**Tema9: Diseño del juego de instrucciones**

**Introducción**

Una instrucción de Lenguaje Máquina es una tira de bits que especifica:

* Código de operación:
  + Operación a realizar.
* Modos de direccionamiento:
  + Dónde localizar los operandos.
  + Dónde dejar el resultado.
* Secuenciamiento:
  + Cuál es la siguiente instrucción a ejecutar.
  + Generalmente es implícito.
  + Instrucciones de Salto, llamada/retorno subrutinas.

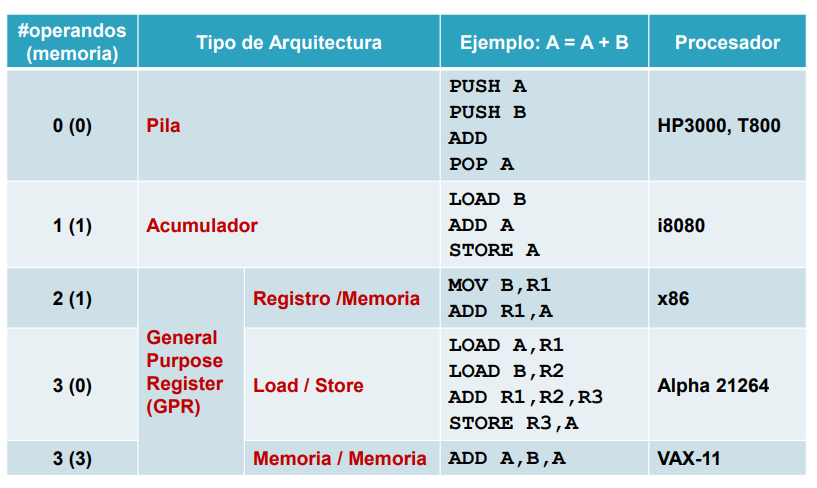
Criterios de diseño de las instrucciones:

* Instrucciones cortas mejor.
* Longitud suficiente.
* Longitud múltiplo de la unidad mínima de direccionamiento (byte).

Formato de las instrucciones:

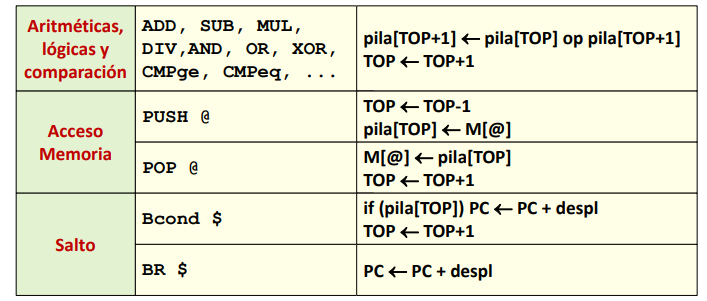
* Fijo:
  + Decodificación rápida y fácil.
  + Desperdicio de memoria.
* Variable:
  + Aprovechamiento de memoria.
  + Decodificación compleja.

**Tipos de Arquitecturas (en función de los operandos explícitos):**



Máquina de PILA:

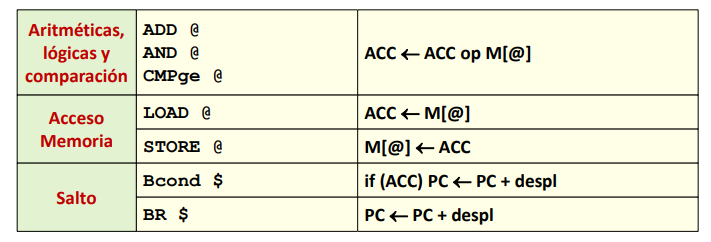
* Sin operandos explícitos a excepción de las instrucciones de salto y acceso a memoria.



* Procesadores antiguos en los que la memoria era un recurso escaso. Máquina Virtual Java.
* La pila es un recurso hardware (equivalente al banco registros coma flotante x86).
* ↑ Instrucciones muy cortas, buena densidad de código.
* ↓ La pila no se puede acceder aleatoriamente, es difícil generar código eficiente.
* ↓ La pila puede ser un cuello de botella.

Máquina de ACUMULADOR:

* Dispone de un acumulador como operando implícito de todas las operaciones.



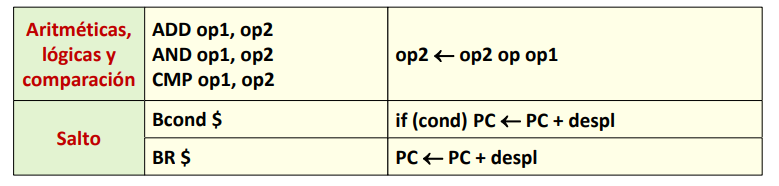
* Acumulador es un término arcaico para referirse a registro.
* ↑ Estado interno de la máquina mínimo (1 registro).
* ↑ Instrucciones cortas.
* ↓ Tráfico con memoria muy elevado.

Arquitecturas GPR:

* Arquitecturas con Registros de Propósito General (GPR, General Purpose Register).
* Características que dividen a las arquitecturas GPR:
  + 2 o 3 operandos:
    - OP2 ← OP1 (op) OP2.
    - OP3 ← OP1 (op) OP2.
* ¿Cuántos de estos operandos pueden estar en memoria?
  + Desde ninguno en las máquinas Load/Store.
  + Hasta todos en las máquinas Memoria/Memoria.
* ↑ Modelo más general para la generación de código, código eficiente.
* ↑ Acceso rápido a los registros.
* ↑ Utilización eficiente de los registros por el compilador.
* ↑ Reducción del tráfico con memoria.
* ↓ Todos los operandos son explícitos, algunas instrucciones pueden ser muy grandes.
* Las arquitecturas GPR son las más adecuadas y extendidas. Interesa que los registros sean equivalentes (ortogonalidad) y numerosos.

Máquina de Registro / Memoria:

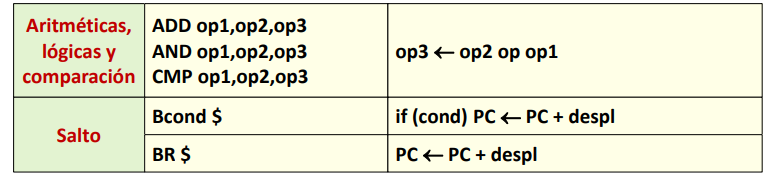
* Procesador con dos operandos explícitos, uno de ellos puede estar en memoria.



* Uno de los operandos es simultáneamente fuente y destino.
* Aparecen instrucciones de movimiento de datos.
* ↑ Los datos son accesibles sin lectura previa de memoria.
* ↓ Los operandos no son equivalentes (un operando fuente se modifica).
* ↓ Codificar una dirección y un registro puede limitar el número de registros.
* ↓ Los ciclos por instrucción varían dependiendo del tipo de acceso (de 0 a 2 accesos a memoria en 1 única instrucción).

Máquina de Memoria/Memoria:

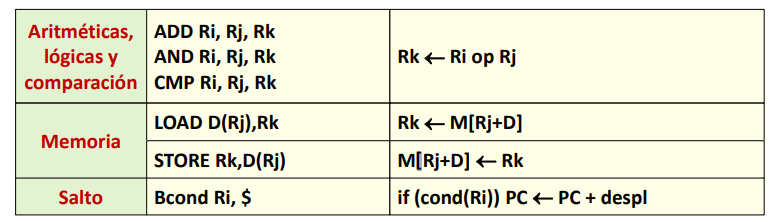
* Procesador con 3 operandos explícitos, cualquiera de ellos puede estar en memoria.



* ↑ Código muy compacto.
* ↑ No hace falta utilizar registros para variables temporales.
* ↓ Diferentes tamaños de instrucción ⇒ Dificulta la búsqueda y decodificación de instrucciones.
* ↓ Diferentes cargas de trabajo por instrucción.
* ↓ La memoria se convierte en el cuello de botella.

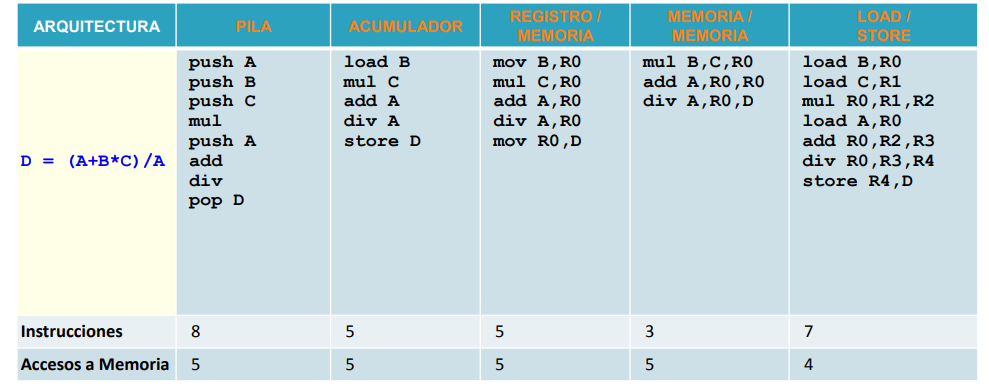
Máquina Load/Store:

* Procesador con 3 operandos explícitos en registros.

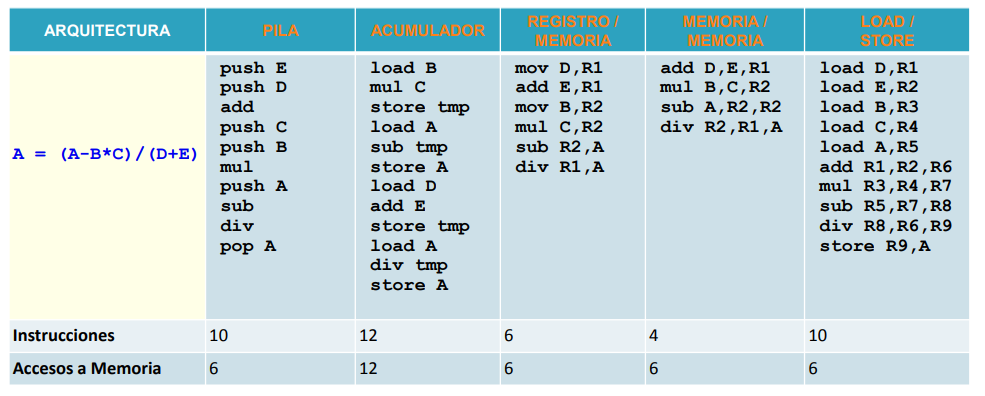


* Todas las operaciones aritméticas se realizan sobre registros.
* Necesita instrucciones específicas para acceder a memoria (load / store).
* ↑ Instrucciones codificadas de forma fija ⇒ Facilita la búsqueda y decodificación de instrucciones.
* ↑ Generación de código sencilla (el compilador tiene pocas alternativas).
* ↑ Todas las instrucciones tardan tiempos parecidos.
* ↓ Hacen falta más instrucciones, p.e. las utilizadas para acceder a memoria.
* ↓ Formato fijo ⇒ Desperdicio de memoria.

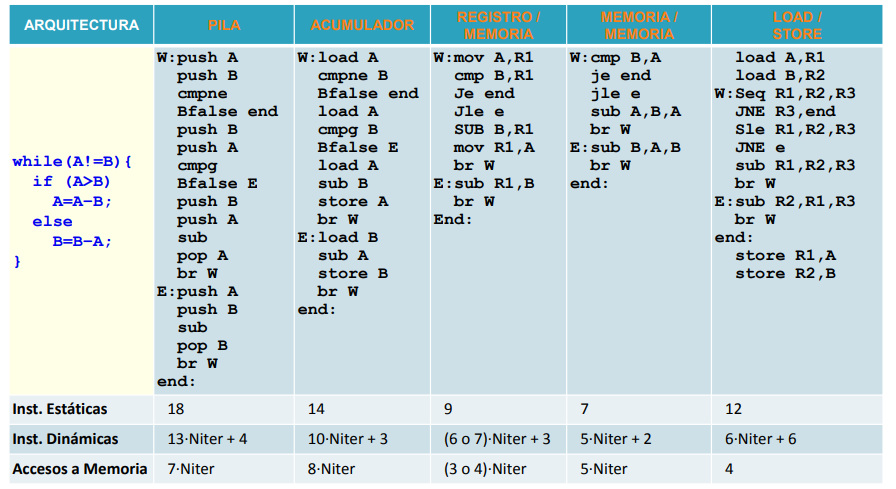
*Ejemplo 1: evaluar una expresión aritmética*



*Ejemplo 2: evaluar una expresión aritmética*



*Ejemplo 3: un código simple en C*



**Modos de direccionamiento**

¿Qué es el Modo de Direccionamiento?

Es un algoritmo utilizado para acceder a los operandos de una instrucción. Para codificar el modo necesitamos 2 campos (implícitos o explícitos):

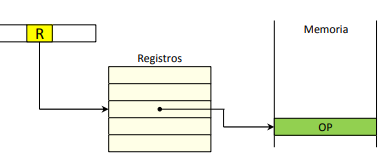
* Modo de direccionamiento: indica el algoritmo a utilizar para calcular la dirección efectiva.
* Una serie de valores, que pueden ser:
  + Dirección.
  + Registro.
  + Desplazamiento.
  + Inmediato.

Criterios de diseño:

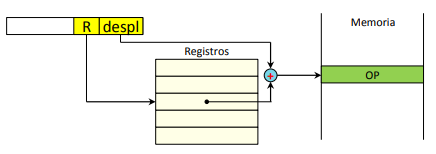
* Acceso a todo el espacio de direcciones.
* Acceso eficiente a estructuras de datos.
* Facilitar la comunicación con subrutinas.

Tipos de Modos de Direccionamiento:

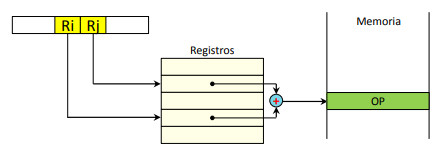
* **Modo registro**:
  + El operando se encuentra en uno de los registros del procesador.
  + ↑ Acceso rápido.
  + ↑ Pocos bits.
  + ↓ Se necesitan instrucciones para mover datos con memoria.
  + Ejemplo: movl %edx, %eax.
* **Modo inmediato**:
  + El operando se encuentra en la instrucción.
  + Muy útil para trabajar con constantes.
  + Ejemplo: movl $34, %eax.
* **Modo Absoluto (Directo)**:
  + La dirección del operando se encuentra en la instrucción.
  + ↓ Se necesitan muchos bits para codificar la dirección.
  + Ejemplo: movl 1242451, %eax.
* **Modo registro indirecto**:
  + La instrucción codifica el registro que contiene la dirección del operando.
  + ↑ Pocos bits.
  + Ejemplo: movl (%ebx), %eax.
  + ↑ Posibilidad de autoincremento / autodecremento para facilitar accesos a arrays, pila, etc. (p.e. VAX11):
    - Autoincremento: MOVL R1, (R2)+
    - Autodecremento: MOVL R1, -(R2)



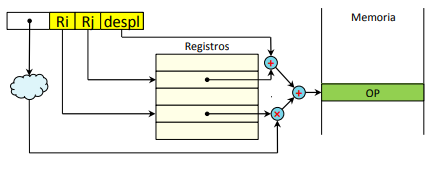
* **Modo Base + desplazamiento**:
  + La dirección se calcula utilizando un registro y un desplazamiento (±).
  + Si el desplazamiento es cero, equivale al modo registro indirecto.
  + Ejemplo: movl -24(%ebp), %eax.
  + ↓ Cálculo de la dirección.



* **Modo indexado**:
  + La dirección efectiva se obtiene sumando el contenido de dos registros.
  + ↑ Pocos bits.
  + ↑ Útil en el acceso a vectores.
  + ↓ Cálculo de la dirección.
  + Ejemplo: movl (%ebx, %esi), %eax.

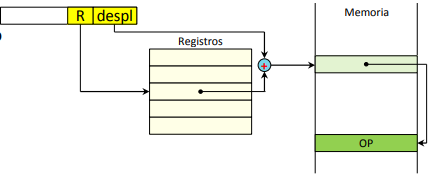


* **Modo escalado**:
  + Especialmente útil para acceder a vectores.
  + El escalado depende de la instrucción.
  + ↓ Cálculo de la dirección.
  + Ejemplo: movl -44(%ebx, %esi, 4), %eax.



Modos de Direccionamiento “combinados”:

* **Post – indirección**:
  + El dato obtenido con los modos anteriores es la dirección del operando.
  + Ejemplo: Base + desplazamiento post-indirecto.
  + Ejemplo: MOVL @32(SP), R3.



* **Post – escalado**:
  + El VAX11, permitía combinar el modo escalado (indexado en terminología de DIGITAL) a cualquier otro.
  + Ejemplo: MOVL @32(SP)[R7], R3.

**Estudio Cuantitativo del nivel LM**

Con pocos bits (8) se codifican un 95% de los valores inmediatos. Muchos de ellos son 0 o 1 (45%).

Con un desplazamiento de 16 bits se cubre el 85% de los casos.

Con el modo base + desplazamiento (el modo registro indirecto se puede implementar con desplazamiento cero) se cubre un porcentaje muy elevado de casos (85%, 70% y 97%, respectivamente).

Tipos de datos soportados a nivel L.M:

* Enteros: complemento a 2, diferentes tamaños.
* Caracteres: usualmente en ASCII (8 bits), aunque cada vez es más habitual es uso de 16bit Unicode (Internacionalización).
* Coma flotante: IEEE 754 (32 y 64 bits).
* Decimal: packed decimal (4 bits por dígito) o unpacked decimal (8 bits por dígito). Imprescindible para contabilidad. Números que se codifican de manera exacta en decimal no son exactos en binario. Calculadora bc (práctica 9).

Muchos de los operandos son direcciones.

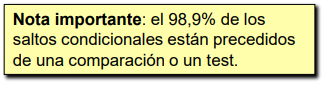
*Las instrucciones más utilizadas son las más sencillas.*

Instrucciones de control de flujo:

* Call / Return: 13%.
* Salto incondicional: 14%.
* Salto condicional: 73%.

Posibles instrucciones de salto condicional:

* En función de bits de condición (x86):
  + ↓ Esta solución restringe el orden de ejecución de las instrucciones.
* Registros de condición. Se realiza test sobre un registro cualquiera que tiene el resultado de una comparación previa (Itanium tiene 128 registros de 1 bit):
  + ↑ Es muy simple.
  + ↑ No restringe el orden de ejecución.
  + ↓ Usa un registro adicional.
* Instrucciones que incluyan la comparación y el salto:
  + ↑ Una sola instrucción.
  + ↓ Puede ser mucho trabajo para una instrucción.



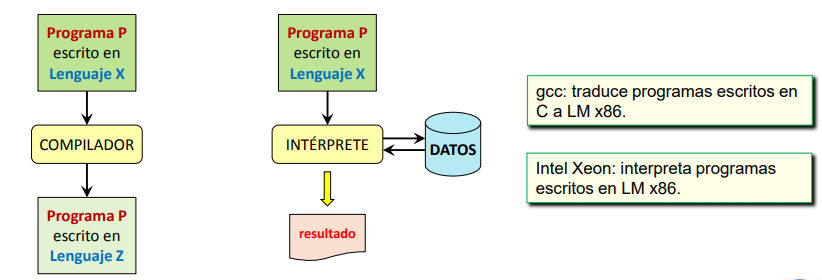
Observaciones:

* Las instrucciones sencillas son las más ejecutadas.
* Las instrucciones complejas son difíciles de usar por el compilador.
* Las instrucciones complejas se pueden sustituir por secuencias de instrucciones sencillas.
* Los modos de direccionamiento más utilizados son los más sencillos.
* Los modos de direccionamiento complejos pueden ser emulados por secuencias de instrucciones que además pueden ser optimizadas.
* Con constantes y desplazamientos de pocos bits se cubren la mayoría de los casos.
* El objetivo final es hacer que los programas se ejecuten lo más rápido posible.

Compilación vs Interpretación:

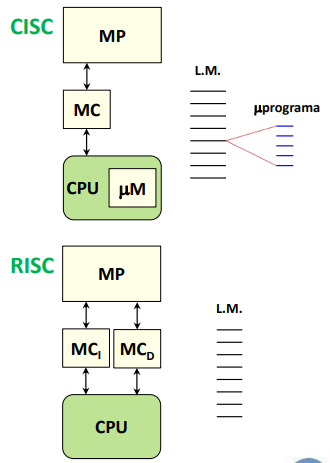
Niveles de un computador:

* Lenguajes de alto nivel (LAN).
* Lenguaje máquina (LM).
* Implementación hardware (HARD) LAN ⇒ compilación ⇒ LM ⇒ interpretación ⇒ HARD.



**RISC vs CISC**

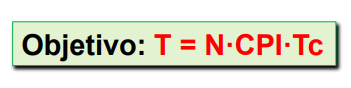
¿Cómo salvar el desnivel entre los Lenguajes de Alto Nivel y el Hardware?

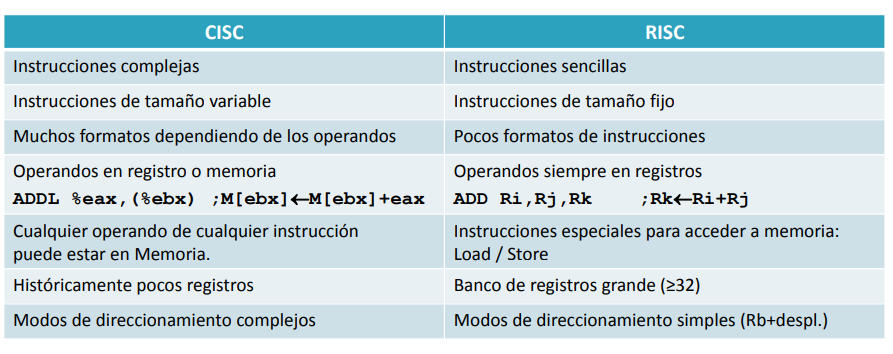
CISC: Complex Instruction Set Computer:

* Instrucciones LM de alto contenido semántico.
* Esfuerzo en interpretación (μcódigo).

RISC: Reduced (complexity) Instruction Set Computer:

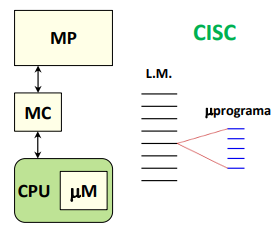
* Instrucciones LM de bajo contenido semántico.
* Esfuerzo en compilación.



****

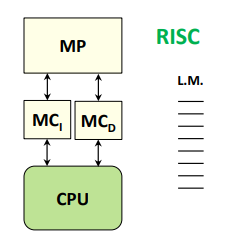
CISC (Complex Instruction Set Computer):

* Usaban instrucciones complejas para acercar el LM a los LAN.
* La memoria era un recurso escaso y caro. La longitud de los programas era uno de los parámetros para evaluar una arquitectura.
* Después de introducir la μprogramación era efectivo mover funciones realizadas por una serie de instrucciones de L.M. hacia el μprograma: aumentaba la densidad de código, reducía el tráfico con memoria y el tamaño del programa ⇒ incrementaba la eficiencia de la máquina.
* Se suponía que mover instrucciones complejas hacia el LM simplificaría la tarea del compilador al eliminar el desnivel semántico entre el LM y los Lenguajes de Alto Nivel (p.e.: INDEX, CASE, CALLS en VAX 11/70).
* Disponían de muchos y complejos modos de direccionamiento.
* Necesitaban instrucciones de longitud variable ⇒ dificulta mucho el fetch y la decodificación.
* La μmemoria alcanzó tamaños desorbitados (480 KB en el VAX 11/70) y se convierte en el cuello de botella del sistema. Además, no está libre de errores.



RISC (Reduced Instruction Set Computer):

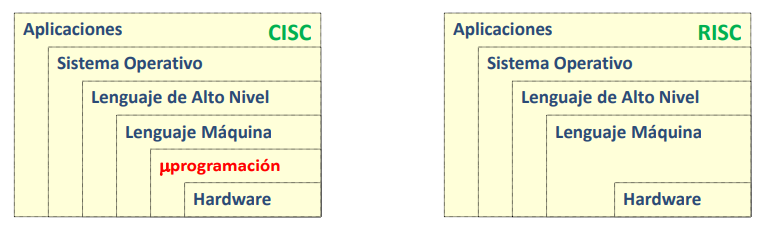
* Los estudios prácticos descubrieron que los procesadores CISC estaban sobredimensionados. Los compiladores no utilizaban eficientemente los modos de direccionamiento e instrucciones de los procesadores CISC.
* Los procesadores RISC utilizan instrucciones muy simples, que al ser generadas por el compilador pueden ser optimizadas.
* Los compiladores generan código más fácilmente y más eficiente para este tipo de máquinas.
* A partir de los años 80, los procesadores empezaron a ser más rápidos que las memorias.
* Los procesadores RISC disponen de muchos registros de propósito general ⇒ la mayoría de los operandos pueden estar en registros, incluyendo los parámetros y variables locales de las subrutinas.
* Sólo se accede a memoria con load y stores.
* Estas máquinas están pensadas para su ejecución segmentada ⇒ se busca ejecutar 1 instrucción por ciclo.



**μprogramación**

Técnica utilizada para simplificar el diseño de la unidad de control de los procesadores CISC.

Visión vertical en niveles de un computador.

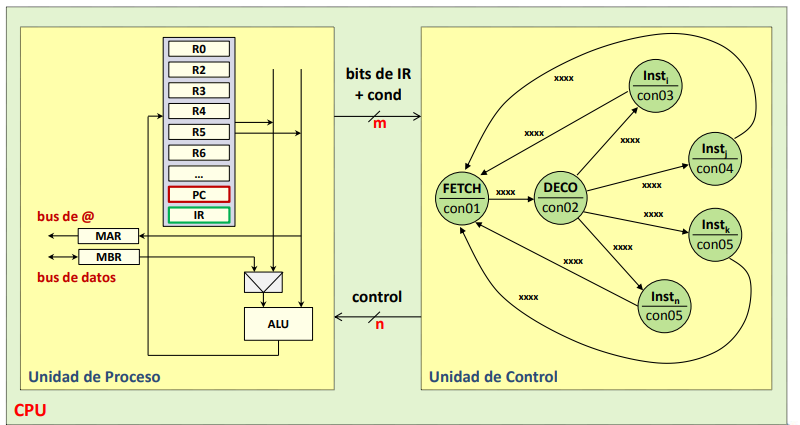


Elementos básicos de un procesador:

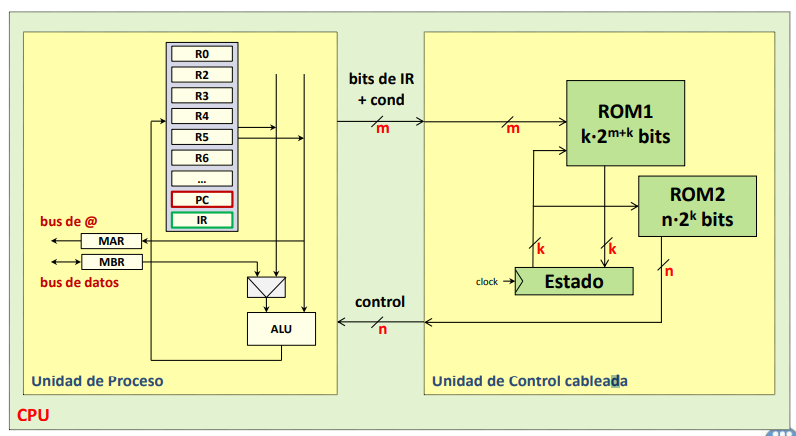
* Unidad de proceso: ALUs, registros, elementos combinacionales, …
* Unidad de control: sistema secuencial.

Unidad de Control cableada vs μprogramada.

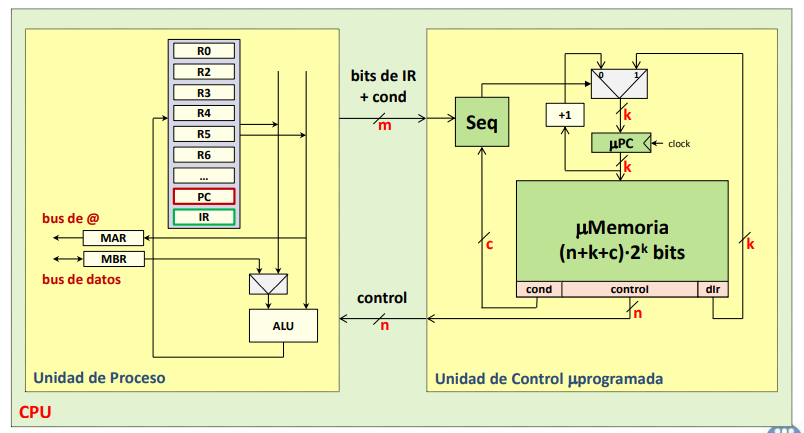
CPU: elementos básicos:

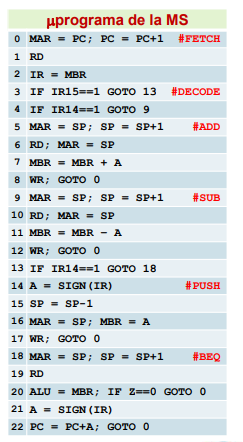


CPU cableada:



CPU μprogramada:



μprograma:

* El contenido de la μmemoria es el μprograma.
* El μprograma está compuesto por μinstrucciones, que equivalen a los estados del sistema secuencial que define la unidad de control.
* En las μ instrucciones se codifican 3 campos:
  + control: bits que gobiernan directamente los circuitos de la UP.
  + dir: dirección de la siguiente μinstrucción en caso de que sea necesario romper el secuenciamento implícito del μprograma.
  + cond: forma de evaluar los flags y bits de IR para decidir si el salto a dir es efectivo.
* En el μprograma se identifican las fases en que se puede dividir la ejecución de una instrucción de Lenguaje Máquina:
  + Fetch.
  + Decodificación.
  + Ejecución detallada.
* El μprograma es el intérprete del Lenguaje Máquina.

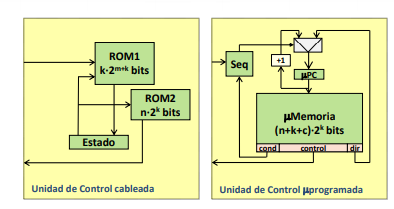
Unidad de Control Cableada vs μprogramada:

Unidad de control cableada:

* X estados, k =log2 X.
* Entradas: m bits (IR + cond).
* Salidas: n bits (control de la UP).
* ROM1: k·2m+k bits para calcular el siguiente estado.
* ROM2: n·2k bits para obtener las señales de control asociadas a cada estado.

Unidad de control μprogramada:

* X μinstrucciones (estados), k =log2 X.
* Entradas: m bits (IR + cond).
* Salidas: n bits (control de la UP).
* Cond: c bits para seleccionar la siguiente μinstrucción.
* μmemoria: (n+k+c) · 2k bits para obtener las señales de control asociadas a cada estado.
* Seq: 2c+m bits para seleccionar la siguiente μinstrucción.

Ejemplo:

* 1024 estados, k = 10.
* Entradas: 14 bits (m = 14).
* Salidas: 24 bits (n = 24).
* Cond: 8 condiciones (c = 3).

Coste UC cableada:

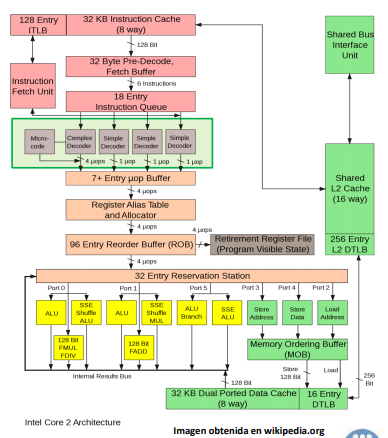
* ROM1: 10 · 224 = 167.772.160 bits.
* ROM2: 24 · 2 10 = 24.576 bits.

Coste UC μprogramada:

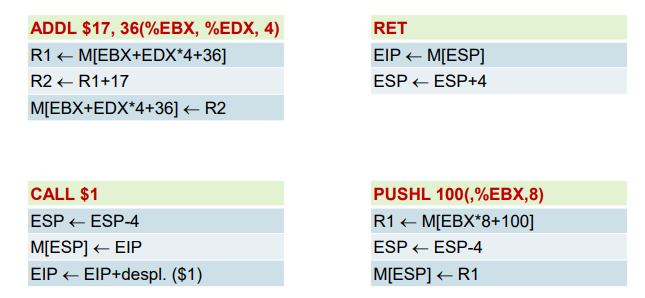
* μmemoria: 37 · 2 10 = 37.888 bits.
* Seq: 217 = 131.072 bits.

**Procesadores x86**

* Las instrucciones x86 son complejas y difíciles de implementar.
* Los primeros procesadores que ejecutaban x86 eran μprogramados.
* En las implementaciones actuales, se utiliza un traductor hardware que traduce de x86 a μoperaciones.
* Las μoperaciones tienen una complejidad similar a instrucciones RISC:
  + R1 ← R4 + EDX.
  + EAX ← R5 – 8.
  + R3 ← M[EAX+EBX\*2-48].
  + M[R3] ← R9.
* La CPU ejecuta las μoperaciones como si fuera un procesador RISC.



Ejemplos de traducción de x86 a μops:



*COMPILACIÓN vs INTERPRETACIÓN*

¿Cómo se ejecuta un programa Java en un procesador x86?

