**1 - Introducción, rendimiento**

**CPU**

* Encargado del control y ejecución de operaciones
* Partes:
  + Unidad de control (CU): Dirige el resto de unidades
  + Unidad de procesamiento (ALU): Realiza operaciones aritmético/lógicas
  + Banco de registros: Elementos de almacenamiento, guardan datos a ser procesados por la ALU
  + Memoria cache: memoria SRAM rápida para un acceso eficiente a datos e instrucciones
* Sistema multinúcleo → varias CPUs en un chip

**ISA (Instructions set architecture)**

* Conjunto de instrucciones máquina que entiende una CPU
* Distintos CPU pueden tener el mismo ISA pero diferenciarse en su microarquitectura
* **ABI:** ISA + interfaz de sistema operativo

**RISC: Reduced Instruction Set Computer**

* ISA pequeño y optimizado, de instrucciones uniformes.
* Accede a memoria con carga y almacenamiento

**CISC: ComplexInstruction Set Computer**

* ISA con muchas instrucciones de tamaño y formato variable
* Accede a memoria con cualquier instrucción

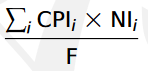
**Rendimiento**

* **Tiempo de respuesta:** tiempo entre comienzo y final de un evento, en segundos.
* **Productividad:** Cantidad de trabajo en un tiempo determinado, medida en transacciones.
* Afectado por algoritmo, lenguaje de programación, compilado e ISA, procesador y sistema de memoria, entrada/salida y sistema operativo.

**Rendimiento relativo !**

* El rendimiento se puede medir como 1 / tiempo de ejecución de un programa determinado.
* Para comparar el rendimiento relativo de B frente a A, se calcula
* Se puede medir tiempo transcurrido (incluyendo todos los aspectos) o sólo tiempo de CPU (no incluye tiempo de E/S ni el debido a la ejecución simultánea de otros prgoramas)
  + El tiempo de CPU se divide en tiempo de usuario y tiempo de sistema.

**CPI !!!!!!!**

* **CPI:** Número de ciclos de reloj que necesita una instrucción máquina para ejecutarse. Depende de la instrucción y del hardware de la CPU.
* Dada una aplicación, podemos determinar el CPI medio de sus instrucciones si conocemos su número de instrucciones total y su número de ciclos.
* CPI de una aplicación puede ser menor que 1 si se ejecutan varias instrucciones en un ciclo.
* TCPU = . Esta fórmula se utiliza para calcular rendimiento relativo.

**Benchmarks**

* Benchmarks sintéticos: Estructuras de programas reales, no son aplicaciones
* Benchmarks naturales: Utilizan programas reales.

**Consumo de potencia !!**

* **Potencia dinámica:** Debido a la carga de los condensadores. Se puede estimar como ½ \* C \* V2 \* f
* **Potencia estática:** Debida a las corrientes de fugas. Se puede estimar como V \* Ifugas.

**Sistemas multicore !!**

* Más de un procesador por chip
* Dificil de programar, requiere equilibrar la carga de trabajo entre cores y optimizar su sincronización.
* **Ley de Amdahl:** la posible mejora del rendimiento está limitada por la proporción en la que se utilice la prestación mejorada.
* Tmejorado =
* La hipotética eficiencia máxima está limitada por la parte no mejorable.
* Otra formulación: Si a un código que se ejecuta en tiempo T le aplicamos una mejora de un F % en un factor M, el nuevo tiempo de ejecución es:

**2 - Representación de la información**

**Codificación de números binarios con signo**

* **Signo-magnitud:** El bit más a la izquierda (MSB) es el bit del signo, 1 negativo y 0 positivo.
  + Los bits de magnitud son el número sin signo.
  + Con n bits se pueden representar números en el intervalo {-(2n-1 -1), (2n-1 -1)}
  + Ej.: -16 = 110000
* **Complemento a 1:** El bit más a la izquierda es el bit del signo. Además, los números negativos son el complemento a 1 del correspondiente número positivo: intercambia los 0 y los 1.
  + Mismo intervalo que signo-magnitud
  + Dos representaciones posibles para el 0: 0000(+0) o 1111(-0)
  + Ej.: -16 = 101111
* **Complemento a 2:** El bit a la izquierda es el signo, los números negativos son el complemento a 2 del positivo:
  + Para calcular el complemento a 2 de un binario N de n bits, se calcula la diferencia 2^n-N y se toman los n bits más a la derecha.
    - Se realiza el complemento a 1 y suma 1 bit. Si el número está en un extremo, se suprime un bit a la izquierda. Ej.: -16 = ~~1~~**10000**
  + Intervalo: {-(2n-1), (2n-1 -1)} (**1 número negativo más** que signomagnitud)
  + 0 sólo tiene una representación, 0. Non existe o -0.
* **NOTA:** Para C1 y C2, los positivos son iguales que en signo magnitud.

**Punto flotante !!!!!!!!!**

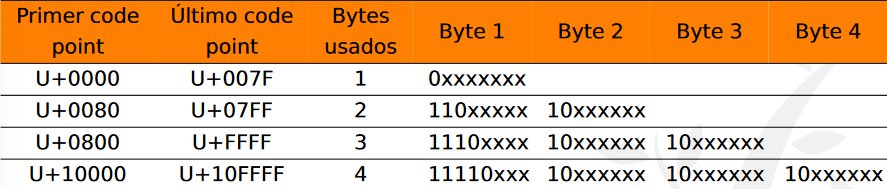
* Representación basada en: **Signo (s), exponente (e) y mantisa (M)**
  + Valor : (-1)S x M x be [siendo b la base]
* Se utilizará el estándar **IEEE 754.**
* Formatos (solo se estudiarán los 2 primeros):
  + Simple precisión: 32 bits, float en C
    - 1 bit de signo, 8 de exponente, 23 de mantisa normalizada
  + Doble precisión: 64 bits, double en C
    - 1 bit de signo, 11 de exponente, 52 de mantisa normalizada
  + Precisión extendida: 80 bits, long double en C. No siempre implementado
  + Cuádruple precisión: 128 bits, tipo \_\_float128 en gcc v>=4.6
* Al leer un número (ej: 11,011x20011) cambiamos el exponente para que en la parte entera quede **solo un 1**.
  + El ejemplo pasa a ser 1,1011x20100
  + De esta forma, sabemos que siempre habrá un 1 en la parte entera, y no es necesario almacenar este bit.[[1]](#footnote-0)
* Se codifica en **exceso:** el exponente que se guarda es expreal + exceso
  + **exceso** es 127 en simple precisión y 1023 si doble (**2n-1-1**, siendo n los bits de exponente)
  + De esta forma, el exponente representado será siempre positivo, y estará en [1,254] o [1,2048] según precisión. [1,2n]
* Tras estos cambios, el valor del número será **(-1)s x 1.M x 2e-exceso**
* **Ejemplo:** Valor decimal de 0x55F20000 en precisión simple.
  + Se pasa a binario: 0101 0101 1101 0010 0000 0000 0000 0000
    - Signo: 0 → positivo
    - Exponente: 101010112 = 17110
      * Exponente real: 171-127 = 44
    - Mantisa: 1,1010012 = 1,64062510
    - Valor: (-1)0 x 1.640625 x 244 **= 2.88621602 x 1013**
* Los exponentes 0 y 255/2047[[2]](#footnote-1) se reservan para **números especiales:**
  + Exp 0 y mantisa 0 representa el **0**.
    - Si la mantisa es 0 y el exponente no, el número será 2-126 y no 0, pues se debe añadir el 1 asumido
  + Exp 0 y mantisa ≠ 0 representa un valor **denormal**
  + Exp max y mantisa 0 representa
  + Exp max y mantisa ≠ 0 representa Not a Number (+-NaN)

**Números denormales**

* Números no normalizados próximos a 0.
* Su valor es **(-1)s x 0.M x 21-exceso**
* El exponente será siempre -126 o -1022 según representación.
* **Ejemplo:** 0 00000000 00010100000000000000000
  + Valor = (-1)0 x 0.000101 x 2-126 = 1.01 x 2-130 = 9.18 x 10-40

**Representación de caracteres !**

* **ASCII**: 7 bits → 128 caracteres. Sólo incluye alfabeto inglés
  + ISO/IEC 646: Similar a ASCII, pero incluye variantes para algunos países
  + **Extended ASCII**: 8 bits → 256 caracteres. ASCII + otros 128 caracteres
    - ISO/IEC 8859: 15 estándares para diferentes idiomas
    - Windows-1252, Mac OS Roman: lol
* **Unicode:** Cada carácter está representado por un **code-point** en hexadecimal, desde U+000016 hasta U+10FFFF16 → **1.114.112 códigos**
  + Los code-points se organizan en 17 planos de 216 code-points cada uno. Los planos se organizan en charts.
  + **UTF-8:** Símbolos de longitud variable, utiliza ASCII de 7 bits.
    - El byte 1, además de tener datos, indica el **nº de bits de longitud!!**:



* + - Ejemplo: El símbolo € es U+20AC (3 bytes). Su UTF-8 será: 11100010 10000010 10101100 → 0xE282AC
  + **UTF-16**: Símbolos de longitud variable: 1 o 2 palabras de 16 bits (2 o 4 bytes), siendo la mayoría de caracteres de 16 bits.
  + **UTF-32:** Exactamente 32 bits por carácter.
    - Más sencillo pero menos eficiente en almacenamiento

**3 - Ensamblador**

**Introducción**

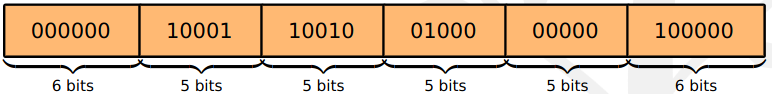
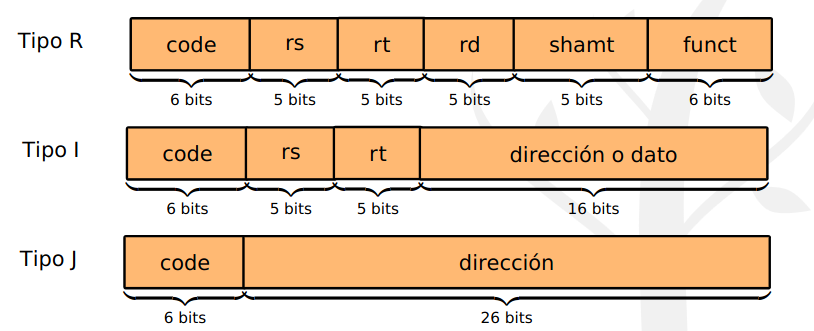
* El lenguaje **ensamblador** es una representación simbólica del código máquina en binario que utiliza el ordenador.
* **PC:** Contador de programa. Indica la dirección de la próxima línea de código a ejecutar.
* **IR:** Registro de instrucción: Contiene el contenido de la última línea de código ejecutada.
* Los programas se dividen en dos partes:
  + .text : sección obligatoria, contiene el código. Empieza siempre con:

.globl main

.main:

* + .data: opcional (normalmente necesaria). Contiene declaraciones de variables
    - Las variables se declaran con var: .tipo

**Instrucciones MIPS !!!**

* Longitud de instrucción, registros y palabras fija de **32 bits[[3]](#footnote-2).**
* Una **instrucción** está organizada en **campos**, distintos según el tipo de operación.
  + **Tipo R:** Instrucciones aritmético/lógicas. Ejemplo: add $t0, $s1, $s2
  + **Tipo I:** Instrucciones de transferencia de datos, saltos, condicionales, operandos inmediatos
  + **Tipo J:** Instrucciones de salto incondicional.

**Direccionamiento**

* Indica la localización de los operandos de la instrucción.
* **Modos:**
  + **Inmediato:** el operando es una constante en un campo de la instrucción.
    - Operaciones AL de tipo inmediato. Ejemplo: addi $t1, $t0, 20
  + **Registro:** El operando está en un registro cuya ubicación se indica en la instrucción.
    - Operaciones aritmético-lógicas. Ejemplo: add $t2, $t0, $t1
  + **Base con desplazamiento:** El operando está en una dirección de memoria, que se obtiene de la suma del desplazamiento + el contenido de un registro base.
    - Operaciones de transferencia de datos. Ejemplo: lw $t1, 20($t0)
  + **Relativo al PC**: El operando está en una dirección de memoria, que se obtiene de la suma del contador del programa + el desplazamiento.
    - Operaciones de salto condicional. Ejemplo: beq $t1, $t0, -25
  + **Pseudodirecto:** El operando está en una dirección de memoria, obtenida como la concatenación de 26 bits indicados en la instrucción y los 4 bits más significativos del contador de programa. Ejemplo: j salto
    - Operaciones de salto incondicional.Ejemplo: j 1200

**Tipos de variables**

* **.ascii** “string a almacenar”: Almacena lista de caracteres
* **.asciiz** “string a almacenar”: Almacena lista de caracteres acabando con 0. Cada caracter ocupa 1 byte, se debe considerar también el espacio del \0.
* **.word** palabra, palabra2, …: Almacena lista de palabras en posiciones contiguas. Cada palabra ocupa 4 bytes.
* **.space** n: Reserva n bytes de espacio en la memoria.
* **.float f1, f2, …:** Almacena lista de números en punto flotante
* **.double d1, d2, …:** Almacena lista de números en punto flotante con prec dobleçç

**Instrucciones aritmético-lógicas**

* **add/sub/and/or $1, $2, $3:** Almacena en $1 el valor de la operación correspondiente de $2 y $3.
* **addi/subi/andi/ori $1, $2, cte:** Almacena en $1 el valor de la operación correspondiente de $2 y la constante.
* **sll $t0,$s1,4**: Almacena en $t0 el resultado de desplazar $s1 4 posiciones a la izquierda.
* Para producto y división se usan **HI** y **LO**, registros para operar con operandos de 64 bits.
  + **div $1, $2** almacena en HI el resto y en LO el cociente de dividir $1 entre $2
  + **mult $1, $2** almacena en LO el resultado de multiplicar $1 por $2
  + **mfhi/mflo $1** copian el valor de HI en $1 y viceversa.
* **slti $1, $2, cte.**: Almacena en $1 un 1 si el valor del registro $2 es menor que cte. Se usa junto con beq y bne para crear blt, bgt, ble, bge.

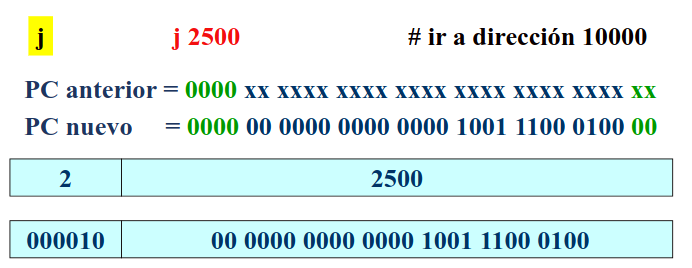
**Instrucciones de carga**

* **lui** **$1, cte10:** Carga la constante en los 16 bits más significativos del registro $1. La etiqueta puede tener signo y se carga en CA2. Parte de la.
* **lw $1, d($2):** Carga en el registro $1 la palabra almacenada en la dir. de memoria [(direccion almacenada en $2) + d]. d debe ser múltiplo de 4.
* **sw $1, d($2):** Carga en la dir. de memoria [(direccion almacenada en $2) + d] la palabra almacenada en $1. d debe ser múltiplo de 4.
* **lb $1, d($2)!:** Carga en el registro $1 el byte almacenado en la dir. de memoria [(direccion almacenada en $2) + d], sobreescribiendo el contenido de $1. d no es necesariamente múltiplo de 4.
  + Tener en cuenta que es little endian. Por ejemplo, si en la dirección que indica ($2) hay 0x1b2b3b4b, lb $1, 3($2) cargará 0x0000002b
* **sb $1, d($2):** Carga en la dir. de memoria [(direccion almacenada en $2) + d] el primer byte almacenado en $1 . d no es necesariamente múltiplo de 4.

**Pseudoinstrucción[[4]](#footnote-3) la [load adress]**

* **la $tn, variable:** carga en el registro $tn la dirección de memoria donde está almacenada la variable.
* Una dirección de memoria ocupa 32 bits, y una instrucción en total puede ocupar 32 bits, de los cuales solo 16 están disponibles.
* La ejecución de está pseudoinstrucción se divide en dos instrucciones, cargándose los bits de 16 en 16.
  + **lui $tn, [primeros 16 bits direccion memoria variable]**: 16 bits más signicativos del registro $tn.
  + **ori $tn, $tn, 0x000[ultimos 16 bits]:** 16 bits menos significativos.

**Saltos !!!!!!!**

* Hacen que el programa salte a otra instrucción que no sea la siguiente en memoria.
* **Salto incondicional (j, jr, jal):** Saltan siempre.
  + No varían los 4 primeros bits del PC, almacenan el valor de la constante en los 26 siguientes, y los últimos 2 bits son siempre 00.
  + 
  + **jr $s1** avanza a la dirección contenida en $s1. Es de tipo R.



* **Salto condicional (beq, bne)!:** Saltan si se cumple una condición
  + **beq $s1,$s2,25:** Si $s1==$s2, se suma al PC el valor de la etiqueta multiplicada por 4, en CA2 (admite valores negativos)

**Subrutinas**

* Se realizan con saltos y pasando parámetros mediante registros.
  + Los parámetros de entrada se facilitan en **$a0-$a3**
  + Los resultados se guardan en **$v0-$v1**
* Se usa la instrucción **jal**, que realiza un salto y además guarda la dirección de retorno en el registro 31 ($ra)
  + Para volver al programa principal, se utilizaría **jr $ra**

**Pilas**

* Parte de la memoria utilizada como una estructura, utilizada para mantener el estado del sistema al llamar a una subrutina.
  + Existen dos posibles convenios: puede ser el invocador o el invocado el responsable de guardar/restaurar.
  + El contenido de los registros temporales ($t0-$t9) debe ser guardado antes de llamar a la subrutina.
  + El contenido de los registros salvados ($s0-$s7) debe ser preservado por la subrutina utilizando la pila.
* Crece de posiciones altas a bajas de memoria. La última posición ocupada es indicada por el **stack pointer** (registro 29, $sp)
* Para realizar **push:**
  + Se resta 4 al puntero de la pila [addi $sp,$sp,-4]
  + Se guarda el valor mediante sw [sw $t0,0($sp)]
  + Se debe restar antes y guardar después. De esta forma, la primera dirección de la pila (0x7000000) estará vacía siempre.
* Para realizar **pop:**
  + Se recupera el valor mediante lw [lw $t0,0($sp)]
  + Se suma 4 al puntero de la pila [addi $sp,$sp,4]

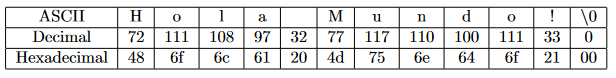
**Syscall**

* Syscall realiza una llamada al sistema operativo para realizar una función, determinada por el contenido del registro $v0.

1. **print\_int:** Imprime como entero el contenido de $a0
2. **print\_float:** Imprime como float el contenido de $f12
3. **print\_double:** imprime como float el contenido de $f12 y $f13 unidos
4. **print\_string**: Imprime como string el contenido en la posición que indica $a0
5. **read\_int:** Pide por teclado un entero que se almacena en $a0
6. **read\_float:** Pide por teclado un float que se almacena en $f0
7. **read\_double:** Pide por teclado un float que se almacena en $f0 y $f1
8. **read\_string:** Pide por teclado un string que se almacena en $a0 y su longitud en $a1

**10. exit:** Finaliza la ejecución.

**Memoria !!!**

* Los números en memoria se almacenan en hexadecimal, en complemento a 2 con signo. Cada word ocupa 4 bytes (32 bits).
* Los strings se almacenan en memoria contigua, divididas en palabras de hasta 4 bytes [32 bits]. Cada char ocupa un byte.
  + Dentro de cada palabra los caracteres están de derecha a izquierda (caracter de la izquierda en posicion menos significativa). Sigue una configuración **Little Endian**. Ejemplo:



|  | b3 | b2 | b1 | b0 |
| --- | --- | --- | --- | --- |
| 0x10010000 | 61 | 6c | 6f | 48 |
| 0x10010004 | 6e | 75 | 4d | 20 |
| 0x10010008 | 00 | 21 | 6f | 64 |

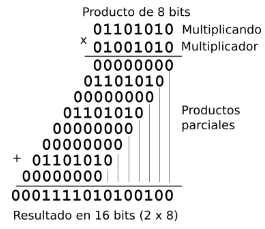
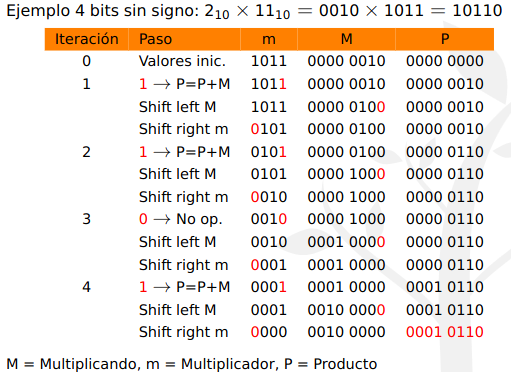
(se en lugar de escribir .asciiz “hola mundo” se escribe directamente .word 0x616c6f48, gárdase a palabra nese mismo orden, 0x616c6f48)

**4 - Aritmética del computador**

**Suma y resta en punto fijo**

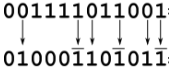
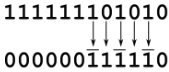
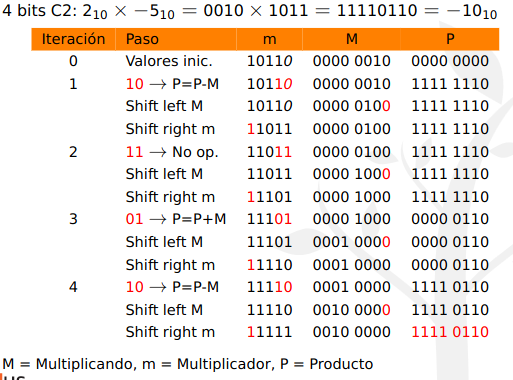
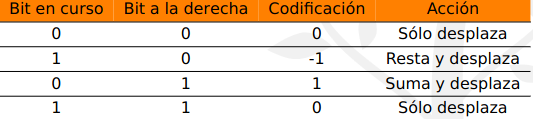
* Los dígitos se suman bit a bit de derecha a izquierda, pasando acarreo
* Para **restar**, el sustraendo se complementa a 2 y luego se suma.
* Para sumar N bits, se utilizan N full adders, conectados formando un sumador de acarreo enlazado.
* Se detecta desbordamiento si el signo del resultado no es correcto
  + En este caso, se puede hacer wrap-around (C, Java), saturar al extremo o convertir a un tipo más grande (python)

**Multiplicación en punto fijo (sin signo)**



* Por cada bit del multiplicador, empezando por el de la derecha, se suma al producto el producto parcial.
* Se desplaza el multiplicador 1 bit a la derecha y el multiplicando a la izquierda. Se repite el proceso hasta que el multiplicador se haya desplazado por completo.
* Para multiplicar sin signo 32 bits, son necesarios:
  + Un registro con desplazamiento de 64 bit para el multiplicando, y uno de 32 bits para el multiplicador
  + Un registro de 64 bits para el resultado
  + Una unidad de control para las condiciones y una ALU para las sumas

**Multiplicación en punto fijo (con signo) !**

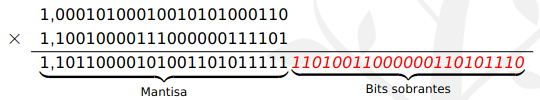
* Se utiliza el **algoritmo de Booth:**
* Recodificamos el multiplicador para aprovechar secuencias de 1s:
  + Se lee del LSB al MSB, dejando los 0 sin cambiar
  + Al encontrar un 1, se cambia por un -1 (1 negado). Si el LSB inicial es un 1, se cambia por -1 igualmente.
  + Luego, todos los 1s consecutivos se cambian por 0s
  + Al llegar a otro 0, este se cambia por un 1, y se reanuda el proceso.
* Ejemplo:
* En circuito:
  + Se añade un bit 0 a la derecha del LSB
  + Se recorre de LSB a MSB siguiendo el criterio de la tabla.
  + La implementación se realiza con **ALU** y **desplazadores**.
  + En el MIPS existen registros especiales de **64 bits** (HI, LO) para almacenar el resultado. (ver instrucciones mult, multi, mflo, mfhi)

**Suma en punto flotante !!**

* **Alineamiento:** Se pone el mismo exponente en ambos operandos
* Se **suma/resta** las mantisas, si es resta se complementa un operando
  + Si hay overflow, el resultado es +-inf
* **Normalización:** Se transforma el resultado para dejar un bit en la parte entera
* Si es necesario, se **redondea** al nº de bits de salida
  + Operaciones no asociativas, debido al error cometido al aproximar

**Multiplicación en punto flotante !!**

* El exponente del resultado es la **suma** de los exponentes de los productos
* Se **multiplican** las mantisas y se **normaliza** el resultado
* Se redondea al número de bits necesarios



**Modos de redondeo !**

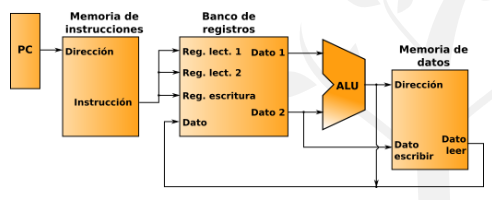
* **Truncamiento:** Se descartan los bits adicionales
* **Redondeo al más próximo, empate a par**: Se redondea al número más proximo y en empate se redondea al número par
* **Redondeo a más/menos infinito:** siempre al número mayor/menor
  + en números negativos, redondeo a +inf é truncamiento. en positivos é ao revés

**5 - Sección de procesamianto y control**

**Ejecución de instrucciones**

* Se siguen los siguientes pasos:
  + Captura: Se accede a la memoria de instrucciones y se lee la instrucción
  + Decodificación: Se decodifica la instrucción y se accede a los operandos, accediendo a memoria si es necesario
  + Ejecución: se realiza la operación necesaria, posiblemente usando la ALU
  + Escritura de resultados: se almacenan los resultados en memoria
* Todo este proceso se produce en **un ciclo de reloj**. Suponemos que cada instrucción se ejecuta en un ciclo (**instrucciones monociclo**)

**Camino de datos !!!!!!**

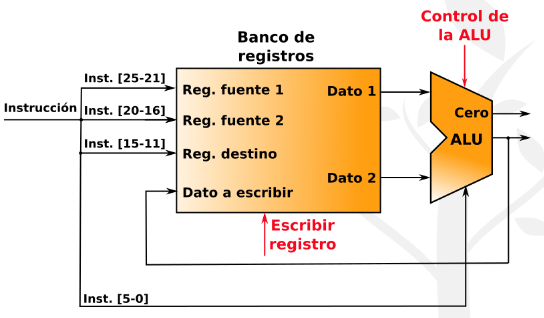


* **Contador de programa (PC)**: Apunta a la dirección de la próxima instrucción que ejecutar. Aumenta en cada ciclo de reloj (por 4 si no hay saltos)

**Incorporación de nuevas instrucciones**

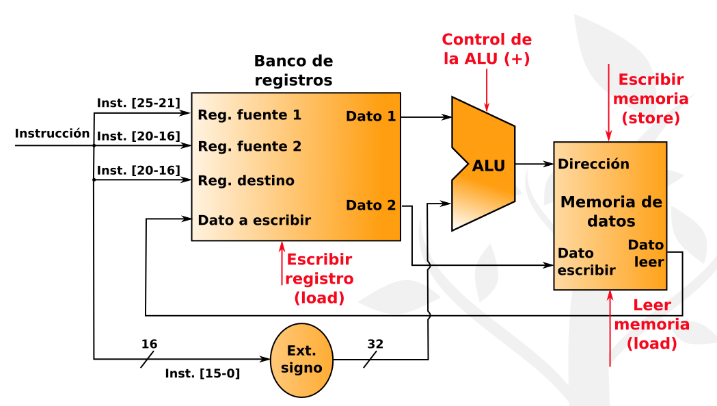
* Añadir nuevas instrucciones puede implicar añadir hardware adicional o modificar la unidad de control.
* Se debe intentar aprovechar el hardware existente.

**Instrucciones aritmético-lógicas !**



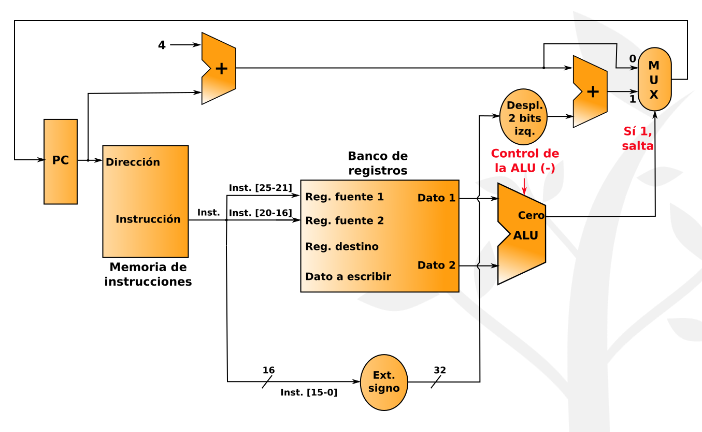
* Los tres registros se almacenan en el banco de registros, que devuelve los dos datos con los que operar
* Los datos se envían a la ALU, junto con el código de la instrucción
* El dato a escribir (resultado) se devuelve al banco de registros

**Carga/almacenamiento**



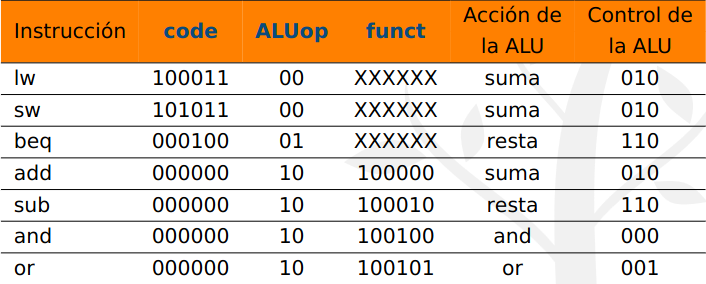
* Los tres registros se guardan en el banco de registros, que devuelve los datos que contienen.
* Si hay desplazamiento, se suma con una ALU
* Se accede a la dirección resultante mediante la memoria de datos, que:
  + Si es una operación de escribir, recibe el dato a escribir
  + Si es una operación de leer, devuelve el dato contenido.

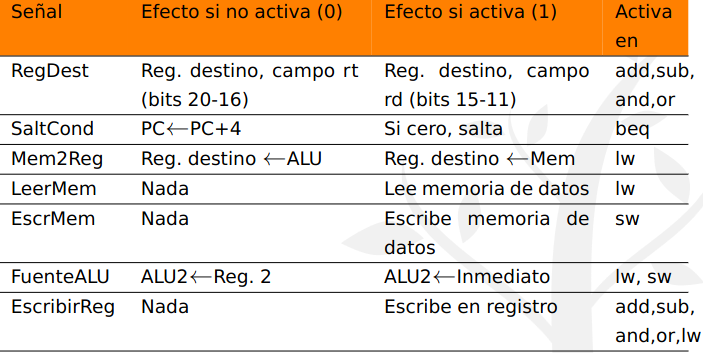
**Salto condicional**



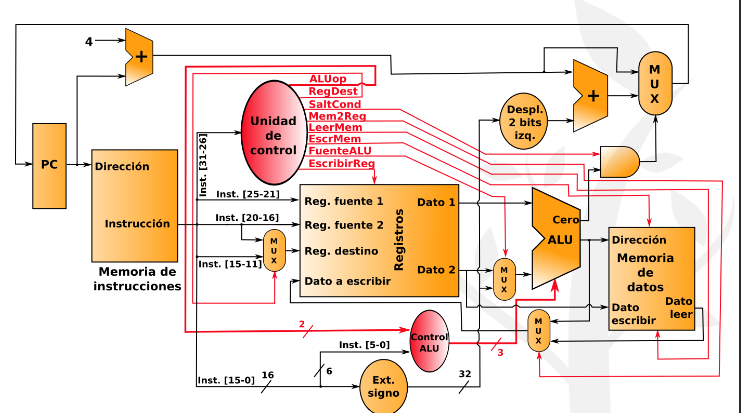
* Los registros se aportan al banco de registros, que devuelve los dos datos que comparar
* Para la condición se utiliza un multiplexor: se restan y si el resultado es igual a 1 (significa que no son iguales) el resultado será distinto de 0 y se salta. Si el resultado es 0, se mantiene el mismo PC pero se le suma 4 (para avanzar a la siguiente instrucción)
* La dirección de la etiqueta se multiplica por 4 (despl. 2 bits izq.) antes de sumarla al PC
* El resultado se guarda en el PC, realizando el salto si es el caso.
* nota: la arquitectura de la imagen completa no permite salto incondicional. Para incorporar esta función, sería necesario un multiplexor adicional que comprobase si SaltIncond==1.

**Unidad de control !!**

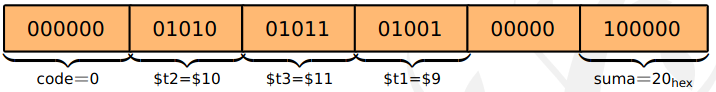
* La **Unidad de control** (UC) recibe los 6 bits de code, y genera:
  + 2 bits de **ALUop** que se pasan a la UCALU (en el esquema, Control ALU), y le indican la operación que debe realizar.
  + 
  + 7 bits separados de señal de control:

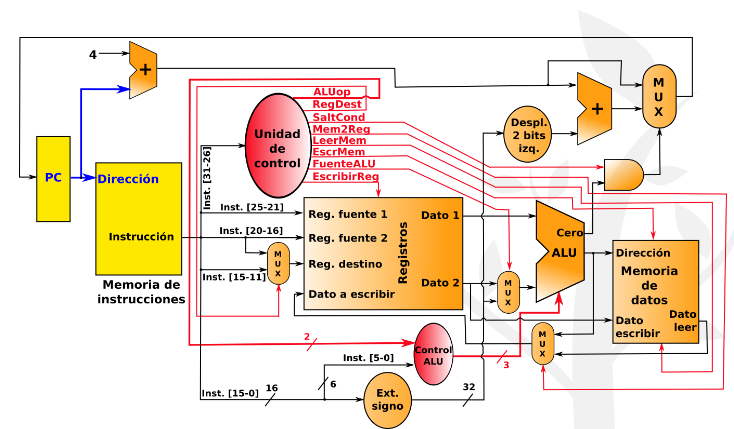


* Las instrucciones aritméticas (add, sub, and, or) además tienen código de función (funct) que recibe directamente la UCALU
* Camino de datos + unidad de control:

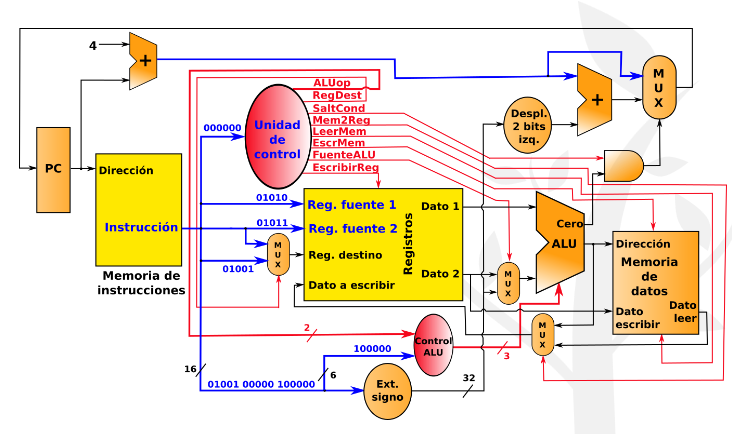


**Ejemplo (suma)**

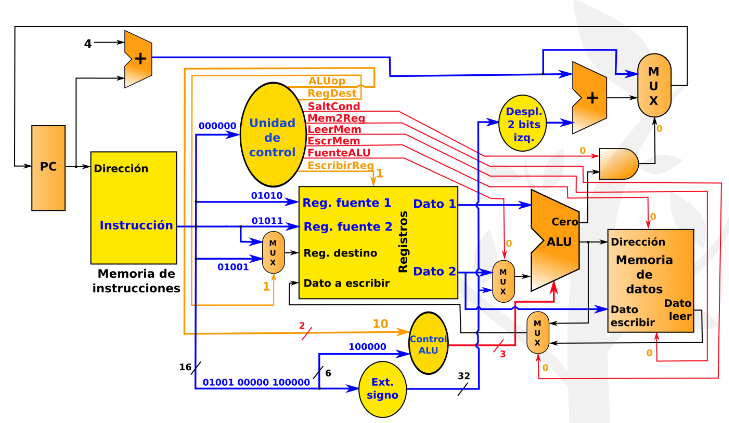
* Instrucción: **add $t1,$t2,$t3**
* 
* Ciclo 1: se pasa la dirección del PC actual, para leer la instrucción.

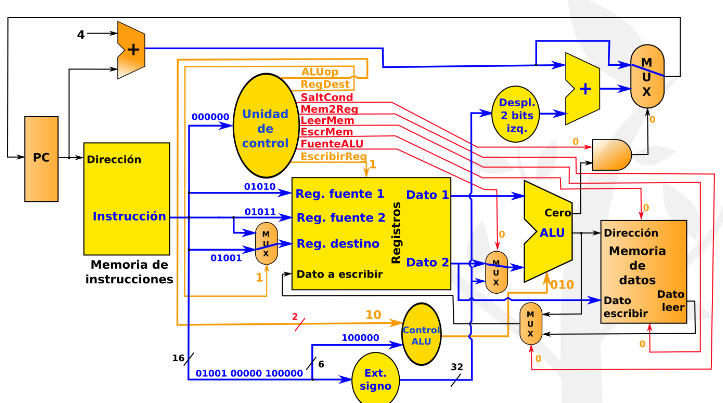


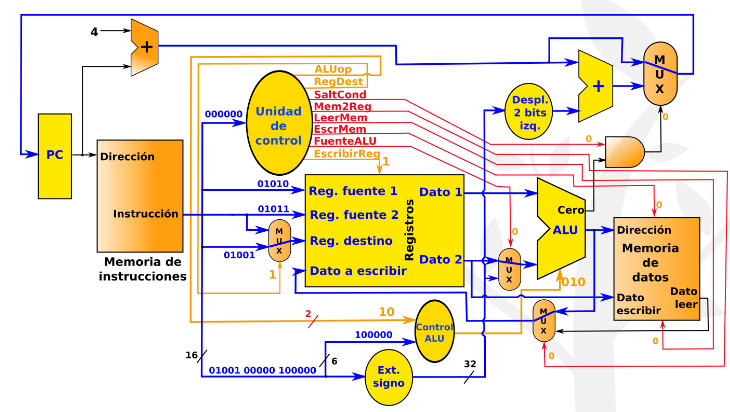
* Ciclo 2: Se leen los registros de fuente que se utilizaarán, se pasa el code a la UC, el funct a la UCALU e non sei o que ao ext. signo



* Ciclo 3: Se accede a la memoria de instrucciones para leer los datos que sumar. La UC genera los bits de ALUop y pone a 1 los de RegDest y EscribirReg.



* Ciclo 4: Se lee el registro de destino. La ALU recibe el código de la operación a realizar (suma
* Ciclo 5: La ALU calcula el resultado de la suma, que se almacena en el registro de destino.



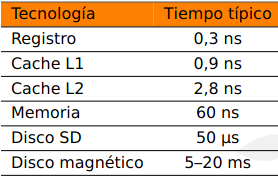
**6 - Jerarquía de memoria**

**Gap CPU-memoria**

* Existe una disparidad de rendimiento entre memoria y CPU: las memorias son mucho más lentas y su rendimiento mejora más lentamente.
* Debido a esto, la memoria limita el rendimiento del sistema.
* Se soluciona organizando la memoria en niveles (jerarquía de memoria)

**Jerarquía de memoria**

1. **Registros de la CPU** (2 KiB[[5]](#footnote-4), 57 GB/s).
2. **Caches** (32Kib-4MiB, 25-45GB/s). Dividida en si misma en varios niveles.
   1. **Principio de inclusión**: Para pasar un dato de un nivel inferior a los registros pasando por cache, se va copiando el dato de un nivel a otro, dejando copias del dato en todos los niveles del cache.
3. **Memoria** **principal** (RAM) (4GiB, 5 GB/s)
4. **Disco duro** (>1000GiB, 150MB/s)
   1. También cintas de back-up, (varios TiB, 30MB/s) e incluye memorias externas (pen drive)

* En niveles superiores, es mayor la velocidad de almacenamiento y lectura, pero mayor coste/byte y menor capacidad.
* La CPU siempre accede al nivel más alto que sea posible para obtener información.
* Ejemplo: con un ciclo de reloj de 0.3ns:

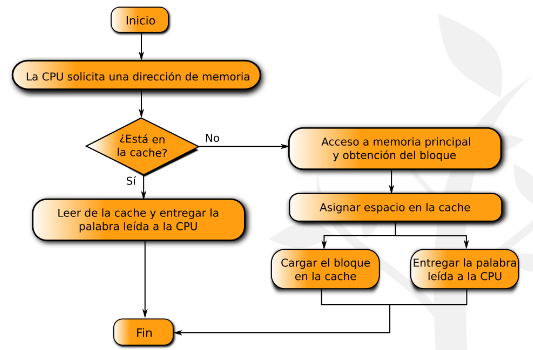
**Gestión de la jerarquía de memoria !!**

* El procesador gestiona registros y cache, los niveles superiores el sistema operativo.
* La jerarquía de memoria está adaptada para seguir los **principios de localidad:**
  + **Princfipio de localidad espacial:** Tras acceder a un dato, es común acceder a un dato cercano. Por ejemplo, al operar con un array.
    - En concreto, es común acceder al dato siguiente en memoria, denominado **principio de localidad secuencial**.
  + **Principio de localidad temporal:** Tras acceder a un dato, es común acceder a la misma dirección en breve. Por ejemplo, en un bucle.
* Debido a esto, cuando se accede a un dato en memoria, no sólo se transfiere este dato a caché, sino que también se mueven con el un conjunto de los datos próximos. Para el movimiento caché ←→ memoria este conjunto se denomina **línea cache**.
* Cuando se pide un dato a un nivel de la jerarquía, puede ocurrir:
  + **Acierto (hit):** El dato se encuentra en el nivel. Tasa de aciertos: nº de aciertos/nº de accesos.
  + **Fallo (miss):** El dato no se encuentra en el nivel. Tasa de fallos: nº de fallos/nº de accesos. En caso de fallo:
    - Se busca en un nivel inferior
    - Se le asigna un lugar en el nivel actual. Es posible que sea necesario desalojar otro dato.
* **Tiempo de acierto:** Tiempo de acceso + tiempo de comprobación
* **Penalización por fallo:** Tiempo de traer un dato desde un nivel inferior
* **Tiempo medio de acceso a nivel de memoria (AMAT)! :** Tiempo de acierto + (Penalización por fallo \* tasa de fallos)

**Ejemplo tiempo de acceso !**

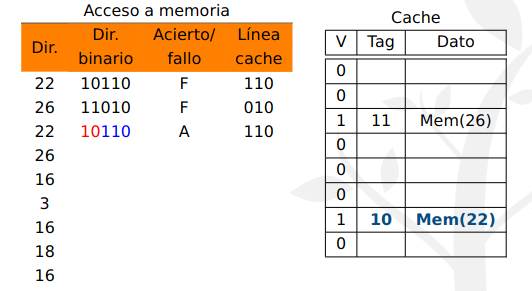
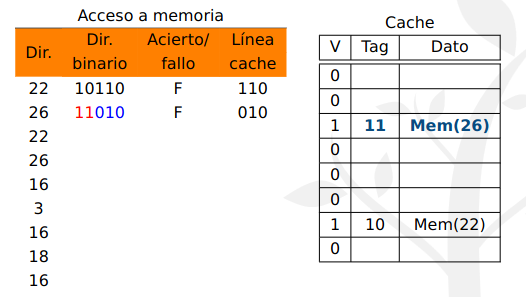
* Un programa lee 500GiB datos de la memoria principal, que lee a 5 GiB/s. Cada dato se utiliza 10 veces. Calcular tiempo sin cache y con cache de 45 GiB/s.
* Sin cache, se accede a 500x10 GiB de datos → 1000 s
* Con cache, se accede 1 vez a 500 GiB con memoria (100 s) y 9 veces con cache (4500GiB a 45 GiB/s) → 100 + 100 = 200 s.

**Memoria cache**

* Memoria pequeña y rápida entre CPU y memoria principal. Almacena la información en uso en un momento dado: datos utilizados en ALU e instrucciones, de uso inmediato.
* **Línea:** cada una de las posiciones de la caché. Puede contener 1 o más palabras (principio de inclusión). Unidad indivisible.
  + Líneas más grandes permiten aprovechar mejor la localidad espacial, pero reducen el nº de líneas que se pueden almacenar.
* **Tipos de cache:**
  + **Cache de asignación directa:** Cada bloque de memoria se coloca en una línea concreta de la cache
  + **Cache totalmente asociativa:** Cada bloque de memoria se puede colocar en cualquier línea
  + **Cache asociativa por conjuntos:** Las líneas se agrupan en conjuntos. Cada bloque de memoria se corresponde con un conjunto, pero dentro del conjunto el bloque se coloca en cualquier línea.

**Cache de asignación directa !**

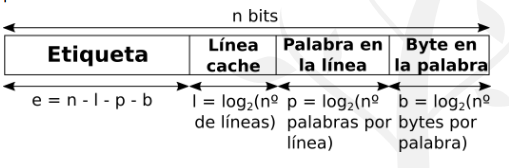
* Siendo **l** el nº de líneas en cache.
* Un dato de la dirección de memoria **d** se aloja en la línea cache número **d%l**.
  + Dada una dirección de memoria, los **log2l** bits menos significativos indican la línea cache. Los bits restantes se denominan etiqueta.
* Se añade a cada línea un bit de validez, que indica si la línea de la cache contiene información válida.
* Cada línea almacena también una **etiqueta**, que indica la dirección de memoria a la que corresponde.
  + El conjunto de todas las etiquetas se denomina **directorio cache**
* Ejemplo:



(si se indica una dirección que no está en caché) (si se indica una dirección que si está)

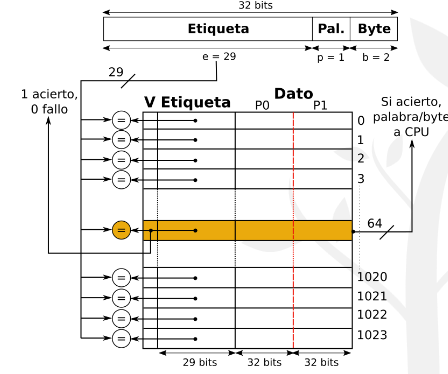
(se baja a memoria y se guarda el dato en la líne (se accede a la línea correspondiente, se

correspondiente, guardando además la etiqueta) comprueba si la etiqueta es la correcta y como lo es, se marca como acierto y devuelve el dato)

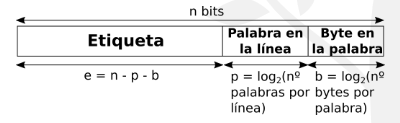
* Si se accede a una línea con una palabra, pero la etiqueta no coincide, se marca como fallo y se carga el dato correcto de memoria.
  + Esto lleva a que se reemplace el dato que había antes.
* Normalmente, las líneas cache contienen **múltiples palabras**. Los p=(log2 palabras/línea) bits menos significativos identifican a la palabra dentro de la línea.
* Además, si la ISA permite acceder a bytes individuales, los log2(bytes/palabra) bits menos significativos indican al byte en la palabra.
* Por ejemplo, en MIPS (cache de 32 líneas, líneas de 8 palabras), b=2, p=3, l=5, e=22
  + Dirección de memoria 0x0700010A (...000100001010): Se accede a la línea 8, a la palabra 4 y a su byte 2.

**Tipos de errores**

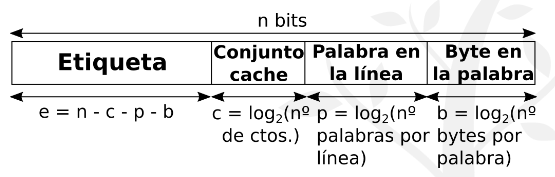
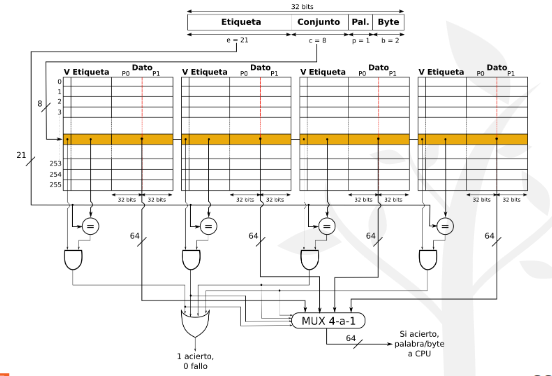
* **Obligatorios:** Se producen la primera vez que se referencia un dato
  + Se reducen aumentando tamaño de línea
  + Representan sólo el 0.006% de los fallos
* **De capacidad:** Cuando se referencia un dato que fue cargado pero después reemplazado porque la cache estaba llena
  + Se reducen aumentando tamaño de cache
* **De conflicto:** Cuando dos líneas compiten por la misma posición en cache
  + Se reducen con caches separadas de datos e instrucciones y aumentando asociatividad

**Cache totalmente asociativa !**

* Un bloque de memoria se puede cargar en cualquier línea de la cache
  + Si está llena, se utiliza un algoritmo de reemplazo para escoger la línea a eliminar
* **tipos de fallos?**
* Para buscar un elemento, se compara su etiqueta con todas las entradas de la cache hasta encontrarlo



**Cache asociativa por conjuntos !!!!!!!!!!**

* La cache se divide en conjuntos de varias líneas
  + **Número de líneas !!!:** nº de líneas por conjunto
* Cada línea tiene asignado un conjunto, pero dentro del conjunto puede ir a cualquier sitio
* Más eficiente que directa y más sencilla/menor coste que totalmente asociativa. Ademais, non requiere leer todas as entradas, sólo as do conxunto correspondente
* Na imaxe, cada fila é un conxunto (en laranxa o conxunto que se está consultando actualmente, leense todas as entradas dese conxunto)
* **direccionamiento por byte?**

**Algoritmos de reemplazo**

* Emprégase cando a caché ou o conxunto a ser empregado está cheo
* Tipos:
  + Aleatorio: lol
  + FIFO: Elimínase o que entrase primeiro
    - Fácil de implementar, pouco eficiente
  + LRU: Elimínase o que leve máis tempo sin ser empregado
    - Máis utilizado, máis eficaz pero máis complexo. Eficacia depende moito do programa/arquitectura
  + LFU: Elimínase o menos empregado
    - Máis eficiente que FIFO

**Reducir penalización por fallo**

* **Cache multinivel:** Consiste en incluir niveles adicionales progresivamente mayores de cache (normalmente, 2 o 3)
  + Los niveles más cercanos a la CPU son más rápidos y pequeños, y suelen estar separadas en datos e instrucciones (otros niveles no)
* También es posible aumentar el ancho de banda (cantidad de datos que la RAM puede pasar en cada ciclo)

**Ancho de banda de memoria**

* Capacidad máxima de transferencia de datos que la memoria admite para realizar los procesos. Medida en múltiplos de B/s o en bytes/ciclo de bus
* Los ciclos de bus/línea es el número total de ciclos de bus que transcurren en caso de fallo.

1. Por eso los bits en los casos previos suman a 63 y no 64 [↑](#footnote-ref-0)
2. 255 si precisión simple, 2047 si precisión doble (basicamente todo 1s no exponente) [↑](#footnote-ref-1)
3. (32 bits=4 bytes) Esto implica que todas las direcciones de memoria deben ser múltiplos de 4, y que la dirección de una palabra es la de su primer byte. [↑](#footnote-ref-2)
4. Otras pseudoinstrucciones: li (carga de 32 bits), blt/ble/bgt/bge (variaciones de bne y slti), move [↑](#footnote-ref-3)
5. KiB → 1024 bytes [↑](#footnote-ref-4)