Tema 1: Introducción a los computadores

Controlador: Sistema secuencial que actúa sobre otros sistemas, activando sus terminales de control

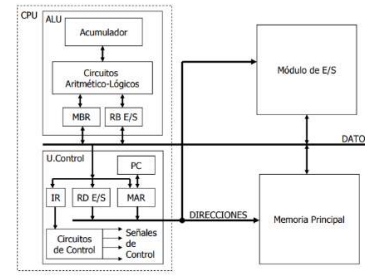
Bit de estado (Flag): Informan al controlador para casos condicionales

Computador: Máquina que procesa información de forma automática

Arquitectura de Von Neumann

En 1945 Von Neumann propuso un modelo de computación que ha constituido, incluso en la actualidad, la base de la arquitectura de los computadores digitales. Está compuesto por 4 unidades:

- Unidad de memoria principal
- Unidad de entrada/salida
- Unidad aritmético-lógica (ALU)
- Unidad de control



Memoria principal:

Compuesta por un conjunto de celdas de almacenamiento agrupadas en bloques del mismo tamaño (*palabra*). El acceso a cada palabra tiene asociado un número único, llamado dirección.

Memoria caracterizada por 2 parámetros: Tamaño (número de direcciones) y ancho de palabra (número de bits por palabra). Los datos e instrucciones se almacenan en la misma unidad de memoria.

Unidad aritmético-lógica:

Posee una serie de microoperaciones que son capaces de tomar datos y aplicarles operaciones elementales. Incorpora registros que almacenan los operandos y guardan el resultado

Unidad de control:

Encargada de manejar todas las demás unidades de la computadora. Toma instrucciones de la memoria principal, las decodifica y prepara las señales para las diferentes unidades funcionales.

Para poder llevar la cuenta de la instrucción del programa que se tendrá que ejecutar, necesita un registro apuntado, denominado *Contador de Programa* (PC)

Unidad de entrada/salida:

La unidad de E/S realiza las transferencias de información con los periféricos. Es capaz de realizar carga de información desde memoria secundaria a memoria principal. Encargada de la visualización por pantalla, impresión, acceso a red…

CPU:

Conjunto formado por la ALU y la unidad de control (también suele incluir la unidad de E/S)

Registros de control

- MBR (*Memory Buffer Register*): Contiene el dato que se va a escribir o leer de memoria
- MAR (*Memory Adress Register*): Especifica la dirección de memoria de la palabra que se va a escribir o leer
- RBE/S (*Registro Buffer para E/S*): Semejante al MBR. Se utiliza para intercambio de datos entre E/S y la CPU
- RDE/S (*Registro Dirección para E/S*): Similar a MAR. Especifica un dispositivo de E/S
- IR (*Instruction Register*): Contiene el código de operación de la instrucción
- PC (*Program Counter*): Contiene la dirección de memoria que contiene la siguiente instrucción
- Ac (*Accumulator*): Almacena temporalmente los operandos y los resultados de las operaciones de la ALU

Formato de instrucción: Se divide en cos campos: Código de operación y de operando

Historia de los computadores

1ª Generación:

- Tubos de vacío: Alta disipación, mucho espacio y consumo energético
- 2 ramas
    - Gran Bretaña: Alan Turing
    - EEUU: Atanasoff y Berry
- ENIAC: Primer computador digital electrónico de propósito general
    - Resolvía problemas balísticos
    - Máquina decimal con programación manual
- IBM: Primera gran empresa de la industria comercial de computadores

2ª Generación:

- Utilización de transistores
- Memoria: Núcleos de ferrita
- Almacenamiento masivo: Disco rígido magnético
- Mejoras en la arquitectura

3ª Generación:

- Circuitos integrados (cientos o miles de transistores)
    - Disminución de tamaños, coste y disipación de potencia
    - Aumento de la frecuencia de funcionamiento
- Sistemas operativos (IBM OS/360, Bell Lab UNIX…)

4ª Generación:

- Aparición de memorias semiconductoras y del microprocesador
- Ordenadores más baratos, pequeños y de menor consumo
- Primer microprocesador (Intel 4004)
    - Palabras de 4 bits
    - Sumaba números de 4 bits
    - Multiplicaba mediante sumas repetitivas
- Aparición de ordenadores personales
    - 1977: Apple II (Steve Jobs y Steve Wozniak)
    - 1981: IBM-PC (procesador Intel 8088/8086)

5ª Generación:

- Circuitos integrados con tecnología VLSI
- Se comienza a usar la arquitectura Harvard (RISC)
- Inteligencia artificial
- Generalización y expansión del uso de redes de comunicación

Tema 4: Unidad de control

Introducción

La unidad de control es un bloque de la arquitectura de Von Neumann. Sus funciones incluyen:

- Asegurar la ejecución de los programas
- Generación de las secuencias de microórdenes para la ejecución de todas y cada una de las instrucciones de la computadora
- Captar y decodificar cada instrucción
- Ejecución de la siguiente instrucción del programa

Técnicas de diseño de la unidad de control:

- Cableada
    - Registros de desplazamiento
    - Sistema secuencial síncrono
    - Decodificadores de tiempo e instrucción
- Microprogramada
    - Mediante ROM de control
        - Microprogramación horizontal
        - Microprogramación vertical

Microprogramación

Instrucciones compuestas por varios pasos (ciclos), tanto de búsqueda como de ejecución. Necesidad de saber cuál será el siguiente paso para ejecutar (Control de la dirección de bifurcación). Presentan distintos formatos:

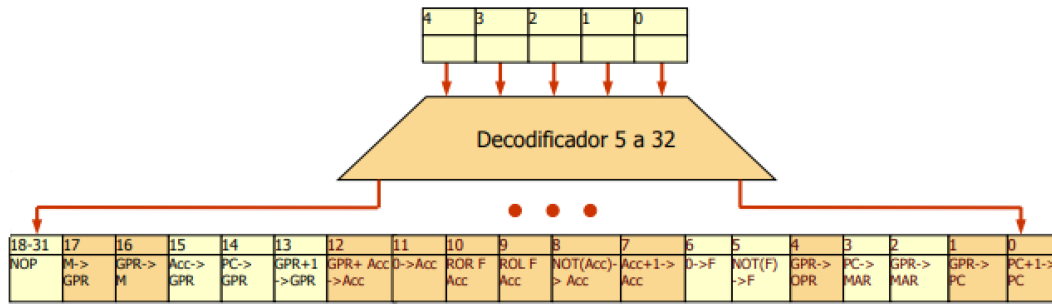
Formato no codificado:

Las micropalabras tienen un bit para cada señal de control, lo que produce una ROM con palabras muy grandes.

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M-> GPR	GPR-> M	Acc-> GPR	PC-> GPR	GPR+1 ->GPR	GPR+Acc ->Acc	0->Acc	ROR F Acc	ROL F Acc	NOT(Acc) ->Acc	Acc+1-> Acc	0->F	NOT(F) ->F	GPR-> OPR	PC-> MAR	GPR-> MAR	GPR-> PC	PC+1-> PC

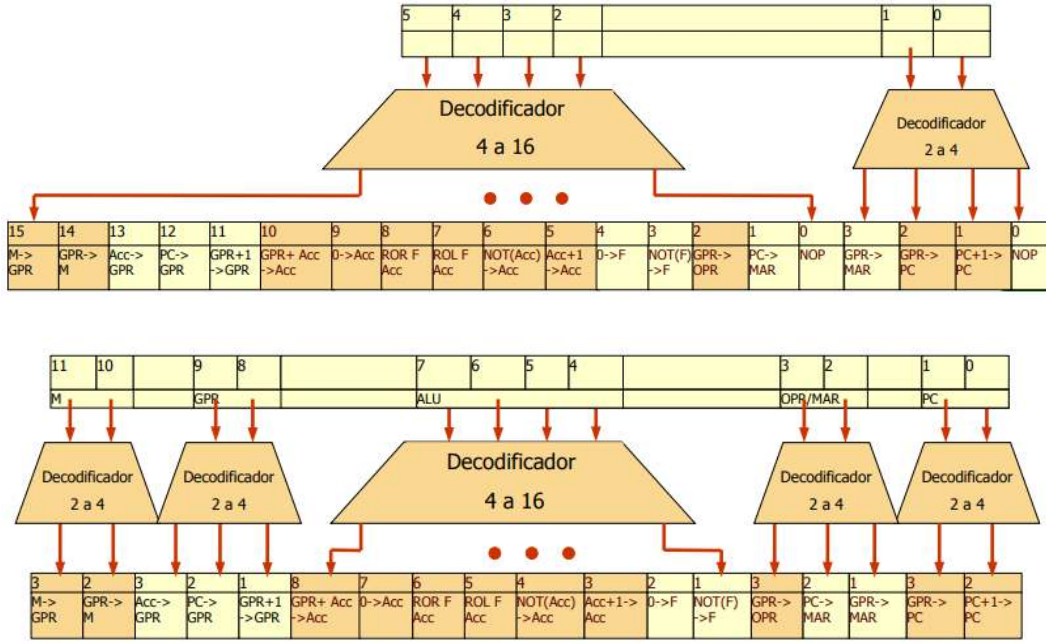
Formato completamente codificado:

Se codifica la activación de todas sus señales de control con menos bits (en cada codificación solo se activa 1 única señal de control)



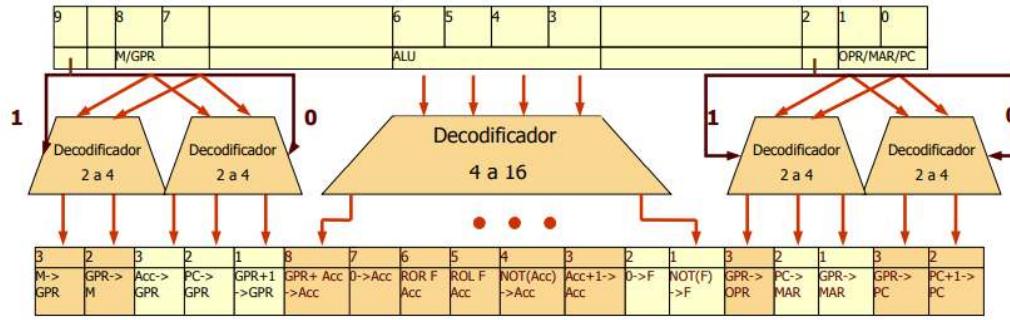
Formato codificado por trozos:

Activación de señales de control con menos bits agrupadas en campos (permite la activación de varias señales a la vez). La microinstrucción es más grande, pero los decodificadores son más pequeños.



Formato con solapamientos:

Si hay señales excluyentes entre sí, se pueden solapar campos, reduciendo el tamaño de las microinstrucciones

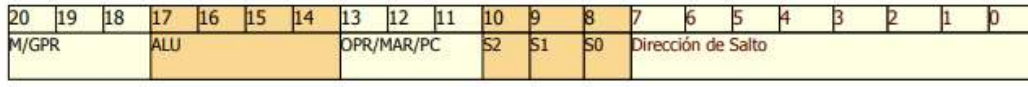


Microprogramación horizontal	**Microprogramación vertical**
Formato no codificado	Formato codificado
Microinstrucciones largas	Microinstrucciones cortas
Alto grado de paralelismo	Bajo paralelismo
Alto consumo de memoria de control	Menor consumo de memoria de control
Más rápida	Más lenta

Secuenciamiento en microprogramación:

Tras la ejecución de cada microinstrucción se tiene que determinar qué nueva microinstrucción se ha de ejecutar. Hay 3 posibilidades:

- Ejecutar la microinstrucción que se encuentra en la dirección de CROM consecutiva (Incremento)
- Ejecutar la microinstrucción que se encuentre en una dirección de la CROM especificada explícitamente (Bifurcación)
- Pasar a ejecutar la primera microinstrucción asociada al ciclo de ejecución de una instrucción determinada (Carga de rutina)

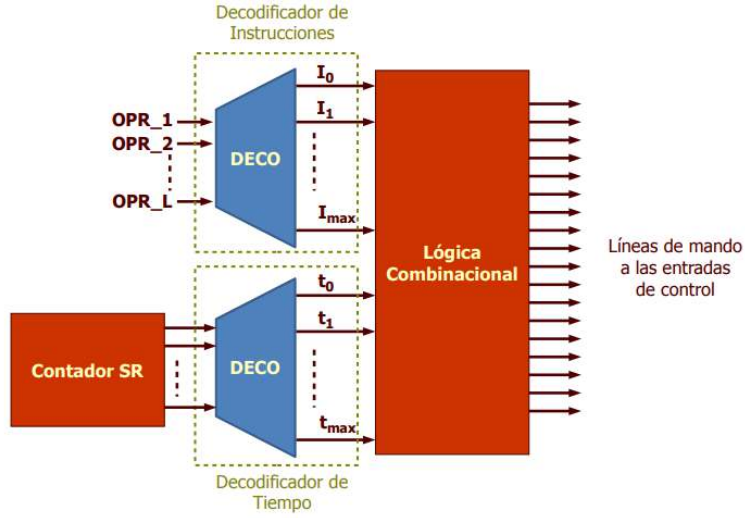


Cableado por Decodificadores Tiempo/Instrucción

Codificación implícita mediante la existencia de cableado específico. Para cada microoperación se activa exclusivamente en un determinado ciclo de tiempod e una instrucción

Ejecución de una microoperación: $M_n = I_i \cdot T_j \cdot C_k$

- M_n representa una microoperación
- I_i representa la instrucción i-ésima del repertorio
- T_j representa el ciclo j-ésimo de búsqueda o ejecución
- C_k representa la condición k-ésima de estado



El contador de tiempo (*SR*) tendrá 3 microoperaciones:

- Cuenta ascendente
- Carga paralela
- Puesta a 0

Cableado mediante registros de desplazamiento

- También llamado “trenes de biestables”
- Implementación sencilla
- Alta flexibilidad
- Facilidad de análisis
- Bastante rápida
- Minimiza la necesidad de lógica adicional
- Alto uso de biestables

Circuito Secuencial Síncrono	CROM	Decodificadores
Tedioso y rígido	Diseño muy fácil	Sencillo de diseñar
Difícil de analizar	Muy flexible	Bastante flexible
Difícil detectar errores	Fácil de analizar	Fácil de analizar
Ejecución más rápida	Mecanismo más lento	Ejecución bastante rápida
Minimiza biestables	Circuitería costosa	Bastante lógica combinacional

Tema 3: Unidad de cálculo

Comparación

Mecanismo para la toma de decisiones en otras operaciones. La comparación se realiza a nivel de magnitud.

- Igualdad
  - Dos bits (i-ésimos) son iguales si $X_i = A_i \cdot B_i + A_i' + B_i' = 1$; i = 0…n-1
  - Dos números son iguales si $\prod_{i=0}^{n-1} Xi = 1$
- Mayor que
  - Un número *a* es mayor que otro *b* si, comenzando desde el dígito más significativo se cumple: $X_j = 1$ (para j = n-1…i+1) y $a_i = 1$ y $b_i = 0$
- Menor que
  - Un número *a* es menor que otro *b* si, comenzando desde el dígito más significativo se cumple: $X_j = 1$ (para j = n-1…i+1) y $a_i = 0$ y $b_i = 1$

Comparación mediante resta

Una forma de saber si dos números son iguales es restándolos o con la suma en complemento a 2 del sustraendo

- Si son iguales, el resultado será 0
- Si el primero es mayor que el segundo, el resultado será positivo
- Si el segundo es mayor que el primero, el resultado será negativo

Representación Signo-Magnitud

Un registro Ac estará compuesto por 2 registros: $A_s|A$, siendo A_s el bit más significativo (bit de signo)

- Si $A_s = 1$, el número es negativo
- Si $A_s = 0$, el número es positivo

Esta representación presenta dos problemas principales: La necesidad de tratar el signo aparte de la magnitud y la aparición de un doble 0 (1 000b // 0 000b)

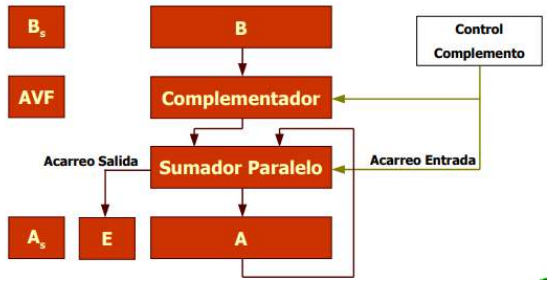
Representación Complemento a 2 con signo

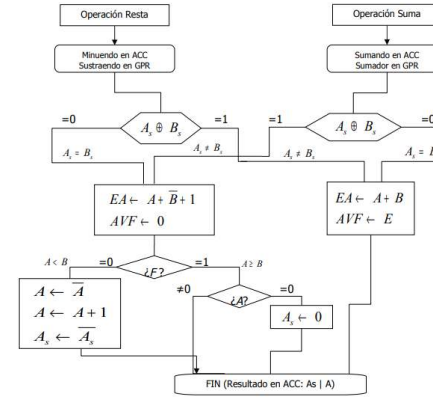
Un registro Ac compuesto por 2 registros igual que en Signo-magnitud, con la diferencia que:

- Si $A_s = 1$, A es la magnitud en complemento a 2
- Si $A_s = 0$, A es la magnitud

Esta forma de representación resuelve el problema del doble 0 y permite tratar A_s de forma integrada.

Suma/Resta en Signo-Magnitud





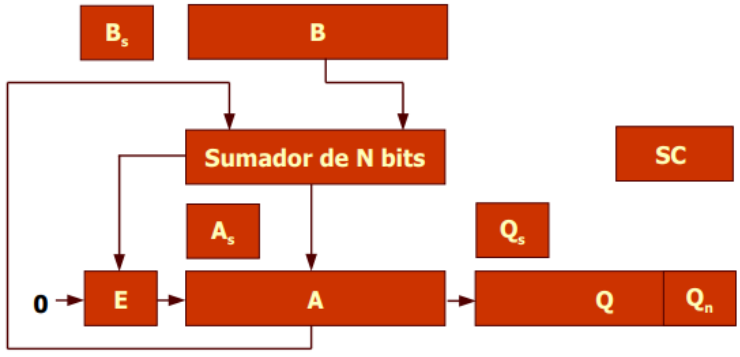
Multiplicación en Signo-Magnitud

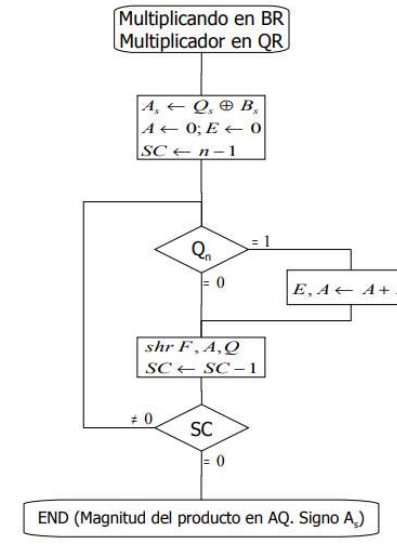
La multiplicación de dos números de N bits (+1 de signo) da como resultado un número de 2·N bits (+1 de signo). El signo será positivo si los dos tienen el mismo signo y la magnitud será el producto de las magnitudes de los operandos.

El algoritmo partirá con el multiplicando en B_sB y el multiplicador en Q_sQ. Se comprobará Q_n:

- Si vale 1 se suman las magnitudes A y B
- Si vale 0 no se hace nada

El bucle terminará desplazando un bit a la derecha el conjunto formado por E, A y Q. Después de N iteraciones, el resultado se encontrará en A_sAQ.





División en Signo-Magnitud

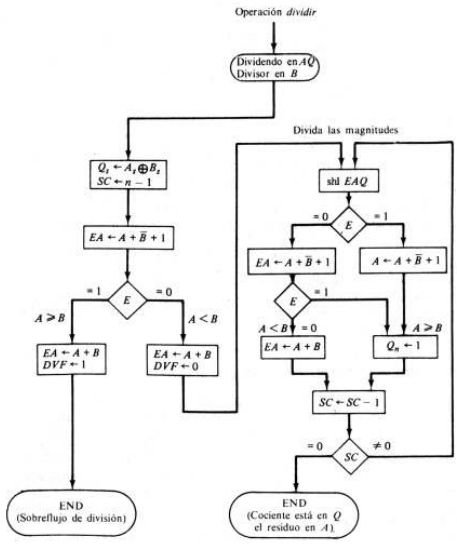
La división de un número de 2N bits por otro de N bits (+1 de signo) da como resultado N bits de cociente (+1 de signo) y N bits de resto (+1 de signo). El signo del cociente será positivo si los dos números tienen el mismo signo y la magnitud será la división de las magnitudes de los operandos.

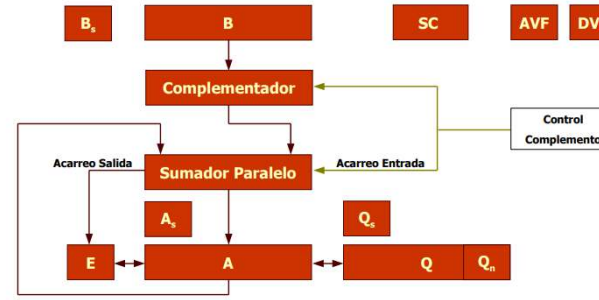
El algoritmo partirá con el divisor en B_sB y el dividendo en A_sAQ. Se empieza desplazando a la izquierda EAQ.

- Si E = 1 sabemos que EA es mayor que B (divisor), así que podemos restar B a A e introducir un 1 en Q_n
- Si E = 0 quiere decir que lo que había en A era menor que lo que había en B. Como no deberíamos haber restado, volvemos a sumar el contenido de B a A

Después de N iteraciones, el cociente estará en Q_sQ y el resto en A_sA

Para comprobar el *overflow* se puede utilizar un biestable (DVF) que sirva de bit de estado del sistema. Si hay *overflow*, DVF = 1.





Suma/Resta en Complemento a 2

Para la suma, realizar la operación utilizando el sumador completo: Ac+BR->Ac

Para la resta, es necesario realizar la operación utilizando el complemento a 2 del sustraendo: Ac+BR'+1->Ac

Existe la posibilidad de desbordamiento, proporcionando un resultado no válido debido a la falta de bits. Una forma de calcularlo es analizando los dos últimos acarreos proporcionados por el sumador:

- Si son iguales, no hay *overflow* (V = 0)
- Si son distintos, hay *overflow* (V = 1)

Desplazamientos en Complemento a 2

Desplazamiento lógico: Consiste en desplazar el contenido de cada bit de la posición actual a la posición inmediatamente anterior o posterior. El bit inicial o final se rellena con 0. No mantiene el signo negativo

Desplazamiento aritmético: Mismo concepto que el desplazamiento lógico, pero manteniendo el signo. Para desplazamientos a la derecha se ingresa en el bit de signo el mismo valor que existía antes del desplazamiento. Para desplazamientos a la izquierda, complementar el bit de signo si varía su valor antes y después del desplazamiento lógico.

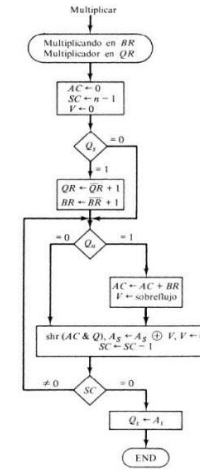
Multiplicación en Complemento a 2

Dependiendo del signo de los operandos se puede aplicar el mismo algoritmo que en signo-magnitud:

- Ambos positivos: Se puede aplicar olvidándose del signo
- Multiplicador positivo y multiplicando negativo: Algoritmo de signo-magnitud realizando desplazamientos aritméticos
- Multiplicador negativo: No se puede realizar el algoritmo

Algoritmo Multiplicación I:

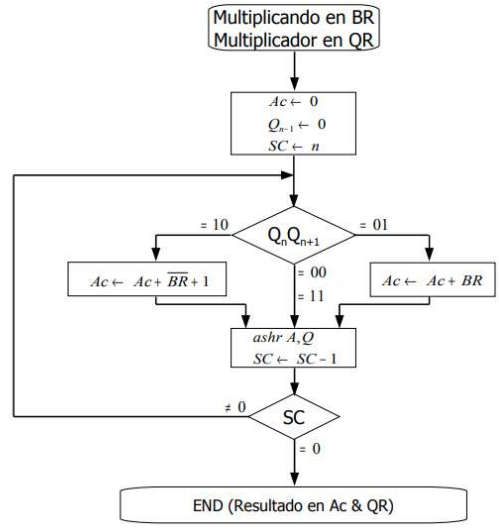
- Calcular el signo del resultado (XOR)
- Convertir ambos en positivos
- Algoritmo signo-magnitud
- Si el signo debe ser negativo, aplicar complemento a 2



Algoritmo Multiplicación II:

- Comprobar el signo del multiplicador (Q_s)
- Si es negativo, cambiar el signo de los dos
- Algoritmo signo-magnitud

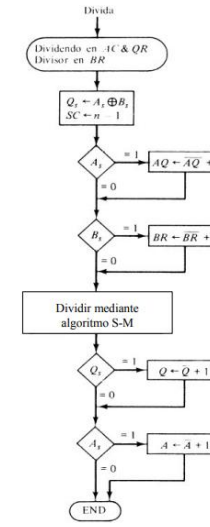
Algoritmo de Booth:



División en Complemento a 2

Debido a las complicaciones que puede generar la división entre sumas y restas en los residuos parciales, lo más conveniente es:

- Determinar el signo del cociente
- Convertir ambos números en positivos
- Dividir siguiendo el algoritmo de signo-magnitud
- Si el resultado debe ser negativo, se complementa



Representación Punto Flotante

$N = S \cdot M \cdot b^e$

- N: Número real
- S: Signo
- M: Mantisa
- b: Base
- e: Exponente

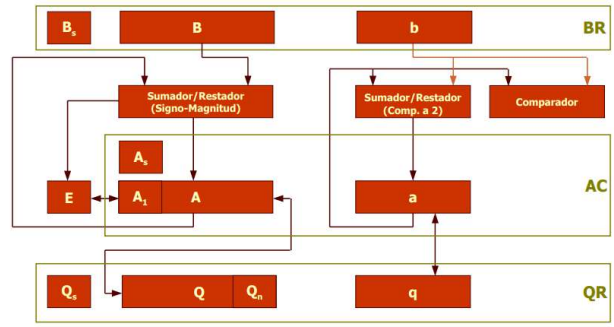
Representación binaria normalizada:

- La mantisa representa solo la parte decimal
- El primer dígito de la mantisa es distinto de 0
- Números negativos en signo-magnitud
- Exponente sesgado (127 para simple precisión y 1023 para doble precisión)
- No se incluye la base

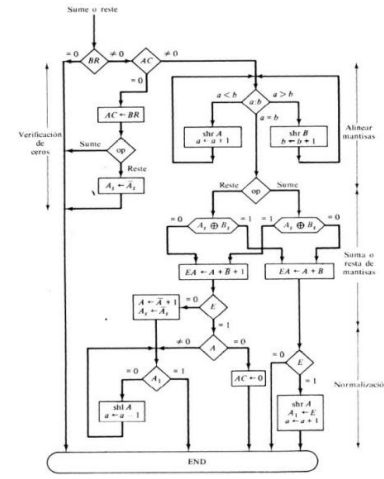
Números singulares:

- Cero: Signo = X; Exponente = 0; Mantisa = 0
- Infinito positivo: Signo = 0; Exponente = 255 ó 2047; Mantisa = 0
- Infinito negativo: Signo = 1; Exponente = 255 ó 2047; Mantisa = 0
- Números denormalizados: Signo = X; Exponente = 0; Mantisa != 0
- No un número (NaN): Signo = X; Exponente = 255 ó 2047; Mantisa != 0

Hardware para Punto Flotante



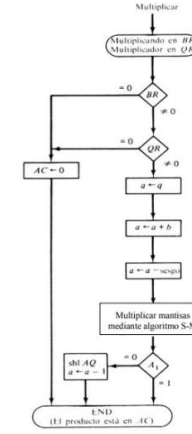
Operaciones en Punto flotante



Suma/Resta:

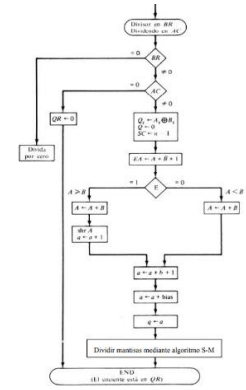
- Operadores con mismo exponente
- El exponente será el común y la mantisa será el resultado de la operación en signo-magnitud de las mantisas de los operandos
- Comprobar que el resultado está normalizado

Multiplicación:



- La mantisa del resultado será el producto en signo-magnitud de las mantisas de los operandos
- El exponente del resultado será la suma de los exponentes de los operandos. Hay que restar el sesgo a la suma de los exponentes
- Comprobar que el resultado está normalizado

División:



- La mantisa del resultado será la división en signo-magnitud de las mantisas de los operandos
- El exponente se obtiene restándoles al exponente del dividendo el del divisor. Hay que sumar el sesgo a la resta de los exponentes
- El cociente estará normalizado

Tema 5: Unidad de Memoria

Memoria principal:

- Acceso directo por CPU
- Almacena los programas en ejecución
- Tipo semiconductor
- Capacidad de MB y tiempo de acceso de nanosegundos
- Alto coste

Memoria secundaria:

- Acceso a través de interfaces
- Almacenamiento de programas y datos que no están en ejecución
- Tipo magnético u óptico
- Capacidad de GB/TB y tiempo de acceso de milisegundos y segundos
- Baratas

Memoria caché:

- Almacena la información más utilizada
- Muy rápida
- Tamaño reducido debido al alto coste
- Acceso directo desde CPU y memoria principal
- Necesidad de un módulo administrador de memoria

Memoria asociativa:

- Memoria de acceso por contenido
- Muy usadas en bases de datos
- Tipos de búsqueda
    - Secuencial (lenta pero sencilla y económica)
    - Paralela (rápida pero compleja y costosa)
- Muy rápidas

Memoria de reserva (Caché)

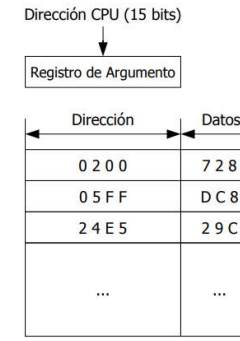
Memoria colocada entre memoria principal y la CPU que almacena instrucciones y datos activos. Es tan rápida como la CPU o, al menos, más rápida que la memoria principal, pero tiene menos capacidad que esta.

La CPU solicita una dirección que se busca primero en la memoria caché. Si está se toma el contenido y se continúa, si no, se accede a memoria principal, se toma el dato y se trae a caché el bloque que contenga la dirección accedida. La eficacia de la memoria caché se mide con la Tasa de Acierto = $\frac{Aciertos}{Aciertos+Fallos}$

Mapeo: Cómo relacionar las palabras de memoria principal y memoria caché. Existen 3 tipos:

- Mapeo asociativo
- Mapeo directo
- Mapeo asociativo por conjuntos

Mapeo asociativo



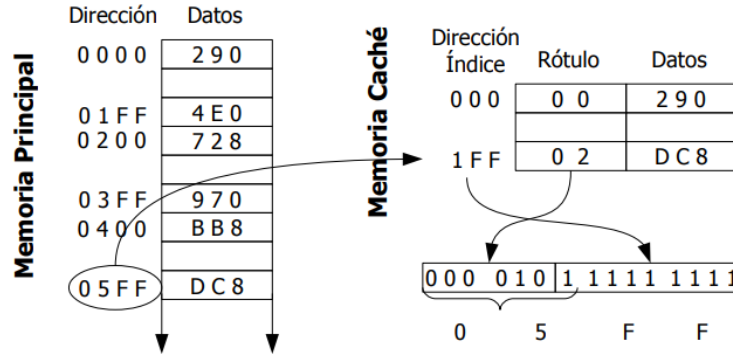
Implementación usando una memoria asociativa (Método rápido pero costoso). El procedimiento es el siguiente:

- La CPU solicita una dirección, colocándola en el registro de argumento
- Si se encuentra se proporcionan los datos
- Si no se encuentra se accede a memoria principal para traer la dirección y datos (si está llena se utiliza un algoritmo de reemplazo)

Mapeo directo

Implementación utilizando RAM (poco costoso y suficientemente rápido, aunque no es demasiado eficiente).

La dirección se divide en 2 campos: Índice y rótulo



Las palabras se agrupan en bloques. El índice de las direcciones se compone de bloque y posición del bloque.

Se busca mediante los bits de índice y se comparan los rótulos. Si coinciden se localiza la palabra, si no, se cambia el bloque completo. Esto provoca un mayor porcentaje de acierto al tener N palabras consecutivas, pero, en caso de fallo, provoca un mayor tiempo de intercambio.

Mapeo asociativo por conjuntos

Agrupación de palabras con su rótulo asociados a un mismo bloque de caché. Mezcla las ventajas de los dos anteriores. El almacenamiento es:

- Como mapeo directo: Cada bloque almacena palabras cuya dirección en memoria principal coincida con el índice asociado
- Como mapeo asociativo: Cada bloque de caché podrá almacenar varias palabras con rótulos diferentes

Búsqueda:

- Utilizando el índice de la posición física se determina la posición en caché
- Se determina si el rótulo está en el bloque mediante memoria asociativa

Escritura en caché

Para garantizar la coherencia de los datos de la memoria caché se utilizan dos mecanismos:

- Escritura directa
    - Se escribe en paralelo en caché y memoria principal
    - La memoria principal siempre está actualizada a costa de mayor lentitud
- Escritura diferida
    - Se escribe solo en caché, marcando con un bit de "modificado"
    - Al retirar la palabra, si el bit está activo se copia a memoria principal
    - Falta de coherencia temporal entre caché y memoria principal

Memoria virtual

Permite la ejecución de programas de gran tamaño, almacenándolos completos en memoria secundaria y parte de las tareas en principal.

La CPU solicita datos o instrucciones utilizando direcciones virtuales. Para ello, es necesario convertir las direcciones virtuales en direcciones físicas. Si el dato no está en memoria principal lo trae desde secundaria.

Mapeo de dirección virtual

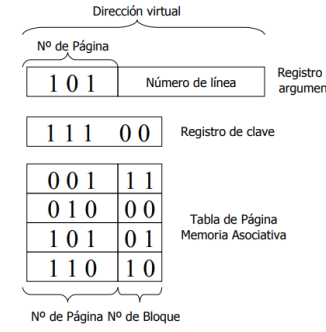
División del espacio de direcciones y de memoria en trozos con tamaño fijo e igual (Páginas para el espacio de direcciones y bloques para el espacio de memoria)

Estructura de una dirección virtual:

- P bits más significativos: Número de página
- Resto de bits: Número de línea dentro de la página

Tabla de página de memoria asociativa

- Tabla con tantas posiciones como bloques en memoria física
- Realiza búsqueda por contenido (muy rápida)
- Cada palabra de la tabla de página tiene 2 campos (nº de página y nº de bloque)
- Detección rápida si está presente una página

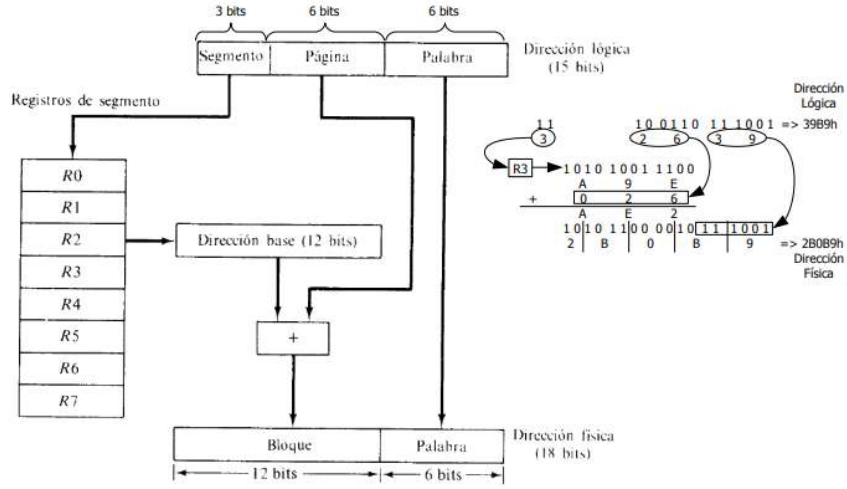


Algoritmos de reemplazo

- FIFO: Necesidad de una pila por páginas
- LRU (Least Recently Used): Necesidad de un contador asociado a cada página que se reinicia cada vez que se accede a dicha página
- LFU (Least Frequently Used): Necesidad de un contador asociado a cada página que se incrementa cada vez que se referencia la página

Mapeo de Página Segmentada

La longitud de segmento es variable. Para traducir de direcciones lógicas a direcciones físicas: Segmento|Página|Palabra => Bloque|Palabra

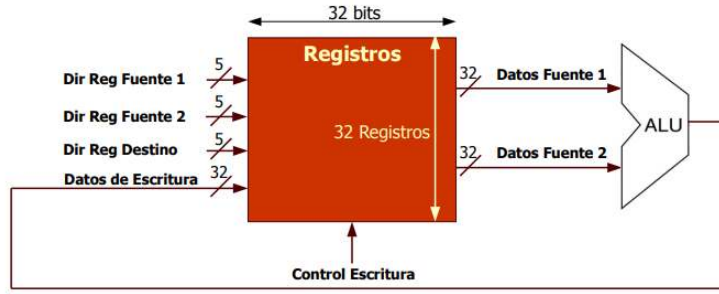


Tema 2: Lenguaje del computador MIPS

Procesador MIPS

Diseñado en la Universidad de Standford por el equipo de John L. Hennessy, utiliza una arquitectura de registros de propósito general (CPU con un banco de registros compuesto por 32 registros de 32 bits).

Utiliza arquitectura RISC, que implica instrucciones de tamaño fijo y presentadas en un reducido número de formatos. Sólo las instrucciones de carga y almacenamiento acceden a la memoria de datos.



Operandos en registros

Los registros están numerados de 0 a 31, pero su representación simbólica está formada por dos caracteres precedidos por el símbolo '$'. Por ejemplo:

- $s0, $s1, …, $s7: Registros utilizados para valores salvados (variables de programas)
- $t0, $t1, …, $t9: Registros utilizados para valores temporales
- $zero: Registro cuyo valor siempre es cero (no modificable)

Ejemplo: add $t0, $s1, $s2 # *El registro $t0 contiene la suma de los contenidos de los registros $s1 y $s2*

Operandos en memoria

Cuando aparecen variables con estructuras complejas hay que definir un formato nuevo para poder trabajar con ellas.

La memoria utilizada por MIPS es de 32 bits por palabra y utiliza 32 bits para ser direccionada (el elemento mínimo direccionable es el byte, los dos último se encargarán de seleccionar el byte dentro de la palabra).

La instrucción que transfiere datos de memoria a algún registro se denomina lw (*load word*) y la de almacenar en memoria sw (*store word*).

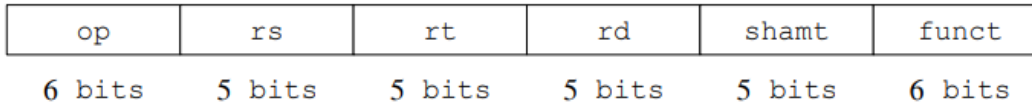
Ejemplo: Supongamos que A es una tabla de 100 palabras y que g y h están asociadas a $s1 y $s2. La base de A se encuentra en $s3. Para hacer g = h + A[8]:

lw $t0, 32($s3) # *El registro $t0 se carga con A[8] (4x8 = 32)*

add $s1, $s2, $st0 # *g = h + A[8]*

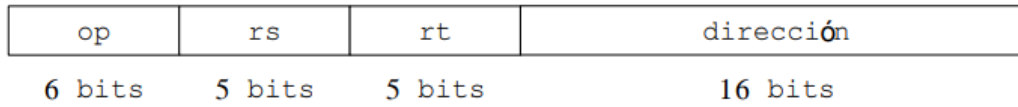
Formatos de instrucción

Las instrucciones de MIPS son de 32 bits y están divididas en los siguientes campos:



- op: Código de operación
- rs: Primer registro operando fuente
- rd: Registro operando destino
- shamt: Tamaño de desplazamiento (*Shift AMounT*)
- funct: Función (selecciona la variante específica de la operación op)

Para lw y sw sólo se usan dos direcciones de registro. Para especificar la constante que se sumará al registro índice necesitamos un nuevo formato:



El formato que usa dos registros operando fuente y un registro operando destino se llama Tipo-R (de registro), mientras que el último descrito se denomina Tipo-I

Instrucciones para la toma de decisiones

beq registro1, registro2, L1: La instrucción beq (Branch if equal) realiza un salto a la sentencia L1 si el valor del registro1 es igual al del registro2

bne registro1, registro2, L1: La instrucción bne (Branch if not equal) realiza un salto a la sentencia L1 si el valor del registro1 no es igual al del registro2

slt $t0, $s1, $s2: slt (Set on less than) guarda en $t0 un 1 si $s1 < $s2, en caso contrario guarda un 0

Instrucciones inmediatas

Utilizando el Tipo-I, rs sería el registro donde está el primer operando, rt el registro destino y la constante estaría codificada en los 16 bits destinados a la dirección.

addi $s0, $s1, 5 # *$s0 = $s1 + 5*

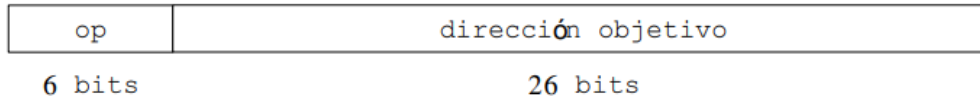
slti $t2, $s3, 8 # *$t2 = 1 si $s3 < 8*

lui (*Load upper inmediate)*: Carga los 16 bits codificados en la propia instrucción en los 16 bits más significativos del registro destino

Saltos incondicionales

j Fin # *Salta a "Fin"*

Para saltos se plantea un tercer formato de instrucción, el Tipo-J:



Operaciones lógicas

Operaciones de desplazamiento lógico:

sll $s0, $s1, 4 #*Transfiere el valor de $s1, desplazado 4 bits a la izquierda, a $s0*

srl $s0, $s1, 4 #*Transfiere el valor de $s1, desplazado 4 bits a la derecha, a $s0*

Operaciones AND y OR lógicas:

and $t0, $t1, $t2 # *$t0 = $t1 & $t2*

or $t0, $t1, $t2 # *$t0 = $t1 | $t2*

Operaciones AND y OR lógicas inmediatas:

andi $t0, $t1, 0xFF00

ori $t0, $t1, 0xFF00

Carga y almacenamiento de bytes

lb $t0, 2($s1) # *Carga el byte seleccionado de memoria y lo transfiere a los 8 bits menos significativos del registro destino*

sb $t1, 4($s1) # *Carga los 8 bits menos significativos del registro fuente y los transfiere al byte seleccionado en memoria*

Instrucciones de llamada a procedimientos

jal dirección # *Salta a dirección y la dirección de retorno se almacena en $ra*

jr $t1 # *Salta a la dirección contenida en $t1*

Pseudoinstrucciones

move $t0, $t1 # *$t0 se carga con el valor de $t1*

El ensamblador convierte esta instrucción en la siguiente que sí está implementada: add $t0, $t1, $zero # *$t0 = $t1 + 0*

blt (Branch if less than), ble (Branch if les or equal to), bgt (branch if greater than) y bge (Branch y great than or equal to)

El coste para esto es la reserva del registro $at

Otra de las pseudoinstrucciones más usadas es li (load inmediate).

Nº de reg	Nombre	Uso	Preservado en las llamadas
0	$zero	Valor constante cero	No aplicable
1	$at	Reservado por el ensamblador	No
2-3	$v0-$v1	Valores para resultados de las llamadas a procedimiento y evaluación de expresiones	No
4-7	$a0-$a3	Argumentos	No
8-15	$t0-$t7	Temporales	No
16-23	$s0-$s7	Salvados	Sí
24-25	$t8-$t9	Más temporales	No
26-27	$k0-$k1	Reservados al núcleo del sistema operativo	No
28	$gp	Puntero global	Sí
29	$sp	Puntero de pila	Sí
30	$fp	Puntero de bloque de activación	Sí
31	$ra	Dirección de retorno	Sí