

Clasificación de Computadores

Clases de Computadores

Las categorías en las que se puede clasificar son las siguientes:

- Dispositivos móviles personales (PMD).
- Escritorio (*Desktop*).
- Servidores.
- *Clusters*.
- Embebidos (*Embedded*).

Característica	PMD	Escritorio	Servidor	<i>Clusters</i>	Embebido
Precio	\$100-\$1000	\$300-\$2500	\$5000-\$10.000.000	\$100.000-\$200.000.000	\$10-\$100.000
Precio μ	\$10-\$100	\$50-\$500	\$200-\$2000	\$50-\$250	\$0,01-\$100
Propósito	Energía, Tamaño	Precio-Rendimiento	Escalabilidad	Rendimiento	Específico

Clasificación de Computadores

Otras clasificaciones

Según generación:

- Primera generación (1946-1959), basado en tubos de vacío.
- Segunda generación (1959-1965), basado en transistores.
- Tercera generación (1965-1971), circuitos integrados.
- Cuarta generación (1971-1980), VLSI (20,000 transistores a 1,000,000).
- Quinta generación (1980-presente), ULSI (más de un millón de transistores).

Según propósito:

- General.
- Específico.

Según procesamiento de datos:

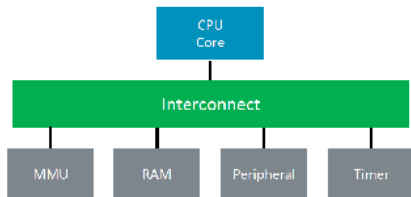
- Analógico.
- Digital.
- Híbrido.

Clasificación de Computadores

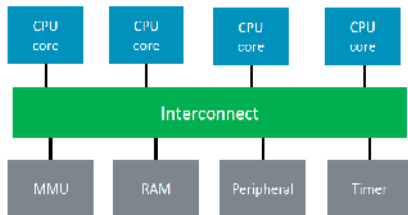
Clasificación por Paralelismo

- Paralelismo a nivel de bit (BLP): Hacer el bus más grande o ancho.
- Paralelismo a nivel de Instrucción (ILP): Pipeline, VLIW, Superscalar, OoOE.
- Paralelismo a nivel de Hilo (TLP): Simultaneous Multithreading Processor (SMT).
- Paralelismo a nivel de datos: Arquitecturas vectoriales y GPUs.

La Era del *SingleCore*



Clasificación por Paralelismo



¿Arquitectura y Microarquitectura son lo mismo?

Arquitectura de un computador

La arquitectura de un procesador corresponde al Set de Instrucciones (ISA) que puede ejecutar dicho procesador.

Arquitectura \Rightarrow Software

Responde a la pregunta: **¿Qué hace/ejecuta/tiene el hardware?**

Componentes de un ISA:

- Clase de ISA: *Register-Memory* o *Load-Store*.
- Direccionamiento de memoria: Endianness, alineamiento.
- Métodos de direccionamiento.
- Tipos y tamaños de operandos.
- Operaciones.
- Control de flujo.
- Encodificación.
- Costo (área, ley de Moore).
- Simplicidad: De diseño y verificación.
- Desempeño.
- Escalabilidad.
- Tamaño (*Memory footprint*).
- Facilidad de programación.
- Seguridad.

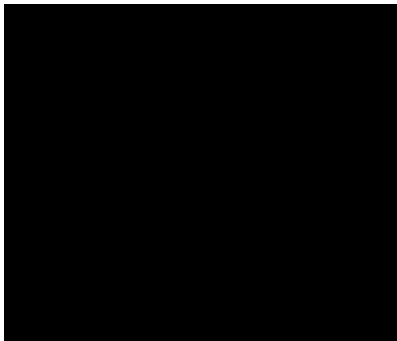
¿Arquitectura y Microarquitectura son lo mismo?

Microarquitectura de un computador

La microarquitectura son los detalles de interconexión, implementación y optimización de una arquitectura. Aspectos de alto nivel de la implementación de un computador. También se conoce como organización.

Microarquitectura \Rightarrow Hardware

Responde a la pregunta: **¿Cómo hace/ejecuta/implementa el hardware?**



Paralelismo, Arquitecturas

Paralelas y Flynn

Parallel Architectures by Flynn

- “...Parallel or concurrent operation has **many different forms** within a computer system...”
- “...A stream is a sequence of objects such as **data**, or of actions such as **instructions**. **Each stream is independent of all other streams, and each element of a stream can consist of one or more objects or actions...**”

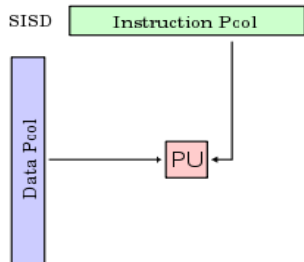
Las arquitecturas más comunes según la cantidad de *streams* son:

- SISD.
- SIMD.
- MISD.
- MIMD.

Paralelismo, Arquitecturas Paralelas y Flynn

SISD

- Significa *Single Instruction Single Data*.
- Arquitectura tradicional de un único procesador.
- Utiliza *pipelining*, realizando concurrentemente diferentes fases de procesamiento de una instrucción.
- Implementa *instruction level parallelism* (ILP) como superescalar *overlong instruction word* (VLIW).
- No se obtiene concurrencia de ejecución, pero si de procesos.



Paralelismo, Arquitecturas Paralelas y Flynn

SIMD

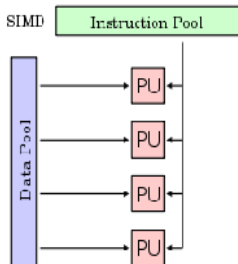
- Significa *Single Instruction Multiple Data*.
- Incluye procesadores de arreglo (*array*) y vectoriales.
 - Procesadores de arreglo: Instrucciones operan en múltiples elementos al mismo tiempo. Se conocen como *massively parallel processor*.
 - Procesadores vectoriales: Instrucciones operan en múltiples elementos en tiempos consecutivos.

$LD\ VR \leftarrow A[3:0]$
 $ADD\ VR \leftarrow VR, 1$
 $MUL\ VR \leftarrow VR, 2$
 $ST\ A[3:0] \leftarrow VR$

SIMD (Tiempo vs Espacio)

LD0	LD1	LD2	LD3
AD0	AD1	AD2	AD3
MU0	MU1	MU2	MU3
ST0	ST1	ST2	ST3

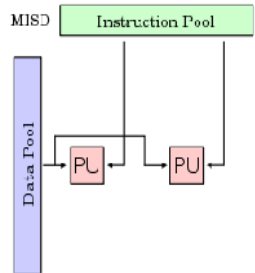
LD0	LD1	LD2	LD3
AD0	AD1	AD2	AD3
MU1	MU2	MU3	MU4
ST0	ST1	ST2	ST3



Paralelismo, Arquitecturas Paralelas y Flynn

MISD

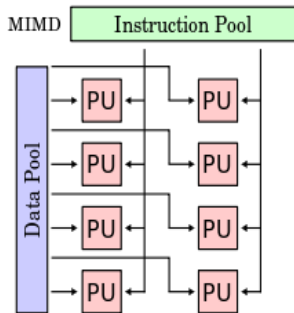
- Significa *Multiple Instruction, Single Data*.
- Se usa en sistemas aeroespaciales y arreglos sistólicos.
- También se puede usar para detectar y enmascarar errores



Paralelismo, Arquitecturas Paralelas y Flynn

MIMD

- Significa *Multiple Instruction, Multiple Data*.
- No necesariamente todos los procesadores deben ser idénticos, cada uno opera independientemente.
- Son: procesadores multinúcleo y superescalares.
- Cuando usan memoria compartida en este tipo de sistemas hay dos problemas:
 - Consistencia de memoria (se resuelve a través de combinación de técnicas de hardware y software).
 - Mantener la coherencia de cach´ (se resuelve mediante técnicas de hardware).



Según Flynn cómo se catalogan:

$$[f(x), g(y), h(z)] = \left[\frac{x+1}{2}, \frac{\sin y}{y}, e^z \right] \quad (1)$$

$$[h(x, y)] = \left[e^{x+y} \right] \quad (2)$$

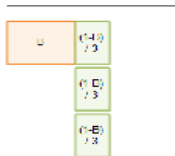
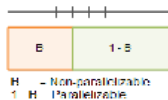
$$[f(x)] = a_0 + x(a_1 + x(a_2 + a_3x)) = a_0 + a_1x^2 + a_2x^3 \quad (3)$$

$$[g(x, y, z)] = x^{a_0} + y^{a_1} + z^{a_2} \quad (4)$$

Ley de Amdahl

¿Qué es?

- Permite obtener la ganancia en el desempeño debido a la mejora en una característica determinada.
- La ley de Amdahl puede servir como una guía para determinar la mejora real y como distribuir los recursos para tener mejor relación costo-desempeño.
- Gene Amdahl establece que todo programa se divide en:
 - Partes paralelizable.
 - Partes **no** paralelizables.



¿Qué es?

$$\frac{T-B}{N}$$

Donde:

$$T(N) = B + \frac{(T-B)}{N}$$

Ley de Amdahl

- B : es la parte no paralelizable.
- T : es el tiempo de duración de una tarea.

La fracción paralelizable está dada por un factor N

Speedup

$$\text{Speedup} = \frac{\text{Tiempo de ejecución de una tarea sin mejora}}{\text{Tiempo de ejecución de una tarea con mejora}}$$

$$\text{Speedup} = \frac{T(1)}{T(N)} = \frac{T(1)}{B + \frac{(T(1)-B)}{N}}$$

Con $T(1) = 1$ (sin mejora):

Speedup Overall

$$\text{Speedup} = \frac{1}{B + \frac{(1-B)}{N}}$$

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Donde

- $\text{Fraction}_{\text{enhanced}}$ es la fracción del tiempo de computación original que se puede ver beneficiado por la mejora.
- $\text{Speed}_{\text{enhanced}}$ es la ganancia del producto de la ejecución en modo "mejorado". Esto es, qué tan rápido ejecutaría la tarea si la mejora se aplicara a todo el programa.

Ley de Amdahl

Ejemplo

Ejemplo: Ley de Amdahl

Supongamos que se desea mejorar un procesador utilizado para un servidor Web. El nuevo procesador es **10 veces más rápido** en tiempo de computación para la aplicación de servidor Web que el procesador antiguo. Asumiendo que el procesador original está **ocupado** un **40 %** del tiempo y el **60 %** del tiempo **esperando** por dispositivos de Entrada/Salida. ¿Cuál es la ganancia total producto de incorporar la mejora?

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

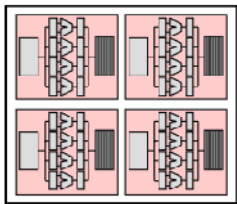
$$\text{Fraction}_{\text{enhanced}} = 40 \% = 0,4 \quad \text{Speedup}_{\text{enhanced}} = 10$$

$$\text{Speedup}_{\text{overall}} = 1,56$$

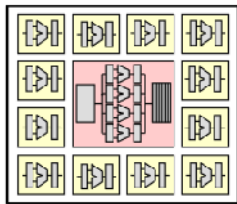
Ley de Amdahl

Amdahl con unidades de procesamiento simétricas y no-simétricas

- La paralelización es uniformemente distribuida en las unidades de ejecución (procesadores).
- No aplica en sistemas heterogéneos y *multicore* (simétricos y no simétricos).
- Supone que no hay conflictos de recursos.



Symmetric: Four 4-BCE cores



**Asymmetric: One 4-BCE core
& Twelve 1-BCE base cores**

Confiabilidad

Probabilidad de que el sistema esté funcionando en el instante t dado que funcionaba en el instante t_0 . Se tiene:

- Tiempo medio para una falla (MTTF).
- Fallos por tiempo (λ): $\lambda = \frac{1}{MTTF}$

$R(t)$: probabilidad de que un sistema falle en t unidades de tiempo después de la última falla.

$$R(t) = e^{-\lambda(t-t_0)}$$

Mantenibilidad

Tiempo requerido para que el sistema este funcionando luego que se dio una falla.

- Tiempo medio para reparar (MTTR): Tiempo de la interrupción del servicio.
- Tasa de reparación (μ): $\mu = \frac{1}{MTTR}$

$M(t)$: probabilidad de que un sistema este funcionando en t unidades de tiempo después de que se presentó una falla.

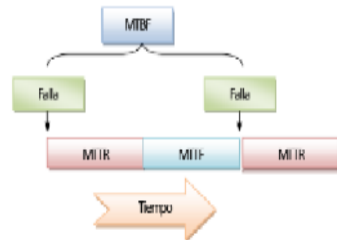
$$M(t) = e^{-\mu t}$$

Disponibilidad

Es el porcentaje del tiempo en el que el sistema estará disponible para brindar un servicio correcto.

Tiempo medio entre fallas (MTBF): $MTTF + MTTR$

$$A = \frac{MTTF}{MTBF}$$



Desempeño

Benchmarking

El benchmark es un instrumento para comparar el desempeño de varios sistemas en aplicaciones reales.

Representa un recurso de software para evaluar un sistema y discriminar la mejor opción para el diseño.

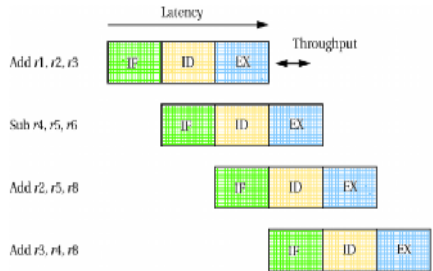
Varios tipos de benchmarks: SPEC, EEMBC, BDTi, Drystone, CoreMark.

Término aplica de manera diferente según el campo:

- ISP: Calidad de imagen.
- Memorias: Accesos a memoria por segundo.
- CPU's: Medida de la tasa en que los programas son ejecutados (IPC, CPI).

Punto de vista microscópico:

- Latencia: Tiempo requerido para ejecutar una instrucción desde inicio hasta finalización.
- Flujo de instrucciones (throughput): Tasa de finalización de instrucciones.



Formas de Organización/Microarquitectura

- Arquitectura Von Neumann: Un único espacio de direccionamiento, y única ruta de acceso al CPU.
- Arquitectura Harvard: Memoria de datos y memoria de instrucciones tienen rutas de hardware diferentes hacia el CPU, además de espacios de direccionamiento separados.
- Arquitectura Harvard Modificada: rutas de hardware diferentes para el CPU Cache, y un espacio de direccionamiento único.

Clasificación de los ISA

Otros ISAs

- ISA ortogonal: El código de operación y el operando son independientes.
- Cualquier instrucción puede usar cualquier operando.

Longitud fija vs longitud variable:

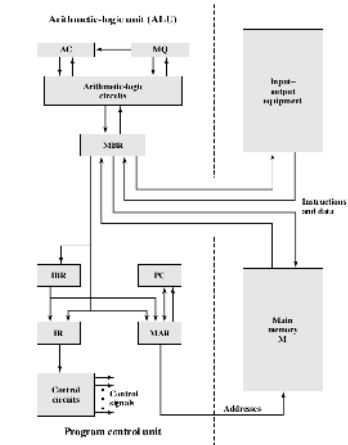
- Longitud fija: *Fetching y decoding* por hardware es rápido.
- Longitud variable: *Fetching y decoding* por hardware es lento.

Arquitectura Von Neumann: Características VIEJA

- La información se representa por medio de direcciones.
- Memoria unificada**, una única memoria para datos y programa.
- Las instrucciones almacenadas y ejecutadas secuencialmente: Program counter debe actualizarse ($PC=PC+1$) para obtener siguiente instrucción.
- Ciclo de Fetch: Búsqueda, Decodificación, Ejecución, Almacenado.
- Cuenta con un ISA de 21 instrucciones.

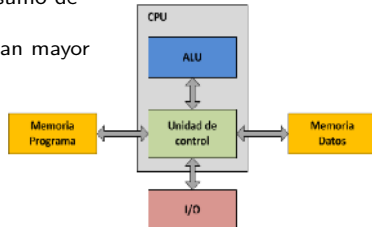
En la versión original se listaron las siguientes partes:

- Central Arithmetic (CA)**: Unidad encargada de llevar a cabo las operaciones aritméticas de suma, resta, multiplicación y división.
- Central Control (CC)**: Lógica de control del computador, encargado de llevar la secuencia correcta del programa.
- Memoria (M)**: Almacena largas cantidades de operaciones (programa). Se ejecuta secuencialmente.
- I/O Equipment (I,O)**: Periféricos del sistema.



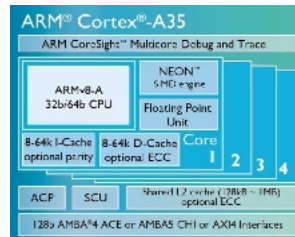
Arquitectura Harvard: Características

- Memoria de datos y memoria de programa están **físicamente** separadas.
- Acceso a memoria de datos e instrucciones puede ser simultáneo.
- Ventaja: mayor rendimiento (paralelismo).
- La mayoría de DSP poseen una arquitectura tipo Harvard pues necesitan buscar datos y operaciones al mismo tiempo.
- Desventajas de la arquitectura de Harvard:
 - El espacio de direccionamiento separado implica 2 memorias físicas diferentes: mayor espacio, consumo de potencia.
 - Rutas diferentes (mayor ancho de banda) generan mayor consumo de potencia dinámica.



Arquitectura Harvard Modificada

- Desventajas de la arquitectura de Harvard:
 - El espacio de direccionamiento separado implica 2 memorias físicas diferentes: mayor espacio, consumo de potencia.
 - Rutas diferentes (mayor ancho de banda) generan mayor consumo de potencia dinámica.
- Arquitectura de Harvard Modificada:
 - Disminuye el impacto de la separación de memoria.
 - Rutas separadas para instrucciones y datos, con único espacio de direccionamiento.
 - Provee instrucciones para acceder a los contenidos de la memoria de instrucciones como si fueran datos.
 - Una única memoria principal.



- Utiliza dos memorias caché de CPU, para la separación de datos e instrucciones.
- Una única memoria principal.
- Desde el punto de vista macro se comporta como una arquitectura Von Neumann, pero internamente separa instrucciones y datos.
- ¿Dónde está lo complicado?

La mayoría de las arquitecturas modernas-Harvard son en realidad Harvard Modificada.

Clasificación de los ISA

Tipo de operando

Tipos de operando que existen:

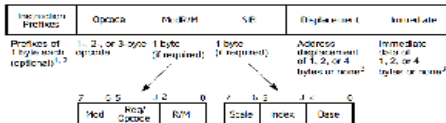
Load/Store → Divide las operaciones en dos categorías:

- Accesos a memoria (instrucciones: Load/Store en memorias y registros).
- Operaciones con ALU (solo entre registros).

Ejemplos: ARM, RISC-V, MIPS.

Tipo de operando: Register/Mem

Operaciones pueden ser entre registros y entre espacios de memoria.



1. The REX prefix is optional, but if used must be immediately before the opcode; see Section 2.2.1, "REX Prefixes" for additional information.

2. For VEX encoding information, see Section 2.3, "Intel® Advanced Vector Extensions (Intel® AVX)".

3. Some new instructions can take an 8B immediate or 8B displacement.

Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Clasificación de los ISA

Complejidad de instrucciones: CISC

Complex Instruction Set Computer:

- Enfoque inicial de arquitectura.
- El ISA contiene gran variedad de instrucciones: instrucciones poderosas y complicadas.
- Facilidad de programación.
- El compilador no realiza traducciones complejas: instrucciones son similares a lenguajes de alto nivel.

Ejemplo: x86.

Características típicas del set:

- Ventajas:
 - Facilidad de programación: Tareas complejas se realizan en menos tiempo.
 - Múltiples modos de direccionamiento simplifican las tareas.
 - Tamaño de código pequeño.
 - Desventajas:
 - Instrucciones de tamaño variable: diferente tiempo de búsqueda y ejecución hacen muy complicado tener sistemas determinísticos. Hardware es complicado (área, dinero).
 - Muchas de las instrucciones especializadas no son utilizadas con frecuencia: El 98 % de las instrucciones en un programa típico corresponden al 20 % de las instrucciones del set.
- Múltiples modos de direccionamiento (forma de acceder a datos en memoria).
 - Formato de instrucciones variable.
 - Duración de instrucciones variable.
 - Bajo número de registros de propósito general. x86: RAX, RBX, RCX, RDX.
 - Las instrucciones son capaces de ejecutar tareas complejas.
 - Decodificación de instrucciones implica mayor lógica de hardware.

Clasificación de los ISA

Complejidad de instrucciones: RISC

Reduced Instruction Set Computer:

- Enfoque moderno: DSPs, CPUs para sistemas embebidos.
- El set está compuesto por pocas instrucciones con funcionalidad simple.
- La dificultad está en el programador (bajo nivel) o el compilador.

Ejemplos: MIPS, ARM.

Reduced Instruction Set Computer:

Características típicas del set:

- Ventajas:
 - Instrucciones de tiempo y tamaño fijo: simplifica hardware y brinda determinismo.
 - Mejor aprovechamiento de hardware.
 - Permite pipeline.
- Desventajas:
 - Tamaño de código mayor.
 - Carga pesada para el software (programa de bajo nivel o compilador).

Pocos modos de direccionamiento (1-4).

Las instrucciones tienen un tamaño fijo.

El tiempo de ejecución de cada instrucción es el mismo.

Alto número de registros de propósito general (16, +32).

Decodificación de instrucciones simple.

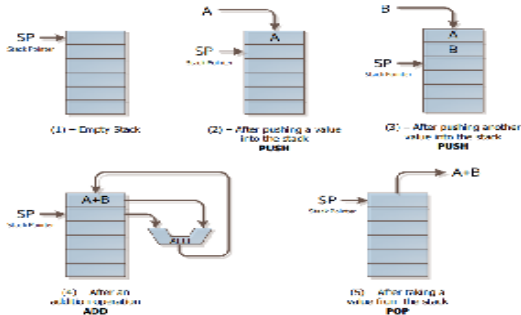
Clases de ISA

El almacenamiento interno en un procesador es la forma más básica de diferenciación. Las cuatro clasificaciones más importantes son:

- Stack.
- Acumulador.
- Registro-Memoria.
- Registro-Registro.

Stack

En la clase Stack, los operandos se encuentran en una *única pila*. Para realizar las operaciones se deben insertar los operandos (*push*) y luego extraer el resultado (*pop*). Los va extrayendo del *Top of Stack Register* (TOS).



Stack: Ejemplo

Suponga la operación: $A = B + C$.

```
push B; // insertar operando 1 hacia latch ALU
push C; // insertar operando 2 hacia latch ALU
add;    // suma
pop A;  // extraer resultado del TOS hacia A
```

Los operandos están **implícitos** en el TOS (no se definen/nombran explícitamente).

Acumulador

En las arquitecturas con acumulador, un operando se encuentra **implícito** en el acumulador y el otro se encuentra **explícito** en memoria.

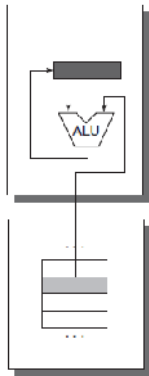
Suponga la operación: $A = B + C$.

Load B; // Carga B en el acumulador.

Add C; // Suma C al operando implícito en memoria.

Store A; //Almacena en memoria el resultado.

Uno de los operandos es explícito (**C**) y el otro implícito cargado previamente desde **B**.



Registro (registro-memoria)

En esta arquitectura se cuenta con registros de propósito general (GPRs). Uno de los operandos corresponde a un GPR y el otro proviene directamente de memoria.

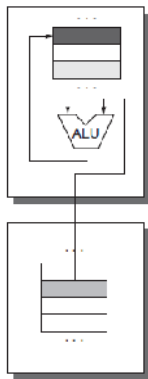
Suponga la operación: $A = B + C$.

Load R1, B; //Cargar B en GPR R1

Add R3, R1, C; //Sumar R1 con C (dir. memoria)

Store R3, A; //Almacenar resultado en A (dir. memoria).

Ambos operandos son explícitos. La mayoría de arquitecturas CISC utilizan esta clase.



Registro-Registro

Una arquitectura de la clase registro-registro no realiza operaciones directamente desde memoria, sino que las realiza enteramente con operandos en registros de propósito general.

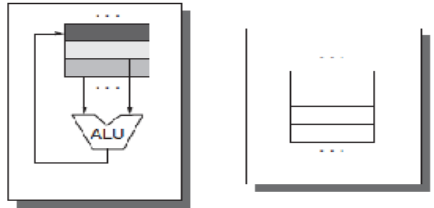
Suponga la operación: $A = B + C$.

Load R1, B; // Cargar B en GPR R1

Load R2, C; // Cargar C en GPR R2

Add R3, R1, R2; // Sumar R1 con R2 y almacena en R3.

Store R3, A; // Almacenar resultado en A (dir. memoria).



Ambos operandos son explícitos. La mayoría de arquitecturas CISC utilizan esta clase.

Direccionamiento de memoria

Independientemente de la arquitectura (reg-reg, reg-mem. etc) se debe tener una forma de interpretar y especificar las direcciones de memoria. Existen diferentes formas de acceder a una dirección:

Endianness

Existen dos formas de ordenar los bytes en una dirección memoria (*Endianness*) (también se conoce *byte ordering*):

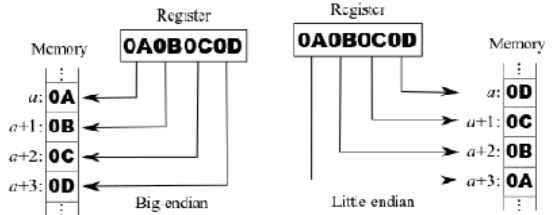
- *Little Endian*: Ordena el byte menos significativo en la dirección más pequeña de la palabra (doble palabra).
- *Big Endian*: Ordena el byte más significativo en la dirección más pequeña de la palabra (doble palabra).

Nivel de byte (8 bits)

Nivel de media palabra (16 bits)

Nivel de palabra / word (32 bits)

Nivel de doble palabra (64 bits)



Alineamiento de memoria

Surge como una limitación de los procesadores modernos.

CPUs son mas eficientes cuando las direcciones de memoria son múltiplos del tamaño del dato (byte, KB, etc).

Algunos lenguajes de programación modernos esconden esta limitación al programador.

[illegible]

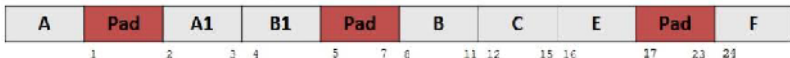
Ejemplo

Represente el alineamiento en memoria para x86 de la siguiente estructura y mejore su orden para un mejor alineamiento.

```
struct:  
char a;  
short a1;  
char b1;  
float b;  
int c;  
char e;  
double f;
```

Data Type	32-bit (bytes)	64-bit (bytes)
char	1	1
short	2	2
int	4	4
long	8	8
float	4	4
double	8	8
long double	16	16
Any pointer	4	8

Table 1: typical alignment requirements for data types on 32-bit and 64-bit Linux systems as used by the Intel® C++ Compiler



Modos de direccionamiento

Los modos de direccionamiento se refiere a la forma en que las arquitecturas **especifican** la dirección de un objeto que van a acceder.

- **Dirección efectiva:** El valor final de dirección especificado por el modo de direccionamiento.

A mayor cantidad de modos de direccionamiento → mayor complejidad (CISC).

A menor cantidad de modos de direccionamiento → menor complejidad (RISC).

Múltiples modos de direccionar datos dentro de una instrucción:

Addressing mode	Example instruction	Meaning	When used
Register	Add R4, R3	$Regs[R4] \leftarrow Regs[R4] + Regs[R3]$	When a value is in a register.
Immediate	Add R4, #3	$Regs[R4] \leftarrow Regs[R4] + 3$	For constants.
Displacement	Add R4, 100(R1)	$Regs[R4] \leftarrow Regs[R4] + Mem[100 + Regs[R1]]$	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	Add R4, (R1)	$Regs[R4] \leftarrow Regs[R4] + Mem[Regs[R1]]$	Accessing using a pointer or a computed address.
Indexed	Add R3, (R1 + R2)	$Regs[R3] \leftarrow Regs[R3] + Mem[Regs[R1] + Regs[R2]]$	Sometimes useful in array addressing; R1 = base of array; R2 = index amount.
Direct or absolute	Add R1, 1001	$Regs[R1] \leftarrow Regs[R1] + Mem[1001]$	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	Add R1, @R3	$Regs[R1] \leftarrow Regs[R1] + Mem[Mem[Regs[R3]]]$	If R3 is the address of a pointer p, then mode yields p .
Autoincrement	Add R1, (R2)+	$Regs[R1] \leftarrow Regs[R1] + Mem[Regs[R2]]$ $Regs[R2] \leftarrow Regs[R2] + d$	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d .
Autodecrement	Add R1, -(R2)	$Regs[R2] \leftarrow Regs[R2] - d$ $Regs[R1] \leftarrow Regs[R1] + Mem[Regs[R2]]$	Same use as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack.
Scaled	Add R1, 100(R2)[R3]	$Regs[R1] \leftarrow Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * d]$	Used to index arrays. May be applied to any indexed addressing mode in some computers.

Operandos

Operandos: Tipo y tamaño de operandos

El tipo de operandos normalmente se encuentra, en la mayoría de los casos, en el código de operación (opcode). Alternativamente los datos son anotados mediante etiquetas (*tags*) que son interpretadas por el HW (mnemónico).

Byte (8 bits).

Half-Word (16 bits).

Word (32 bits) / punto flotante de precisión simple (SP-FP).

DoubleWord (64 bits) / punto flotante de precisión doble.

Enteros: Típicamente en representación complemento a 2.

Flotantes: IEEE 754*.

Caracteres: ASCII.

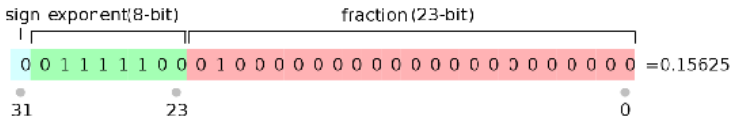
Decimal: BCD.

Conversión de Decimal a Punto Flotante IEEE 754

IEEE 754

¿Cómo se representan decimales?

Ejemplo: Representar $0,10_{10}$, en base 2: $0,00011$



El exponente es desplazado mediante un sesgo para poder representar exponentes negativos.

- 1 Convertir valor decimal a binario (solo número ignorar el signo).
- 2 Colocar el número de la forma: $\text{numero} \times 2^0$.
- 3 Denotar el número de la forma $1, a_1 a_2 \dots a_i \times 2^n$ (se corre la coma n espacios).
- 4 Determinar el signo: 0 si N es mayor que 0, 1 si N es menor que 0.
- 5 Determinar el exponente como $E = n + 127$, luego pasar a binario.
- 6 Determinar la mantisa F como $F = a_1 a_2 \dots a_i$.
- 7 Escribir el número según IEEE, completando con ceros a la derecha los 23 bits de la mantisa.

IEEE 754

Ejemplo: Convertir el número 22,625 a flotante

- ➊ Paso 1: Convertir 22_{10} a binario: 10110_2 . Convertir la parte decimal $0,625_{10}$ a binario: $0,625_{10} = 101_2$.
- ➋ Paso 2: Por lo tanto $22,625_{10} = 10110,101_2$ en notación científica sería $22,625_{10} = 10110,101_2 \times 2^0$.
- ➌ Paso 3: Se normaliza a $1,0110101_2 \times 2^4$. Por lo tanto $n = 4$.
- ➍ Paso 4: Se obtiene el signo (0).
- ➎ Paso 5: Se obtiene el exponente
 $E = 127 + n \rightarrow E = 127 + 4 = 131_{10} = 10000011_2$.
- ➏ Paso 6: Se obtiene la mantisa
 $F = 01101010000000000000000_2$.
- ➐ Paso 7: Se coloca en notación IEEE Signo + Exponente + Mantisa = $01000001101101010000000000000000_2$.

Operands

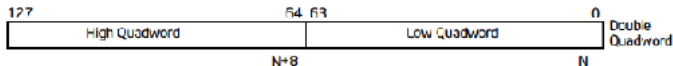
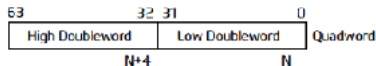
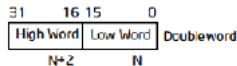
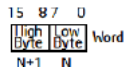
Operands: Ejemplo x86

Type	A32	A64	Description
int/long	32-bit	32-bit	integer
short	16-bit	16-bit	integer
char*	8-bit	8-bit	byte
long long	64-bit	64-bit	integer
float	32-bit	32-bit	single-precision IEEE floating-point
double	64-bit	64-bit	double-precision IEEE floating-point
bool	8-bit	8-bit	Boolean
wchar_t ^a	16-bit unsigned	16-bit unsigned	short (compiler dependent)
	32-bit unsigned	32-bit unsigned	int (compiler dependent)
void* pointer	32-bit	64-bit	addresses to data or code
enumerated types	32-bit	32-bit ^b	signed or unsigned integer
bit fields	not larger than their natural container size		

ABI defined extension types

__int128/ __uint128	128-bit	128-bit	signed/unsigned quadword
__float	16-bit	16-bit	half precision

- Environment dependent. in GNU-based systems (such as Linux) this type is always 32-bit.
- If the set of values in an enumerated type cannot be represented using either int or unsigned int as a container type, and the language permits extended enumeration sets, then a long long or unsigned long long container may be used.



Codificación

Al tener un tamaño de palabra finito y determinado para las instrucciones, es vital maximizar su uso para poder describir cada instrucción.

Respecto al tamaño de instrucción existen 3 formas comunes de longitud de instrucción:

Variable → CISC.

Fija → RISC.

Híbrido.

Dirección relativa

Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, Super11	Tests special bits set by ALU operations, possibly under program control.	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.
Condition register	Alpha, MIPS	Tests arbitrary register with the result of a comparison.	Simple.	Uses up a register.
Compare and branch	PA-RISC, VAX	Compare is part of the branch. Often compare is limited to subset.	One instruction rather than two for a branch.	May be too much work per instruction for pipelined execution.

Codificación

Ejemplo

Un set de instrucciones de **16 bits**, está constituido por instrucciones de **0, 1 y 2 operandos**, los operandos tienen un tamaño de 6 bits, si en él ya existen 5 instrucciones de 2 operandos y 33 instrucciones de 0 operandos. ¿Cuál es el número máximo de instrucciones de 1 operando que se pueden codificar con dicho ancho de palabra y como sería la codificación del ISA?

A	B	Y
0	0	0 op
0	1	1 op
1	0	2 op
1	1	X

Se necesita diferenciar entre si es un operando de 0, 1 o 2 operandos. Para identificarlos se usan 2 bits.

1operandos \rightarrow 6bits \rightarrow **256instrucciones**

15	14	13	8	7	0
# Op	1 operandos (6 bits)			ld inst. (256 instrucciones)	

0operandos \rightarrow 0bits \rightarrow 33instrucciones

15	14	13	0
# Op	ld inst. (14 bits o 16384 instrucciones)		

2operandos \rightarrow 12bits \rightarrow 5instrucciones

15	14	3	2	0
# Op	2 operandos (12 bits = 2×6 bits)			ld inst.

Paralelismo

Técnica de programación e implementación en la que se pretende realizar operaciones simultáneamente, con el fin de reducir tiempos de ejecución, en un procesador.

Existen diferentes tipos de paralelismo:

- Paralelismo a nivel de bit.
- Paralelismo a nivel de instrucciones.
- Paralelismo a nivel de datos.
- Paralelismo a nivel de tareas.
- Paralelismo a nivel de hilos.

Paralelismo a nivel de instrucción (ILP)

Técnica de **paralelismo** basada en la ejecución simultánea de instrucciones.

Posee dos enfoques:

- Paralelismo por hardware (dinámico) - ejecución

- Paralelismo por software (estático) - compilación

Tipos de ILP:

- Segmentación Pipeline

- Ejecución fuera de orden (OoOE)

- VLIW

- CPU's superescalares



ILP - Bloque básico

Corresponde a una sección de código secuencial que no presenta ramificaciones (branches) hasta el final del mismo. En un bloque básico, el flujo de control es **secuencial** y no se detiene hasta terminar el bloque.

Tiempo de ejecución en el peor de los casos (WCET)

Tiempo máximo que tarda en ejecutar un código en un hardware específico. Fundamental en sistemas de tiempo real.

- Análisis estático, típicamente.
- x_i puede tener restricciones estructurales y/o dadas por el programador.

Dado un programa con N bloques básicos, donde cada bloque B_i , que posee un tiempo de ejecución máximo c_i , se ejecuta un número de veces x_i , el WCET es:

$$\sum_{i=1}^N c_i \cdot x_i$$

```
w = 0;  
x = x + y;  
y = 0;  
if( x > z)  
{  
    y = x;  
    x++;  
}  
else  
{  
    y = z;  
    z++;  
}  
w = x + z;
```

Source Code

```
w = 0;  
x = x + y;  
y = 0;  
if( x > z)
```

```
y = x;  
x++;
```

```
y = z;  
z++;
```

```
w = x + z;
```

Basic Blocks

Mejoras ILP mediante Hardware

Técnicas que permiten hacer *hardware* capaz de procesar mas instrucciones simultáneamente

Segmentación Pipeline.

OoOE.

VLIW.

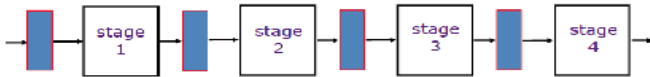
Superescalar.

Especulación*.

Renombramiento de registros.

Segmentación - Pipeline

Técnica utilizada en el diseño de CPUs para aumentar el rendimiento, mediante la separación de las etapas en el proceso de ejecución de una instrucción



El objetivo del diseñador del pipeline es balancear la longitud de cada etapa del pipeline.

$$\frac{\text{Time per instruction on an unpipelined machine}}{\text{Number of pipeline stages}}$$

De esta forma se obtiene un *speedup* teórico de N (número de etapas).

Etapas básicas de un pipeline

- Búsqueda de instrucción (**IF**): Enviar el PC a memoria, traer nueva instrucción, actualizar el PC.
- Decodificación de instrucción (**ID**): "Traducción de instrucción", lectura de registros operandos.
- Ejecución /Dir efectiva (**Ex**): Operaciones en ALU: Memoria efectiva, R-R, R-I.
- Acceso a memoria (**MEM**): Instrucciones L/S.
- Escritura a registros (**WB**): Escritura de resultados R-R o instrucciones L/S a banco de registros.

Número de instrucción	Ciclo de reloj								
	1	2	3	4	5	6	7	8	9
Instrucción i	IF	ID	EX	MEM	WB				
Instrucción i + 1		IF	ID	EX	MEM	WB			
Instrucción i+2			IF	ID	EX	MEM	WB		
Instrucción i+3				IF	ID	EX	MEM	WB	
Instrucción i+4					IF	ID	EX	MEM	WB

Ganancia en desempeño

La ganancia en el desempeño debido al pipeline está dada por

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} \\ &= \frac{\text{CPI}_{\text{unpipelined}} \times \text{Clock cycle}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}} \times \text{Clock cycle}_{\text{pipelined}}} \\ &= \frac{\text{CPI}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}}} \times \frac{\text{Clock cycle}_{\text{unpipelined}}}{\text{Clock cycle}_{\text{pipelined}}}\end{aligned}$$

Ventajas

- Aumenta el rendimiento del CPU.
- Brinda determinismo en ejecución de instrucciones.

Desventajas

- Etapas e instrucciones lentas afectan el rendimiento general
- Mayor complejidad, más hardware.
- Latencia es ligeramente mayor.

Riesgos

Ventajas y desventajas

Riesgos en la segmentación

Riesgo: Situación que previene que la siguiente instrucción pueda ser ejecutada en el ciclo de reloj correspondiente.

Riesgos estructurales: conflictos de hardware entre instrucciones.

Riesgos de datos: Causado por dependencias **reales** entre datos de instrucciones

Riesgos de control: Saltos y branches.

Los riesgos reducen el desempeño ideal ganado por la técnica de pipeline.

Stalls

Los riesgos provocan que el pipeline se *detenga* (**stall**)

- Las instrucciones calendarizadas antes de la instrucción detenida deben terminar su ejecución.
- Las instrucciones calendarizadas después de la instrucción detenida deben ser detenidas igualmente.

Se debe tomar en cuenta el tiempo detención por instrucción:

$$CPI_{\text{pipelined}} = \text{IdealCPI} + \frac{\text{Pipeline stall clock cycles per instruction}}{\text{instruction}}$$

Si se toma IdealCPI = 1:

$$\text{Speedup} = \frac{CPI_{\text{unpipelined}}}{1 + \text{Pipeline stall clock cycles per instruction}}$$

Riesgos

Definición

Unidades Funcionales
- FU

Riesgos estructurales

Riesgos de Datos

Adelantamiento

Referencias

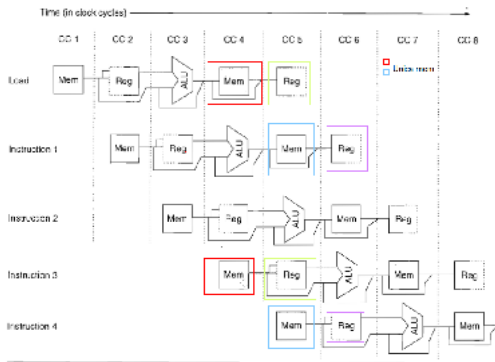
Unidades Funcionales - FU

Una unidad funcional es un elemento, dentro del hardware de un procesador, que realiza una función específica: Ejemplos:

Riesgos Estructurales

ALU Multiplicadores Contadores
fpALU Comparadores Entre otros.

En un procesador con pipeline se requieren unidades funcionales duplicadas para alojar recursos en todas las posibles combinaciones de instrucciones.



Cuando no hay recursos necesarios para evitar conflictos en uso de hardware se tiene un **riesgo estructural**

Posibles Soluciones

Uso de stalls.

Solución simple

Disminuye el rendimiento -> 1 ciclo más

Duplicar hardware

Solución más compleja. Puede requerir lógica de control adicional

Puede ser cara (más hardware, más potencia)

No disminuye el desempeño

Riesgos de datos

Los **riesgos de datos** ocurren cuando en el pipeline se cambia el orden de acceso a lectura/escritura de operandos, de forma que el orden difiere de la ejecución secuencial en un procesador sin pipeline.

DADD	R1,R2,R3
DSUB	R4,R1,R5
AND	R6,R1,R7
OR	R8,R1,R9
XOR	R10,R1,R11

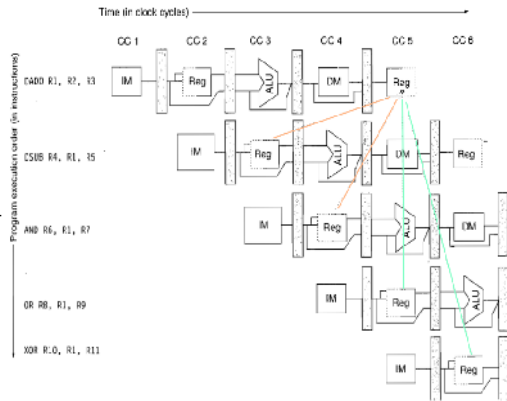
Posibles Soluciones

Uso de stalls.

Solución simple

Disminuye el rendimiento -> requiere más de 1 ciclo adicional

Adelantamiento: Consiste en mover el resultado de un registro directamente hacia la siguiente etapa donde se necesita, si esperar al WB.



Riesgos de control

Los **riesgos de control** ocurren cuando al tener la ejecución de una instrucción *branch* se puede modificar o no el valor del PC, alterando el flujo de ejecución del programa.


- Dependiendo de si el salto se toma o no (etapa ID), la siguiente instrucción será o no la correcta.

Posibles Soluciones

Stall de un ciclo.

- Después de cada salto se realiza dos IF.

Predicción de salto: Estrategia basada en métodos estadísticos o probabilistas para tratar de predecir el si salto es tomado o no. En caso de fallar la predicción, se debe vaciar (flush) el pipeline.

Stall							
Branch instruction	IF	ID 	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

Ejecución fuera de orden (OoOE)

Tipo de ILP en el que las instrucciones de **ejecutan** en un orden distinto al que fueron programadas.

OoOE

1. SUB R1, R2, R3
2. ADD R4, R3, **R1**
3. ROR R2, R2, **#4**

En lugar de 1,2,3 (riesgo de datos), se puede cambiar el orden a 1,3,2 y ganar desempeño.

Dependencias en pipeline

En una arquitectura que implementa pipeline (y/o otras formas de ILP) se pueden presentar 3 tipos de dependencias entre instrucciones:

- ➊ Dependencias de datos (reales).
- ➋ Dependencias de nombre.
- ➌ Dependencias de control.

Las dependencias, en general, son **producto de los programas**

Si una dependencia lleva a un riesgo, su detección y corrección son propiedades de la **organización** del pipeline.

Dependencias de datos (reales)

Una dependencia de datos entre instrucciones puede surgir en los siguientes casos:


- La instrucción i produce un resultado que puede ser utilizado por la instrucción j .
- La instrucción j depende de un dato de la instrucción k , y la instrucción k depende de un dato de la instrucción i .

```
L.D    F0,0(R1)    ;F0=array element
ADD.D  F4,F0,F2    ;add scalar in F2
S.D    F4,0(R1)    ;store result
DADDUI R1,R1,#-8   ;decrement pointer 8 bytes
BNE    R1,R2,LOOP  ;branch R1!=R2
```

Ejemplo

1. ADD **R3**,R2, R1
2. SUB R1, **R3**, 1

```
Loop:   L.D    F0,0(R1)    ;F0=array element
        ADD.D  F4,F0,F2    ;add scalar in F2
        S.D    F4,0(R1)    ;store result
```



Componentes de una dependencia de datos

Al tratar con dependencia de datos se debe tomar en cuenta:

- La **posibilidad** de un riesgo.
- El orden en que las instrucciones deben ejecutarse (caso OoOE).
- Límite máximo de paralelismo que puede ser explotado.

Soluciones a una dependencia de datos

Una dependencia no implica necesariamente un riesgo, pero deben ser atendidas.

- Mantener la dependencia, pero evitar el riesgo
- Eliminar la dependencia por la transformación del código**

Pueden ser implementadas por software o por hardware.

Dependencia de nombre

Ocurre cuando dos instrucciones usan el mismo registro (o dirección de memoria), pero **NO** hay relación o flujo entre las instrucciones.

Dos tipos:

Antidependencia

Ocurre cuando una instrucción j escribe a un registro o posición de memoria que una instrucción i lee.

Ejemplo

1. ADD R3, **R2**, R1
2. SUB **R2**, R5, 1

Dependencia de salida

Ocurre cuando una instrucción i y una instrucción j escriben al mismo registro o dirección de memoria.

Ejemplo

1. ADD R3, R1, R2
2. SUB **R4**, R3, 1
3. ADD **R4**, R5, R5

Solución dependencias de nombre

Dado que no hay transmisión entre las instrucciones, **no son dependencias** reales.

- Pueden ser ejecutadas en paralelo

Solución: **Renombramiento de registros**

- Por hardware: Calendarización dinámica de instrucciones. RRU: register renaming unit.
- Por software: Calendarización estática. Compilación.

Dependencias de control

Una dependencia de control determina el orden de ejecución de una instrucción *i*, con respecto a una instrucción de salto previa.

```
if p1{  
    S1;  
}  
if p2{  
    S2;  
}
```

No debe invertirse el orden de ejecución cuando existen dependencia de control.

- 1 Una instrucción dependiente de control en un salto NO puede moverse antes del salto.
- 2 Una instrucción que NO es dependiente de control NO puede moverse justo después de un salto.

Implicaciones

Ejemplo 1

_init:

```
ADD R1, R1, R2  
BEQZ R1, T0, L1  
SUB R1, R2, R3
```

L1:

done

Riesgos de datos

Un riesgo de datos se puede tener cuando se presenta una dependencia de **nombre** o real de **datos** entre instrucciones lo suficientemente cercanas para que se pueda producir un cambio en el orden de acceso a los operandos.

Tres tipos de riesgos de datos:

ReadAfter Write - RAW)

Se presenta cuando una instrucción *j* trata de leer un operando antes de que la instrucción *i* lo escriba, obteniendo un valor antiguo.

Ejemplo

1. ADD **R3**, R2, R1
2. SUB R1, **R3**, 1

Escritura después de lectura (WAR)

Se presenta cuando la instrucción *j* trata de escribir un destino **antes** de que sea leído por la instrucción *i*, lo que provoca que *i* lea el nuevo valor (incorrecto).

Ejemplo

- i ADD R4, R2, **R1**
- j SUB **R1**, R3, 1

Escritura después de escritura (WAW)

Se presenta cuando la instrucción *j* trata de escribir un operando **antes** de que se escrito por la instrucción *i*. Las escrituras se realizan en el orden incorrecto.

Ejemplo

- i ADD **R1**, R2, R3
- j SUB **R1**, R3, 1

Técnicas de software para mejorar ILP

Se consideran estáticas.

Se realizan durante tiempo de compilación.

- 'Información' para branch prediction.
- Reordenamiento de código (memoria).

Durante compilación se puede reorganizar el código de forma que se optimice el uso de procesador.

- Renombramiento de registros.

Durante compilación se puede detectar falsas dependencias de datos y renombrar registros para aumentar el ILP.

- *Loopunrolling*.

La idea principal es reducir la cantidad de iteraciones y lógica de control (instrucciones condicionales) en un bloque de código para mejorar el ILP.

Puede darse como optimización del compilador o en el código fuente directamente.