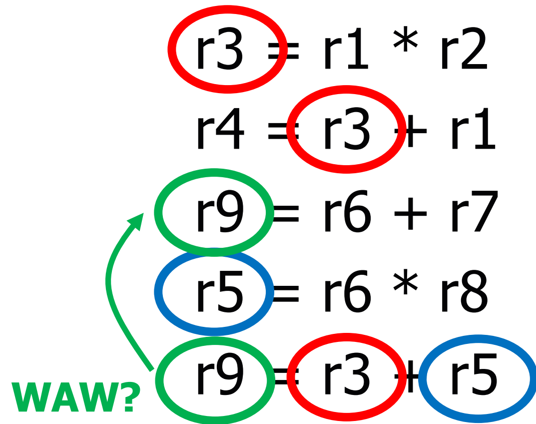
r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5

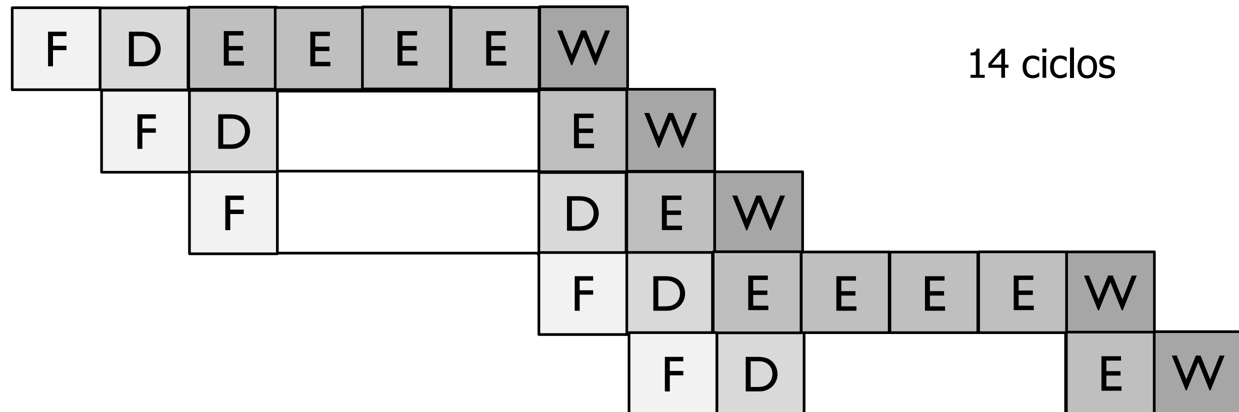


2.4 – Paralelismo

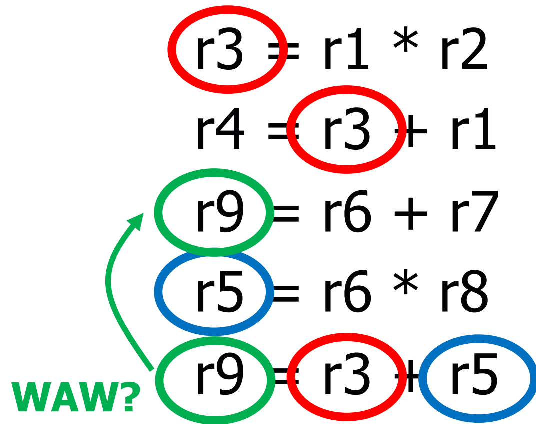
Problemas (hazards)

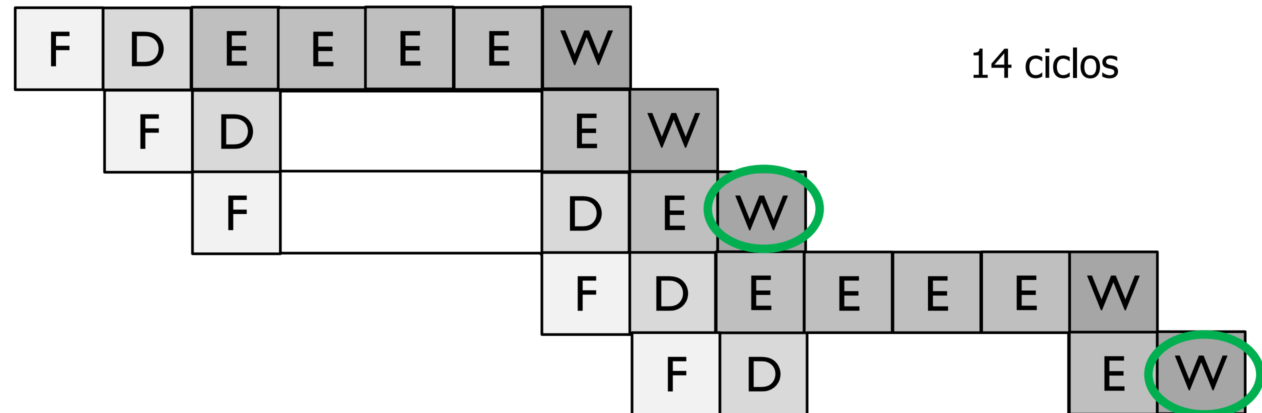
- In-order execution





- In-order execution

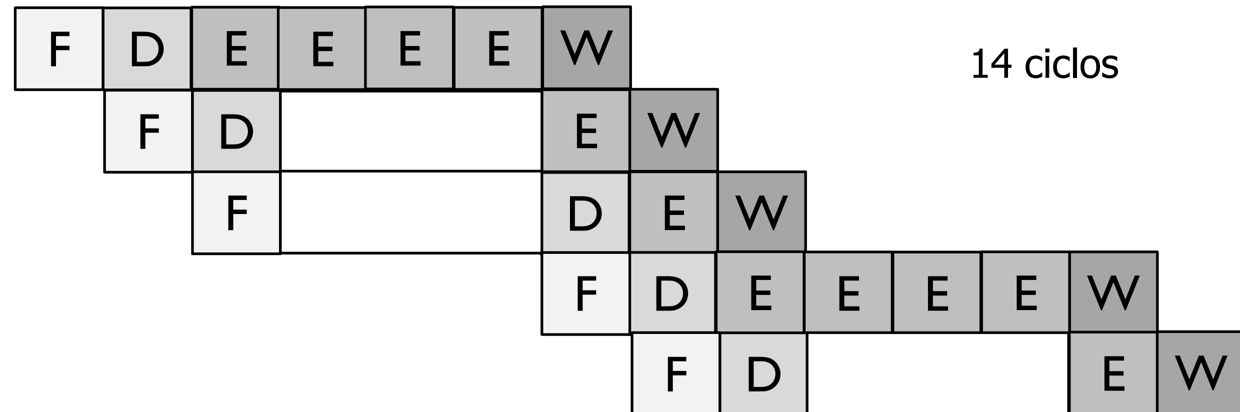




2.4 – Paralelismo
Problemas (hazards)

- In-order execution

r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5



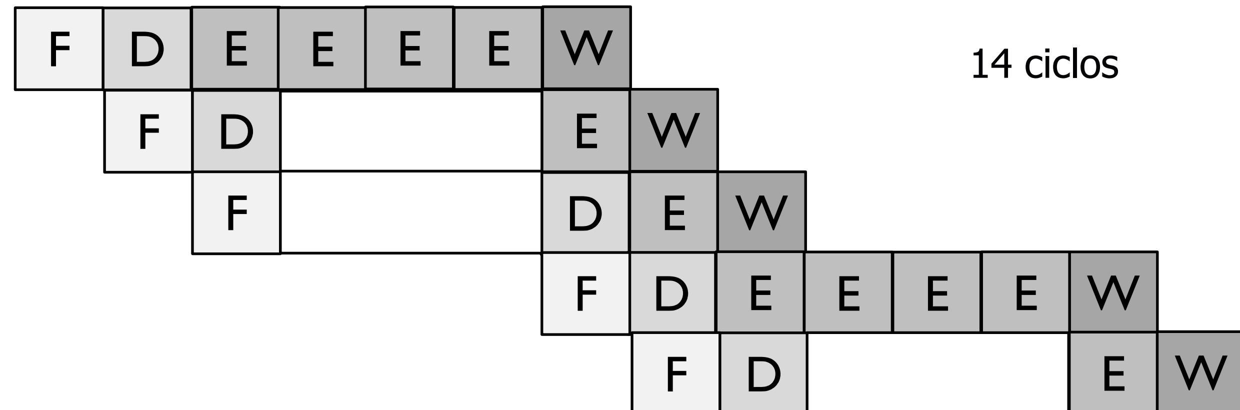
- Out-of-order execution

r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5

2.4 – Paralelismo
Problemas (hazards)

- In-order execution

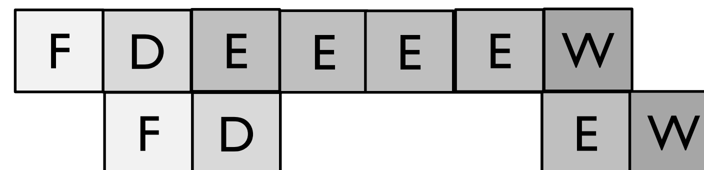
r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5



14 ciclos

- Out-of-order execution

r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5

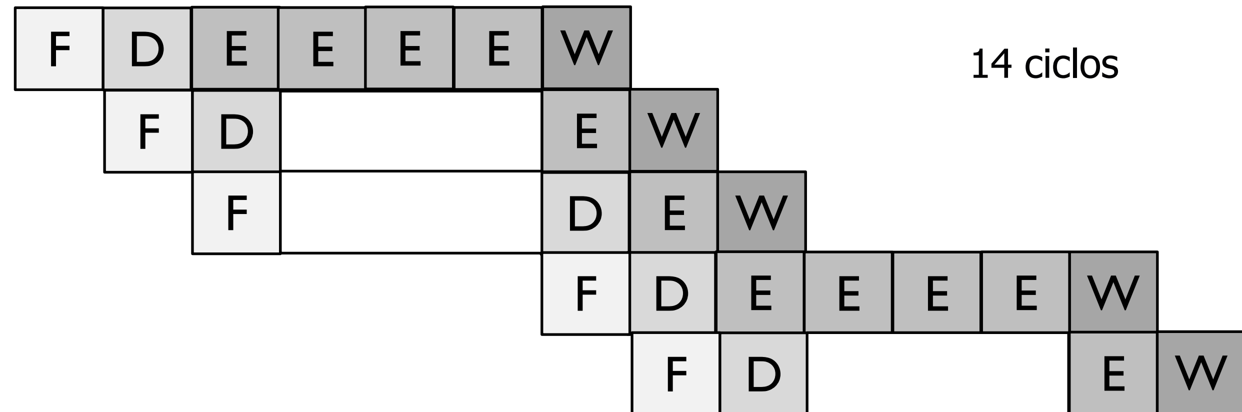


2.4 – Paralelismo

Problemas (hazards)

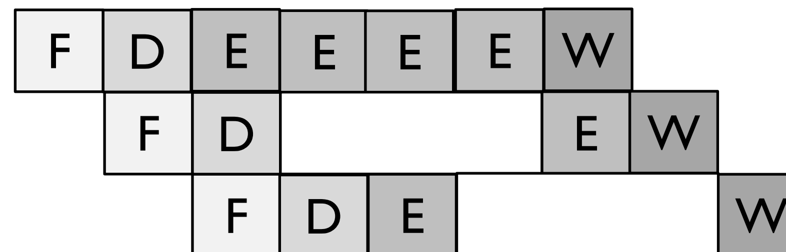
- In-order execution

r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5



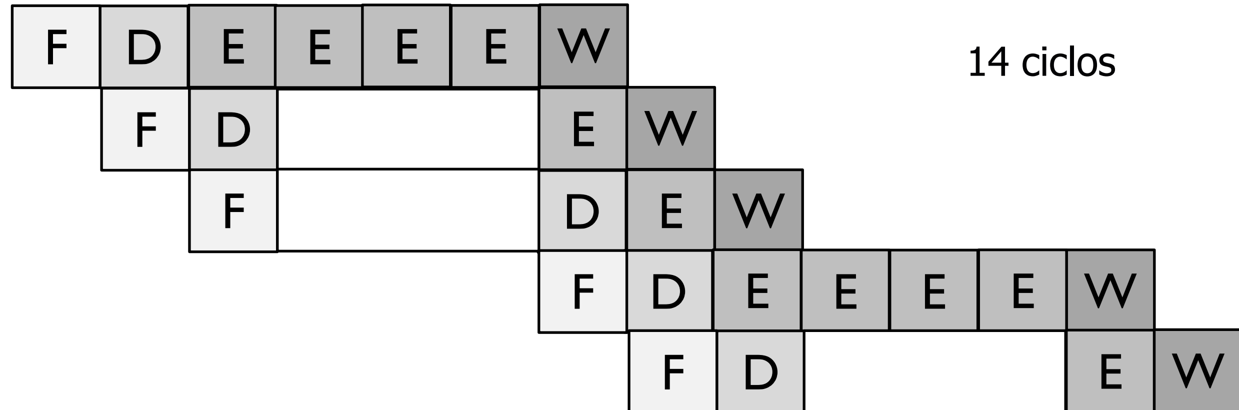
- Out-of-order execution

r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5



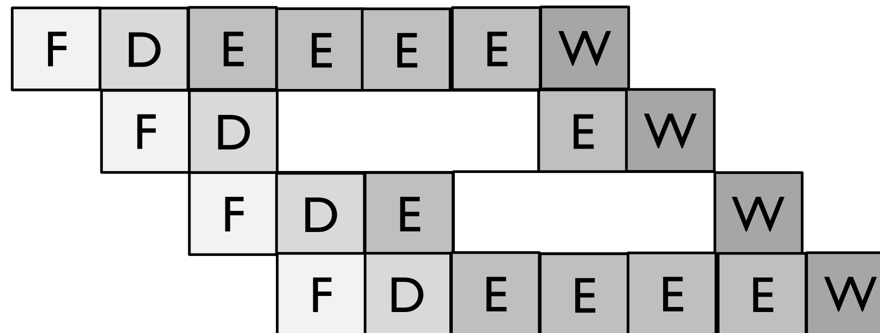
- In-order execution

r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5



- Out-of-order execution

r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5

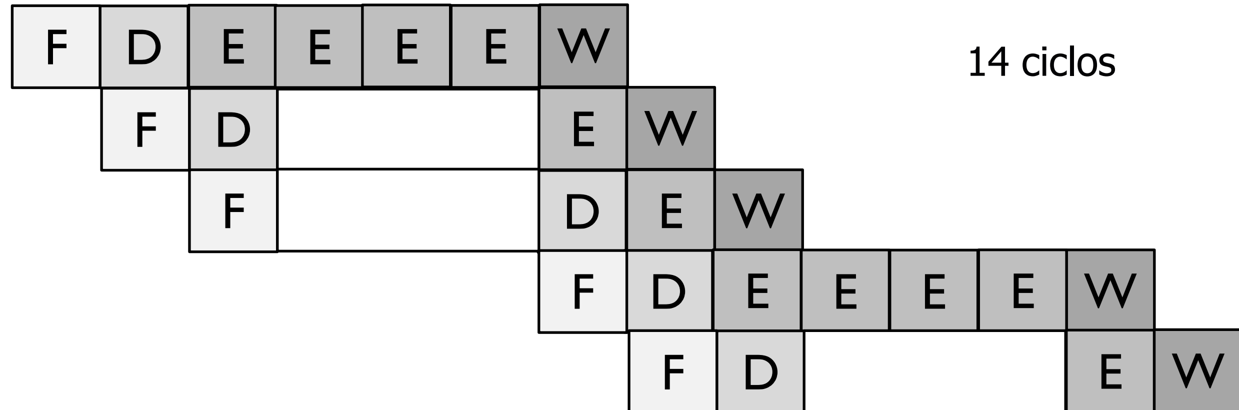


2.4 – Paralelismo

Problemas (hazards)

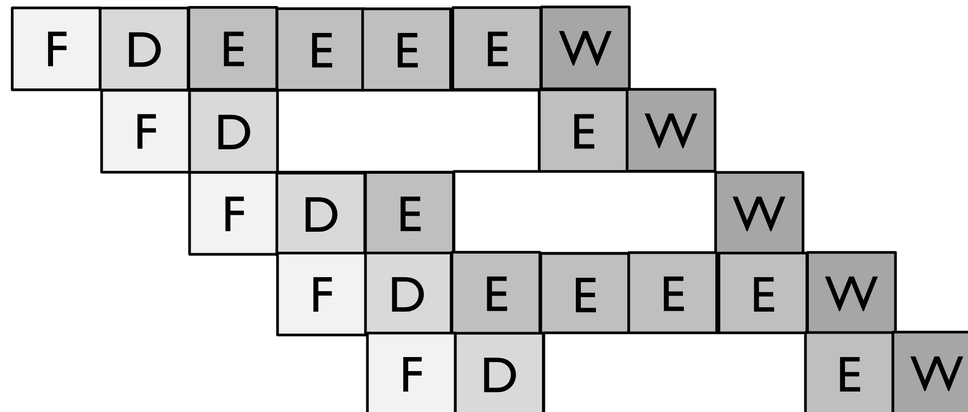
- In-order execution

r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5



- Out-of-order execution

r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5

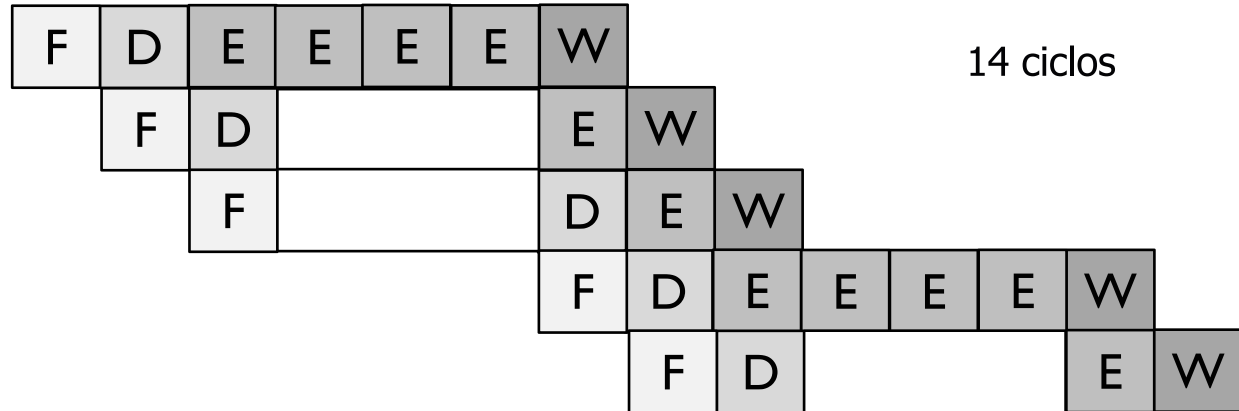


2.4 – Paralelismo

Problemas (hazards)

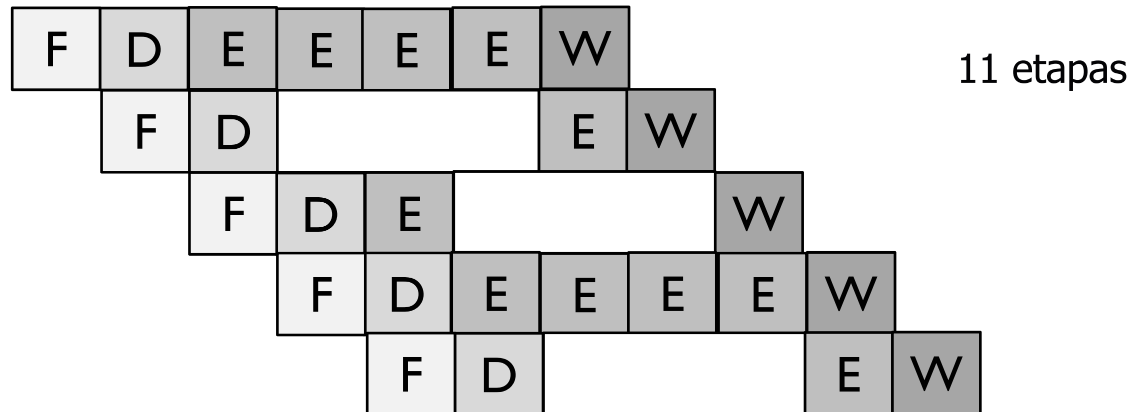
- In-order execution

r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5



- Out-of-order execution

r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5

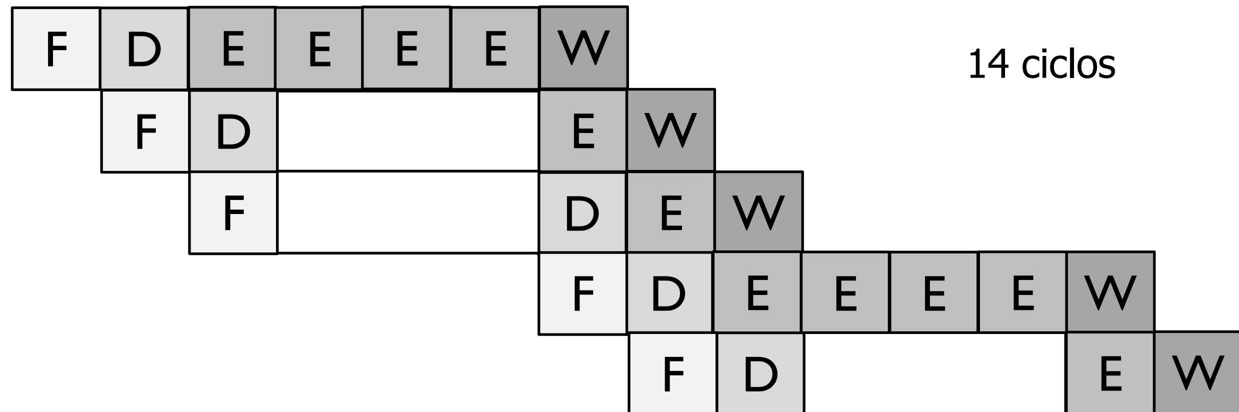


2.4 – Paralelismo

Problemas (hazards)

- In-order execution

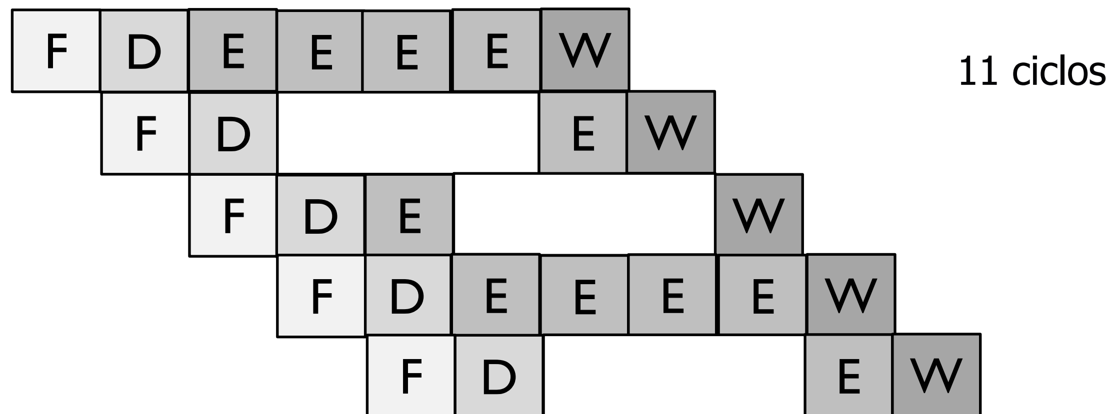
```
r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5
```



- Out-of-order execution

```
r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5
```

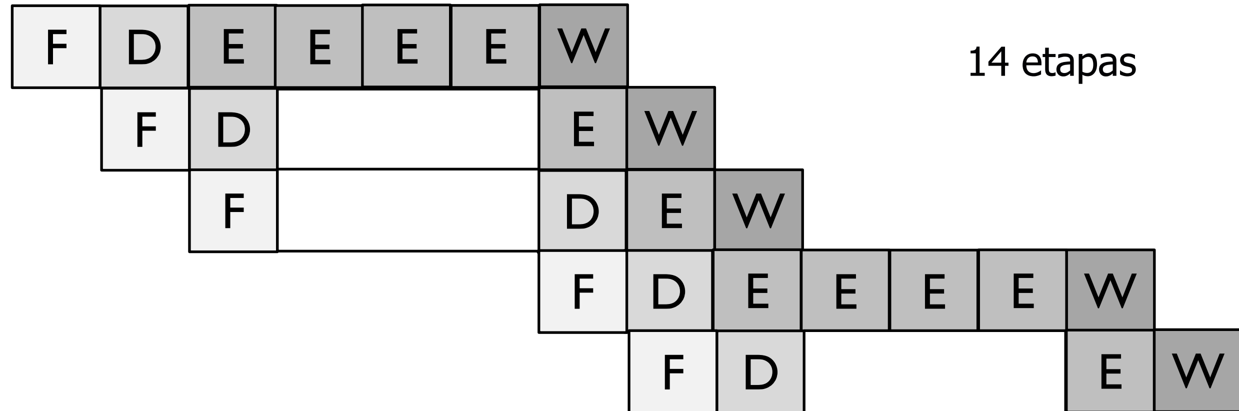
WAW?



2.4 – Paralelismo
Problemas (hazards)

- In-order execution

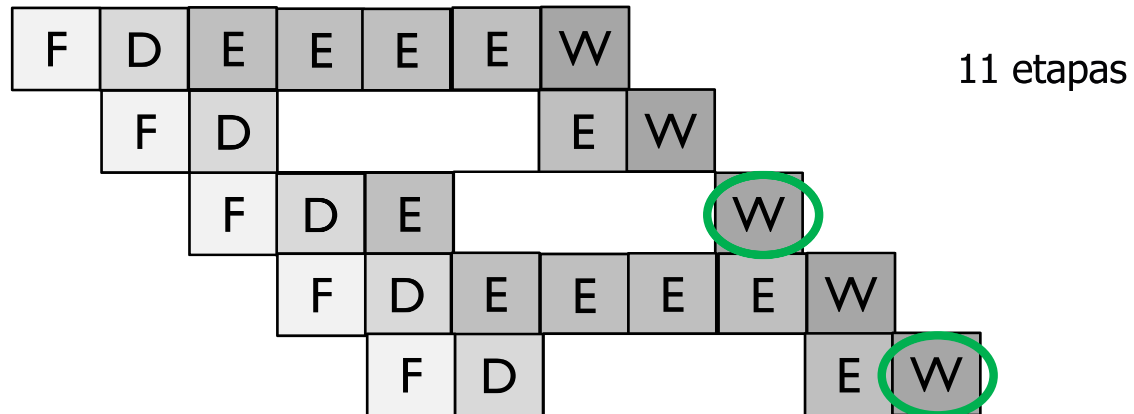
r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5



- Out-of-order execution

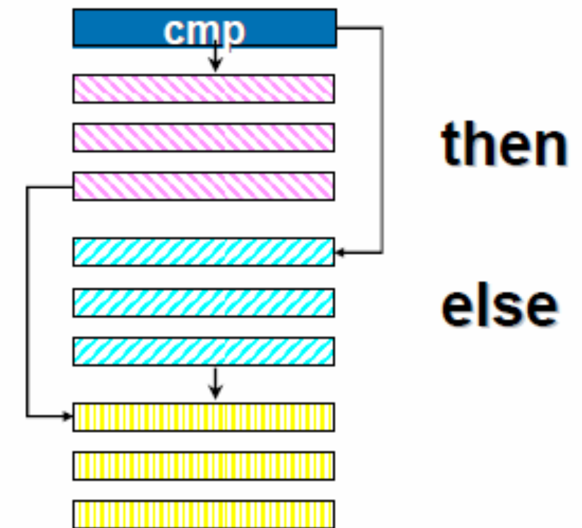
r3 = r1 * r2
r4 = r3 + r1
r9 = r6 + r7
r5 = r6 * r8
r9 = r3 + r5

WAW?



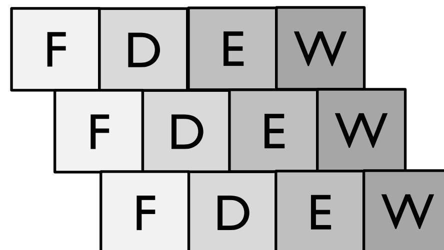
2.4 – Paralelismo
Problemas (hazards)

- Problema en las instrucciones de salto condicional
- Aprovechar los períodos de la ejecución de una instrucción en que no se ha de acceder a memoria para captar la siguiente instrucción. Si la dirección de la siguiente instrucción a captar es desconocida a causa de una instrucción de salto condicional, se captaría la siguiente instrucción secuencial.
- Cada vez que el código máquina encuentra un salto y se rompe la secuencia del código, todo el trabajo acumulado se pierde.



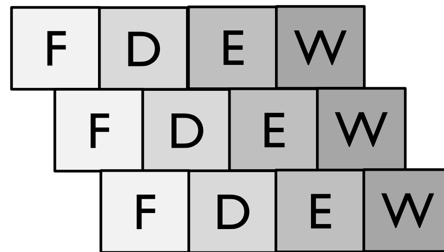
Problemas (hazards)

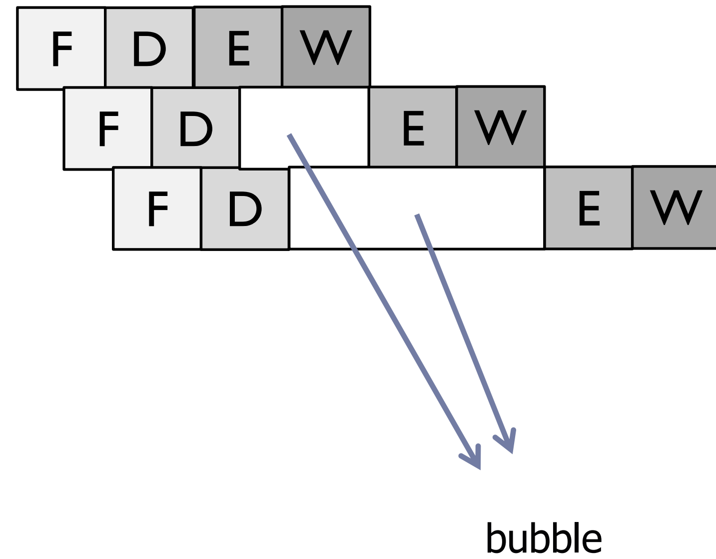
```
if (a == b)
    a = a + b
b = a + b
```



2.4 – Paralelismo
Problemas (hazards)

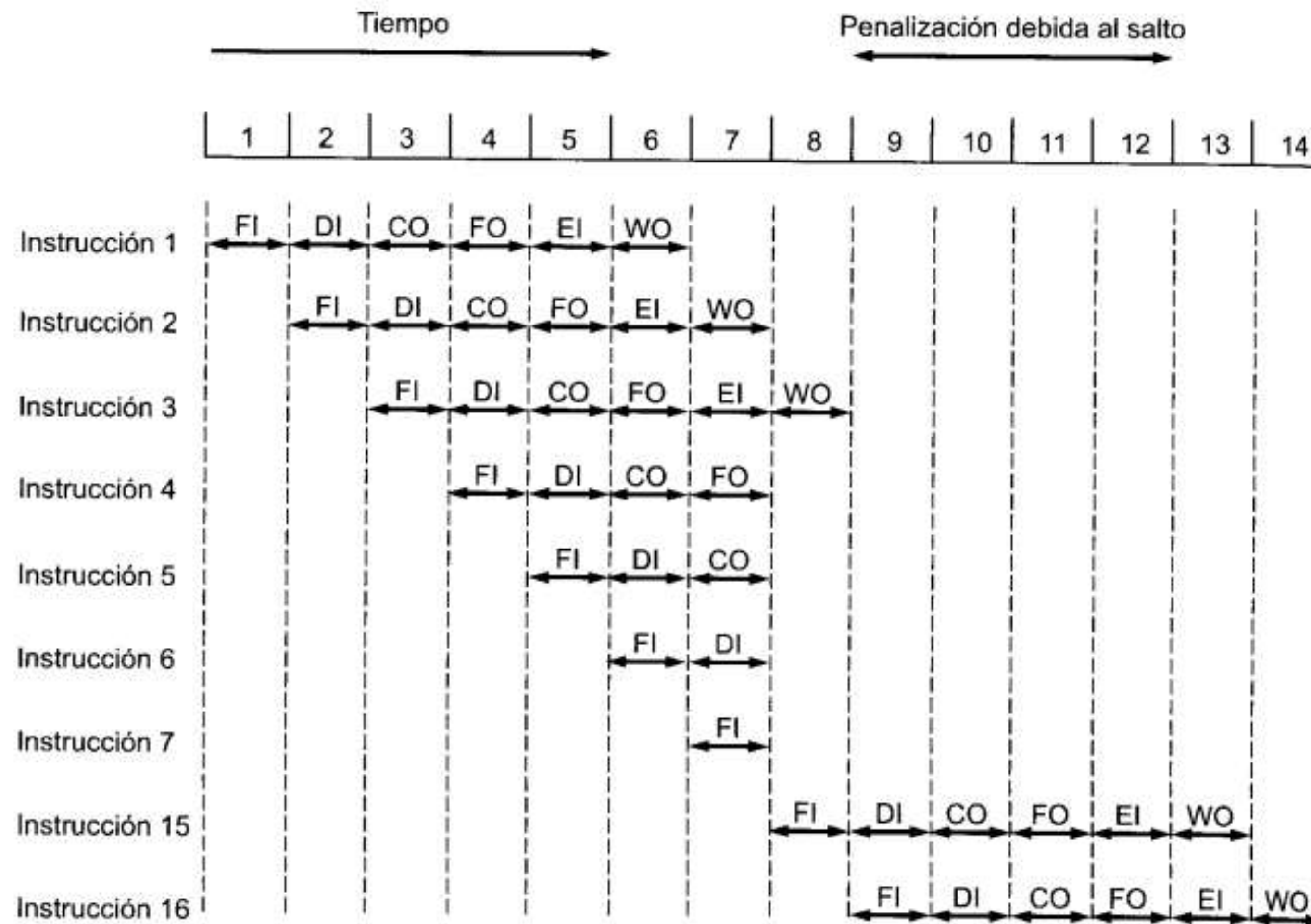
```
if (a == b)
    a = a + b
b = a + b
```





2.4 – Paralelismo
Problemas (hazards)

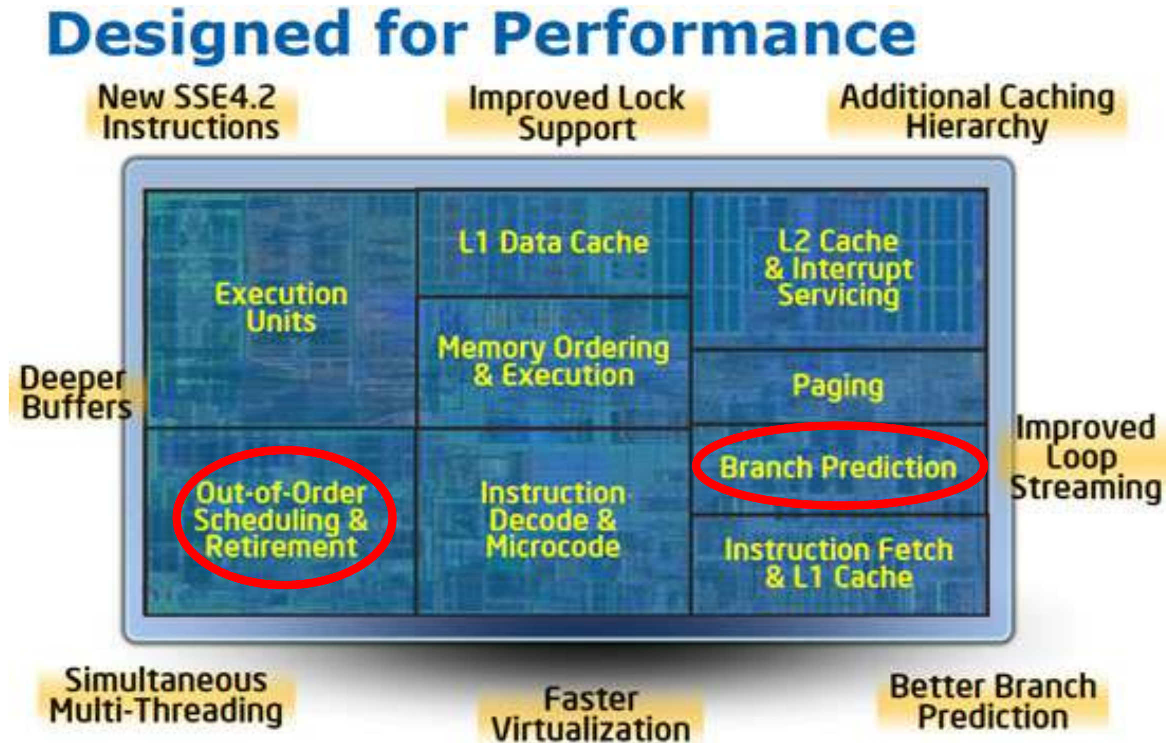
- Efecto de un salto condicional



2.4 – Paralelismo
Problemas (hazards)

- Posibles soluciones en caso de saltos
  - Detectarlo y esperar que se sepa donde se salta (bubbles)
  - Predecir si se salta o no para saber cual es la siguiente instrucción
    - verificar luego si la predicción ha sido correcta y, si no lo es, hacer un flush de todo lo que se ha hecho mal
    - muchos mecanismos basado en decisiones fijas (siempre cierto, siempre falso), análisis del pasado, profiling, etc.
  - Mientras se resuelva el salto, se ejecutan etapas de otros hilos (multithread)
  - Se empiezan las etapas de ambos y cuando se resuelve el saldo, se finaliza solo una de ellas

- Scheduler/Dispacher es el elemento que decide cuando y que instrucciones ejecutar



Intel Core i7 (quad core)

- Entre el 30 y 70% del área de un chip es típicamente cache

2.4 – Paralelismo

Arquitectura VLIW

- Los procesadores VLIW se caracterizan por tener instrucciones de gran tamaño. Esto se debe a que en cada instrucción se especifica el estado de todas las unidades funcionales del sistema.
- Es una forma de paralelismo que permite al compilador especificar directamente el nivel de paralelismo.
- La decisión de qué instrucciones se deben ejecutar simultáneamente corresponde al compilador (hardware más sencillo que el de un superescalar)
  - El compilador genera un paquete (bundle) de p instrucciones independientes y un "template" que indica al procesador cómo y cuando ejecutar las instrucciones
  - El compilador conoce el hardware y puede generar un bundle para ejecutar en paralelo múltiples instrucciones dependiendo del número de unidades ejecutivas disponibles

Traditional

1000:	op 1
1001:	op 2
1002:	op 3
1003:	op 4
1004:	op 5
1005:	op 6

VLIW

1000:	op 1	op 6	op 7	NOP
1001:	NOP	NOP	op 3	op 4
1002:	NOP	op 2	NOP	NOP
1003:	NOP	op 5	op 12	NOP
1004:	NOP	NOP	NOP	op 17
1005:	NOP	NOP	op 8	op 16

No VLIW

Inst.1 r1 = load(0x52fe)

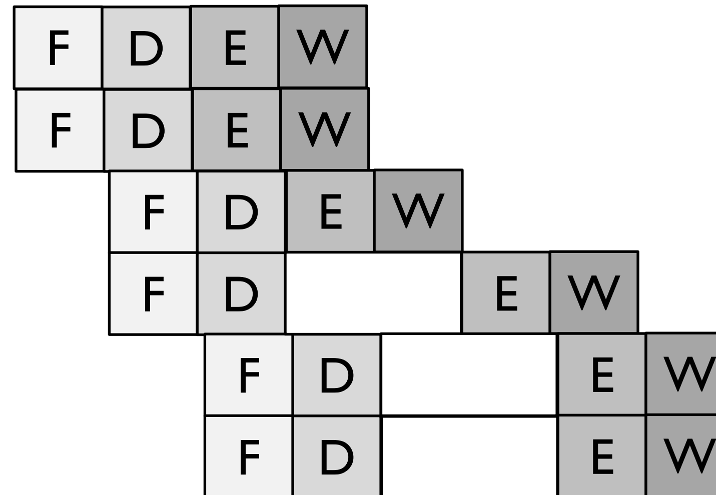
Inst.2 r2 = r3 + r4

Inst.3 r5 = r5 * 2

Inst.4 write(0xab43) = r5

Inst.5 r3 = r2 + r5

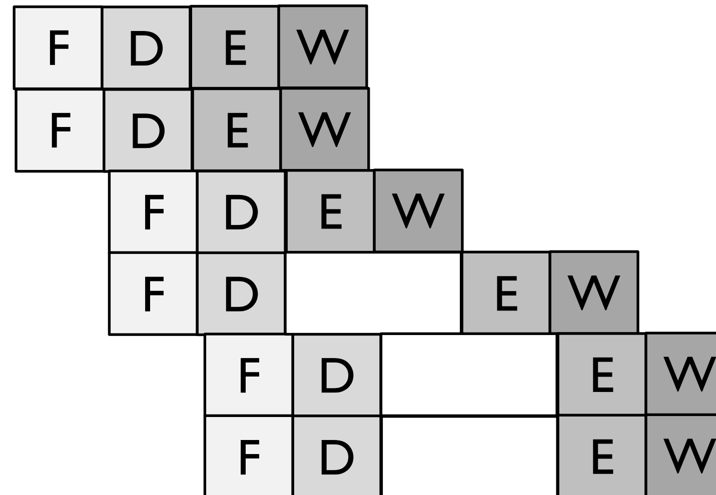
Inst.6 r6 = r2 mod 4



Arquitectura VLIW

No VLIW

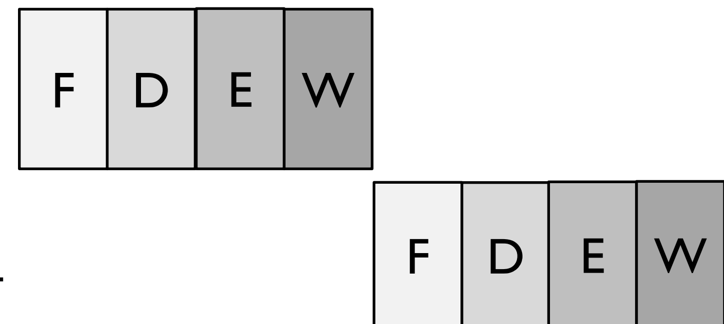
Inst.1 r1 = load(0x52fe)
Inst.2 r2 = r3 + r4
Inst.3 r5 = r5 * 2
Inst.4 write(0xab43) = r5
Inst.5 r3 = r2 + r5
Inst.6 r6 = r2 mod 4



VLIW

Inst.1
r1 = load(0x52fe) r2 = r3 + r4 r5 = r5 * 2

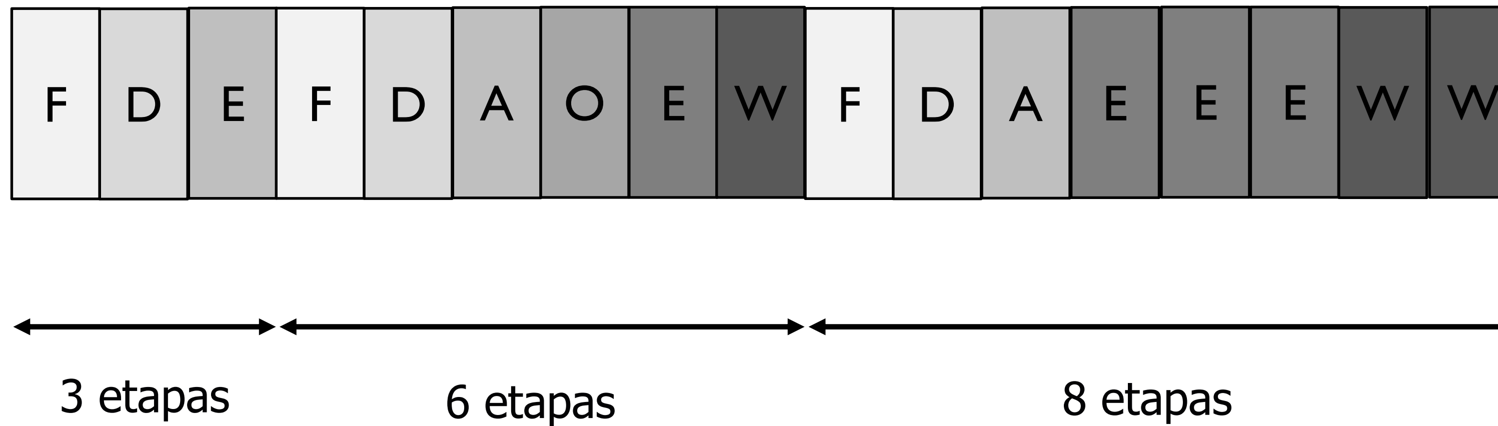
Inst.2
write(0xab43) = r5 r3 = r2 + r5 r6 = r2 mod 4



2.4 – Paralelismo

Arquitectura VLIW

- Hasta se pueden agrupar operaciones que no tienen determinadas etapas para reducir mas el tiempo de ejecución



2.4 – Paralelismo
Arquitectura VLIW ideal

- El tiempo de ejecución de un programa es producto de tres términos:

$$Te = Inst / p \times CPI \times Tc$$

donde

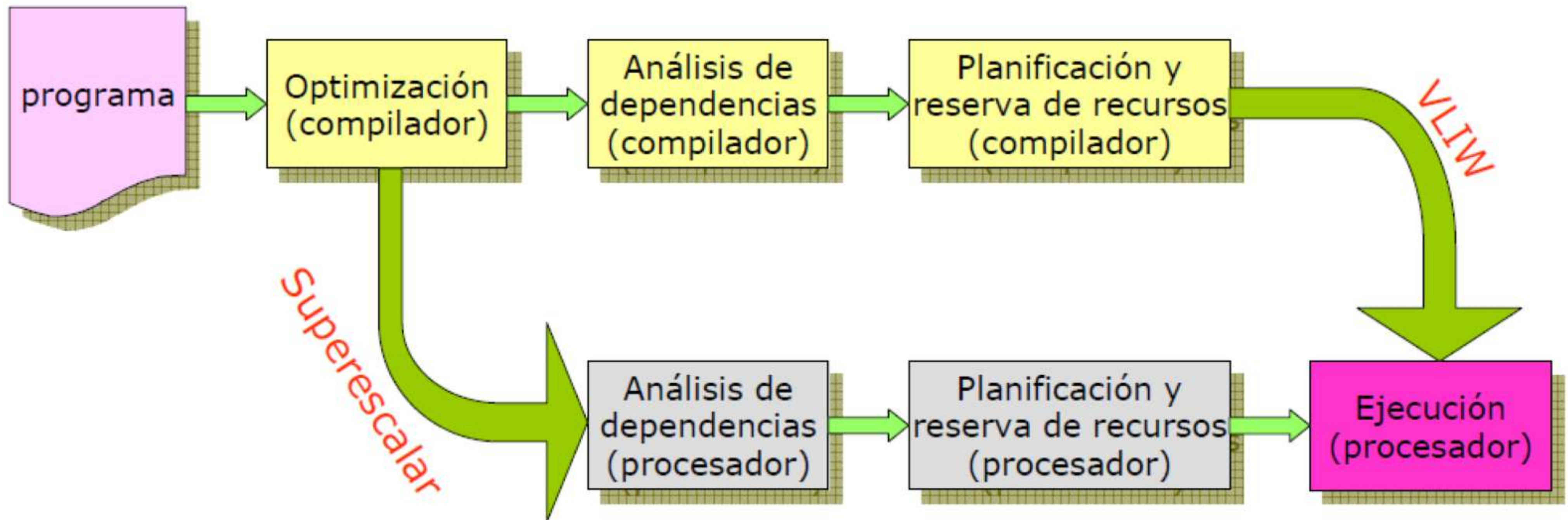
Inst: instrucciones/programa

CPI: número medio de ciclos por instrucción

Tc: tiempo de ciclo de reloj

p: instrucciones por bundle

2.4 – Paralelismo
VLIW vs. superescalar



2.4 – Paralelismo
VLIW vs. superescalar

- Superescalar
  - Planificación dinámica de la ejecución de las instrucciones
  - Hardware complejo
  - Código compacto
  - Adaptable a cualquier ISA
- VLIW
  - Planificación estática, no se necesita comprobar las dependencias durante la ejecución
  - Hardware simple, mayor frecuencia
  - Código mas grande
  - Compilador complejo, si no encuentran p instrucciones independientes que se puedan ejecutar en paralelo, hay que poner una nop (no operation)
  - ISA no compatible con otras arquitecturas

2.4 – Paralelismo

Arquitectura VLIW

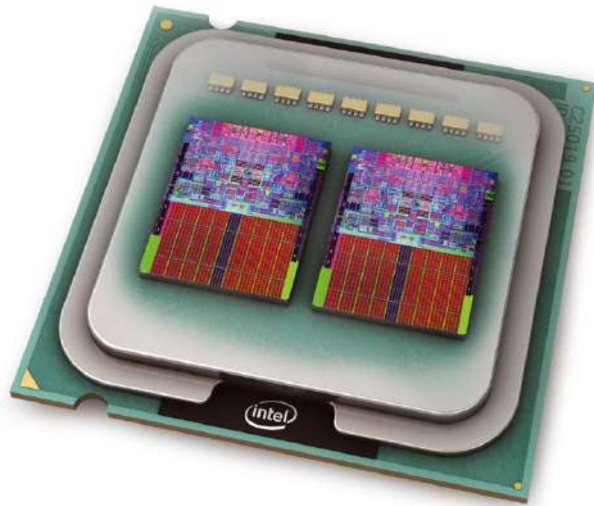
- Idea de principio de los 80
- Primer procesador VLIW fue Multiflow trace (1987) con 28 operaciones en paralelo por instrucción
- Intel i860 (1989), primer VLIW de 64 bits
- Intel Itanium IA-64 (2001)
  - No completamente VLIW pero basado en el mismo principio
  - EPIC (Explicitly Parallel Instruction Computing)
  - En este caso se crean bundle de instrucciones que pueden ser dependientes y ejecutarlas en lo que llaman EPIC operation
  - Realmente una mezcla entre superescalar y VLIW

2.4 – Paralelismo
Arquitectura VLIW

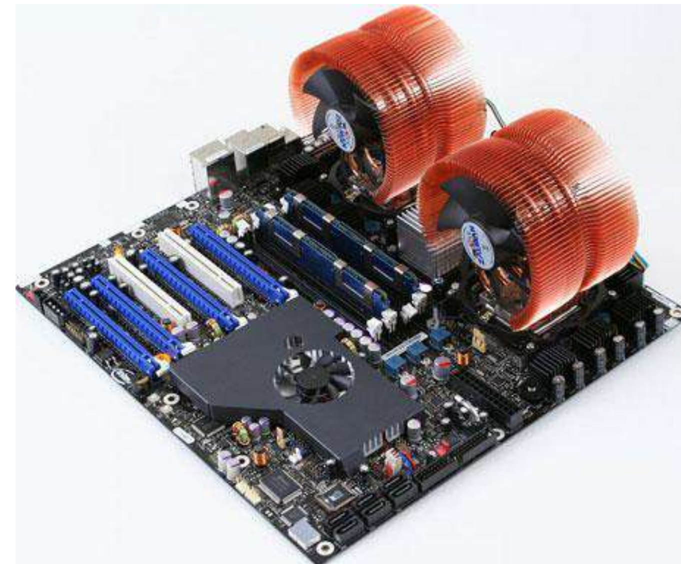
- Poco éxito en el mercado de ordenadores, éxito en otros como sistemas incrustados (coches, aviones, electrodomésticos, equipos médicos, etc.) y procesadores DSP
  - Fujitsu, TI, STM, TriMedia, etc.
- Debido principalmente a los muchos NOP que hay que insertar haciendo VLIW más lento que un superescalar
- Hoy en día los compiladores creados por VLIW se emplean para superescalares
  - Si el compilador ya ayuda a eliminar las inter-dependencias, entonces también es un beneficio para los superescalares

2.4 - Paralelismo
Multicore

- Llegados al limite de frecuencia, de disipación de calor, de segmentación y superescalaridad, la siguiente generación de procesadores implementan múltiples núcleos para incrementar el nivel de paralelismo y por lo tanto de velocidad de ejecución
- Procesador con mas de un núcleo donde cada núcleo es una unidad de procesado independiente capaz de ejecutar instrucciones completas
- No confundir con multi-CPU



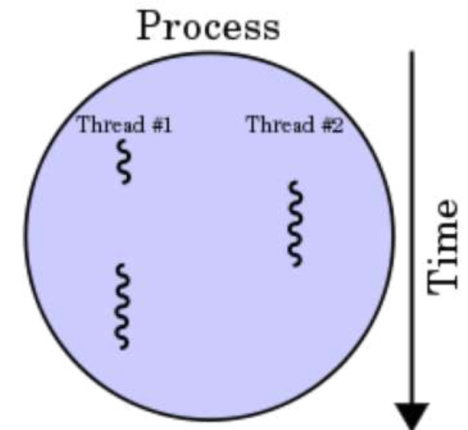
1 CPU, 2 nucleos



2 CPU

2.4 - Paralelismo
Multicore

- Proceso (process)
  - Una instancia de un programa en ejecución con los estados necesarios para permanecer en ejecución (una aplicación es generalmente un proceso)
  - Cada proceso es generalmente independiente
  - Cada proceso tiene su espacio de memoria donde se guardan las instrucciones y los datos
- Hilo (thread)
  - En un mismo proceso, se pueden definir hilos (partes de un proceso) distintos que pueden ejecutarse en determinados momentos según determinadas dependencias entre sí
  - Los hilos comparten los recursos del proceso, es decir tienen acceso a los mismos datos



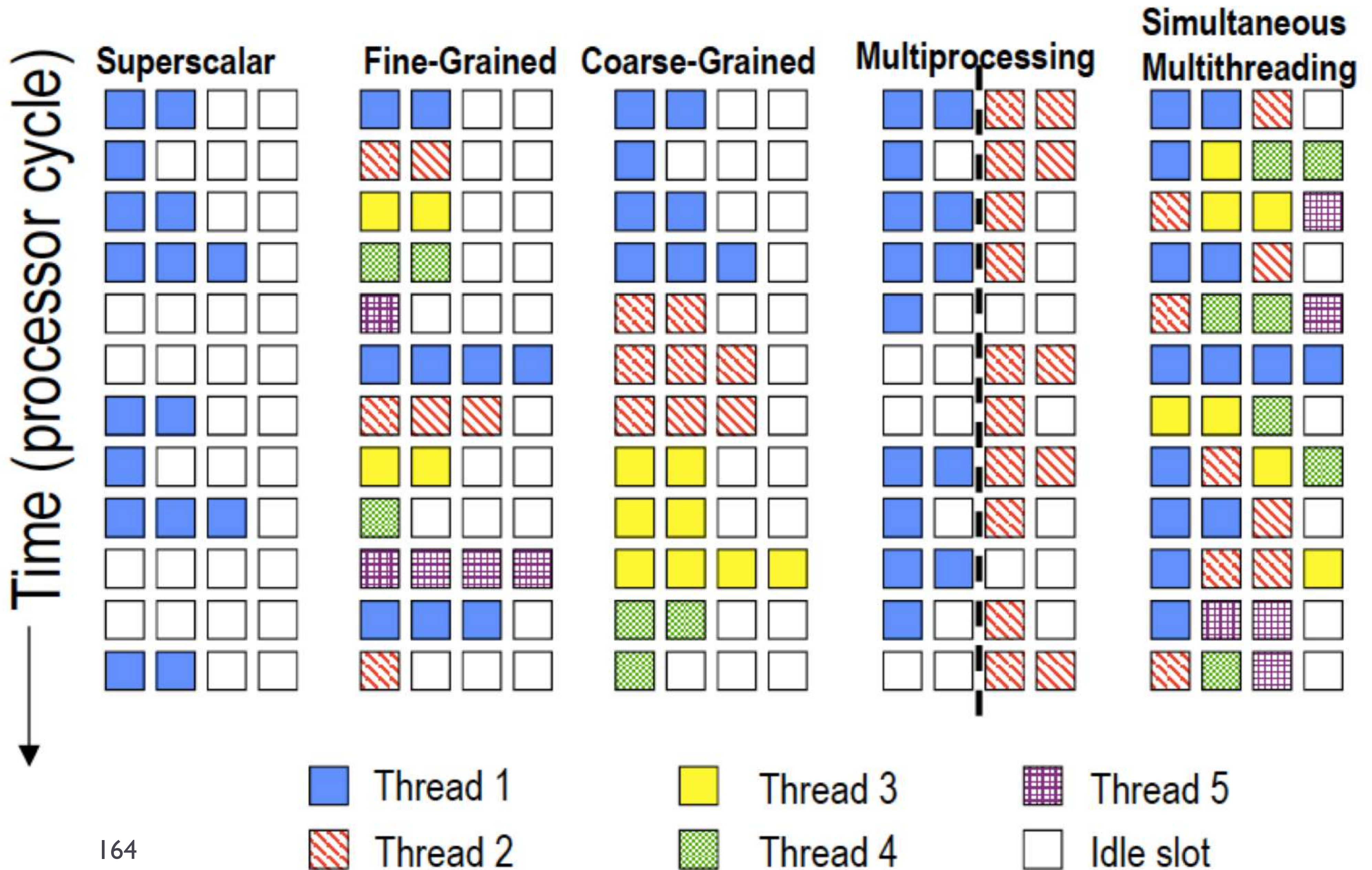
2.4 - Paralelismo
Multicore

- Multiproceso
  - Varios procesos que se ejecutan concurrentemente (o paralelamente si multicore) controlado por software
  - Se implementa a nivel de OS
- Multihilo (multithread)
  - Varias tareas que se ejecutan concurrentemente (o paralelamente si superescalar) dentro de un mismo proceso
  - Temporal multithreading (TMT)
  - Simultaneous multithreading (SMT)

2.4 - Paralelismo
Multicore

- Temporal multithreading (TMT)
  - La ejecución de cada thread es intercalada
  - Se ejecuta un thread durante un tiempo, luego otro durante otro tiempo, luego se vuelve al primero, etc.
  - Por ejemplo cuando un thread necesita un dato aún no disponible, se pasa a otro thread
- Simultaneous multithreading (SMT)
  - Se aprovecha la superescalaridad de un procesador para ejecutar los threads en paralelo (al mismo tiempo)

2.4 - Paralelismo
Multicore



- Scheduling/Planificador
- Dato un conjunto de tareas **J** donde cada j_i tiene una longitud l_i y un número de cores m, encuentra una manera para ejecutar todos las tareas en el menor tiempo posible sin que estas se solapen
- Problema de optimización NP-hard
  - (non-deterministic polynomial time)

2.4 - Paralelismo
Multicore

- Intel Pentium 4 (2002) fue el primer procesador para ordenadores sobremesa con SMT
  - 1 core
  - Dos SMT con rendimiento del 30% superior al con 1 thread
- IBM Power5 2 SMT por core (2, 4 o 8 core)
- IBM Power8 8 SMT por core (4, 6, 8, 10 o 12 core) (96 thread max)

- Oracle Corporation Sparc T3 8 fine-grained threads por core

- Sparc T4, Sparc T5, Sparc M5 y M6 8 fine-grained threads por core de los cuales 2 SMT

2.4 - Paralelismo

Multicore

- Ejemplo: cuenta los números primos hasta 100,000

```
#include <stdio.h>
#include <time.h>

void main() {
  clock_t start, end;
  double runTime;
  start = clock();
  int i, num = 1, primes = 0;
  int limit = 100000;

  while (num <= limit) {
    i = 2;
    while (i <= num) {
      if(num % i == 0)
        break;
      i++;
    }
    if (i == num)
      primes++;
    num++;
  }
  end = clock();
  runTime = (end - start) / (double) CLOCKS_PER_SEC;
  printf("Aquesta maquina ha calculat els %d nombres primers per sota de %d en %g segons\n", primes,limit, runTime);
}
```

2.4 - Paralelismo
Multicore

- Ejemplo: cuenta los números primos hasta 100,000

```
void main() {
  double start, end;
  double runTime;
  start = omp_get_wtime();
  int num = 1,primes = 0;

  int limit = 100000;

  #pragma omp parallel for schedule(dynamic) reduction(+ : primes)
  for (num = 1; num <= limit; num++) {
    int i = 2;
    while(i <= num) {
      if(num % i == 0)
        break;
      i++;
    }
    if(i == num)
      primes++;
  }

  end = omp_get_wtime();
  runTime = end - start;
  printf("Aquesta maquina ha calculat els %d nombres primers per sota de %d en %g segons\n",primes,limit,runTime);
}
```

omp parallel for…
Usado para dividir iteraciones de un bucle entre diferentes hilos

OpenMP es una API para la programación multihilo de memoria compartida.
Permite añadir paralelismo a los programas escritos en C, C++ y Fortran
Conjunto de directivas de compilador, rutinas de biblioteca, y variables de entorno que influyen en el comportamiento en tiempo de ejecución.

2.4 - Paralelismo

Multicore

- Ejemplo: resultado usando un PC con 3 cores

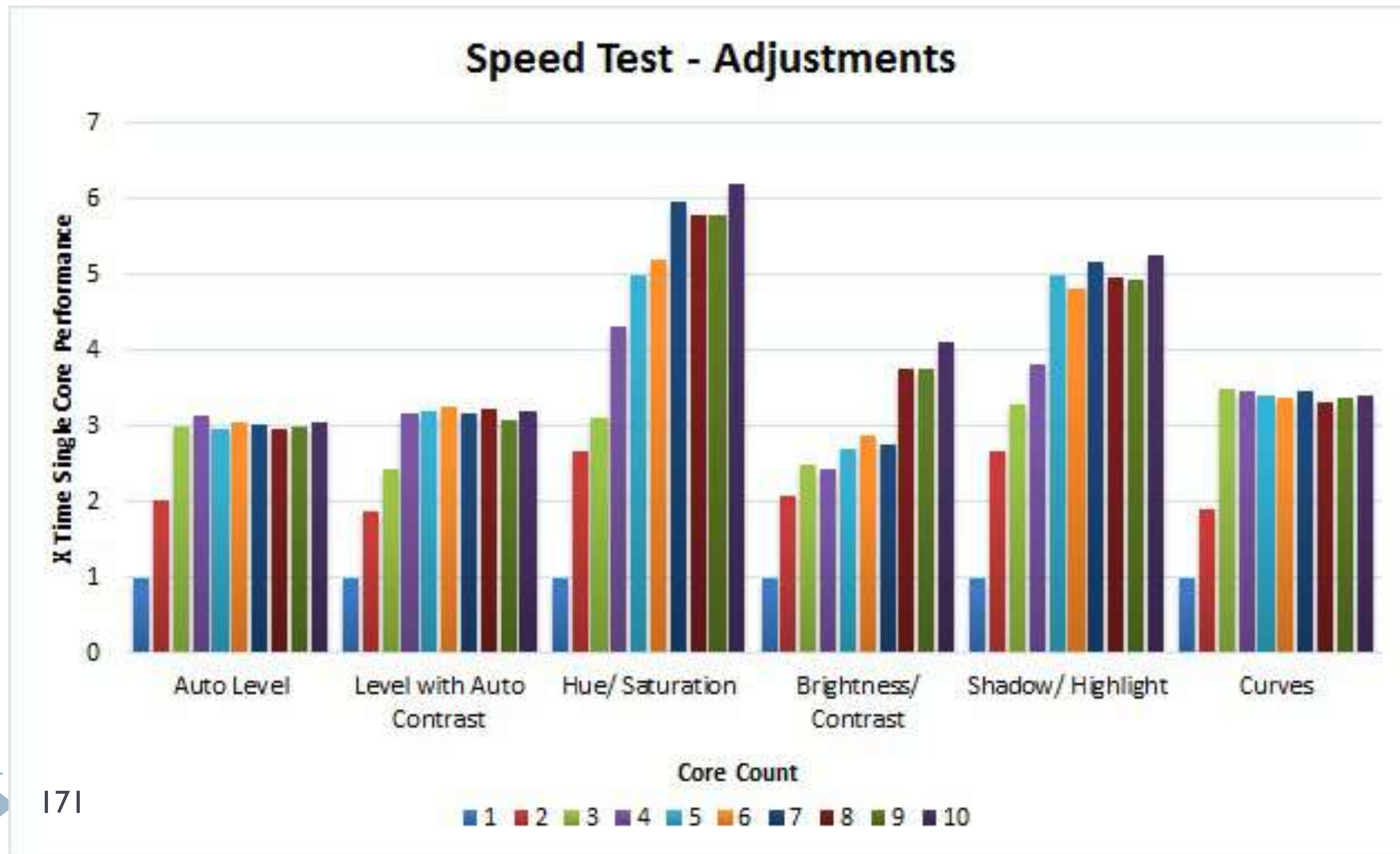
```
ac@ac-server:~$ ./primers
Aquesta maquina ha calculat els 9592 nombres primers per sota de 100000 en 1.29 segons
ac@ac-server:~$
ac@ac-server:~$
ac@ac-server:~$
ac@ac-server:~$ ./primers2
Aquesta maquina ha calculat els 9592 nombres primers per sota de 100000 en 0.500744 segons
ac@ac-server:~$
```

2.4 - Paralelismo
Multicore

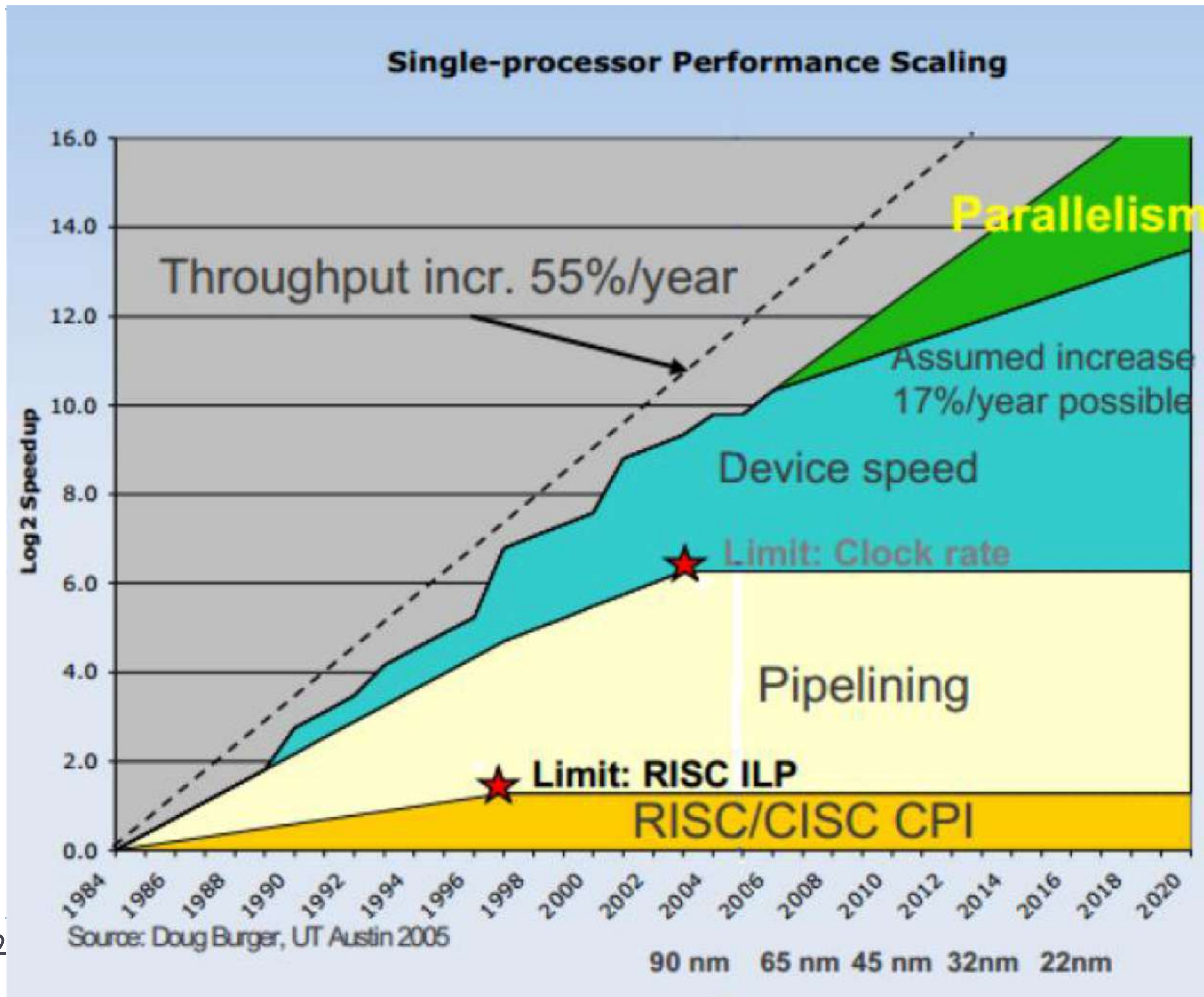
- Ventajas
  - Estos procesadores pueden ejecutar varias aplicaciones al mismo tiempo
  - Señales entre distintos núcleos se propagan más rápido: menores distancias
  - Mejora calidad de transmisión: menos fallos en comunicación (evitando reenvíos de información)
  - Menor consumo de energía
  - Si la aplicación demanda mucho ancho de banda en memoria, la mejora de pasar de uno a dos núcleos es entre 30% y 70% (ver desventaja *)
  - Si la aplicación no demanda mucho ancho de banda en memoria, se puede alcanzar una mejora del 90% (ver desventaja *)
- Desventajas
  - La habilidad de mejorar el rendimiento depende de la capacidad de la aplicación de crear y explotar hilos (threads). Si se ejecuta una aplicación que no sea paralelizable (no se pueda descomponer en hilos) no se aprovecha el potencial de estos procesadores
  - Los núcleos comparten y compiten por el acceso al bus de sistema y de memoria: conflictos (*)

2.4 - Paralelismo
Rendimiento

- Comparación de speedup usando múltiples núcleos para diferentes ajuste en Adobe Photoshop



2.5 - Resumen



Temario

- Tema 1. Introducción
- Tema 2. El microprocesador
  - Introducción
  - Instruction Set Architecture
  - Arquitectura interna
  - Paralelismo
  - Ejercicios
- Tema 3. Memoria
- Tema 4. Dispositivos de E/S y buses
- Tema 5. DataCenters y modelos de comunicación

2.5 – Ejercicios
Recordatorio

- CPI (Clock cycles per instruction)
  - Es un valor medio que indica el número medio de ciclos para ejecutar una instrucción
  - Depende de varios factores
    - Diferentes instrucciones tardan diferentes números de ciclos para ejecutarse
    - Una adición puede tardar 1 ciclo y una división 10
    - De la frecuencia con la que se ejecuta cada tipo de instrucción
  - A veces se usa el reciproco IPC (Instruccions per clock cycle)
- Ejemplo
  - Un programa ejecuta un igual numero de operaciones con enteros, reales y de acceso a la memoria
  - CPI por tipo de instrucción
    - Entero = 1
    - Real = 3
    - Acceso memoria = 2
  - CPI = 1 * 33% + 3 * 33% + 2 * 33% = 2

2.5 – Ejercicios
Recordatorio

- Prestaciones de una CPU para ejecutar instrucciones

  - Latency (L): segundos / instrucción

    → segundos / instrucción = (ciclos / instrucción) * (segundos / ciclo)

    = CPI / Frecuencia

  - Throughput (T): instrucciones / segundo

    → instrucciones / segundo = (instrucciones / ciclo) * (ciclos / segundo)

    = Frecuencia / CPI

2.5 – Ejercicios
Recordatorio

- El tiempo de ejecución (Le) de un programa es producto de tres términos:

$$Le = Inst \times CPI \times Tc$$

donde

Inst: instrucciones/programa

CPI: número medio de ciclos por instrucción

Tc: tiempo de ciclo de reloj

2.5 – Ejercicios
Recordatorio

- El tiempo de ejecución (Le) de un programa es producto de tres términos:

$$Le = Inst \times CPI \times Tc$$

donde

Inst: instrucciones/programa

CPI: número medio de ciclos por instrucción

Tc: tiempo de ciclo de reloj

- Por ejemplo
  - 100.000 instrucciones
  - 1 GHz
  - CPI = 2

$$Le = 100.000 * 2 * (1 / 10^9) = 0,2 \text{ ms}$$

2.5 – Ejercicios

Speedup

- El speedup es la comparación entre dos sistemas y determina cuanto las prestaciones de un sistema A son mejores que de un sistema B

$$\text{Speedup} = S = \frac{B}{A}$$

- En el caso de latency, el speedup S_L de A respecto a B es

$$\text{Speedup} = S_L = \frac{L_B}{L_A}$$

- En el caso de throughput, el speedup S_T de A respecto a B es

$$\text{Speedup} = S_T = \frac{T_A}{T_B}$$

2.5 – Ejercicios
Speedup

- El speedup es la comparación entre dos sistemas y determina cuanto las prestaciones de un sistema A son mejores que de un sistema B

$$\text{Speedup} = S = \frac{B}{A}$$

- En el caso de latency, el speedup S_L de A respecto a B es

$$\text{Speedup} = S_L = \frac{L_B}{L_A}$$

- En el caso de throughput, el speedup S_T de A respecto a B es

$$\text{Speedup} = S_T = \frac{T_A}{T_B}$$

¿Throughput al revés que latency? ¿Un error?

2.5 – Ejercicios
Speedup

- El speedup es la comparación entre dos sistemas y determina cuanto las prestaciones de un sistema A son mejores que de un sistema B

$$\text{Speedup} = S = \frac{B}{A}$$

- En el caso de latency, el speedup S_L de A respecto a B es

$$\text{Speedup} = S_L = \frac{L_B}{L_A}$$

- En el caso de throughput, el speedup S_T de A respecto a B es

$$\text{Speedup} = S_T = \frac{T_A}{T_B}$$

¿Throughput al revés que latency? ¿Un error?
No, porque Latency cuanto más bajo mejor, Throughput cuanto más alto mejor

2.5 – Ejercicios
Speedup - ejemplo

- Un sistema A tarda 10s en ejecutar un programa y el B tarda 15 s.
- ¿cuál es mejor?

2.5 – Ejercicios
Speedup - ejemplo

- Un sistema A tarda 10s en ejecutar un programa y el B tarda 15 s.
- ¿cuál es mejor?
- ¿cuál es la mejora (speedup) de A respecto a B?

2.5 – Ejercicios

Speedup - ejemplo

- Un sistema A tarda 10s en ejecutar un programa y el B tarda 15 s.
- ¿cuál es mejor?
- ¿cuál es la mejora (speedup) de A respecto a B?

$$\text{Speedup} = S_L = \frac{L_B}{L_A} = \frac{15}{10} = 1.5$$

2.5 – Ejercicios

Speedup - ejemplo

- Un sistema A tarda 10s en ejecutar un programa y el B tarda 15 s.
- ¿cuál es mejor?
- ¿cuál es la mejora (speedup) de A respecto a B?

$$\text{Speedup} = S_L = \frac{L_B}{L_A} = \frac{15}{10} = 1.5$$

- ¿cuál es la mejora (speedup) de B respecto a A?

2.5 – Ejercicios

Speedup - ejemplo

- Un sistema A tarda 10s en ejecutar un programa y el B tarda 15 s.
- ¿cuál es mejor?
- ¿cuál es la mejora (speedup) de A respecto a B?

$$\text{Speedup} = S_L = \frac{L_B}{L_A} = \frac{15}{10} = 1.5$$

- ¿cuál es la mejora (speedup) de B respecto a A?

$$\text{Speedup} = S_L = \frac{L_A}{L_B} = \frac{10}{15} = 0.66$$

- Es peor! (speedup menor que 1)

2.5 – Ejercicios
Speedup - ejemplo

- Un sistema A tiene un throughput de 2000 i/s y el B de 500 i/s.
- ¿cuál es mejor?

2.5 – Ejercicios

Speedup - ejemplo

- Un sistema A tiene un throughput de 2000 i/s y el B de 500 i/s.
- ¿cuál es mejor?
- ¿cuál es la mejora (speedup) de A respecto a B?

2.5 – Ejercicios
Speedup - ejemplo

- Un sistema A tiene un throughput de 2000 i/s y el B de 500 i/s.
- ¿cuál es mejor?
- ¿cuál es la mejora (speedup) de A respecto a B?

$$\text{Speedup} = S_L = \frac{T_A}{T_B} = \frac{2000}{500} = 4$$

2.5 – Ejercicios
Speedup

- Si se dice que A es X% más rápido que B cuando

$$X = \left(\frac{L_B - L_A}{L_A}\right) \times 100$$

- Por ejemplo, si L_A es 10 s y L_B es 15 s

$$X = \left(\frac{L_B - L_A}{L_A}\right) \times 100 = \left(\frac{15 - 10}{10}\right) \times 100 = 50\%$$

- A es 50% más rápido que B

2.5 – Ejercicios

Speedup

- Por ejemplo, si L_A es 10 s y L_B es 15 s

$$X = \left(\frac{L_B - L_A}{L_A}\right) \times 100 = \left(\frac{15-10}{10}\right) \times 100 = 50\%$$

- A es 50% más rápido que B

- ¿y cuanto es B más lento que A?

2.5 – Ejercicios
Speedup

- Por ejemplo, si L_A es 10 s y L_B es 15 s

$$X = \left(\frac{L_B - L_A}{L_A}\right) \times 100 = \left(\frac{15-10}{10}\right) \times 100 = 50\%$$

- A es 50% más rápido que B

- ¿y cuanto es B más lento que A?

Sugerencia, B no es 50% más lento que A

2.5 – Ejercicios

Speedup

- Por ejemplo, si L_A es 10 s y L_B es 15 s

$$X = \left(\frac{L_B - L_A}{L_A}\right) \times 100 = \left(\frac{15-10}{10}\right) \times 100 = 50\%$$

- A es 50% más rápido que B

- ¿y cuanto es B más lento que A?

$$X = \left(\frac{L_B - L_A}{L_B}\right) \times 100 = \left(\frac{15-10}{15}\right) \times 100 = 33.3\%$$

- B es 33% más lento que A

2.5 – Ejercicios
Coste y throughput

- También se puede considerar el coste y determinar la relación entre rendimiento y coste
- Si C_A es el coste de A y C_B es el coste de B, interesa que el aumento del throughput de un sistema sea a menor coste posible
- Si T_A es el throughput de A y T_B es el throughput de B, entonces interesa comparar

$$\frac{T_B}{C_B} \; vs \; \frac{T_A}{C_A}$$

- Por ejemplo
  - throughput de A es 500 y el coste es 100
  - throughput de B es 2000 y el coste es 200
- ¿Cual es mejor?

2.5 – Ejercicios

Coste y throughput

- También se puede considerar el coste y determinar la relación entre rendimiento y coste
- Si C_A es el coste de A y C_B es el coste de B, interesa que el aumento del throughput de un sistema sea a menor coste posible
- Si T_A es el throughput de A y T_B es el throughput de B, entonces interesa comparar

$$\frac{T_B}{C_B} \ vs \ \frac{T_A}{C_A}$$

- Por ejemplo
  - throughput de A es 500 y el coste es 100
  - throughput de B es 2000 y el coste es 200
- ¿Cual es mejor?

$$\frac{T_B}{C_B} = 10 \qquad \frac{T_A}{C_A} = 5$$

2.5 – Ejercicios
Coste y throughput

- Por ejemplo
  - throughput de A es 500 y el coste es 100
  - throughput de B es 2000 y el coste es 200
- ¿Cual es mejor?

$$\frac{T_B}{C_B} = 10 \qquad \frac{T_A}{C_A} = 5$$

- ¿Cuanto es mejor?

$$\text{X} = \left(\frac{10-5}{5}\right)\times 100 = 100\%$$

- B es un 100% mejor que A (es decir el doble)

2.5 – Ejercicios
Rendimiento del paralelismo

- El tiempo de ejecución de un programa es producto de tres términos:

$$L_e = Inst \times CPI \times T_c$$

- Si se usa segmentación y k son las etapas, en el caso ideal

$$L_e^{seg} = Inst \times CPI \times {}^{T_c}/_{k}$$

El speedup es por lo tanto

$$S_L = \frac{L_e}{L_e^{seg}} = \frac{Inst \times CPI \times T_c}{Inst \times CPI \times {}^{T_c}/_{k}} = k$$

- Si se usa supersegmentación y n es su grado, en el caso ideal

$$L_e^{sup} = Inst \times CPI \times {}^{T_c}/_{kn}$$

El speedup es por lo tanto

$$S_L = \frac{L_e}{L_e^{sup}} = \frac{Inst \times CPI \times T_c}{Inst \times CPI \times {}^{T_c}/_{kn}} = kn$$

2.5 – Ejercicios
Rendimiento del paralelismo

- El tiempo de ejecución de un programa es producto de tres términos:

$$L_e = Inst \times CPI \times T_c$$

- Si se usa superescalaridad y m es su grado, en el caso ideal

$$L_e^{esc} = Inst \times {}^{CPI}/_{m} \times T_c$$

El speedup es por lo tanto

$$S_L = \frac{L_e}{L_e^{esc}} = \frac{Inst \times CPI \times T_c}{Inst \times {}^{CPI}/_{m} \times T_c} = m$$

- Si se usa VLIW y p es el número de instrucciones por bundle, en el caso ideal

$$L_e^{vliw} = {}^{Inst}/_{p} \times CPI \times T_c$$

El speedup es por lo tanto

$$S_L = \frac{L_e}{L_e^{vliw}} = \frac{Inst \times CPI \times T_c}{{}^{Inst}/_{m} \times CPI \times T_c} = p$$

2.5 – Ejercicios
Rendimiento del paralelismo

- Sin embargo, la mejora solo afecta al porcentaje de código que se pueda paralelizar
- En caso de saltos o instrucciones dependientes entre ellas, no valen estas formulas

- Se define la ley de Amdahl que indica el efecto del paralelismo en el speedup

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}}$$

donde

S_L es el speedup real en tiempo de ejecución

S_L^I es el speedup ideal en tiempo de ejecución

p es la fracción del tiempo de ejecución que se puede paralelizar

2.5 – Ejercicios
Rendimiento del paralelismo

- Si p es la fracción que se puede paralelizar idealmente con un speedup S_L^I y L^{no} es el tiempo de ejecución sin paralelismo, el tiempo de ejecución real con paralelismo L^p de esta parte paralalizable sería

$$L^p = \frac{p}{S_L^I} L^{no}$$

- La fracción no paralelizable es $1 - p$ y siendo L^{no} es el tiempo de ejecución sin paralelismo, el tiempo total de ejecución con paralelismo es

$$L = (1-p)L^{no} + \frac{p}{S_L^I} L^{no}$$

- Por lo tanto el speedup S_L real es la relación entre el tiempo de ejecución sin paralelismo y el tiempo de ejecución con paralelismo

$$S_L = \frac{L^{no}}{L} = \frac{L^{no}}{(1-p)L^{no} + \frac{p}{S_L^I} L^{no}} = \frac{1}{1 - p + \frac{p}{S_L^I}}$$

2.5 – Ejercicios

Ley de Amdahl - ejemplo

- El 10% de ejecución de un programa consiste en operaciones en coma flotante. Para estas operaciones se consigue un paralelismo que permite reducir su tiempo de ejecución a la mitad
- ¿Que mejora se obtiene?

2.5 – Ejercicios

Ley de Amdahl - ejemplo

- El 10% de ejecución de un programa consiste en operaciones en coma flotante. Para estas operaciones se consigue un paralelismo que permite reducir su tiempo de ejecución a la mitad
- ¿Que mejora se obtiene?

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.1 + \frac{0.1}{2}} = 1.053$$

- ¿En porcentaje cuanto es?

2.5 – Ejercicios

Ley de Amdahl - ejemplo

- El 10% de ejecución de un programa consiste en operaciones en coma flotante. Para estas operaciones se consigue un paralelismo que permite reducir su tiempo de ejecución a la mitad
- ¿Que mejora se obtiene?

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.1 + \frac{0.1}{2}} = 1.053$$

- ¿En porcentaje cuanto es?

La mejora es del 5.3%

2.5 – Ejercicios

Ley de Amdahl - ejemplo

- Para mejorar la velocidad de una aplicación, se ejecuta el 90% del trabajo sobre 100 hilos en paralelo. El 10% restante no admite la ejecución en paralelo.
- ¿Que mejora se obtiene?

2.5 – Ejercicios

Ley de Amdahl - ejemplo

- Para mejorar la velocidad de una aplicación, se ejecuta el 90% del trabajo sobre 100 hilos en paralelo. El 10% restante no admite la ejecución en paralelo.
- ¿Que mejora se obtiene?

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.9 + \frac{0.9}{100}} = 9.17$$

- Se ejecuta 9.17 veces más rápido

2.5 – Ejercicios

Ley de Amdahl - ejemplo

- Para mejorar la velocidad de una aplicación, se ejecuta el 90% del trabajo sobre 100 hilos en paralelo. El 10% restante no admite la ejecución en paralelo.
- ¿Que mejora se obtiene?

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.9 + \frac{0.9}{100}} = 9.17$$

- Se ejecuta 9.17 veces más rápido

  La mejora es del ???%

Ley de Amdahl - ejemplo

- Para mejorar la velocidad de una aplicación, se ejecuta el 90% del trabajo sobre 100 hilos en paralelo. El 10% restante no admite la ejecución en paralelo.
- ¿Que mejora se obtiene?

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.9 + \frac{0.9}{100}} = 9.17$$

- Se ejecuta 9.17 veces más rápido

  La mejora es del 817%

2.5 – Ejercicios

Ley de Amdahl - ejemplo

- Se quiere ejecutar un programa con N núcleos. El 30% del tiempo se ejecuta código que no se puede paralelizar.
- Calcular la ganancia para N=2, 3, 4, 5, ∞.

2.5 – Ejercicios

Ley de Amdahl - ejemplo

- Se quiere ejecutar un programa con N núcleos. El 30% del tiempo se ejecuta código que no se puede paralelizar.
- Calcular la ganancia para N=2, 3, 4, 5, ∞.

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.7 + \frac{0.7}{N}}$$

- $N = 2, S_L = 1.54$

2.5 – Ejercicios

Ley de Amdahl - ejemplo

- Se quiere ejecutar un programa con N núcleos. El 30% del tiempo se ejecuta código que no se puede paralelizar.
- Calcular la ganancia para N=2, 3, 4, 5, ∞.

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.7 + \frac{0.7}{N}}$$

- $N = 2$, $S_L = 1.54$
- $N = 3$, $S_L = 1.85$

2.5 – Ejercicios

Ley de Amdahl - ejemplo

- Se quiere ejecutar un programa con N núcleos. El 30% del tiempo se ejecuta código que no se puede paralelizar.
- Calcular la ganancia para N=2, 3, 4, 5, ∞.

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.7 + \frac{0.7}{N}}$$

- $N = 2, S_L = 1.54$
- $N = 3, S_L = 1.85$
- $N = 4, S_L = 2.1$
- $N = 5, S_L = 2.3$

2.5 – Ejercicios

Ley de Amdahl - ejemplo

- Se quiere ejecutar un programa con N núcleos. El 30% del tiempo se ejecuta código que no se puede paralelizar.
- Calcular la ganancia para N=2, 3, 4, 5, ∞.

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.7 + \frac{0.7}{N}}$$

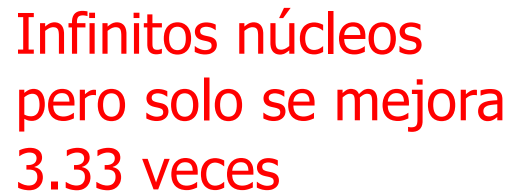
- $N = 2, S_L = 1.54$
- $N = 3, S_L = 1.85$
- $N = 4, S_L = 2.1$
- $N = 5, S_L = 2.3$
- $N = \infty, S_L = 3.33$

2.5 – Ejercicios
Ley de Amdahl - ejemplo

- Se quiere ejecutar un programa con N núcleos. El 30% del tiempo se ejecuta código que no se puede paralelizar.
- Calcular la ganancia para N=2, 3, 4, 5, ∞.

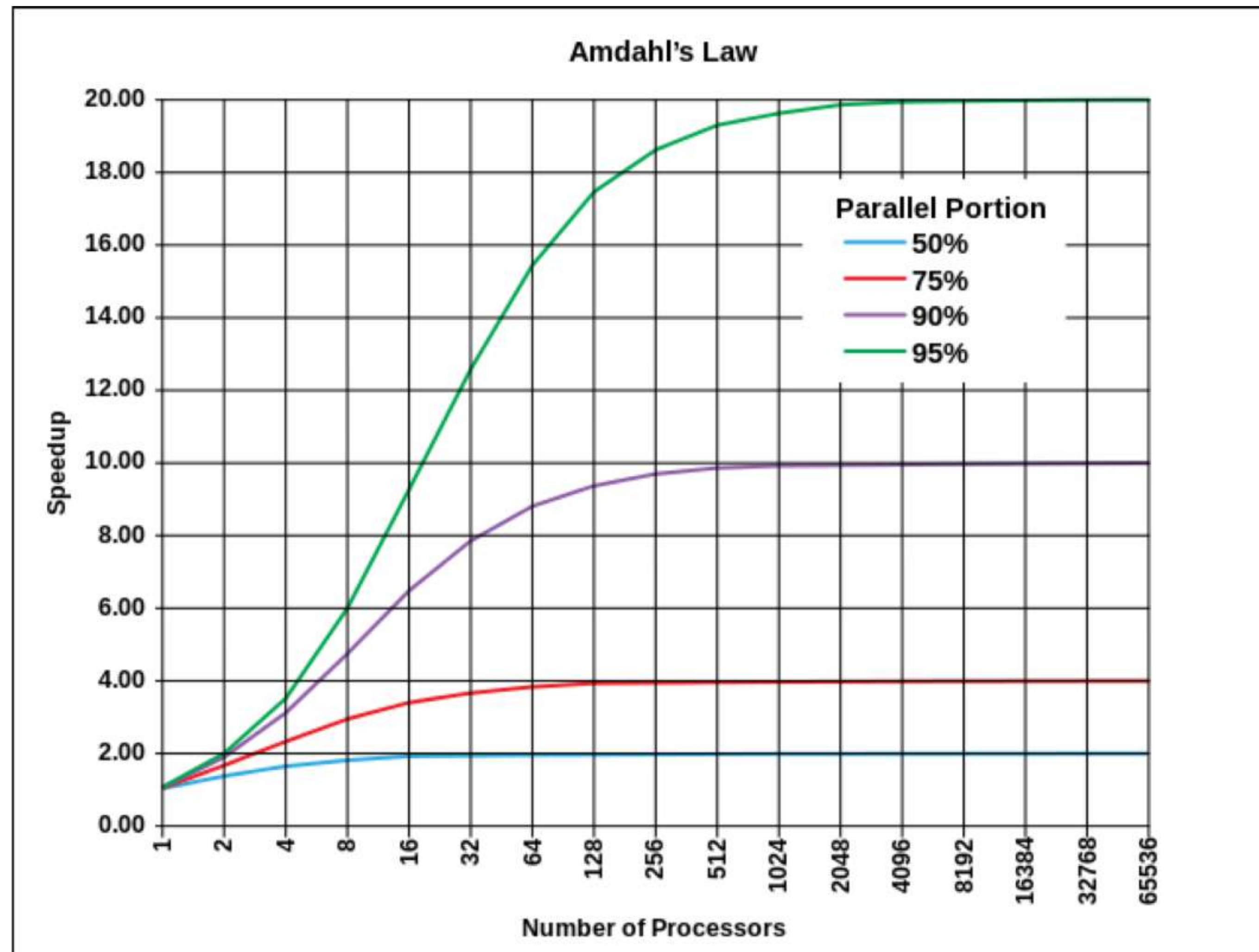
$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.7 + \frac{0.7}{N}}$$

- $N = 2, S_L = 1.54$
- $N = 3, S_L = 1.85$
- $N = 4, S_L = 2.1$
- $N = 5, S_L = 2.3$
- $N = \infty, S_L = 3.33$



2.5 – Ejercicios
Ley de Amdahl

- Speedup real en función del numero de núcleos/procesadores y *p*

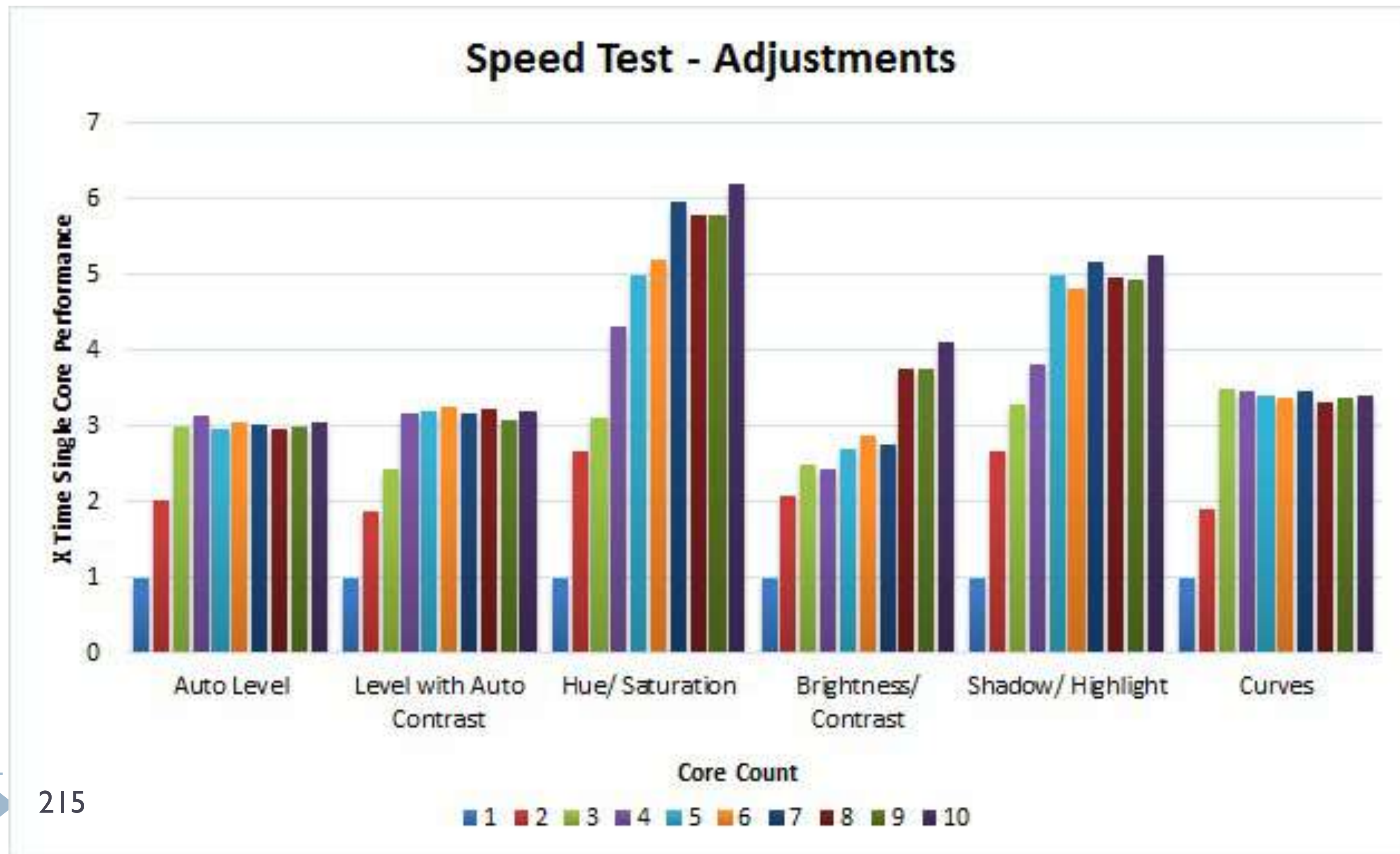


2.5 – Ejercicios
Limitaciones de la ley de Amdahl

- La ley de Amdahl tiene ahora más de 40 años
- Aunque proporciona una buena aproximación, hoy en día se buscan alternativas más correctas (corolarios para la era multinúcleo)
- Limitaciones
  - Solo es valida cuando el cuello de botella es la CPU (no la RAM, la tarjeta video, HDD, etc.)
  - Muchos códigos están programados para que se ejecuten con un número fijo de núcleos, de forma que cualquier núcleo adicional no cambiaría el resultado
  - El número de núcleos usados en un programa puede variar según que tipo de actividad se está haciendo con este programa
  - Considera que todos los núcleos son homogéneos, cuando realmente puede que no (por ejemplo un núcleo puede hacer de master y mandar los otros)
  - El mismo proceso de paralelismo de una ejecución conlleva un cierto overhead que no se tiene en cuenta en la ley

2.5 – Ejercicios
Limitaciones de la ley de Amdahl

- Comparación de speedup usando múltiples núcleos para diferentes ajuste en Adobe Photoshop



2.5 – Ejercicios

Ejemplos

- Se añade un bloque de planificación y predicción que consigue que el procesador vaya 10 veces más rápido que en la máquina original. Este bloque se utiliza sólo el 40% del tiempo de ejecución.
- ¿Cuál es la ganancia de velocidad global?
- ¿y la máxima ganancia que podríamos obtener?

2.5 – Ejercicios

Ejemplos

- Se añade un bloque de planificación y predicción que consigue que el procesador vaya 10 veces más rápido que en la máquina original. Este bloque se utiliza sólo el 40% del tiempo de ejecución.
- ¿Cuál es la ganancia de velocidad global?

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.4 + \frac{0.4}{10}} = 1.56$$

- ¿y la máxima ganancia que podríamos obtener?

- Se añade un bloque de planificación y predicción que consigue que el procesador vaya 10 veces más rápido que en la máquina original. Este bloque se utiliza sólo el 40% del tiempo de ejecución.
- ¿Cuál es la ganancia de velocidad global?

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.4 + \frac{0.4}{10}} = 1.56$$

- ¿y la máxima ganancia que podríamos obtener?

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.4 + \frac{0.4}{\infty}} = 1.66$$

2.5 – Ejercicios
Ejemplos

- Suponer que queremos conseguir una aceleración de 80, con 100 procesadores.
- ¿Qué fracción del programa debe ser paralela?

2.5 – Ejercicios
Ejemplos

- Suponer que queremos conseguir una aceleración de 80, con 100 procesadores.
- ¿Qué fracción del programa debe ser paralela?

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}}$$

- Suponer que queremos conseguir una aceleración de 80, con 100 procesadores.
- ¿Qué fracción del programa debe ser paralela?

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} \rightarrow 80 = \frac{1}{1 - p + \frac{p}{100}}$$

- Suponer que queremos conseguir una aceleración de 80, con 100 procesadores.
- ¿Qué fracción del programa debe ser paralela?

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} \rightarrow 80 = \frac{1}{1 - p + \frac{p}{100}}$$

$$p = \frac{1 - {}^{1}/_{80}}{1 - {}^{1}/_{100}} = 0.9975$$

- Hay que paralelizar el 99.75%

2.5 – Ejercicios
Ejemplos

- Un computador tiene una CPU que se utiliza el 50% del tiempo, el resto se consume en accesos a memoria y al sistema de E/S. Supongamos que el coste de la CPU representa 1/3 del coste total del computador
- ¿mejora la relación coste/rendimiento si se reemplaza la CPU por otra 5 veces más rápida que cueste 5 veces más?

2.5 – Ejercicios
Ejemplos

- Un computador tiene una CPU que se utiliza el 50% del tiempo, el resto se consume en accesos a memoria y al sistema de E/S. Supongamos que el coste de la CPU representa 1/3 del coste total del computador
- ¿mejora la relación coste/rendimiento si se reemplaza la CPU por otra 5 veces más rápida que cueste 5 veces más?

- Speedup

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.5 + \frac{0.5}{5}} = 1.67$$

La nueva CPU es 1.67 veces más rápida

2.5 – Ejercicios
Ejemplos

- Un computador tiene una CPU que se utiliza el 50% del tiempo, el resto se consume en accesos a memoria y al sistema de E/S. Supongamos que el coste de la CPU representa 1/3 del coste total del computador
- ¿mejora la relación coste/rendimiento si se reemplaza la CPU por otra 5 veces más rápida que cueste 5 veces más?

- Coste

$$C_{nuevo} = \left(\frac{2}{3} + \frac{1}{3} \times 5\right) \times C_{original} = 2.33\ C_{original}$$

La nueva CPU cuesta 2.33 veces la original

- Un computador tiene una CPU que se utiliza el 50% del tiempo, el resto se consume en accesos a memoria y al sistema de E/S. Supongamos que el coste de la CPU representa 1/3 del coste total del computador
- ¿mejora la relación coste/rendimiento si se reemplaza la CPU por otra 5 veces más rápida que cueste 5 veces más?

- Relación

La nueva mejora 1.67 pero cuesta 2.33 más

$$\frac{1.67}{2.33} = 0.717$$

La relación es menor de 1, desde el punto de vista coste/rendimiento, la nueva CPU no conviene ya que el coste aumenta más que el speedup

2.5 – Ejercicios
Problema 1

- Un programa tarda 32 minutos en ejecutarse en un computador con un único procesador. El 80% del tiempo de ejecución del programa se podría ejecutar en paralelo. El 20% restante de este tiempo se divide en partes iguales: una correspondiente a la parte puramente secuencial y la otra dedicada a las operaciones de entrada/salida. Calcular:
- La aceleración máxima alcanzable por el programa suponiendo que el grado de paralelismo es un número suficientemente grande. ¿Cuál será el tiempo de ejecución del programa en este caso?
- El grado de paralelismo necesario para conseguir una aceleración de 3 en la ejecución del programa. ¿Cuál será el tiempo de ejecución del programa en este caso?
- La aceleración que se conseguirá si la unidad de entrada/salida se hace tres veces más rápida.

2.5 – Ejercicios
Problema 1

- La aceleración que se conseguirá si la unidad E/S se hace tres veces más rápida
- Datos:
  - Se tarde 32 minutos
  - 80% procesador con posibilidad de paralelismo
  - 10% dispositivo de E/S
  - 10% otro

32 min = 80% x 32 min + 10% x 32 min + 10% x 32 min

2.5 – Ejercicios
Problema 1

- La aceleración que se conseguirá si la unidad E/S se hace tres veces más rápida
- Datos:
  - Se tarde 32 minutos
  - 80% procesador con posibilidad de paralelismo
  - 10% dispositivo de E/S
  - 10% otro

Parte de E/S

32 min = 80% x 32 min + 10% x 32 min + 10% x 32 min

Si 3 veces más rápida

80% x 32 min + 10% x 32 min + 10% x 32 / 3 min = 29.87 min

S = ?

2.5 – Ejercicios
Problema 1

- La aceleración que se conseguirá si la unidad E/S se hace tres veces más rápida
- Datos:
  - Se tarde 32 minutos
  - 80% procesador con posibilidad de paralelismo
  - 10% dispositivo de E/S
  - 10% otro

Parte de E/S

32 min = 80% x 32 min + 10% x 32 min + 10% x 32 min

Si 3 veces más rápida

80% x 32 min + 10% x 32 min + 10% x 32 / 3 min = 29.87 min

S = 32 min / 29.87 min = 1.071

2.5 – Ejercicios
Problema 1

- La aceleración máxima alcanzable por el programa suponiendo que el grado de paralelismo es un número suficientemente grande. ¿Cuál será el tiempo de ejecución del programa en este caso?
- Datos:
  - 80% procesador con posibilidad de paralelismo

2.5 – Ejercicios
Problema 1

- La aceleración máxima alcanzable por el programa suponiendo que el grado de paralelismo es un número suficientemente grande. ¿Cuál será el tiempo de ejecución del programa en este caso?
- Datos:
  - 80% procesador con posibilidad de paralelismo
- Speedup

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.8 + \frac{0.8}{\infty}} = 5$$

2.5 – Ejercicios
Problema 1

- La aceleración máxima alcanzable por el programa suponiendo que el grado de paralelismo es un número suficientemente grande. ¿Cuál será el tiempo de ejecución del programa en este caso?
- Datos:
  - 80% procesador con posibilidad de paralelismo

- Speedup

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.8 + \frac{0.8}{\infty}} = 5$$

- Tiempo de ejecución

$$L = \frac{32\,min}{S_L} = \frac{32\,min}{5} = 6.4\,min$$

2.5 – Ejercicios
Problema 1

- El grado de paralelismo necesario para conseguir una aceleración de 3 en la ejecución del programa. ¿Cuál será el tiempo de ejecución del programa en este caso?
- Datos:
  - 80% procesador con posibilidad de paralelismo
  - Speedup = 3

2.5 – Ejercicios
Problema 1

- El grado de paralelismo necesario para conseguir una aceleración de 3 en la ejecución del programa. ¿Cuál será el tiempo de ejecución del programa en este caso?
- Datos:
  - 80% procesador con posibilidad de paralelismo
  - Speedup = 3

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.8 + \frac{0.8}{n}} = 3$$

2.5 – Ejercicios

Problema 1

- El grado de paralelismo necesario para conseguir una aceleración de 3 en la ejecución del programa. ¿Cuál será el tiempo de ejecución del programa en este caso?
- Datos:
  - 80% procesador con posibilidad de paralelismo
  - Speedup = 3

$$S_L = \frac{1}{1 - p + \frac{p}{S_L^I}} = \frac{1}{1 - 0.8 + \frac{0.8}{n}} = 3$$

$n = 6$

- Tiempo de ejecución

$$L = \frac{32\ min}{S_L} = \frac{32\ min}{3} = 10.67\ min$$

2.5 – Ejercicios
Problema 2

- Un computador dispone de un procesador que funciona con un reloj a 100 MHz. Este procesador dispone de tres tipos de instrucciones A, B y C, con un número medio de ciclos por instrucción de 1, 2 y 3, respectivamente. Este procesador se utiliza para comparar el rendimiento de dos compiladores distintos C1 y C2. Para ello se ha monitorizado un programa de prueba compilado con ambos compiladores, obteniendo el número de instrucciones ejecutadas de cada tipo indicadas en la tabla.

Compilador	A	B	C
c1	5×10^6	10^6	10^6
c2	10×10^6	10^6	10^6

- Calcular, para cada compilador, el tiempo de ejecución del programa, el CPI medio y los MIPS.
- ¿Con qué compilador el programa se ejecuta más rápidamente? ¿en qué % es más rápido el programa con este compilador?

2.5 – Ejercicios
Problema 2

- Calcular, para cada compilador, el tiempo de ejecución del programa, el CPI medio y los MIPS.
- Datos
  - Frecuencia = 100 MHz

Compilador	A	B	C
c1	$5x10^6$	10^6	10^6
c2	$10x10^6$	10^6	10^6

- Calcular, para cada compilador, el tiempo de ejecución del programa, el CPI medio y los MIPS.

2.5 – Ejercicios
Problema 2

- Calcular, para cada compilador, el tiempo de ejecución del programa, el CPI medio y los MIPS.
- Datos
  - Frecuencia = 100 MHz

Compilador	A	B	C
c1	$5x10^6$	10^6	10^6
c2	$10x10^6$	10^6	10^6

- Calcular, para cada compilador, el tiempo de ejecución del programa

$$L_1 = \frac{ciclos}{frecuencia} = \frac{(1\times5 + 2\times1 + 3\times1)\times10^6}{100\times10^6} = 0.1\ s$$

$$L_2 = \frac{ciclos}{frecuencia} = \frac{(1\times10 + 2\times1 + 3\times1)\times10^6}{100\times10^6} = 0.15\ s$$

2.5 – Ejercicios
Problema 2

- Calcular, para cada compilador, el tiempo de ejecución del programa, el CPI medio y los MIPS.
- Datos
  - Frecuencia = 100 MHz

Compilador	A	B	C
c1	$5x10^6$	10^6	10^6
c2	$10x10^6$	10^6	10^6

- Calcular, para cada compilador, el CPI medio

2.5 – Ejercicios
Problema 2

- Calcular, para cada compilador, el tiempo de ejecución del programa, el CPI medio y los MIPS.
- Datos
  - Frecuencia = 100 MHz

Compilador	A	B	C
c1	$5x10^6$	10^6	10^6
c2	$10x10^6$	10^6	10^6

- Calcular, para cada compilador, el CPI medio

$$CPI_1 = \frac{\sum_{i=1}^{n} CPI_i \times Inst_i}{Inst} = \frac{(1\times5 + 2\times1 + 3\times1)\times10^6 ciclos}{(5+1+1)\times10^6 instrucciones} = 1.428\ c/i$$

$$CPI_2 = \frac{(1\times10 + 2\times1 + 3\times1)\times10^6 ciclos}{(10+1+1)\times10^6 instrucciones} = 1.25\ c/i$$

2.5 – Ejercicios
Problema 2

- Calcular, para cada compilador, el tiempo de ejecución del programa, el CPI medio y los MIPS.
- Datos
  - Frecuencia = 100 MHz

Compilador	A	B	C
c1	5×10^6	10^6	10^6
c2	10×10^6	10^6	10^6

- Calcular, para cada compilador, los MIPS

2.5 – Ejercicios
Problema 2

- Calcular, para cada compilador, el tiempo de ejecución del programa, el CPI medio y los MIPS.
- Datos
  - Frecuencia = 100 MHz

Compilador	A	B	C
c1	5×10^6	10^6	10^6
c2	10×10^6	10^6	10^6

- Calcular, para cada compilador, los MIPS

$$MIPS_1 = \frac{Instrucciones}{L \times 10^6} = \frac{(5+1+1) \times 10^6 instrucciones}{0.1 \times 10^6 s} = 70$$

$$MIPS_2 = \frac{Instrucciones}{L \times 10^6} = \frac{(10+1+1) \times 10^6 instrucciones}{0.15 \times 10^6 s} = 80$$

Arquitectura i Configuracions Informàtiques

Tema 2. El microprocesador

Davide Careglio