ARQUITECTURA DE COMPUTADORAS

UNIVERSIDAD NACIONAL DEL SUR

PRIMERA EDICIÓN: MARZO 2003

ÚLTIMA EDICIÓN: DICIEMBRE 2012

::: ACORDE A
ARQUITECTURA DE COMPUTADORAS PARA INGENIERIA
2DO CUATRIMESTRE 2012 :::

NOTAS:
0) EN LA PÁGINA 24 DEL PRESENTE DOCUMENTO SUGIERO ESTUDIAR EL APUNTE DE DIVISIÓN RÁPIDA SRT BRINDADO POR LA CÁTEDRA, ES LO ÚNICO QUE NO ESTÁ INCORPORADO EN EL RESUMEN. AL AGREGARLO SOBREESCRIBIR LA COPIA DE SEGURIDAD EN PDF.
1) ESTE RESUMEN PERMITE ALCANZAR UNA APROBACIÓN SEGURA DEL EXAMEN FINAL REGULAR SIEMPRE Y CUANDO SE LO ESTUDIE EN SU TOTALIDAD, LO CUAL QUEDA A TOTAL DISCRECIÓN DEL LECTOR
2) SI BIEN FALTAN ILUSTRACIONES (PEQUEÑOS ESQUEMAS Y DIAGRAMAS DE GANTT QUE COMPLEMENTEN LA TEORÍA), SE LAS PUEDE ENCONTRAR EN LA BIBLIOGRAFÍA DE LA CÁTEDRA.
3) SE BRINDA TOTAL DERECHO DE MODIFICACIÓN, AUNQUE SE PROVEE UNA VERSIÓN EN PDF COMO COPIA DE SEGURIDAD.

Capítulo 1- Circuitos Digitales

Teoremas

- 1a. 0.X = 0 — 1b. 1+X=1
- 2a. 1.X=X — 2b. 0+X=X
- 3a. X.X=X — 3b. X+X=X
- 4a. X . not(X)=0 — 4b. X+ not(X)=1
- 5a. X.Y = Y.X — 5b. X+Y = Y+X
- 6a. X.Y.Z = X.(Y.Z)= (X.Y).Z — 6b. X+Y+Z = X+(Y+Z)=(X+Y)+Z
- 7a. not(X.Y....Z)= not(X) + not (Y) + ... + not (Z). — 7b. not(X+Y+...+Z)= not(X).not (Y). ... not (Z).
- 8. not f (X,Y,...,Z, .,+)= f (not(X), not(Y),..., not(Z), +, .)
- 9a. X.Y + X.Z = X.(Y+Z) — 9b. (X+Y) . (X+Z) = X + (Y.Z)
- 10a. X.Y + X. not(Y) = X — 10b. (X +Y).(X + not (Y))=X
- 11a. X + X.Y =X — 11b. X.(X + Y) = X
- 12a. X + not (X) . Y = X+Y — 12b. X.(not(X) + Y) = X.Y
- 12a’. Z.X + Z. not(X). Y =Z.X + Z.Y
- 12b’. (Z + X).(Z + not (X)+Y) = (Z + X) (Z + Y)
- 13a. XY + not(X).Z + Y.Z = X.Y + not(X).Z
- 13b. (X+Y). (not(X) + Z).(Y + Z)= (X + Y)(not(X)+ Z)

Formas Especiales de Expresiones Booleanas

Hay cuatro formas especiales de expresiones Booleanas:

- Suma expandida de productos
- Producto expandido de sumas
- Mínima suma de productos
- Mínimo producto de sumas

La suma expandida de productos y el producto expandido de sumas son usados para el análisis de expresiones booleanas y están asociadas a circuitos y, también, son el punto de partida de otros métodos de simplificación. La mínima suma de productos y el mínimo producto de sumas son muy interesantes porque los circuitos son más frecuentemente implementados directamente por estas expresiones.

Suma expandida de Productos (miniterm [1’s de la función])

Aquí cada término contiene cada variable, complementada o sin complementar. Para obtener la suma expandida de productos desde una suma de productos, las variables perdidas son puestas en todas las posibles combinaciones de cada producto. Lo que se aplica es el teorema 10a en reversa

$$\sum(0,1,4,5) =$$
$$= not(A).not(B).not(C) + not(A).not(B).C + A.not(B).not(C)+A.not(B).C$$

Producto expandido de Sumas (maxiterm [0’s de la función])

El producto expandido de sumas puede ser obtenido de un producto de sumas de la misma manera que en la suma expandida de productos. Las variables perdidas son puestas en todas las posibles combinaciones, a cada suma.

$$\Pi(2,3,6,7) =$$
$$= (A+not(B)+C).\ (A+not(B)+not(C)).\ (not(A)+not(B)+C).\ (not(A)+not(B)+not(C))$$

Expresiones Booleanas Mínimas

El objetivo principal de la minimización de una expresión booleana consiste en llegar a una representación que corresponda a un costo mínimo de switch’s en el circuito. Para este fin, el mínimo número de bloques lógicos es generalmente el criterio primario y el mínimo número de entradas en los bloques lógicos es típicamente el criterio segundo. Un bloque lógico esta compuesto de AND’s y OR’s. Se habla de AND seguido de OR en una suma de productos y de OR seguido de AND en un producto de sumas.

Una mínima suma de productos puede ser obtenida de una suma de productos por la aplicación de teoremas de simplificación. De igual manera se puede obtener un mínimo producto de sumas de un producto de sumas.

Una suma de productos puede ser obtenida de una expresión que no está en esta forma, “multiplying out” la expresión, es decir aplicando el teorema 9a en reversa.

Un producto de sumas puede ser similarmente obtenido por la operación dual de “adding out”, es decir, aplicando el teorema 9b en reversa.

Método de Minimización

Los métodos de minimización se clasifican en tabulares o gráficos. El método de Quine–McCluskey es un método tabular. Los métodos Karnaugh y Veitch son métodos gráficos.
Ambos tipos de métodos no dan soluciones, bridan un punto de partida ayudando a determinar aquellos términos irreducibles. Salvo que el problema sea trivial, la obtención de la/s solución/es optimas requerirán de un proceso adicional.

Combinaciones Opcionales

Como sabemos, para una deseada función de un circuito, todas las combinaciones posibles de entrada pueden ser consideradas en dos grupos; un grupo consiste de aquellas combinaciones para las cuales se desea una salida, y en el otro grupo todas las combinaciones para las cuales no se desea ninguna salida.
Un ejemplo es:

- La salida es de la forma: `not(A).not(B) + A.B.C`
- No se desea salida en:

  `not(A).C + A.not(C) + A.not(B)`
  o
  `not(A).C + A.not(C) + not(B).C`

Notar que la expresión de salida es el complemento de la de no salida
Ahora, todas las posibles combinaciones de entrada, pueden ser divididas en tres grupos:

- Un grupo consiste en aquellas combinaciones para la cual se desea una salida.
- Otro grupo son aquellas combinaciones para la cual no se desea una salida.
- Y un tercer grupo de combinaciones opcionales.

Estas combinaciones de entradas especiales pueden provenir de dos condiciones distintas:

- La combinación opcional es inválida, es sabido que nunca puede ocurrir.
- La combinación opcional es “don’t care”, es decir no influye en la salida si esta ocurre o no a la entrada.

No es necesario diferenciar entre combinaciones inválidas y don´t care; ambas tienen influencia en el circuito de la misma manera. Si una combinación opcional es adherida a una expresión booleana para la salida de un circuito, la expresión se puede volver más simple o más compleja.

Minimización con el Método Tabular (Quine–McCluskey)

Este método se basa en el teorema `X.Y + X.not(Y) =X`
X representa una o más variables e Y representa una sola variable.
El primer paso es el de llevar la expresión a una suma expandida de productos. Este teorema es aplicado a exhaustivamente para llegar a los términos irreducibles.
Luego se separan de la tabla cada término en función de su peso (cantidad de 1’s).
El tercer paso es comparar cada término de un determinado peso con todos los términos del peso siguiente. Si coinciden en todos los unos y solo no coinciden en un cero, se marcan como que se juntan y se crea una nueva tabla donde se tacha (-) el cero que no concuerdan.
Esto se hace hasta que no se puedan hacer coincidir ningún término con otro.
Cada término que queda al final se llama término irreducible.
Una vez terminado de encontrar todos los irreducibles, se crea una tabla donde cada columna son los términos de un principio y cada fila son los irreducibles.
Primero se elige cada terminal que sea el único y que cubra determinada columna luego se elige los terminales que cubran más columnas.

Solución algebraica para la tabla de implicantes primos

Una forma de realizar la selección de los mini términos es a través de una forma algebraica. Se le colocan una letra mayúscula a cada término irreducible y según quien cubra cada columna se realizan un productos de sumas donde cada suma son los mini términos que cubren cada columna. A través de aplicaciones de las propiedades y de los teoremas se reduce la expresión. Se llega a una suma de productos donde cada variable representa un término irreducible. Se elige uno de los términos de la suma de productos, dependiendo de la cantidad de entradas de cada término o de la cantidad de irreducibles que contenga.

Diagrama Lógico Detallado

El diagrama lógico detallado es de interés principal para el fabricador y para el personal de servicio donde es usado, por ejemplo en testeo, mantenimiento, etc. El diagrama lógico detallado representa funciones lógicas y no

lógicas e incluye los aspectos físicos y eléctricos del circuito. El diseñador lógico debe saber como producir un diagrama lógico detallado de un diagrama lógico básico y como leer un diagrama lógico detallado.
Las funciones lógicas son implementadas por circuitos lógicos o dispositivos. La información y las condiciones eléctricas se mantienen separadas de las funciones lógicas hasta este punto. Las entradas y las salidas de los circuitos lógicos debe encontrase en uno de dos estados. Estos dos estados, son típicamente dos valores de voltaje o niveles de corriente. Uno de los principales puntos aquí es relacionar el estado lógico 1 y 0 con, por ejemplo, el más alto o bajo nivel de voltaje. Esta relación, a la que llamamos "polaridad", es uno de los aspectos más importantes de el diagrama lógico detallado.
El nivel de voltaje es inmaterial; representamos el más alto (más positivo) de los dos nivel con H y el más bajo (el más negativo) de los niveles con L. Existen dos posibilidades:

- H representa 1 y L representa 0
- H representa 0 y L representa 1

Si pasa lo primero (H = 1, L = 0), un circuito o sistema se dice que tiene u opera con "lógica positiva". Si pasa lo contrario (H = 0, L = 1) se dice que el circuito o sistema tiene "lógica negativa".
Salvo que se indique lo contrario, por default las señales que entran o salen de un circuito se asumen en lógica positiva.

Circuito AND

Tabla *Verdad*			*Tabla Verdad*			*Tabla Verdad*		
Voltaje			Lógica Positiva			Lógica Negativa		
A	*B*	*F*	*A*	*B*	*F*	*A*	*B*	*F*
L	L	L	0	0	0	1	1	1
L	H	L	0	1	0	1	0	1
H	L	L	1	0	0	0	1	1
H	H	H	1	1	1	0	0	0

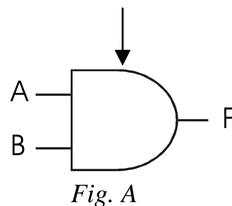


Fig. A

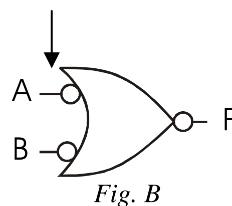


Fig. B

La *Fig. A* es un diagrama detallado, donde la función AND es implementada con un circuito AND, con una lógica positiva.
La *Fig. B* es un diagrama detallado, donde la función AND es implementada con un circuito OR, con una lógica negativa. Se indica que es una lógica negativa a través de los círculos tanto a las entradas como las salidas.

Circuito OR

Tabla Verdad			*Tabla Verdad*			*Tabla Verdad*		
Voltaje			Lógica Positiva			Lógica Negativa		
A	*B*	*F*	*A*	*B*	*F*	*A*	*B*	*F*
L	L	L	0	0	0	1	1	1
L	H	H	0	1	1	1	0	0
H	L	H	1	0	1	0	1	0
H	H	H	1	1	1	0	0	0

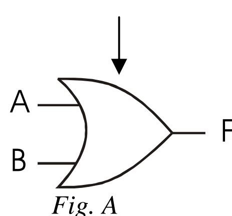


Fig. A

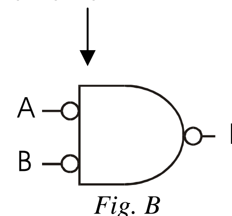


Fig. B

Funciones NAND y NOR

Los circuitos más comunes son NAND y en menor medida NOR. El punto es que no es deseo del fabricante el and(or) negado. La cuestión es que para evitar la negación se debe introducir una etapa más de inversión, luego mayor retardo (NAND(2) 3 msg, AND(2) 6 msg). La idea para diseñar será emplear NAND y NOR en lógica mixta.

La función NAND tiene como salida 0 solo si todos las entradas son 1's. En manera inversa, la salida es solo 1, si alguna de las entradas, o más de una, son 0's. La función NAND es lo mismo que negar un AND común (Ej. not(X AND Y) = X NAND Y).
La función NOR tiene como salida 0 solo si una o más de las entradas son 1's. Inversamente, la salida es solo 1 si todas las entradas son 0`s. La función NOR es lo mismo que negar el OR común. (Ej. **NOT**(X**OR**Y) = X**NOR**Y).
La álgebra de Boole puede ser realizada con solo funciones NAND o solo con funciones NOR.

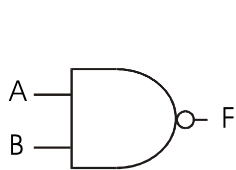
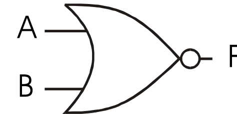


Tabla Verdad Lógica		
A	*B*	*F*
0	0	1
0	1	1
1	0	1
1	1	0

Tabla Verdad Lógica		
A	*B*	*F*
1	1	1
1	0	0
0	1	0
0	0	0



F= NOT (A AND B)
F= NOT(A) + NOT(B)

F= NOT (A OR B)
F= NOT(A) . NOT(B)

Circuito NAND

Tabla Verdad Voltaje

A	*B*	*F*
L	L	H
L	H	H
H	L	H
H	H	L

Tabla Verdad Lógica
H=1,L=0 en la entrada
L=1, H=0 en la salida

A	*B*	*F*
0	0	0
0	1	0
1	0	0
1	1	1

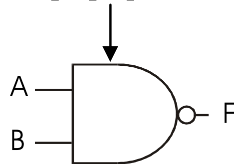
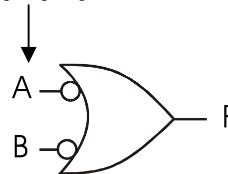


Tabla Verdad
H=0,L=1 en la entrada
L=0, H=1 en la salida

A	*B*	*F*
1	1	1
1	0	1
0	1	1
0	0	0



Circuito NOR

Tabla Verdad Voltaje

A	*B*	*F*
L	L	H
L	H	L
H	L	L
H	H	L

Tabla Verdad Lógica
H=1,L=0 en la entrada
L=1, H=0 en la salida

A	*B*	*F*
0	0	0
0	1	1
1	0	1
1	1	1

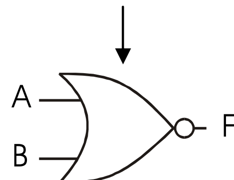
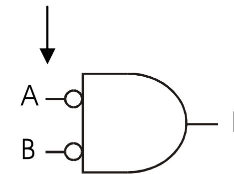


Tabla Verdad
H=0,L=1 en la entrada
L=0, H=1 en la salida

A	*B*	*F*
1	1	1
1	0	0
0	1	0
0	0	0



Una característica importante de los diagramas lógicos detallados, es que el diseñador puede mirar los símbolos e inmediatamente reconoce las funciones lógicas realizadas, sin conocerlas, sabiendo que tipos de dispositivos o que polaridad es usada para implementar la función. Por el lado contrario, mirando los símbolos y observando las funciones lógicas y las polaridades, se puede determinar no solo el tipo de dispositivo que se está usando, sino también los niveles de voltajes que se esperan.

Circuito Inversor
Su rol es un cambio de polaridad en la lógica, NO la inversión lógica. La inversión lógica se logra con la discontinuidad en la polaridad asignada a una variable en su recorrido.
Este dispositivo tiene una entrada y una salida, con la característica que la salida es L solo si la entrada es H y la salida es H solo si la entrada es L. Para mantener la consistencia en el diagrama lógico detallado, desde el punto de vista de los niveles de voltaje, un circulo pequeño debe aparecer en la línea de la señal de entrada o en la línea de la señal de salida.

Tabla Verdad Voltaje		*Tabla Verdad Lógica* H=1,L=0 en la entrada L=1, H=0 en la salida		*Tabla Verdad* H=0,L=1 en la entrada L=0, H=1 en la salida	
A	*F*	*A*	*F*	*A*	*F*
L	H	0	0	1	1
H	L	1	1	0	0

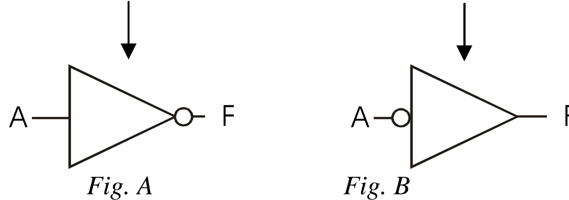


Fig. A *Fig. B*

Diseño Lógico MSI y LSI

El multiplexor Digital
Este dispositivo es llamado también "selector" ya que funciona muy parecido a una llave selectora. Por ejemplo, un multiplexor de ocho entradas selecciona una de las entradas para que sea la salida. Un código binario de tres bits determina que entrada aparecerá en la salida. Además es útil para simplificar funciones lógicas.

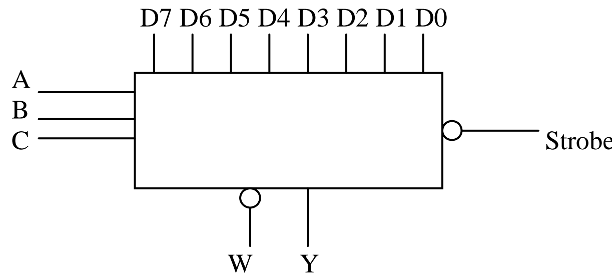


Fig. A Multiplexor de ocho entradas

La Fórmula de este sería
Y =ABC D7 + ABC' D6 + AB'C D5 + AB'C' D4 + A'BC D3 + A'BC' D2 + A'B'C D1 + A'B'C' D0
Aquí vemos que en este mux de 8 a 1, con 3 líneas de selección, pasamos de tener que implementar una función de M variables a 8 funciones residuo de M-3 variables que corresponden a la entrada de datos del integrado.

El Decodificador o Demultiplexor
Es lo opuesto al multiplexor. En un decoder una dirección binaria de entrada determina cuál de las salidas será más baja (0). Por ejemplo en un decoder decimal la entrada es D'CBA'=0110 = 6 en decimal. Entonces la pata 6 del decoder será 0 y todas las demás serán 1. No se lo utiliza para seleccionar una de varias entradas, se lo emplea para implementar una función lógica. Con el decoder podemos reflexionar en cuanto a que cada salida del mismo es asimilable a un miniterm, de aquí lo podemos entender como una "familia" de miniterm.

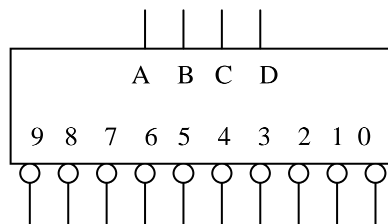


Fig. A Decoder Decimal

Cabe aclarar que el decoder y el mux se complementan, esto es, el mux removiendo variables y el decoder en la implementación de funciones residuo.

Memorias de solo Lectura (ROM)

La ROM la podemos emplear en un proceso “estándar” de fabricación de circuitos combinacionales.
En las líneas de entrada presentamos las variables y se apuntara a una u otra locación según la combinación de verdadero/falso de las mismas.
Luego si en la función expandida (suma de productos) se tiene un dado miniterm, en la locación asociada se asocia un 1, 0 en caso contrario.
La programación se hará con la tabla de verdad (0’s y 1’s de la función), es por esto, que es un proceso estándar, no se sabe como es el circuito.
Si tenemos una ROM de n líneas de address tendremos 2^n locaciones. De aquí se tiene que el tamaño de la ROM es 2^n palabras si n es el número de variables. Cada locación es asimilable a un miniterm. Luego con n variables, tenemos 2^n miniterminos y $2^{2^{\wedge}(n)}$ funciones distintas. Esto induce a pensar que dada la ROM, esta admite infinitas funciones distintas por lo que es un hardware desproporcionado.

Reducción del tamaño de la ROM

Algo muy redituable sería codificar las entradas. Para cada variable que elimina la ROM pasa a la mitad de tamaño. Si podemos almacenar codificadas las salidas también podremos reducir el tamaño, pero ahora solo linealmente. Sin embargo, esto provoca que las ecuaciones lógicas se vuelvan más complejas.
También la podemos pensar a esta memoria de solo lectura como un arreglo de *X*-Y de puntos. Por ejemplo una ROM de 16.384 bits es un arreglo de 128x128 puntos. El circuito tiene ocho patas de salidas y 11 patas de entrada (direccionamiento). Por lo tanto está organizada como 2048 palabras de 8 bits cada una. Por esto, para cada combinación de entrada, se produce una salida diferente de 8 bits.

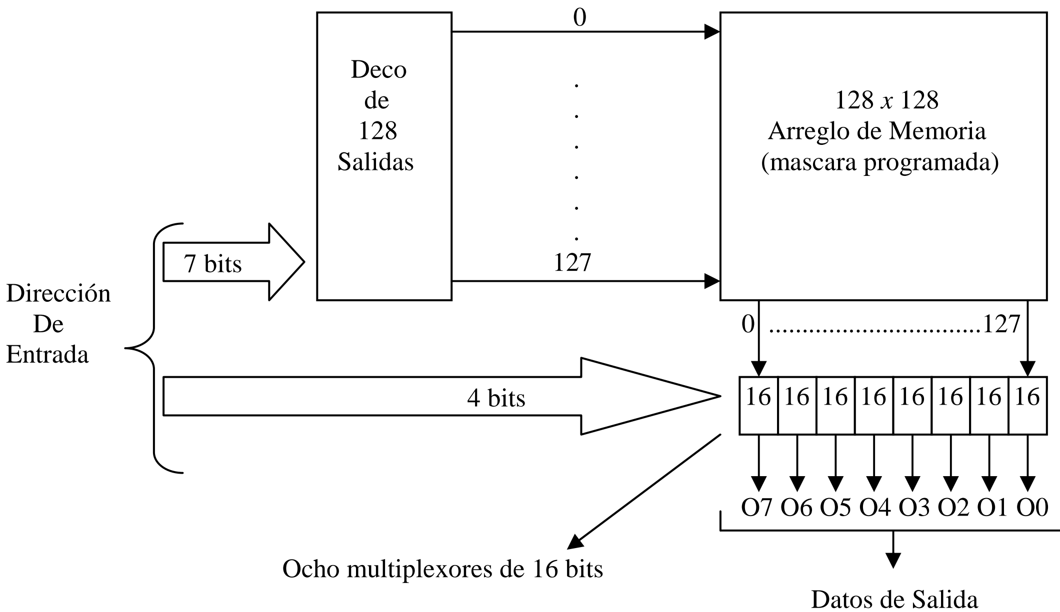


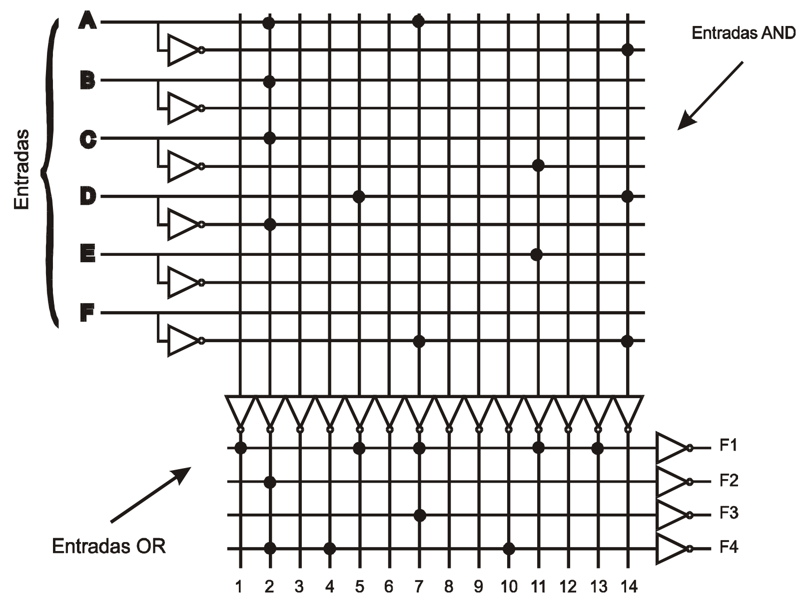
Fig. A ROM de 16.384 bits

Memorias de Solo lecturas Programables

La mayor desventaja de las ROMs programadas, es que es virtualmente imposible hacer algún cambio si un error se encuentra en el diseño. Por esto existen otros tipos de ROMs para producciones más pequeñas y para chequeos iniciales y producciones piloto.
Esta se llama Memoria de Solo Lectura Programable (PROM). Para programar el dispositivo, el usuario simplemente usa un Programador PROM, el cual escribe el bit deseado en el dispositivo. Algunas PROM son reprogramables, es decir que se pueden borrar eléctricamente o por exposición a una luz ultravioleta, y volverlas a escribir (EEPROM, EPROM).

Arreglos Lógicos Programables (PLA)

No es otra cosa que un AND seguido de OR. Con estas se especifica, no solo un bit de salida para cada word, sino la dirección del word. Esto hace posible tener más entradas que en una ROM del mismo tamaño.
Al igual que una ROM, podemos ver a una PLA como una memoria restringida, como un convertidor de código o como un generador de funciones lógicas. Además ambas matrices del PLA son especificadas por el consumidor mediante una tabla de verdad. La ROM también implementa una matriz que permite su decodificación dada una entrada. Tal como sucede con la ROM, los datos de la tabla de verdad se perforan en tarjetas que se utilizan para generar automáticamente una máscara especial en uno de los pasos de la fabricación de la PLA. En ambos integrados, generalmente es más económico decodificar salidas mutuamente excluyentes fuera del PLA con circuitos más económicos.
Lo que tenemos acá sería como una máquina de minitérminos. Las entradas son las variables y cada línea para abajo serían las variables que forman (mediante AND's) un término. Luego las líneas para la derecha que van hacia la salida son las uniones (OR's) de los minitérminos.
Además las redundancias se resuelven automáticamente. Es fácil observar como el layout de las matrices hace un uso muy eficiente del chip. El único factor a considerar es reducir el número de términos producto dado que el número de variables en cada término es irrelevante. Reducimos el número de términos productos solo hasta que se ajuste a las limitaciones del PLA, debido a que no existe beneficio alguno en obtener una reducción mayor.



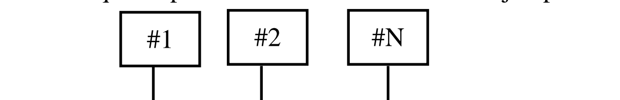
Por ejemplo:
El la columna 2 tenemos el siguiente minitérmino: A B C D E F
1 1 1 0 X X

El la columna 14 tenemos el siguiente minitérmino: A B C D E F
0 X X 0 X 0

En la fila F1 (la salida) tenemos: D + AF' + C'E

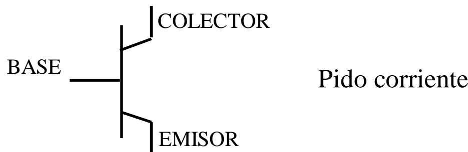
Open Colector

Existen situaciones en las que se plantean unir salidas entre sí. Ejemplo un bus.



En la comunicación, en el recurso compartido, se están uniendo salidas.
Una etapa de salida convencional no resistirá esta acción. La condición para que sea viable es que un nivel predomine sobre el otro.
El elemento activo de un circuito electrónico es el transistor, ya sea bipolar o MOS.

En general al transistor se lo opera como una llave, como un contacto, abierto (OFF) o cerrado (ON). A estos estados se los conoce también como cortado o saturado respectivamente.
A continuación presentamos el esquema del transistor bipolar



A continuación presentamos el esquema del transistor MOS

Unir una o más salidas open collector

Si hay algún bajo el punto de unión es bajo. Sea alto si todos están en alto.

LP <u>AND</u> WIRED
LN <u>OR</u> WIRED

Indico que la salida de una compuerta es open collector

Si alguien escucha mal la información se corrompe. A través de un bit de paridad se detecta la falla. Es tolerante ante una falla, nada se quema.
Tiene aspectos positivos y negativos. Por un lado resiste <u>físicamente</u> a errores de habilitación. Por el otro es relativamente lento.

Three State

Habilita a disponer de varias salidas unidas entre si a condición de que:

a) Todas estén en el tercer estado o
b) Uno de ellos sea sacado del tercer estado

Bus Three State

En esta implementación de un bus por un lado se destaca su rapidez. Sin embargo, si hay habilitación múltiple el mismo falla. Si esto ocurre se debe arreglar la falla del direccionado y reemplazar el/los circuitos quemados. Se lo debe emplear dentro de una tarjeta o un chip. Además este esquema implementa de manera muy eficiente el multiplexado de señales.
Cabe aclarar que Three State no resuelve funciones lógicas.

Lógica Secuencial

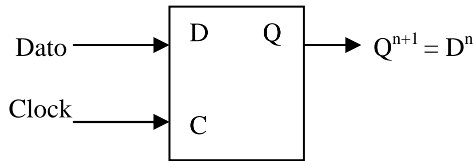
Elementos de Memoria

Aquí introduciremos el término de memoria cambiante. Si cambiamos el circuito dentro de una ROM, podemos cambiar los bits como queramos, y usarla para recordar cosas. La PROM es cambiable, pero no tan fácil como una RAM. Para cambiar el contenido de una RAM simplemente presentamos una muestra del bit deseado en la entrada de datos y pulsamos el reloj. La palabra es inmediatamente cambiada. La memoria nos permite recuperar todos los datos juntos en el instante en que se realiza el cómputo.
Al igual que tenemos compuertas lógicas como elementos básicos combinacionales y una ROM como un gran arreglo de elementos, tenemos los flip-flop como un elemento básico de memoria y una RAM como un gran arreglo de memoria. Esencialmente, compuertas y ROM's son análogos a flip-flops y RAM's, pero con flip-flops y RAM's tenemos la adición de la dimensión del tiempo. Ambos dispositivos tienen una entrada de reloj y sus estados cambian únicamente después de la transmisión del reloj.

Tipos de Flip-Flops

Los flip-flops se pueden definir a través de tablas de verdad y de diagramas, como los elementos combinacionales. En los flip-flops la salida siempre depende de la entrada. Como elementos de memoria, las salidas no cambian en función de la entrada hasta que el clock es transmitido.

- Flip-Flop D. Este es el flip flop Delay (retraso), y la ecuación muy simple nos dice que la entrada D es guardada en el flip-flop cuando ocurre el clock y aparece en la salida Q en el próximo tiempo de reloj. Este flip flop es muy parecido a una RAM de un solo bit, y es muy usada para guardar datos.

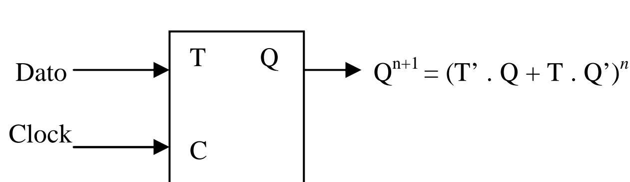


D^n	Q^{n+1}
0	0
1	1

Flip Flop "D"

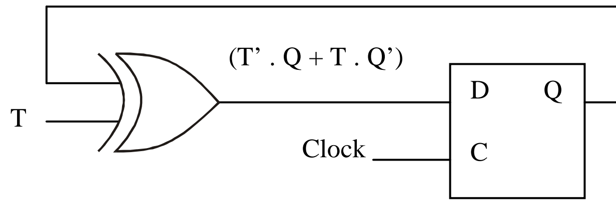
- Flip-Flop T. Este es el flip-flop toogle. Este da como salida el estado anterior si la entrada T es falso antes del clock. Si la entrada T es verdadera, la salida cambia hacia el estado contrario con el clock. Este es muy usado para contadores binarios.

T^n	Q^{n+1}
0	Q^n
1	$(Q')^n$



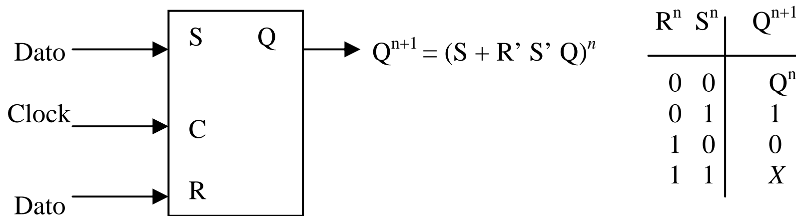
Flip Flop "T"

Cualquier flip-flop puede ser construido a través de un flip-flop D y usando algunas compuertas.



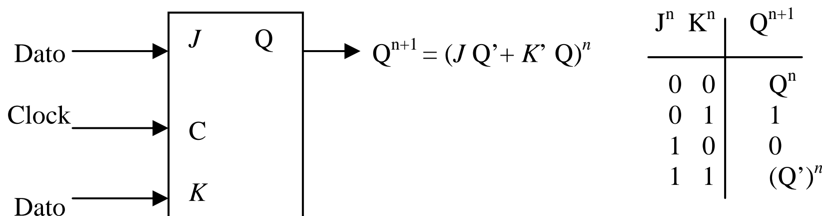
Ej. Flip-Flop T usando un flip-flop D.

- Flip-Flop R-S, este setea después que la entrada S sea true y resetea después que la entrada R es true. La salida es indefinida si ambos R y S son true



Flip Flop “R-S”

- Flip – Flop *J-K*, este setea después que *J* sea true y resetea después que *K* sea true. Es similar al R-S salvo que si *J* y *K* son true, la salida cambia al estado contrario.



Flip Flop “*J-K*”

Master-Slave
Se logra comportamiento de <u>tipo pulso</u>. Esto es, <u>la salida</u> cambiara luego de que la entrada (excitación haya desaparecido)

¿Por que en un registro de desplazamiento los flip-flops deben ser master-slave?

Cada flip-flop será leído y escrito simultáneamente. Es requisito para diseñar un circuito en modo pulso. Sino fuera MS cambiaria durante la exitación.

Capítulo 2 – Algoritmos Aritméticos

Suma

Sean
$X = (X_{n-1}, \ldots, X_0)$
$Y = (Y_{n-1}, \ldots, Y_0)$
Ambos de n dígitos y de una base b genérica.
Luego $X + Y = S = (S_n, S_{n-1}, \ldots, S_0)$
¥ b, $S_n = 0$ ó $S_n = 1$. Si $S_n = 1$ entonces se produce un overflow

```
1    c0=0, (c0, carry inicial)
2    For i=0 step 1 until n-1 do
          begin
               si = (xi + yi + ci) mod b;
               ci+1 = └ (xi + yi + ci) / b ┘
          end
     sn = cn
3    end
```

Se puede verificar que cualquier C_i será 0 o 1 (¥ b). Considerando que $X_i + Y_i \leq 2\,(b - 1)$ y con $C_0 = 0$ (o eventualmente $C_0 = 1$ si se restase en 2's complemento), el máximo valor para cualquier C_i será
$\lfloor (2(b-1) + 1) / b \rfloor = 1$
En este algoritmo se analiza cada digito una vez. Por lo tanto, se dice que el algoritmo es de complejidad O(n). Una función g(n) es de complejidad de orden f(n) si para cualquier n, $g(n) \leq k\, f(n)$. Menos que n es imposible, al menos habrá que procesar cada digito. Luego los caminos de mejora se dan a nivel de implementación.

Para obtener el circuito lógico que resuelve la suma recurrimos a la tabla de verdad.
Dado X_i, Y_i, C_i

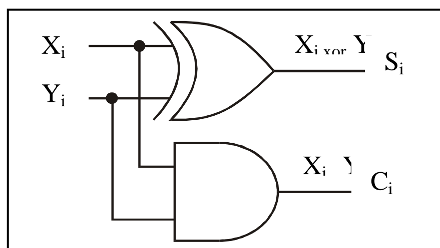
Half Adder (HA, Semi Sumador)

Este toma dos operandos de entrada y produce dos salidas:

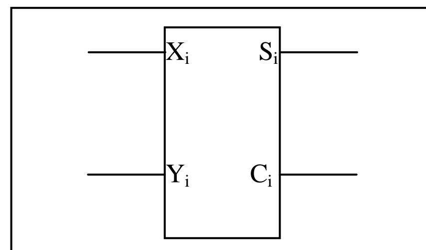
- S_i = suma de los operandos = X_i XOR Y_i
- C_i= carry de salida = $X_i.Y_i$

Tabla de verdad

X_i	Y_i	S_i	C_i
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Circuito Lógico



Esquemático

Full Adder (FA, Sumador Completo)

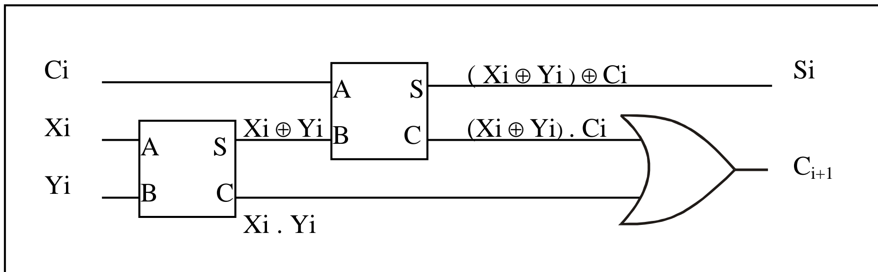
Tomas tres entradas (los dos operandos y el carry de entrada) y produce dos salidas:

- S_i = suma de los operandos con el carry de entrada = X_i XOR Y_i XOR C_i
- C_{i+1} = Carry de salida = $X_i.Y_i + (X_i + Y_i).C_i$ o equivalentemente
- C_{i+1} = Carry de salida = $X_i.Y_i$ + (X_i XOR Y_i).C_i o equivalentemente

Tabla de verdad

X_i	Y_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Full Adder a partir de dos Half Adders

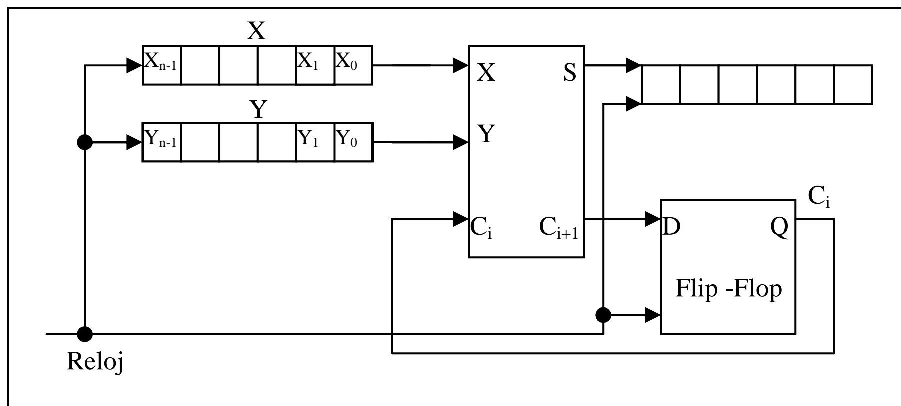


Circuito Lógico

Para sumar *n* bits lo que se hace es: empezar por la posición 0, pongo de entradas X_0,Y_0 y C_0. Esto me da el digito S_0 de la suma y utilizo el C_{out} para el C_{in} de la posición 1, y así hasta llegar hasta la posición *n*-1. (Orden (*n*)).

Sumador Serie: Un circuito para sumar n dígitos, muy básico, sería un sumador serie con un único Full Adder y un Flip-Flop; en la medida que se entreguen de manera serial los X_i e Y_i y se vayan almacenando los S_i resultantes.

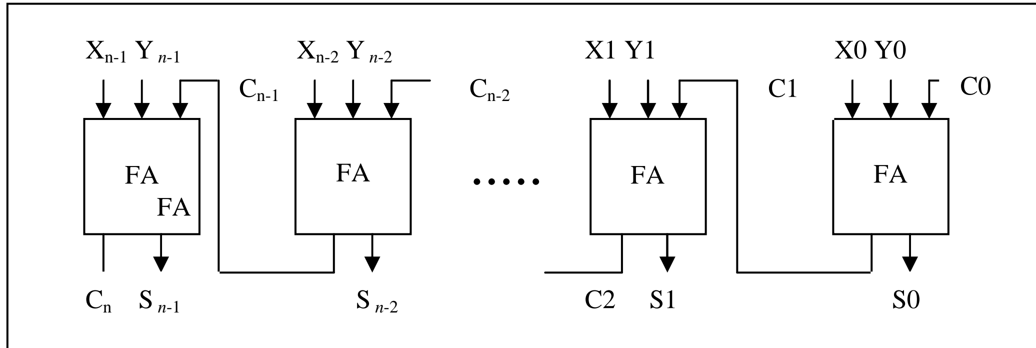
Los sumadores seriales no son usados en el diseño de procesadores centrales o ALU's debido a que son muy lentos y también muy restrictivos.



Un ejemplo de este tipo yace en un controlador de disco, el RR06 (DEC). El disco decidía que cilindro alcanzar de forma serial, de manera serial realizaba la diferencia.

Sumadores Paralelos

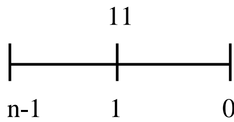
- **Ripple Adder**. Es el más simple. Consiste de n F.A. donde el carry de entrada a un F.A. es el carry de salida del F.A. precedente.



$C_{i+1} = \underbrace{X_i \cdot Y_i}_{g_i} + \underbrace{\underbrace{(X_i + Y_i)}_{p_i}.C_i}$

carry generado   carry propagado

Cabe aclarar que si los dos operandos son 1, se genera carry independientemente del carry de arrastre.
El tiempo de suma dependerá del valor particular de los operandos.



La situación más crítica en cuanto a disponer del carry de entrada es todos propagando y ninguno generando.
En este caso la complejidad es O(n).
Si operase en forma sincrónica deberé considerar para el tiempo el del peor caso. Asincrónicamente, detectando cuando termina la suma se verifica que el tiempo promedio es proporcional al $\log_2(^{5n}/_4)$.
Sin embargo, operar asincrónicamente no interesa por:

a) Complica el control
b) Un compilador optimizante no podrá tomar provecho de ello.

En particular, la suma (resta) es una operación básica que determinara muy posiblemente el periodo del reloj.

- **Carry Look-Ahead Adder.**
  Podemos comparar la circunstancia de tener un tiempo de suma proporcional a 2n (pensando que cada F.A. implica 2 niveles de compuertas) con el hecho de que una función lógica se resuelve con 2 niveles AND seguido de OR u OR seguido de AND.
  La respuesta a este planteo es "generar" el carry como una función independiente de la suma (De allí el nombre CLAA).

  $c_{i+1} = x_i \cdot y_i + (x_i \oplus y_i) \cdot c_i = g_i + p_i \cdot c_i$
  Donde G = carry generado y *P*=carry propagado
  $c_{i+1} = g_i + p_i \cdot (g_{i-1} + p_{i-1} \cdot c_{i-1})$
  $= g_i + p_i \cdot g_{i-1} + p_i \cdot p_{i-1} \cdot c_{i-1}) = \ldots\ldots$
  $= g_i + p_i \cdot g_{i-1} + p_i \cdot p_{i-1} \cdot (g_{i-2} + p_{i-2}\, c_{i-2})$
  Seguimos desarrollando hasta C_0.
  Luego tengo una función lógica para obtener el carry en los diversos full adders en la cual interviene los Xi e Y_i de las posiciones anteriores y el carry inicial.
  En cuanto a lo que retardo respecta tenemos:
  1 nivel para computar G y P.
  1 nivel para resolver los productos lógicos
  1 nivel par resolver la suma lógica
  3 niveles para determinar los carry de entrada a los F.A. + 2 niveles para la suma en sí
  Por lo tanto, la suma se resuelve en 5 niveles de compuertas cualquiera sea n, muy distinto de los 2n del ripple.
  La cuestión es que este planteo encuentra "rápidamente" limitaciones prácticas por:

a) Cuestiones de fan in
b) Cuestiones de fan out
c) Cuestiones de layout (caminos de interconexión largos e irregulares) que lo tornan, impracticable con n que supere valores 4-5.

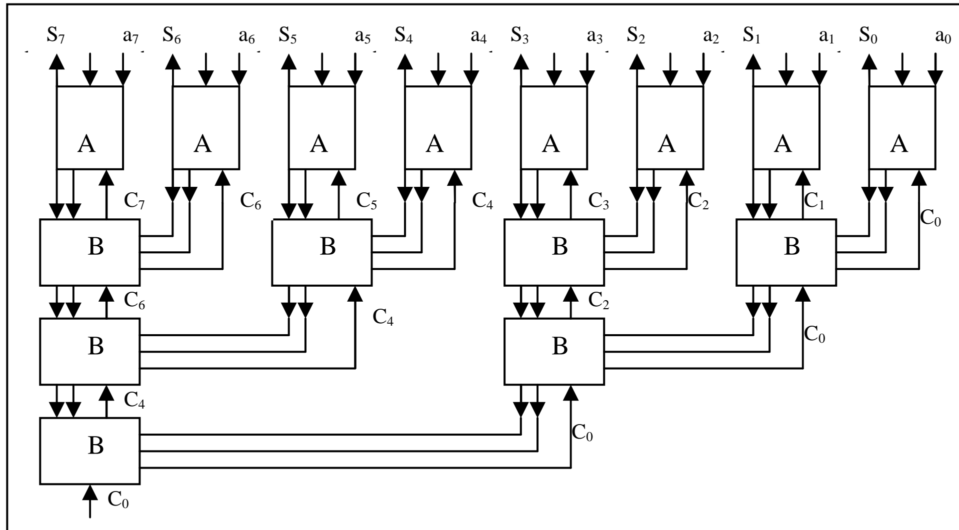
¿Cual es la salida a esto?
Sin apartarnos ápice de la idea (de carry por adelantado) el recurso será generar los p y g en etapas, esto es, instanciar la obtención de los mismos. No resolverlos en un nivel sino en varios niveles lógicos pagando el precio de un mayor retardo pero haciéndolo técnicamente rentables.
Si observamos las ecuaciones de P, G y C podemos obtener las siguientes relaciones recursivas:

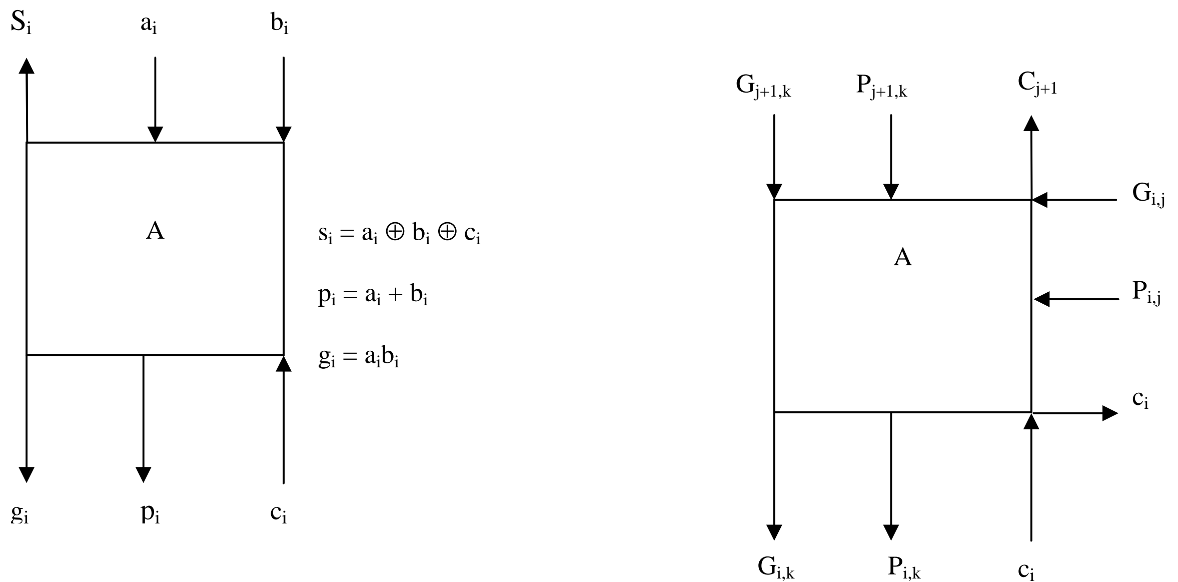
1. $C_{k+1} = G_{ik} + P_{ik} . C_i$ con $i < j$, $j+1 < k$ i — j — k
2. $G_{ik} = G_{j+1\,k} + P_{j+1\,k} . G_{ij}$
3. $P_{ik} = P_{ij} . P_{j+1\,k}$

Estas ecuaciones también valen si $i \le j$, $j < k$, si asumimos $G_{ii} = g_i$, $P_{ii} = p_i$

A modo de ejemplo veamos expresar P_{03} y G_{03}
$P_{03} = P_{01} . P_{23} = P_{00} . P_{11} . P_{22} . P_{33} = p_0p_1p_2p_3$
$G_{03} = G_{23} + P_{23}\, G_{01} = (G_{33} + P_{33}\, G_{22}) + P_{22}\, P_{33}\, (G_{11} + P_{11}\, G_{00}) = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$.

Con esto se puede construir un CLA práctico. El sumador consiste de dos partes. La primera computa los valores de P y G. La segunda usa estos valores de P y G para calcular los Carries. Dejando propagar las señales en los niveles expresados, se plantea recorrer el camino inverso a partir del C_o y de los P y G que fueron calculados en la primera instancia.
Los bits en un CLA deben pasar a través de log n niveles lógicos, comparados con los 2n del ripple-carry adder. Mientras que el ripple tiene n celdas, el CLA tiene 2n celdas, que usan un tamaño de n.log(n). Esto es lo que se paga por un incremento en la velocidad, espacio. Sin embargo, conserva una estructura regular lo que favorece la integración a gran escala.
Sea n=8:

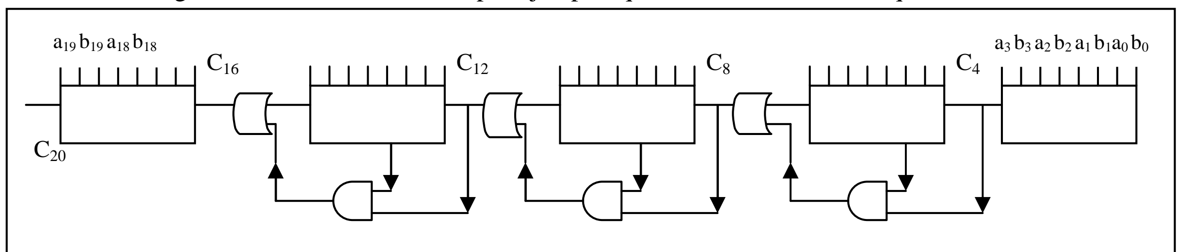




Existen alternativas, no tan costosas como el CLAA, pero de performance superior a la del ripple y algo inferior a la de este.

Básicamente construimos con pequeños sumadores ripple (de algunos pocos bits) sumadores complejos apelando a dos alternativas posibles:
a) Carry Skip Adder
b) Carry Select Adder

- **Carry Skip Adder.** Este está situado en el medio de ripple y CLA, en términos de velocidad y costo. La idea de este sumador comienza con el hecho que computar el propagado (*P*) es mucho más simple que computar el generado (G), por lo tanto el carry skip adder solo computa el propagado. Esto surge de la salida de carry del propio sumador, a partir de forzar inicialmente C_{in} = 0 con lo que luego del tiempo de suma el C_{out} no es otra cosa que el carry generado por el bloque.
Para analizar la velocidad, asumimos que toma 1 unidad de tiempo pasar la señal a través de dos niveles lógicos. Entonces si tenemos, por ejemplo, que sumar 20 bits, en bloques de 4 bits cada uno



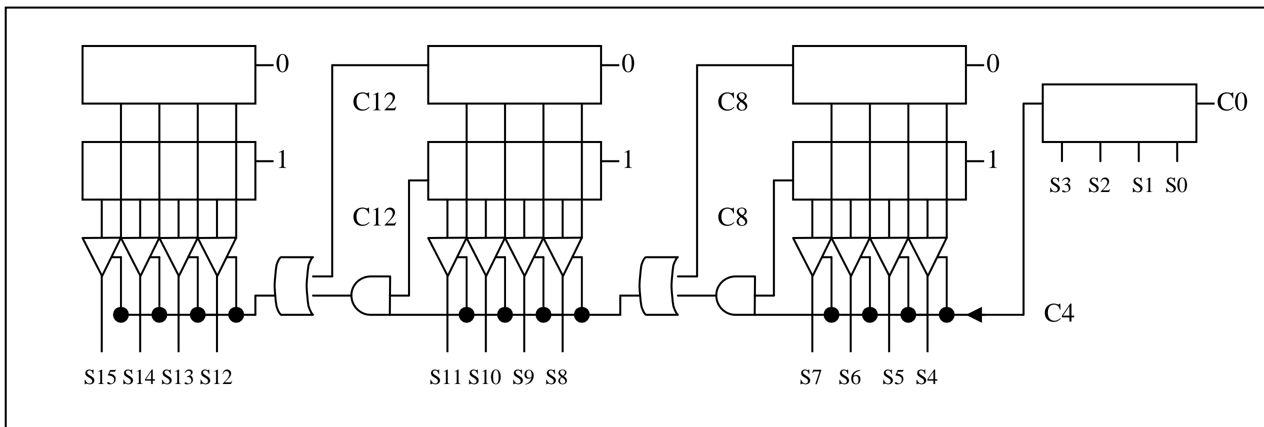
tenemos:
  - 4 unidades de tiempo para producir todos los propagados y el primer carry in (del primer bloque).
  - (*n*/*k* - 2) unidades de tiempo para pasar el bloque, es decir, (20/4 – 2) = 3 unidades de tiempo
  - 4 unidades de tiempo para que el último bloque realice la suma.

Esto nos da 11 unidades de tiempo para realizar la suma.
Una variante para mejorar el tiempo es tener bloques de distinto tamaño de bits. Resulta beneficioso ensayar ir aumentando hacia el centro en la idea de que la propagación del carry se va retrasando, dando luego posibilidad de mayor tiempo para generación sin prejuicio de temporizado.
A su vez conviene que los sumadores más alejados sumen menos bits (porque su carry se resuelve más tarde).
Por ejemplo si, en el caso anterior, tenemos el primer bloque de 2 bits, el segundo de 5 y los restantes de 6, 5 y 2 bits, el tiempo total se reduce a 9 unidades de tiempo.
¿Con equispartición dado n hay algún valor óptimo para m?
Retardo = Tiempo suma primero (coincide con la generación de los restantes) + Tiempo en hacerse del carry del último sumador + Tiempo de suma de este
$= m\,t + (n/m - 2)\,t + m\,t$
Retardo = $(n/m + 2m - 2)\,t$

Derivamos respecto a m
$\delta/ \delta m (n/m + 2m -2) t = n/m^2 -2 = 0$ entonces $m= \sqrt{(n/2)}$

Así Si n=32 m=4
Si n=128 m=8
Este circuito tiene una complejidad en tiempo del orden de $O(\sqrt{n})$.
Justamente con n=32 da 28t y con n=128 da 60t

- **Carry Select Adder**. Este trabaja con el siguiente principio: se realizan dos sumas en paralelo, una asumiendo carry in 0 y la otra asumiendo carry in 1. Cuando el carry es finalmente conocido, se selecciona la suma correcta. Con este método obtenemos el doble de velocidad con un costo de 50% más. Sin embargo los carry's deben atravesar varios multiplexores, los cuales incorporan un retraso. Para aprovechar esto, una variante es que cada bloque sea un bit más grande que su antecesor. Esto produce que se necesiten bloques de varios tamaños, por lo tanto no conviene equispartición, en su lugar, una rampa.
  El retardo puede calcularse, despreciando el retardo de los multiplexores, como:
  Retardo = m + (n/m – 2) y la complejidad en tiempo es también $O(\sqrt{n})$.



Sumador	*Tiempo*	*Espacio*
Ripple	$O(n)$	$O(n)$
CLA	$O(\log n)$	$O(n \log n)$
Carry Skip	$O(\sqrt{n})$	$O(n)$
Carry Select	$O(\sqrt{n})$	$O(n)$

Comparación de los Tipos de Sumadores

Multiplicación de enteros no signados

Sean *x*, *y* y *p* el multiplicando, multiplicador y producto respectivamente.
$X = (X_{n-1}, \ldots, X_0)$ e $Y = (Y_{n-1}, \ldots, Y_0)$ en una base b posicional.
Su producto P es un numero de 2n digitos, $P = (P_{2n-1}, P_{2n-2}, \ldots, P_0)$, los cuales se obtendrán como sigue:

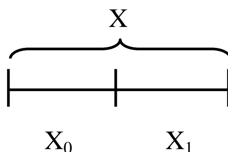
```
1       pj=0, 0<=j<=2n-1
2       For i=0 step 1 until n-1 do
3           If yi≠0 then
                begin
                    k=0
                    For j=0 step 1 until n-1 do
                        begin
                            z = xj yi + pi+j + k (en base b)
                            pi+j = Z mod b
                            k = └ z / b ┘
                        end
                    pi+n=k    {…P acumulador total…}
                end
```

La complejidad del algoritmo es $O(n^2)$. Esto indica que a nivel del algoritmo habrá mucho para hacer (cosa que no ocurre con la suma). Mucho se ha investigado sin precisar cuan cerca de O(n) se puede estar. El punto es que en la mayoría de los casos resulto de interés teórico más que practico.

Mejora del algoritmo

Si dividimos multiplicador y multiplicando en mitades:
$X = (X_1 2^n + X_0)$
$Y = (Y_1 2^n + Y_0)$
$X_1 = (X_{2n-1}, \ldots, X_n)$, $X_0 = (X_{n-1}, \ldots, X_0)$

Gráficamente



$P = X \ . \ Y = (2^{2n} + 2^n) X_1 Y_1 + 2^n (X_1 - X_0)(Y_0 - Y_1) + (2^n + 1)(X_0 Y_0)$
Estamos reemplazando un producto de operandos de 2n bits por tres productos de operandos de n bits.
[$T(2n) \leq 3T(n) + C_n$]
C se relaciona con sumas y corrimientos
La solución a esta ecuación es que $T(n) \leq 3\ C_n^{\ \log 3}$, esto es, de $O(n^2)$ se pasa a $O(n^{1,59})$.

Circuito Básico

A acumulador
MQ multiplier – quotient
Concatenados con capacidad de corrimiento a derecha
A = 0
MQ = Y (multiplicador)
B = X (multiplicando)
En cada iteración se analiza el bit menos significativo de MQ.
Si 1 sumo A+B y corro (con carry) a derecha
Si 0 simplemente corro
Y así hasta iterar n veces con lo que en AMQ queda el producto de X e Y

Multiplicación de enteros signados

Si es signo magnitud, el mismo hardware funciona.
Consideramos $X = (X_{n-1}, \ldots, X_0)$ e $Y = (Y_{n-1}, \ldots, Y_0)$ donde X_{n-1} e Y_{n-1} son los dígitos del signo.
Luego $P = (P_{2n-1}, P_{2n-2}, P_{2n-3}, \ldots, P_0)$
$P_{2n-3} \ldots P_0$ sale de hacer $X_{n-2}, X_{n-3}, \ldots, X_0 \,.\, Y_{n-2}, Y_{n-3}, \ldots, Y_0$
$P_{2n-2} = 0$
$P_{2n-1} = X_{n-1}$ XOR Y_{n-1}
En notación 2's complemento recordando que un numero negativo x es un numero positivo de valor $2^n - |Y|$, si aplicamos el hardware visto tanto con $Y < 0$ como $X < 0$ se induce a error!

Caso A

$X < 0$ $\quad P = X \,.\, Y = 2^{n-x} \,.\, Y = 2^n Y - |X| Y$
$Y \geq 0$
El resultado correcto es un numero negativo menos $|X| \,.\, Y$
$2^{2n} - |X| Y$
Para corregir el resultado tenemos dos posibilidades

a) Proceder como se vio, sin cambio (normalmente), y corregir al finalizar
$P + 2^{2n} - 2^n Y = 2^n Y - |X| Y + 2^{2n} - 2^n Y = 2^{2n} - |X| Y$

b) Asumir X de doble extensión
$P = (2^{2n} - |X|)\, Y = 2^n Y - |X| Y =$ ~~$2^{2n} (Y-1)$~~ $+ 2^{2n} - |X| Y$
Dado que en 2's complemento el carry se descarta
Observar que esto no implica A y ALU de 2n bits, la extensión es virtual. Se implementa trabajando con el bit que ingresa a AMQ (por izquierda) que no será el de carry. Sino:

{ 0 hasta la primer suma
Signo de X de allí al final

Caso B

$Y < 0$ $\quad P = (2^n - |Y|)\, X = 2^n X - |Y| X$
$X \geq 0$

a) Sumar $2^{2n} - 2^n |X|$
b) Asumir Y de doble extensión (2n bits)
La opción b implica el doble de iteraciones. Luego claramente conviene la opción a
Luego el resultado correcto es $2^{2n} - |Y| X$

Recodificando el multiplicador

Durante la implementación y con objeto de acelerar el proceso de multiplicación se puede recodificar de a más de un bit del multiplicador (Y). Si recodifico de a c bits tendré n/c iteraciones y deberé disponer de 2^c múltiplos de X.
La recodificación de Booth propone trabajar en digito signado para el multiplicador.
Para justificar esta operación consideremos una subcadena de 1´s en el multiplicador de la forma (*X* es 0 o 1):

```
Y          XXXX011110XX      (bits del multiplicador)
               ^    ^^
               j    i(i-1)   (posición)
```

La contribución al producto de este String de 1's es:
$x*(2^i+2^{i+1}+ \ldots + 2^{j-1}) = x*(2^j-2^i)$

Reemplazo el string de 1's por una resta al principio y una suma al final. Observar que en 2's complemento la resta se resuelve tan fácil como la suma. Se necesita un flag para controlar si sumar, restar o saltear.

Reestructurando el algoritmo:

```
1 pj=0,i≤j≤2n; flag = 0;
2 for i=1 to n
        if (yi=1 y flag=0)o(yi=0 y flag=1)
3               if yi=0
                        p=p+x
                        flag=0
                else
                        p=p-x
                        flag=1
                end
        end
  end
```

Es de observar que en la operación producto si trabajamos de forma "asincrónica" se podría tomar ventaja de las posibilidades que da el multiplicador a nivel de

a) Shift Over Cero's (*correr sobre 0´s.*)

b) Skip Over Cero's (*saltar sobre 0´s.*)

Skip Over Cero's asegura:

- Poder detectar una cadena de 0's
- Disponer de una unidad de corrimiento variable

En operación sincrónica, ¿Cuál es el aporte de Booth?

1. Resuelve de forma automática el caso y < 0 trabajando en 2's complemento. Esto resulta del hecho de que el bit signo es 1 y la extensión de signo virtual es como una cadena de 1´s. Bastara que en el proceso de análisis del multiplicador (codificado según Booth) no vea la finalización del String de 1's para que al no hacer la suma respectiva estaré efectivamente restando 2^nx que es lo mismo.

El peor caso es no saltear sobre 0´s cuando se implementa la versión original del algoritmo. Cuando el multiplicador es la forma 0101...0101 y saltamos sobre 0´s y 1´s, aún tenemos que realizar *n* operaciones. Esto es porque a veces el decodificador realmente revisará si es un solo 1 rodeado de 0´s para realizar solo una suma en lugar de una suma y una resta. Si podemos saltar sobre 0´s en cualquier longitud, se demuestra que los desplazamientos promedio son de un largo de 2. Si pueden saltarse 0´s y 1´s, entonces el promedio sube a 3. Pero la circuitería necesaria para implementar saltos variables de cualquier longitud puede ser muy engorrosa.

Una alternativa a saltar sobre 0´s y 1´s es recodificar *c* bits del multiplicador a la vez y tener $2^c - 1$ múltiplos del multiplicando antes de comenzar la multiplicación. Por ejemplo, con $c = 2$, tendríamos x, $2x$ y $3x$. Si los bits y_i e y_{i+1} del multiplicador son 00, se realiza un desplazamiento doble a derecha. Si son 01, entonces se suma x antes del desplazamiento. Si son 10, sumaremos $2x$ y si son 11 $3x$. En el paso 2 del algoritmo original, i es incrementado en 2. Hay dos esquemas, mirar al futuro o soportarse en el pasado.

Multiplicador			*Producto agregado*	*Explicación*
y_{i+2}	y_{i+1}	y_i		
0	0	0	0	Sin cadena
0	0	1	+2	Fin de cadena de 1´s
0	1	0	+2	Un solo 1
0	1	1	+4	Fin de cadena
1	0	0	-4	Comienzo de cadena
1	0	1	-2	+2 por final y –4 por comienzo
1	1	0	-2	Comienzo de cadena
1	1	1	0	Cadena de 1´s

Recodificación del multiplicador de 2 bits con salto

Este esquema de recodificación puede implementarse con saltos sobre 0´s y 1´s. La Tabla 1 muestra cómo esto puede realizarse simultáneamente para 2 bits con la generación de solo $2x$ y $4x$, o sea, solo desplazando el multiplicando a izquierda. Para entender la tabla debemos poner atención al hecho de que los bits y_i e y_{i+1} son los recodificados, mientras que el bit y_{i+2} se usa para indicar la terminación posible de una cadena. Además se asume que hay dos bits 00 extras implícitos en el final menos significativo y otro en el extremo izquierdo. Por ejemplo, un multiplicador de 6 bits (011001) = 25 es recodificado como:

```
0  01 10 01 00
^           ^
```

bit extra bits extras

$-4.2^{-2}.x$

$2.2^{0}.x$

$-2.2^{2}.x$

$2.2^{4}.x$

que es (32-8+2-1).x = 25x

Usando CSA

En realidad no es un sumador paralelo. Si tenemos operandos de n bits un CSA se constituye con n F.A desconectados entre sí. No hay procesamiento de carry alguno.
La entrada C_{in} de los F.A. se destina a un tercer operando, luego al CSA entran 3 operandos y entrega dos resultados:

- Pseudosuma
- Pseudocarry

La verdadera suma se alcanza si hacemos la suma de estas dos sumas.

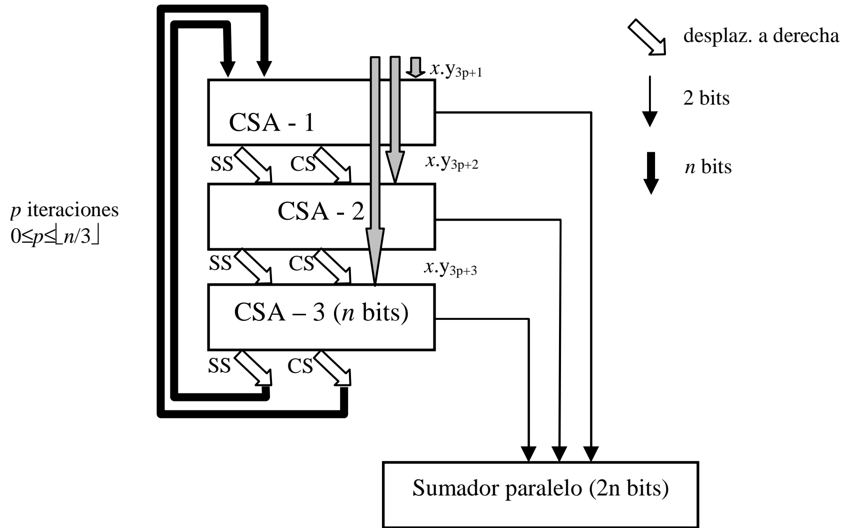
Ejemplo

```
X  = 0 0 1 1 0
Y  = 0 1 1 0 0
Z  = 0 1 0 1 1
SS = 0 0 0 0 1
SC = 0 1 1 1 0
     1 1 1 0 1
```

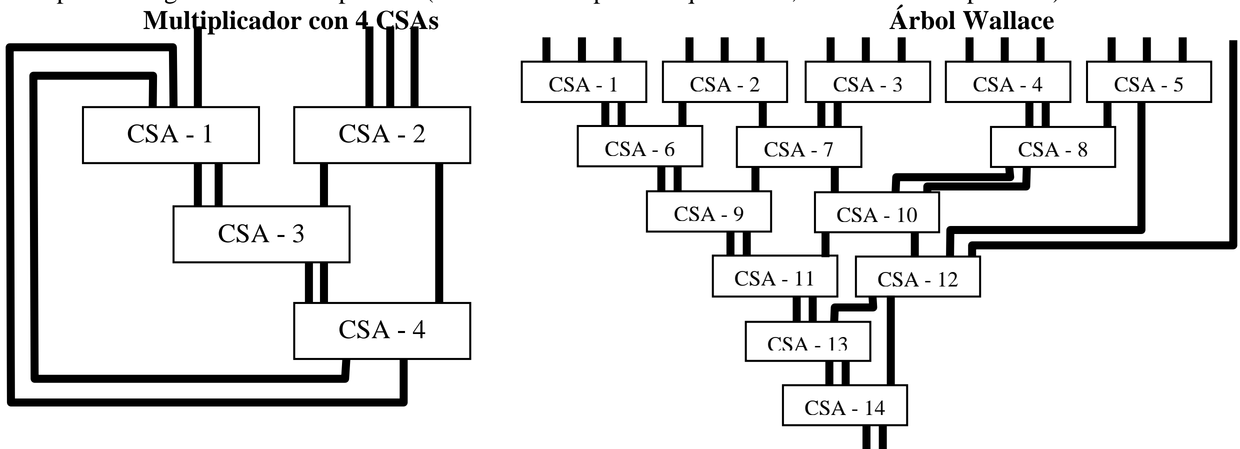
El tiempo de ejecución es mínimo, corresponde a un F.A. (ni carry anticipado, ni ripple, el C_{in} es un operando).
Si hay que sumar tres o más operandos resultara ventajoso el empleo de estos "sumadores", ejemplo de esto el producto. Para tal fin, podemos usar el siguiente algoritmo:

1. $ss_j = 0$, $cs_j = 0$, $1 \leq j \leq 2n$
2. **for** i=1 **to** n
   if $y_i \neq 0$
   for k=0 **to** (n-1)
3. $ss'_{i+k} = ss_{i+k} \oplus cs_{i+k} \oplus x_i$
   $cs'_{i+k+1} = ss_{i+k} * cs_{i+k} + (ss_{i+k} \oplus cs_{i+k}) * x_i$
   $ss_{i+k} = ss'_{i+k}$
   $cs_{i+k+1} = cs'_{i+k+1}$
   end
4. Sumar ss y cs en un sumador convencional de 2n bits.

Como los CSAs son mucho menos complejos que los sumadores paralelos, podemos usar más de uno sin un incremento indebido en el costo. En la figura se muestra cómo usar los CSAs



En una fase de reloj las operaciones de cada CSA, menos la última suma ripple, toma *n* niveles de CSA. No parece haber ventajas entre colocar 1 o 3 CSAs. Pero si permitimos superponer las operaciones, por ejemplo, realizar la recodificación del multiplicador en paralelo con la acumulación de productos parciales, el duplicar hardware implica una gran mejora. Además, si incrementamos el número de CSAs, podemos realizar concurrentemente partes del proceso de multiplicación. En la figura, mostramos cómo la inclusión de un CSA más permite algunas sumas en paralelo(solo una descripción esquemática, sin el sumador paralelo).



El número de iteraciones se decrementa de $n/3$ a $n/4$ para un tiempo total de $\frac{3}{4}n$ niveles de CSAs. Más generalmente, si tenemos (n-2) CSAs podemos construir un árbol de profundidad de O(log n), o sea, genera la suma de las sumas ripple en O(log n) niveles de CSAs. Este orden de magnitud se obtiene de la siguiente construcción. Agrupamos de a n operandos en tríos como entradas a los CSA superiores. Se obtienen (aproximadamente) $2/3.n$ salidas, las cuales alimentan de nuevo en tríos al segundo nivel. Continuando la construcción hasta tener solo 2 salidas. Evidentemente tenemos profundidad logarítmica, ya que decrementamos el número de entradas un tercio cada nivel. Se necesitan (n-2) CSAs, y se prueba por inducción sobre el número de operandos. Esto se llama Wallace Tree o sumador Wallace.

En general, no deseamos invertir en (n-2) CSAs. Pero solo un número limitado de ellos pueden ser conectados como en la figura y emplear un algoritmo iterativo. Recodificar el multiplicador puede usarse concurrentemente, con dos desplazamientos (o más) en lugar de uno.

La suma es computada por el árbol de CSAs, el cual produce una suma de $2n$ bits y un carry de $2n$ bits. La absorción final del carry es elaborada por un sumador paralelo con propagación interna de carry.

El multiplicador estrictamente combinacional de la figura es práctico para valores moderador de n, dependiendo del nivel de integración que se use. Para valores de n grandes, el número de CSAs necesarios se vuelve excesivo. Las técnicas de CSAs pueden seguir usándose sí particionamos el multiplicador en k segmentos de m bits. Se repite el proceso k veces y se va acumulando el resultado. El resultado se obtiene por lo tanto luego de k iteraciones. La multiplicación carry-save es apropiada para implementaciones pipelined.

VLSI

La alternativa para alcanzar profundidad logarítmica en n (log n) es el árbol binario del producto.
Se necesita ya no un sumador (3,2) como el CSA sino uno (2,1) entran dos bit suma y sale un bit.
¿Y el carry? La notación digito signado hace posible la suma sin carry. El precio que se paga, como se verá, es:

a) Se necesita un doble registro para almacenar (1, 0, -1)

b) La lógica de suma es más compleja que la del CSA, en el cómputo de S_i en una dada posición intervienen no solo los bits de la posición i sino que además los de la posición i-1 o i-2.

Esto último, no obstante, es de mucho menor alcance que en el caso de una suma convencional (con carry) donde se debe considerar desde la posición i hasta la inicial 0.

Desarrollamos el algoritmo en 2 pasos:

a) En un primer paso calculamos los S_i y C_{i+1} para que en un paso siguiente, en el que se computa S_i + C_i (resultante del anterior) no se genere carry en las diversas posiciones. Según tabla se calculan los S_i y C_{i+1} y luego se acomete la suma entre S_i y C_i, como sigue:

X_i → S_i
Y_i C_i
S_i S_i

Se tendrá la siguiente tabla para $X_i + Y_i$

X_i		1		1		not(1)		0		1		not(1)
Y_i	+	1	+	not(1)	+	not(1)	+	0	+	0	+	0
	1	0	0	0	-1	0	0	0	1	not(1) $(X_{i-1}\geq 0)$e$(Y_{i-1}\geq 0)$	0	not(1) $(X_{i-1}\geq 0)$e$(Y_{i-1}\geq 0)$
									0	1 caso contrario	not(1)	1 caso contrario
	C	S	C	S	C	S	C	S	C	S	C	S

Ejemplo

X_i 1 not(1) 0   2
Y_i +   0 0 1   1
R   3

0 + 1 = 1 not(1) $(X_{i-1}, Y_{i-1} \geq 0)$
not(1) + 0 = 0 not(1) $(X_{i-1}, Y_{i-1} \geq 0)$
1 + 0 = 0 1 $(X_{i-1} < 0, Y_{i-1} \geq 0)$

$S_i + C_i$

1 not(1)
0 not(1)
0 1

---

0 1 0 not(1)

Para mostrarlo
100
-
001
011

S_{i+1}
X_i
$\underline{Y_i}$
C_i, el cual es función de los X_{i-1} e Y_{i-1} y de los X_{i-2} e Y_{i-2}.
Se ve que la lógica a desarrollar toma en cuenta para cada posición los dígitos de la misma y los de las dos posiciones inmediatamente precedentes.

División de enteros no signados

En la división para un dividendo X de 2n bits y un divisor Y de n bits se tendrán dos resultados de salida; un cociente de n+1 dígitos y un resto de n dígitos tales que

$$x = y.q + r,\ \ 0 \leq r < y$$

¿Por qué n+1? Recordar que el producto de dos números de n bits podrá dar un resultado de 2n bits o 2n-1 bits. Para este último caso si a uno de los multiplicando o multiplicador lo multiplico por 2 (Paso a n+1 bit). El resultado del producto en tal caso será de 2n bits. Luego si a este lo someto al cociente por el otro factor es claro que el resultado es n+1 bit.

El método normal de papel y lápiz, asumiendo un sistema de números posicional de base *b*, necesita de un trabajo de suposición. Al primer dígito de *q*, q_n, se le da un valor; se calcula el producto $y.q_n$ y se lo resta a *x*. Si esta operación produce un valor negativo, q_n era muy grande y se intenta con otro valor (uno más chico). Si el resultado fuese positivo pero más grande (o igual) que *y*, entonces q_n era muy chico y se necesita un valor más grande. El dígito correcto q_n es tal que:

$$0 \leq x - y.q_n.b_n < y.b_n$$

En este proceso de prueba y error voy ajustando el digito del cociente. Una vez que q_n fue generado, el proceso continua con q_{n-1} reemplazando *x* por $x - y.q_n$ y así hasta obtener q_0. El último dividendo parcial es el resto *r*.
En la base 2 la división es trivial; probamos si esta contenido. Si esta contenido, q_i=1, sino simplemente q_i=0 y sigo adelante con la determinación de q_{i-1}.
La división <u>con restoring</u> plantea hacer la resta si ≥ 0, q_i=1 y el resultado es el dividendo para el próximo paso. Si es < 0 (FALLÉ), q_i=0 y restauro el dividendo para el próximo paso.

Sean
$x = (x_{n-1}, \ldots, x_0)$
$y = (y_{n-1}, \ldots, y_0)$
$q = (q_{n-1}, \ldots, q_0)$
$r = (r_{n-1}, \ldots, r_0)$

1. Expandir *x* en $x' = (x_{2n-2},\ldots,x_{n-1},\ldots,x_0)$, con $x_i = 0$, $n \leq i \leq 2n$, A 0 (extensión de signo).
   Cabe aclarar que el hardware está preparado para 2n bits

	x
00000	

2. **for** *i*=1 **step** 1**until** *n*
   set $z = x' - 2^{n-i}.y$
   if $z \geq 0$
   $q_{n-i} = 1$ **and**
   $x' = z$
   else
   $q_{n-i} = 0$ **and not modify** *x'*
3. $r = x'$

¿Cómo se ganan posiciones menos significativas? Con los corrimientos a izquierda; comienzo por la más significativa y voy corriendo hasta llegar a la posición 0 (Donde calculo el q_0)
El "else" del paso 2 es engañosamente simple. Si la resta resulta negativa, tenemos que sumar para restaurar *x´*. Esto puede evitarse usando una técnica sin restoring. Aunque este último esquema no está limitado al sistema binario, nos restringimos a lo que pasa en él porque es el caso más importante (y para más claridad).

Como dijimos antes las únicas dos alternativas en el sistema binario son 0 y 1. Volviendo al paso 2, cuando z es negativo(cuando suponemos mal), tenemos:

$$x' - 2^{n-i}.y \leq 0$$

Luego de la resta, y asumiendo que $i \neq n$, sumamos $2^{n-i}.y$ y restamos $2^{n-i-1}.y$, o sea, realizamos:

$$z_1 = x' - 2^{n-i}.y + 2^{n-i}.y - 2^{n-i-1}.y, \text{ o}$$

$$z_1 = x' - 2^{n-i-1}.y$$

Si en su lugar no restauramos z y sumamos $2^{n-i-1}.y$ cuando z es negativo, obtenemos:

$$z_2 = x' - 2^{n-i}.y + 2^{n-i-1}.y, \text{ o}$$

$$z_2 = x' - 2^{n-i-1}.y = z_1$$

La única dificultad de este método es cuando el último bit el cociente provoca un resto negativo. Esto sucede si tenemos, para algún j:

$$r' = x' - 2^{j+1}.y + (2^j + 2^{j-1} + ... + 2^0).y < 0$$

El resto correcto debe haber sido el resto parcial generado al mismo tiempo como el bit $j^{\underline{mo}}$ del cociente, que es:

$$r = r' + y$$

De esta manera el algoritmo SIN RESTORING para números binarios POSITIVOS queda:

1. Expandir x en $x' = (x_{2n-2},...,x_n,x_{n-1},...,x_0)$, con $x_i = 0$, $n \leq i \leq 2n-2$, (extensión de signo)
   signo = 1
2. **for** i=1 **to** n
   - $z = x'$ - signo.$2^{n-i}.y$
   - **if** $z \geq 0$
     - $q_{n+1-i} = 1$
     - signo = 1
   - **else**
     - $q_{n+1-i} = 0$
     - signo = -1
   - $x' = z$
3. **if** $q_0 = 1$
   - $r = x'$

   else
   - $r = x' + y$

#Txt Ref = Estudiar apunte División Rápida SRT brindado por la cátedra

División con multiplicación repetida

En los sistemas conteniendo multiplicadores de alta velocidad, la división puede realizarse eficientemente y a bajo costo utilizando repetidas multiplicaciones. En cada iteración, se genera un factor F_i y se usa para multiplicar el divisor V y el dividendo D. Es elegido de manera que la secuencia $V \times F_0 \times F_1 \times F_2$ ... converja rápidamente a 1. Esto implica que $D \times F_0 \times F_1 \times F_2$ ... converge hacia el cociente buscado Q, ya que:

$$Q = \frac{D \times F_0 \times F_1 \times F_2 \cdots}{V \times F_0 \times F_1 \times F_2 \cdots}$$

Si el denominador converge hacia 1, el numerador lo hace hacia Q.
La convergencia del método depende de la elección de los F_is. Para simplificar, asumamos que D y V son fracciones positivas normalizadas tales que $V = 1 - x$, donde $x < 1$. Poner $F_0 = 1 + x$. Ahora podemos escribir:

$$V \times F_0 = (1 - x)(1 + x) = 1 - x^2$$

Claramente $V \times F_0$ está más cerca de 1 que V. Después poner $F_1 = 1 + x^2$. Luego:

$$V \times F_0 \times F_1 = (1 - x^2)(1 + x^2) = 1 - x^4$$

y así sucesivamente. Vi denota $V \times F_0 \times F_1 \times \ldots \times F_i$. El factor de multiplicación en cada etapa se computa como sigue:

$$F_i = 2 - V_{i-1}$$

lo cual es simplemente el 2 complemento de V_{i-1}. Sigue $F_i = 1 + x^2$ y $V_i = 1 - x^{2^{i+1}}$. A medida que i aumenta, V_i converge rápidamente a 1. El proceso termina cuando $V_i = 0.11 \ldots 11$, el número más cercano a 1 para el tamaño de palabra dado.

Capítulo 3 – Pipeline

Introducción

La alternativa para reducir el tiempo empleado en una dada operación (tarea) es concurrencia. Para ello tenemos dos alternativas:

- Paralelismo: duplicar recursos para realizar actividades simultáneas e independientes.
- Pipelining u Overlap: implica dividir una tarea en subtareas, dedicando un hardware específico a cada una de ellas, etapa o segmento del pipeline y alcanzar una ejecución "solapada" de distintas tareas cada una en distinta fase de ejecución.

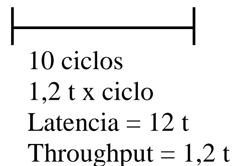
Este concepto de pipelining es asimilable a una línea de montaje.

El objetivo del pipe es throughput (cantidad de trabajo por cantidad de tiempo). El throughput del pipeline es determinado por la frecuencia con la que una instrucción sale del pipeline. El tiempo requerido para mover una instrucción de una etapa a otra se llama ciclo de máquina. Con frecuencia un ciclo máquina es un ciclo de reloj (a veces más).

Observar que en general funciona de manera antagónica con otro concepto que es latencia. (tiempo que media entre el inicio y la culminación de una tarea).

La operación del pipe es sincrónica. Debe observarse que se avanza a la velocidad de la etapa más lenta.

Ejemplo



Otra cuestión a remarcar es que independientemente de la profundidad del pipe (numero de etapas o segmentos) en régimen se obtendrá un resultado por ciclo.

Requisitos para organizar un pipeline

a) Dividir en subtareas una tarea de forma tal que la ejecución secuencial de las mismas resulte equivalente.

b) Disponer para cada subtarea de hardware específico, conocido como etapa o segmento del pipe (Para posibilitar overlap)

c) Solo hay intercambio de información entre la salida de una etapa y la entrada de la siguiente (en secuencia). No intercambian información interna.

d) Cada etapa dura aproximadamente lo mismo. Se avanza a la velocidad de la etapa más lenta por lo que se puede constituir en un cuello de botella afectando la performance.

e) A las etapas hay que aislarlas entre sí para posibilitar el avance discontinuo de la información. Para ello si no tienen memoria propia habría que incorporar buffers inter-etapa.

Tiempo de ejecución de n tareas

$T = (k - 1)t + t\,n$

T es el periodo, tiempo de la etapa más lenta

¿Cuál sería el throughput? 1/t

SpeedUp (k etapas, n operaciones)

La performance ideal en un pipeline se mide por la ganancia de velocidad contra una máquina sin pipeline:

$$S_k = \frac{\text{Promedio de Tiempo de instrucciones sin pipeline}}{\text{Promedio de Tiempo de instrucciones con pipeline}} = \frac{T_1}{T_k} = \frac{n \cdot k \cdot t}{((k-1)+n)t} = \frac{n \cdot k}{k-1+n}$$

Está claro que si $n >> k$ entonces $S_k \to k$

Esquema de los datos que fluyen en el pipe

S4				A	B	…
S3			A	B	C	…
S2		A	B	C	D	…
S1	A	B	C	D	E	…

Clasificación del pipeline

De acuerdo al uso:

- ***Datos individuales o Pipeline Aritmético.*** Estructura las unidades funcionales (ALU) en pipeline, con lo cual solapamos el procesamiento sobre distintos datos. Las unidades lógicas pueden ser segmentadas para operaciones de pipeline.
- ***Pipeline de Instrucciones***. Estructura el CPU en pipeline. Se divide el procesamiento de una instrucción en fases, se asigna hardware específico a cada fase y luego se acomete el procesamiento concurrente de múltiples instrucciones. En un comienzo se lo refirió como "look-ahead de instrucciones". Este tipo de pipelining busca explotar el paralelismo a nivel de instrucciones (I.L.P.)
- ***Procesos completos***. Aplica pipelining a nivel de múltiples procesadores, cada uno a cargo de un determinado proceso y los resultados de uno serán consumidos por el siguiente en secuencia.

Según las configuraciones y las estrategias de control tenemos 3 clasificaciones:

- ***Unifuncional vs. Multifuncional***. El unifuncional siempre es estático ya que posee una función "fijada".
  Los multifuncionales pueden realizar diferentes tareas, bien sea simultáneamente o en distintos tiempos. Obviamente las tareas se soportan variando la interconexión entre etapas.
- ***Estáticos vs. Dinámicos*** (configuración). Los estáticos realizan una función y admiten una configuración por vez. Podrían ser unifuncionales o multifuncionales en cuyo caso una sola configuración por vez.
  Lo podemos referir como multifuncional estáticamente configurado. Esta alternativa será provechosa en tanto los cambios de configuración sean poco frecuentes. El pipelining dinámico permite que varias configuraciones existan simultáneamente. Claramente debe ser multifuncional. Sin embargo este requiere de un control más sofisticado que los static.
- ***Escalar vs. Vectorial***. Depende del tipo de instrucciones, o del tipo de datos.
  - Escalares. Procesa una secuencia de operandos escalares bajo el control del software (do, loop).
  - Vectorial. Tiene instrucciones para operar con vectores, luego la sucesión de operaciones sobre los componentes escalares de los vectores se manejan a través de hardware o firmware. Deberá ser configurado para cada operación, aunque sean las mismas, dado que se los instruye además de la operación, respecto a la ubicación de los vectores, dimensión, etc. Ej. Pipeline multifuncional estáticamente configurado. Por caso, un procesador vectorial tiene un ISA que entiende operaciones vectoriales.

Secuenciamiento del pipeline
Pipeline Estáticos

En los pipelines lineales, se sigue una secuencia y cada etapa a lo sumo se visita una vez. En otras palabras no hay feedback ni feedfoward. Luego el Secuenciamiento es trivial. Lo único que hay que contemplar para su temporizado es el tiempo de la etapa más lenta.
En cada nuevo ciclo podría realizar una nueva tarea. Ese no es el caso de un pipe no lineal (en don de que alguna o algunas etapas podrán usarse más de un ciclo).
La tarea es alcanzar regímenes óptimos libres de colisiones.

El primer paso es construir la tabla de reservación. La misma constituye un método para el secuenciamiento óptimo de un pipeline estáticamente configurado.

- Tantas filas como etapas.
- Tantas columnas como ciclos de reloj consuma una operación.
- Tendrá una marca para indicar la ocupación de una dada etapa en un determinado ciclo.

Ejemplo

	1	*2*	*3*	*4*	*5*	*6*
S1	X				X	
S2		X				X
S3			X			
S4				X		

La posibilidad de colisionar en una etapa la da la presencia de múltiples cruces en una fila.
Si hubiera múltiples cruces en una columna en ningún caso indica posible colisión, solo que en ese ciclo el dato hace uso de más de una etapa.
¿Cuándo habrá colisión?
Si dos iniciaciones entran separadas en ciclos en lo mismo que separa a dos cruces en una fila en algún momento se producen colisiones.
Esas latencias (intervalo entre iniciaciones) son prohibidas o criticas.
Con ellas construimos un vector de colisiones C con una cardinalidad dada por la mayor latencia crítica.
$C = (C_n, C_{n-1}, \ldots, C_1)$
Donde
$C_i = 1$ si prohibida
$C_i = 0$ en otro caso.
En nuestro ejemplo: C = (1,0,0,0 [la 1])
Con este vector construimos un diagrama de inicializaciones exhaustivo que exponga todos los ciclos posibles "libres de colision".
La idea de ciclo es un camino cerrado que recorra una serie de estados.
El ciclo que recorrimos se caracteriza porque cada nueva iniciación es con la latencia mínima permitida, en particular, se lo conoce como Greedy.
No por ser Greedy asegura la menor latencia promedio. Un ciclo Greedy no podrá tener una latencia mayor que el numero de 1's en el vector de colisión + 1.

Pipeline de instrucciones

Busca un aprovecho del paralelismo a nivel de instrucciones, ILP por sus siglas en inglés.

Pipeline de instrucciones
Procesador DLX

Refleja el estado de las arquitecturas RISC al principio de los '90. Fue desarrollado por Hennesey y Patterson.
Características:

- Instrucciones de tamaño fijo (32 bits).
- Formato fijo.
- RAR (registro a registro), solo load/store con memoria.

RISC proviene de reduced instrution set computer. Es reducido a nivel de complejidad no de número de instrucciones. RISC favorece el desarrollo de organizaciones del CPU más eficientes, esto es, proporcionan ventajas en orden a reducir CPI (ciclos por instrucciones). Complementariamente, IPC proviene de instrucciones por ciclo. Además facilitan las mejoras a nivel de "microarquitectura". Se buscaba reducción del CPI o aumento del IPC.

Formato de Instrucciones DLX

I-TYPE

6	5	5	16
OPCODE	Rs1	Rd	INMEDIATO

Se codifican LOAD y STORE
Todas las operaciones inmediato (Rd ← Rs1 op inmediato

R-TYPE

6	5	5	5	11
OPCODE	Rs1	Rs2	Rd	FUNCION

ALU Operation Rd ← Rs1 op Rs2

J-TYPE

6	26
OPCODE	OFFSET (SUMA AL PC)

Jump and Jump and link

Data Path

Es la parte del CPU en donde se almacenan, se procesan, se comunican los datos dentro del mismo.
Es el responsable de la performance última alcanzable.
Por otro lado, el control path es responsable de organizar la secuencia de señales de control que activando sobre el data path posibilitan la ejecución de las distintas instrucciones. Debe asegurar la "correctitud" de las mismas en algún sentido es "la parte inteligente del CPU."

Pipeline de Instrucciones para DLX

Asumamos que el procesamiento de cada instrucción requiere los siguientes 5 pasos.

1. ***Instrucción Fetch***. Se envía el PC y se trae la instrucción de memoria poniéndola en el IR (registro de instrucción).
   MAR ← PC; IR ← M[MDR]
2. ***Instrucción Decode/Register Fetch***. Se decodifica la instrucción y accede al register file para leer los registros (operandos). También se incrementa el PC para apuntar a la próxima instrucción. La decodificación puede ser hecha en paralelo junto a la lectura de los registros. Esto puede lograrse gracias al formato fijo de las instrucciones.
   A ← Rs1; B ← Rs2; PC ← PC + 4 (avanzo 32 bits, 4 bytes)
3. ***Ejecución / Cálculo dirección efectiva***. La ALU opera con los operandos de la etapa anterior realizando una de las siguientes tres funciones:
   - Referencia a memoria. La ALU suma los operandos para formar la dirección efectiva y el MDR es cargado si es un store.
     MAR ← A + $(IR_{16})^{16}$ ## $IR_{16..31}$ campo inmediato con extensión de signo.
     Si es un store MDR ← B
   - ALU instrucción. La ALU realiza la operación especificada por el opcode.
     ALU OUTPUT (C) ← A op (B o $(IR_{16})^{16}$ ## $IR_{16..31}$)
   - Branch/Jump. La ALU suma el PC con el valor inmediato para computar la dirección del target del jump. Si es un branch, un registro, que ha sido leído en la etapa anterior, es chequeado para decidir si la dirección debe ser insertada en el PC.
     Si es un jump:
     ALU OUTPUT (C) ← PC + $(IR_{16})^{16}$ ## $IR_{16..31}$
     Si es un branch (salto condicional)
     COND ← (A op 0)
4. ***Acceso a Memoria/Completado del branch***. Realiza una de las siguientes operaciones:
   - Referencia a Memoria. Si la instrucción es un load, los datos son retornados de memoria. Si un store los datos son escritos en memoria.
     MDR ← M[MAR]   Si LOAD
     M[MAR] ← MDR   Si STORE
   - Branch. Si es un branch, el PC es reemplazado por la dirección destino del branch.
     if(COND) {   PC ← ALU OUTPUT(C)};
5. ***Write Back***. Escribe el resultado en el register file, ya sea que venga de memoria o de la ALU.
   RD ← ALU OUTPUT (C) O MDR

Para reducir el tiempo de fetch se pueda utilizar un buffer para realizar un look-ahead de instrucciones.
De forma independiente de la actividad del CPU (ejecutando) ir a memoria y buscar las instrucciones por adelantado. Así, procesadores elementales como 8086 tenían un buffer de 6 bytes y el 8088 de 8 bytes.
Mejor aún es un buffer con look-ahead y con look-behind.

Este concepto resulta "sublimado" con la utilización de la memoria cache.

DECODE

En esta etapa se realiza la búsqueda de operandos. Se acelera si los operandos los manejamos en registros. El acceso a registros es en menor tiempo que a memoria por dos factores:
a) No existe cálculo de dirección efectiva
b) Aunque se emplee cache (memoria rápida, tanto para instrucciones como para datos) y a pesar de usar la misma tecnología que los registros, estos últimos son más rápidos porque son de mucho menor tamaño.

EXECUTE

Aquí se busca la mejora del algoritmo y de la implementación. Interesa favorecer el uso de unidades funcionales dedicadas (no multipropósito). Además al tener diferentes unidades podre hacer un uso concurrente de estas, ganando así velocidad.
El paso siguiente, en orden a mejorar el tiempo de ejecución de los programas es Pipelining.
En una situación ideal el tiempo de ejecución por instrucción seria:

$$\frac{\text{Tiempo por instrucción en una maquina } \textbf{no} \text{ pipelined}}{\text{K(n° de etapas del pipeline)}}$$

Este límite no será alcanzable en general por dos cuestiones, a saber:
a) la dificultad en alcanzar tiempos iguales por etapa
b) los latch interetapa

Ejemplo

El DLX estructurado en 5 etapas:

IF	inst i	IF	ID	EX	MEM	WB				
ID	i+1		IF	ID	EX	MEM	WB			
EX	i+2			IF	ID	EX	MEM	WB		
MEM	i+3				IF	ID	EX	MEM	WB	
WB						IF	ID	EX	MEM	WB
		1	2	3	4	5	6	7	8	9

Ejemplo

El tiempo seria (en orden) (IF, ID, EX, MEM, WB) [50 ns, 50 ns, 60 ns, 50 ns, 50 ns] y 5 ns latch interetapa.
Tiempo por instrucciones sin pipe = 50 + 50 + 60 + 50 + 50 = 260 ns
Para el pipe, el tiempo de ciclo seria: 60 + 5 = etapa más lenta + latch = 65

$$\text{SpeedUp} = \frac{\text{Tiempo de ejecución sin pipe}}{\text{Tiempo de ejecución con pipe}} = \frac{260}{65} = 4 \text{ veces (no 5)}$$

Requerimientos básicos para estructurar un CPU con pipeline:

a) Toda vez que una etapa no cuente con su memoria (local) deberá introducir buffer a la entrada
b) Todo recurso que es usado en más de una etapa o bien se modifica o alternativamente se duplica (permite solapamiento)
c) Todo dato que es usado más adelante en el pipe (más allá de la etapa siguiente) deberá ser convenientemente buffereado.
d) En cuanto a la profundidad del pipe, esto no podrá ser arbitrariamente profundo, habida cuenta que para asegurar la fluidez hay ciertas operaciones que tienen que ser atómicas, esto es, resolverse en un ciclo de reloj. Por caso, la operación básica de suma (resta).

A continuación indico los eventos en cada etapa del pipe

STAGE	*ALU INST.*	*LOAD OR STORE INST*	*BRANCH INST*
IF	IR ← M[PC] PC ← PC + 4	IR ← M[PC] PC ← PC + 4	IR ← M[PC] PC ← PC + 4
ID	A ← Rs1 B ← Rs2 PC1 ← PC IR1 ← IR	A ← Rs1 B ← Rs2 PC1 ← PC IR1 ← IR	A ← Rs1 B ← Rs2 PC1 ← PC IR1 ← IR
EX	ALU OUTPUT (C) ← A op B ALU OUTPUT (C) ← A op $(IR_{16})^{16}$ ## $IR_{16..31}$ Rd2 ← Rd1	MAR ← A + $(IR_{16})^{16}$ ## $IR_{16..31}$ SMDR ← B	ALU OUTPUT1 ← PC1 + $(IR_{16})^{16}$ ## $IR_{16..31}$ COND ← Rs1 op 0
MEM	ALU OUTPUT1 ← ALU OUTPUT Rd3 ← Rd2*	LMDR ← M[MAR]; OR M[MAR] ← SMDR	IF(COND) PC ← ALU OUTPUT (C)

WB	Rd3 ← ALU OUTPUT1	Rd3 ← LMDR	

Al MDR lo debo duplicar, tendré LMDR (para load) y SMDR (para store).
El campo que identifica el registro destino, según vimos con los formatos de instrucción podrá ocupar 2 lugares distintos IR_{20-16} ó IR_{16-11}
Asumo que el banco de registros tiene 2 pórticos de lectura y un pórtico de escritura.

Sin pipe 2 accesos a memoria cada 5 ciclos
Con pipe 2 acceso a memoria por ciclo
Con cache independiente busco instrucciones y datos en el mismo ciclo

Pipeline Hazzard

Hay situaciones que podrán apartar al pipe de la performance ideal (un IPC=1 o un CPI=1)
Estas situaciones se conocen como "Hazzards", riesgos, los cuales en general se detectan en la etapa decode.
Los hazards "provienen", son atributos del código (programa). Estos podrán o no traducirse en perdida de performance de la organización (o microarquitectura).
Hay tres tipos de hazzards:

- **Estructural**: Se produce si alguna combinación de instrucciones no puede aceptarse debido a conflicto de recursos, o sea, una instrucción en la etapa decode requiere recursos que están siendo afectados por instrucciones previas en el pipeline. El HW no puede soportar todas las posibles configuraciones de instrucciones.
- **Datos**: Se da si no hay independencia de rango-dominio entre las instrucciones. En etapa decode e instrucciones previas que aun no finalizaron su ejecución. La idea es que el resultado de la ejecución sea el mismo del que se alcanza con una simple ejecución secuencial.
- **Control:** Aparecen a partir de instrucciones que cambian el flujo secuencial de las mismas. Ejemplo de esto: Jump, Branch, Call, Return.

Si en la etapa decode se detecta un hazzard y este no está resuelto (a nivel de microarquitectura) el recurso de mínima (de hardware) es pipeline interlock.
Este frena al pipeline, no se sigue haciendo el fetch, y consecuentemente nada con las instrucciones siguientes, permitiendo que las previas en el pipeline avancen normalmente.
Recién cuando desaparezca el conflicto se libera el procesamiento de las nuevas instrucciones.
Esto es tan así porque la decodificación es en orden, lo cual es crítico que sea así.
Esto da lugar a ciclos "STALL". Estos ciclos perdidos son asimilables a "burbujas" que viajan en el pipe.

I_i	F	D	EX	M	WB				
I_{i+1}		F	D	STALL	STALL	EX	M	WB	
I_{i+2}			F	STALL	STALL	D	EX	M	WB

¿En que se traducen los ciclos stall's? En un apartamiento del CPI = 1

$$CPI_{c/pipe} = CPI_{ideal} + \text{ciclos stall por instruccion.}$$

$$\text{Speedup} = \frac{\text{Average instruction time sin pipe}}{\text{Average instruction time con pipe}} = \frac{CPI_{s/pipe} * \text{clock cycle s/pipe}}{CPI_{c/pipe} * \text{clock cycle c/pipe}}$$

$$CPI_{ideal} = \frac{CPI_{s/pipe}}{\text{Profundidad del pipe}}$$

Realizo sustitución:

$$\text{Speedup} = \frac{CPI_{s/pipe} * \text{clock cycle s/pipe}}{CPI_{ideal} + \text{ciclos stall por instrucción} * \text{clock cycle c/pipe}}$$

Simplificando, esto es, sin considerar lo relativo a periodo de reloj con y sin pipe:

Speedup = $\underline{CPI_{ideal} * \text{profundidad del pipe}}$

CPI_{ideal} + stall por instrucción

Resulta claro que sin stall's el SpeedUp está dado por el pipeline depth, es decir, el numero de etapas. Cabe aclarar que cuanto mayor sea la profundidad del pipe mayor podrá ser los ciclos stall por instrucción.

Hazard estructurales

En general una alternativa es "aumentar" el recurso en cuestión (como sería el caso del renombramiento de registros). En particular a nivel de unidad funcional se tienen tres alternativas para entender con esto (esto se podrá dar si la ejecución demanda más de un ciclo).

a) Agregar unidades funcionales que podrán operar en paralelo.

b) Pipeline de la unidad funcional que operan de forma solapada

Ambas alternativas son de aplicación si la frecuencia de estas operaciones es alta (evitando se constituyan en un cuello de botella)

c) D-virtuales. Sea asocia a la unidad funcional un buffer donde encolamos hasta un cierto número de instrucciones, donde despachamos instrucciones decodificadas con sus operandos que quedan aguardando que finalicen operaciones previas.

¿Qué logro? Evitar los stall's, evitar que el pipe se frene pues completo la decodificación y podre seguir con las instrucciones siguientes. Destrabar al decode. Luego esto da lugar a ejecución fuera de orden.

Hazard de datos

Se originan en la no independencia entre rango y dominio entre la instrucción a despachar y las instrucciones presentes en el pipe.

Casos:

O_i rango (salida)

O_j dominio (entrada)

$i<j$

Si se verifica que:

$O_i \cap O_j \neq 0$

$I_i \cap O_j \neq 0$

$O_i \cap I_j \neq 0$

$I_i \cap I_j \neq 0$ no da lugar a hazzard

Esto se puede dar tanto a nivel de registros como de locaciones de memoria. Son más fáciles de detectar, es inmediato (explicito) a nivel de registros. No así con locaciones de memoria, un mismo direccionado puede dar lugar a distintas direcciones, y distintos direccionados conducir a una misma dirección.

Luego la detección a nivel de registros es tarea que puede acometer tanto el hardware como el software.

En cambio a nivel de locaciones de memoria el software se verá muy limitado, no así el hardware. Por que podrá trabajar en la instancia ultima del acceso, cuando ya se conoce la dirección. Esto se conoce como memory desambiguation.

Enfocaremos el problema a nivel de registros, asumiendo que el acceso a memoria es el orden de las instrucciones por lo cual el conflicto no se daría a ese nivel.

¿Cómo se denotan los conflictos? Por el orden de acceso que se debe PRESERVAR. Pueden ser clasificados en uno de tres tipos dependiendo del orden de acceso a lectura y escritura en las instrucciones.

Dada una instrucción *j* a ser decodificada y una instrucción genérica *i*(previa) activa en el pipeline, se podrá provocar los siguientes hazzard de datos:

- *RAW (read after write)*: el conflicto será que *j* trata de leer su fuente antes de que *i* lo escriba. Es estrictamente procedural.
- *WAR (write after read)*: aquí *j* trata de escribir en un destino antes que este sea leído por *i*. Luego *i* obtendrá un valor incorrecto.
- *WAW*: aquí *j* trata de escribir en su destino antes de que este sea escrito por *i*. En ese contexto la escritura termina realizándose fuera de orden.

WAR y WAW son meros conflictos de nombre.

La solución para estos últimos es renombramiento, por software o por hardware.

Por software el compilador trabajara hasta agotar la posibilidad que le da el numero de registros lógicos.

En cambio, el renombramiento por hardware que se acomete en la etapa de decode, no estará limitado por los registros lógicos de la arquitectura sino por el banco de registros físicos de que disponga para tal efecto. Cabe aclarar que se dispone de un conjunto de registros físicos mucho mayor que el numero de registros lógicos.

Para observar cómo trabaja el renombramiento por hardware sean :

$I_1 = R_1 \leftarrow R_3 * R_2$

$I_2 = R_4 \leftarrow R_1 + R_6$ } WAR } WAW

$I_3 = R_1 \leftarrow R_9 + R_{10}$

$I_4 = R_5 \leftarrow R_1 + R_7$

Si como veremos luego podemos avanzar más allá (con prescindencia) del conflicto RAW (I_1 – I_2 y I_3-I_4) se podrá arribar al momento de direccionar I_4 que se tienen en el pipe I_1, I_2 e I_3.
¿Cómo trabaja el renombramiento?
Cuando se decodifica I_1 se asigna R_x a R_1 y al decodificar I_3 se asigna R_4 a R_1.
Cuando se decodificaba I_2 se hace intervenir (fuente) a R_x el cual contendrá el resultado de I_1; luego para nada se verá afectado por el resultado de I_3.
El conflicto WAR entre I_2-I_3 desaparece a partir del renombramiento.
Análogamente I_4 espera por R_4 y para nada interfiere I_1; a este nivel el conflicto WAW también desaparece.
¿Qué pasa con R_1? Al momento de despachar I_1 al R_1 se lo instruye para que se haga del resultado de R_x.

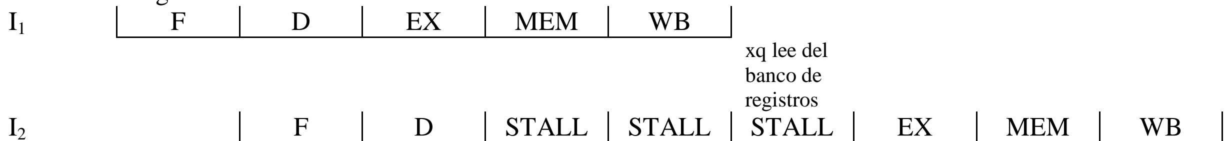
Forwarding

RAW

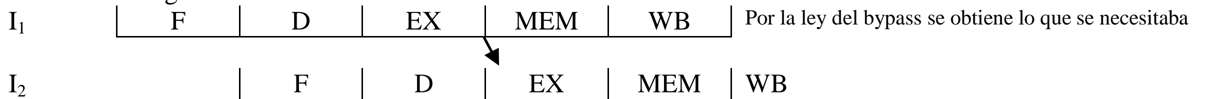
```
I1 ADD R1, R2, R3
I2 SUB R4, R1, R5
```

La cuestión es que SUB no podrá progresar (pasar a execute) hasta tanto no se conozca el resultado del ADD. En otras palabras, la instrucción ADD escribe el registro en la etapa WB, mientras que la instrucción siguiente la necesita en su etapa ID.
Veremos en primer lugar una alternativa de hardware, no ya para solucionar el conflicto RAW, sino que posibilita una reducción en la latencia efectiva de las operaciones (de las funciones), que podría reducir los ciclos stall's o en ciertos casos eliminar.
La técnica en cuestión es "forwarding" o bypass. Toda vez que el dato sea conocido (cierto) dentro del CPU se pueden organizar caminos alternativos para llevar el dato al lugar (Unidad Funcional) que la requiera sin esperar a la etapa write-back, que es cuando el decode podría extraerlo del register file.

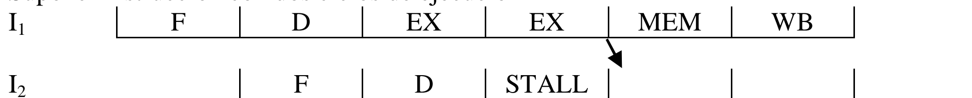
Sin forwarding



Con forwarding

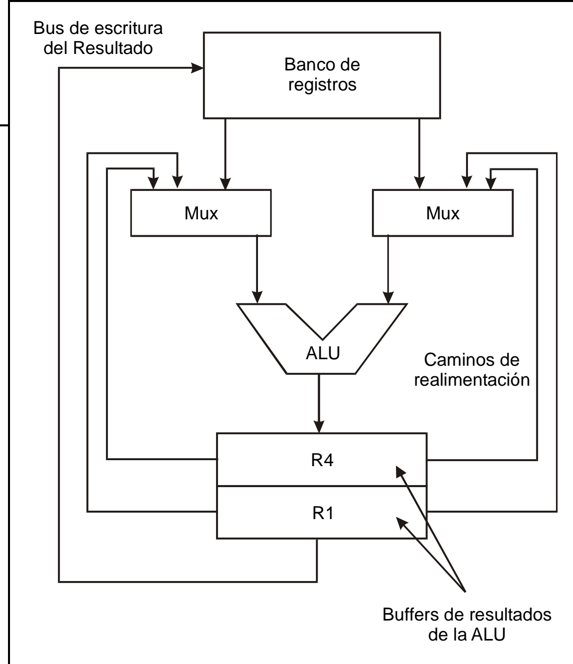


Suponer instrucción con dos ciclos de ejecución



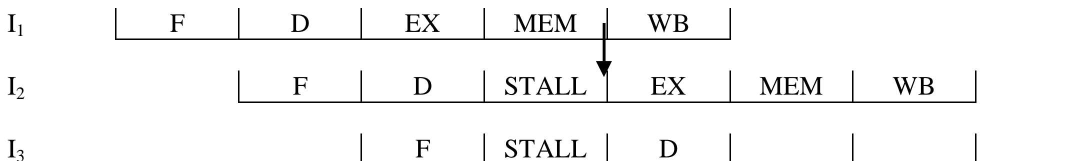
El fowarding requiere lógica de detección y buses (caminos alternativos).
Además hay que entender que el fowarding es de "todos contra todos" vale decir las salidas de las distintas unidades funcionales poderlas llevar por adelantado a las distintas entradas.
Esto último presenta la mayor dificultad hoy en día dado que, como se ha dicho, los retardos de interconexión solo los que dominan el temporizado.

Si al banco de registros se lo accede dos veces en un ciclo, podría escribir (WB) en la primer mitad y leer (decode) en la segunda mitad con lo cual se elimina un nivel de forwarding. Debe considerarse de todas formas



que en los pipelines profundos acceder al banco de registros en un ciclo es difícil, por lo tanto, no da leerlos en un ciclo dos veces.

LOAD R_1, M[-]
ADD R_4, R_1, R_5
MUL R_6, R_3, R_2



Ninguna instrucción puede decodificar porque el decode está trabado.
¿Cómo manejar el hazzard RAW para reducir los ciclos stall's? Puede implementarse static scheduling a través de software o dynamic scheduling (ejecución fuera de orden o despacho condicional) a través de hardware.

Static Scheduling

En esta política el compilador procesa el código reordenando instrucciones de forma tal que, en la manera de lo posible, aquellas instrucciones que son dependientes entre sí resulten separadas adecuadamente (de minima la latencia de la operación fuente).
La desventaja del compilador es que si bien las dependencias a través de registros son explicitas, lo condicionan los conflictos de nombres y el estar limitado por los registros lógicos.
Más aún tendrá ciertas dificultades cuando la posible dependencia se dé a través de locaciones de memoria. Por lo que, deberá ser muy conservativo en el reordenamiento.

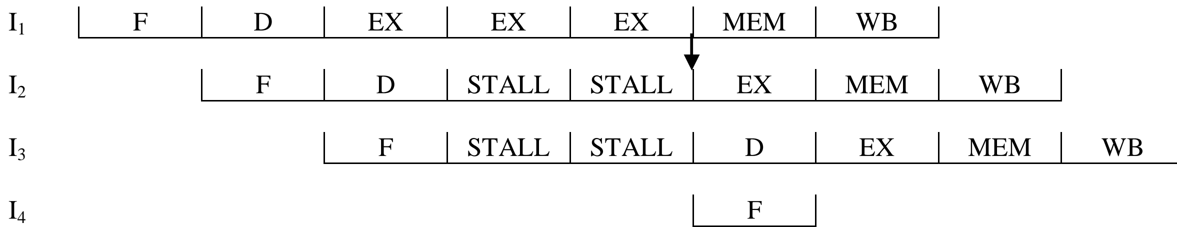
Dynamic Scheduling

La solución de hardware se alcanza a partir de dividir la etapa de decode en dos:
1. Issue (despacho)
2. Read of operand (lectura de operandos)

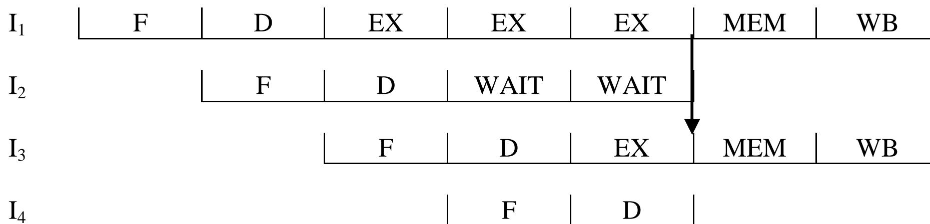
Al sacarle al decode la responsabilidad de leer operando posibilito que no se vea frenado por el conflicto RAW. La tarea de read se la transfiere a otra etapa, con lo cual se podrá seguir con las instrucciones posteriores (a la del conflicto) y solo se vería frenada la o las instrucciones con dependencia RAW, dando lugar a ejecución fuera de orden. Cabe aclarar no obstante que el issue se realiza en orden.

I_1 MULT R_4, R_1, R_2
I_2 ADD R_6, R_4, R_1
I_3 SUB R_9, R_8, #25

Con forwarding, sin dynamic scheduling



Con forwarding, con dynamic scheduling



Scoreboard

En la etapa read operand eventualmente una instrucción "en espera" podrá ser bypaseada por instrucciones posteriores en el pipe, dando lugar a ejecución fuera de orden.
El primer sistema que implemento esto fue CDC 6600. La idea era que se contaba con múltiples UF y se buscaba mejorar su aprovechamiento, vale decir que el conflicto RAW no frenara al pipe.
Se valía de un "scoreboard" que centralizaba todo el control.
Al momento del issue, posterior al fetch, analizaba:
1. Disponibilidad de UF; de no haberla se frena el issue.
2. La instrucción en cuestión tiene como destino un registro que también es destino de una previa (no ejecutada). En tal caso, si ocurre un conflicto WAW se frena el issue.
Realizado el issue, se instruye a la UF para que vaya a buscar sus operandos, en caso de que no exista conflicto RAW. Caso contrario, se difiere dicha orden al tiempo que efectivamente los resultados de instrucciones previas hayan actualizado los respectivos registros.
Esta es la forma en que se resuelve el conflicto RAW.
Ahora debemos preguntarnos, ¿Cómo resuelve el WAR?
Bastara que el scoreboard también tenga la atribución de ordenar cuando una instrucción puede almacenar su resultado.
Si bien funcionara correctamente, una limitación significativa en cuanto a performance es que si no resuelve adecuadamente los conflictos de nombre, WAW lo frena, WAR de forma precaria.
IBM para su sistema 370, unos tres años después, desarrollo el "Algoritmo de Tomasulo".

Algoritmo de Tomasulo

¿Cuál era el desafío?
Partiendo de un procesador de punto flotante con solo cuatro registros, y en un contexto en el cual las latencias de los accesos a memoria eran significativas diseñar un hardware que tuviese alta performance.
No solo le sale al encuentro al conflicto RAW sino también a los de nombre WAR y WAW. ¿Cómo? Con renombramiento implícito.
Más aún también resuelve los conflictos a nivel de accesos a memoria en lo que se conoce como memory desambiaguation.

Esta es otra técnica para el scheduling dinámico. En muchos aspectos es parecido al SC, pero tiene dos diferencias importantes:

- La detección de los hazard y el control de ejecución está *distribuido* (estaciones de reservación en cada unidad funcional controlan cuándo una instrucción puede comenzar la ejecución en esa unidad). En el SC está centralizado en él.
- Los resultados son pasados directamente a la unidad funcional, en lugar de que tengan que ir a buscarlo a los registros. Se usa un bus común de resultados(common data bus o CDB) que permite que todas las unidades reciban simultáneamente un operando que deben cargar.

Para analizar el algoritmo, nos enfocamos en la implementación hecha para una unidad de punto flotante. La estructura básica del algoritmo de Tomasulo es:

- **Tabla de Reservación**: mantiene las instrucciones que fueron despachadas y están siendo ejecutadas en la unidad funcional, así como la información necesaria para controlar la instrucción.
- **El load y store buffer**: mantienen los datos que vienen y van a memoria.
- **Registros de Punto Flotante**: están conectados por un par de buses a las unidades funcionales y por un solo bus al store buffer.
- **CDB**: todos los resultados de las unidades funcionales y de memoria son enviados en el CDB, el cual va a todos los lugares, excepto el load buffer.
- Todos los buffers y estaciones de reservaciones tienen **tags**(identificadores) que son usados para el control de los hazard.

Las etapas por las que atraviesa una instrucción en el algoritmo de Tomasulo son:

1- *Issue*. Se toma una instrucción de la cola de operaciones de punto flotante. Si la operación es de punto flotante, se despacha si hay una estación de reservación vacía y se envían los operandos a la estación de reservación si están en los registros. Si la operación es un load o un store, puede ser despachada si hay un buffer disponible. Si no hay estaciones de reservación vacías o buffer vacíos, existe un hazzard estructural y la instrucción es frenada hasta que se libera una estación o un buffer.

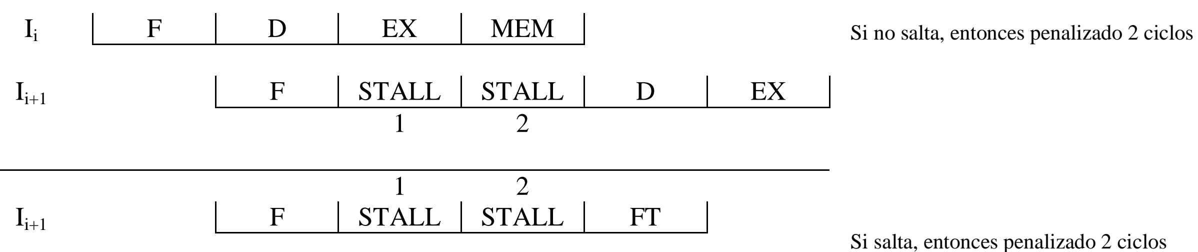
2- *Ejecución*. Si uno o más operandos no están disponibles, se monitorea el CDB mientras se espera porque el registro sea computado. Este paso chequea por RAW hazzard. Cuando ambos operandos están disponibles, se ejecuta la operación.

3- *Escribir Resultado*. Cuando el resultado está disponible, se escribe este en el CDB y de este a los registros y en cualquier unidad funcional esperando por este resultado.

Una observación importante es que los hazard WAR y WAW no son chequeados, porque son eliminados a través de la utilización de renombramientos de registros implícitos usando las estaciones de reservación.

Hazard de Control

Los conflictos de control pueden causar una mayor pérdida de performance en nuestro pipeline que los data hazard. Cuando un branch es ejecutado este puede o no cambiar el flujo secuencial del programa. Si la instrucción *i* es un branch, no sabemos cuál será la instrucción a ejecutar luego de ella. Esto causará un stall en el pipeline hasta resolver el branch.
En el DLX el branch se resuelve en el ciclo memory de la instrucción.
En las máquinas donde la evaluación de las condiciones y calcular la dirección target del branch toma más tiempo y no pueden ser resueltas en el ID, los branch hazard tienen más ciclos de reloj de penalidad.



¿Cómo se puede mejorar la performance?
En un pipe poco profundo como el DLX, se podrá pensar en adelante todo lo posible la reducción del cálculo si el

- Si el branch es tomado o no tomado.
- Computar el PC tomado (dirección target del branch) más temprano.

Para realizar lo primero, se puede completar la decisión del branch al final del ciclo ID usando lógica especial puesta para este test. Para tomar ventaja de esto, se debe computar la dirección target del branch.
Esto requiere un sumador adicional, que puede sumar durante la etapa de ID. Con esto solo hay un stall de un ciclo de reloj en los branch.
Como quedaría el temporizado

Existen soluciones estáticas y soluciones dinámicas al problema. Las soluciones estáticas son cuatro:
a) Lo visto, hacer stall hasta tanto se resuelva el branch
b) Predecir no tomado, vale decir avanzo con las instrucciones del FALL THROUGH y en caso de fallar la predicción estas instrucciones se "nulifican", es decir, se traducen en NOOPS.

Si predigo bien no pierdo ciclos, si predigo mal pierdo tres ciclos.
En cualquier caso no debemos interpretar esto como "especulación". La especulación, ejecución especulativa, sugiere que se ejecuta una instrucción y eventualmente instrucciones que dependan de esta sin saber a ciencia cierta si corresponde o no su ejecución.
c) Predecir tomado, valida si conozco la dirección del target antes de saber si salta o no.
d) Branch retardado (Delayed branch). Este es un recurso de software.
Como se vio en el DLX tendremos luego del branch una cantidad de ciclos "de incertidumbre" en cuanto a cómo sigue la ejecución. En el DLX eran 3 ciclos.

A estos los referimos como slots de retardo. ¿Qué hace el compilador? Abre el código y se hace cargo de introducir instrucciones que:

- No afecten la ejecución, salte o no, tener en cuenta que estas siempre se ejecutan
- Convenientemente realicen trabajo útil.

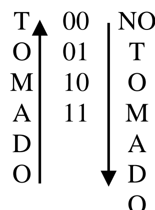
El branch se procesa como cualquier instrucción, lo que si podrá ocurrir es que superados los slots de retardo el PC podría ser el siguiente en secuencia PC ← PC + 4 o el del target.
Existen tres escenarios posibles:
1. Instrucciones anteriores al branch, en tanto no condicionen su ejecución.
De no ser posible:
2. Instrucciones del fall throught, en la hipótesis de que es probable NO salte
3. Instrucciones del target, en la hipótesis de que es probable QUE salte.
En ambos casos, 2 y 3, se está especulando.
La especulación no es tarea simple. Debemos garantizar:
a) flujo de datos
b) comportamiento de excepciones
En caso de traer instrucciones del target se deberá duplicar estas instrucciones dado que puede ser el caso de que se llegue a estas desde otro lugar del programa.

Si bien es una alternativa de software, a nivel del hardware necesita por el tema de interrupciones multiplicidad de PC's, ¿Cuántos? N° de slots + 1.

Una mejora es introducir predicción dinámica de los branch.
Ajustar, en ejecución, la predicción relativa a si es tomado o no.
El esquema más simple es manejar con un bit que ayuda al último comportamiento del branch. Este bit P, será 1 si fue tomado, 0 si no fue tomado. Cuando se decodifica el branch se consulta al predictor
¿Cómo se implementa? Una pequeña memoria, rápida, con no más de algunos miles de entradas de un bit c/u. Esto se conoce como branch prediction buffer", BPB.

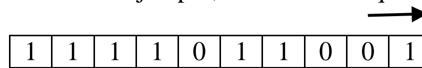
Este esquema de predictor de un bit es mejorable.
Supongamos que un ciclo se ejecuta diez veces. Con un predictor de 1 bit falla 2 de las 10 veces, 80% de acierto.
La idea de mejora se inspira en dotar al predictor de cierta inercia.
Por ejemplo, si planteamos cambiar la predicción luego de dos comportamientos contrarios, cuando el ciclo se está ejecutado el predictor dirá tomado, al salir dada la inercia no cambia sigue tomado.
¿Luego cuantas veces de las diez falla? Una sola, 90% de acierto.

En realidad predictores de más de un bit se construyen de forma aproximada, se emplean contadores saturables.
Sea P un contador con dos 2 bits.



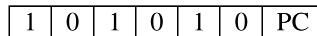
Donde el bit más significativo indica la predicción.
Si 0 tomado
Si 1 no tomado

También hay contadores saturables de 3 bits
Se puede seguir, al menos intentando, con mejoras al predictor empleado predictores en dos niveles o "correlation predictor". Vale decir asociar el comportamiento del branch a una historia pasada que podrá ser local (solo la determina el branch en cuestión) o global (en la que intervienen todos los branch a ejecutar).
P es para el local. ¿Cómo armamos, codificamos una historia pasada?
Simple, con un registro de desplazamiento. Por ejemplo, diez entradas que contemplan las diez últimas ejecuciones del branch.

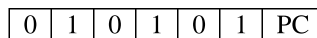


1	1	1	1	0	1	1	0	0	1

Con esta información, concatenada con (como antes) un cierto número de bits inferiores del PC direcciono el buffer.
Un branch que alternativamente salta y luego no salta y así siguiente sería básicamente imposible de predecir en un nivel, pero que si en 2 niveles.



1	0	1	0	1	0	PC



0	1	0	1	0	1	PC

Branch Target Buffer (BTB)

Si al momento de hacer el fetch de una instrucción consultamos al BTB (aún no se sabe que hace la instrucción) podrá ser que en el BTB se supiese que es una instrucción de salto y si incluye como parte de la información la dirección del salto, de funcionar bien la predicción la penalización del branch es 0. Se habla de un HIT BTB.
El predictor requiere una decodificación mientras que el BTB no, utiliza una simple comparación.

PC	DIRECCION SALTO	INST

Tabla BTB

Cada entrada tiene el PC completo, en cambio, el BPB emplaza los bits inferiores. Guardo todo el PC porque todavía no sé que es un branch. La consulta deberá hacerse simultánea en todas las entradas, se requiere por ello una memoria asociativa (content addresable memory)

Branch Folding

Se da con jump (saltos incondicionales)
Para los jump podríamos agregar a las entradas del BTB la instrucción/es del target.

En lugar de cargar en el IR la instrucción del jump cargo la instrucción del BTB. Luego en tal caso el jump cuesta cero ciclos.

Para avanzar aún más en el aprovechamiento del ILP el recurso que queda es especulación.
Se tiene una serie de alternativas para su manejo, con mayor o menor asistencia del hardware. La arquitectura aquí lograda es "hardware especulativo" que incluye:
- Ejecución fuera de orden
- Predicción dinámica del branch
- Soporte por hardware para ejecución especulativa, según lo requerido y además la capacidad de, ante una predicción fallida, retomar en pocos ciclos el camino correcto. Para ello implementa "check pointing". ¿Cómo funciona la ejecución especulativa? Se agrega al esquema del pipe propuesto una etapa más al final de "commit". Las instrucciones se ejecutan, según el esquema visto, pero lo que se controla es el commit de las mismas. Hasta que este no se produzca, no hay cambio de estado, "de afuera" no se ve nada, nada cambio.
El commit se hará en orden (de las instrucciones) cuando efectivamente sea cierto que corresponda ser ejecutado.
A lo visto agregamos un "reorder buffer". En el ingresan las instrucciones despachadas (ISSUE) en orden.

La dinámica es, se traen instrucciones del FETCH, se las despacha (ISSUE) en orden, se utilizan predictores para establecer un camino posible, se ejecutan fuera de orden (dynamic scheduling).

Los resultados irán ingresando al reorder buffer, del cual tomaran su entrada instrucciones dependientes de ellas, estableciendo así un flujo de ejecución especulativa.
¿Cómo se produce el commit? Se analiza el tope del RB:
Si se trata de instrucciones que ya se ejecutaron o
Si en el tope hay instrucciones pendientes de ejecución no se avanza con el commit. Claramente esto es en orden.
¿Qué ocurre con los branch, particularmente aquellos mal predichos? Un branch mal predicho en el tope implica que lo actuado no sirve, hago un flush del RB y hago uso del checkpoint.
Referido a las excepciones estas se atienden recién cuando la instrucción que la provoca llega al tope del buffer. En tal caso es claro que corresponde ejecutarlas, luego se accionara correctamente. De ser originada en una especulación fallida no hubiera llegado al tope del RB, luego nunca hubiese sido atendida.
Arribado a este punto, esto permite estar lo más cerca de un CPI = 1 (IPC = 1).

Además, con esta dinámica, está suministrando manejo de interrupciones precisas.

Multiple Issue

¿Cómo se puede mejorar más aún? Esto es, un CPI < 1 o IPC > 1, la respuesta es multiple issue.
Vale decir despachar más de una instrucción por ciclo, esto es, hacer el fetch de 2,4,6 instrucciones por ciclo. (no mucho más que esto).
Para el issue se tiene dos alternativas:

- Static Issue, Ej. VLIW (very long instruction Word)
- Dynamic Issue, Ej. SUPERESCALAR (Static Scheduling & Dynamic Scheduling).

Static Issue

Realiza un planteo con un hardware minimal. Es el software el que debe acometer la tarea de extraer “paralelismo” a la aplicación.
¿Por qué large? EL compilador empaqueta un conjunto de instrucciones (u operaciones) en una única instrucción. Cabe aclarar que es un planteo estático, el formato está fijado.
Por ejemplo, una instrucción de este tipo podrá ser:

MEM	MEM	PF	PF	INT	INT	BRANCH

Lo que se trae (VLI) se ingresa sin más, no hay chequeo entre instrucciones del mismo paquete, como tampoco entre estas e instrucciones aún sin ejecutar.
Esto pone de manifiesto cuanto y como se simplifica el hardware. Se podrá decir que este y (no otros), son aspectos meritorios de esta arquitectura.
Tiene limitaciones, algunas compartidas con el superescalar y otras que le son propias.
Comparte el tema de múltiples accesos a memoria por ciclo, imponiendo un condicionamiento al periodo, originalizando lentitud en las múltiples unidades y el forwarding asociado.
Para facilitar la tarea del compilador, existen recursos como loop unrolling y algo que se desarrollo específicamente para estas arquitecturas que se denomina “trace scheduling”. Observar que el compilador debe entregar instrucciones independientes. Una medida de su mínimo es el ancho de la instrucción por la profundidad promedio del pipe de las U.F. Estas cuestiones de dependencia, más allá de la dificultad en resolverlas son conocidas. Pero una cuestión no manejable y con severos daños a la performance son los “miss en cache”.
¿Por qué? Por cómo opera el despacho. Si hay un miss todo se ve frenado, seguramente un número importante de ciclos, esto no sería manejable, la alternativa básica para al menos intentar reducir esta pérdida es trabajar con caches lo más grandes posibles.
Una dificultad que le es propia es incompatibilidad de código. Intel, en su momento, argumentando que su arquitectura de 64 bits sería EPIC (explicit paralelism instruction computer), que no es otra cosa que una evolución del VLIW.

Superescalar - Dynamic Issue

Sugiere que se procede a despachar instrucciones, hasta un número máximo (por ejemplo 4, si se tiene un superescalar de 4 instrucciones por ciclo), acorde a lo que determine el hardware.
Si resultan despachadas menos instrucciones (que el ancho) estas pasan al ciclo siguiente.

El procesador superescalar está en mejores condiciones que el vectorial cuando el paralelismo no resulta evidente.

Si por ejemplo es superescalar de dos, trae dos instrucciones en cada fetch. Luego el hardware decide dinámicamente si despacha las dos, despacha 1 o ningún en ese ciclo. En un inicio la propuesta fue muy simple, una instrucción de entero y otra de punto flotante. Más aun en ese orden.
De esta forma simplifica el chequeo de dependencias. Con la evolución de este tipo de arquitecturas, estas restricciones se reservan más que significativamente.

Con las supercomputadoras se empleaba una medida "muy engañosa" que era performance pico.
Eran procesadores vectoriales, su performance cae muy rápidamente a partir de no alcanzar una vectorizacion ideal. La performance pico se calculaba sin más en esas condiciones idealísimas, lejos de cualquier realidad.

Capítulo 4 – Jerarquía de Memoria

Prácticamente desde siempre, desde el comienzo de las computadoras de programa almacenado la memoria no se limito a un único nivel sino que organizo de manera jerárquica, de aquí "jerarquía de memoria". En esta jerarquía cada nivel tendrá atributos distintos, diferenciados de:
- Velocidad
- Tamaño
- Costo por bit

Cuanto más cerca del CPU, mayor velocidad, mayor costo por bit, menor tamaño.

Objetivos de la Jerarquía de Memoria

Los objetivos son:

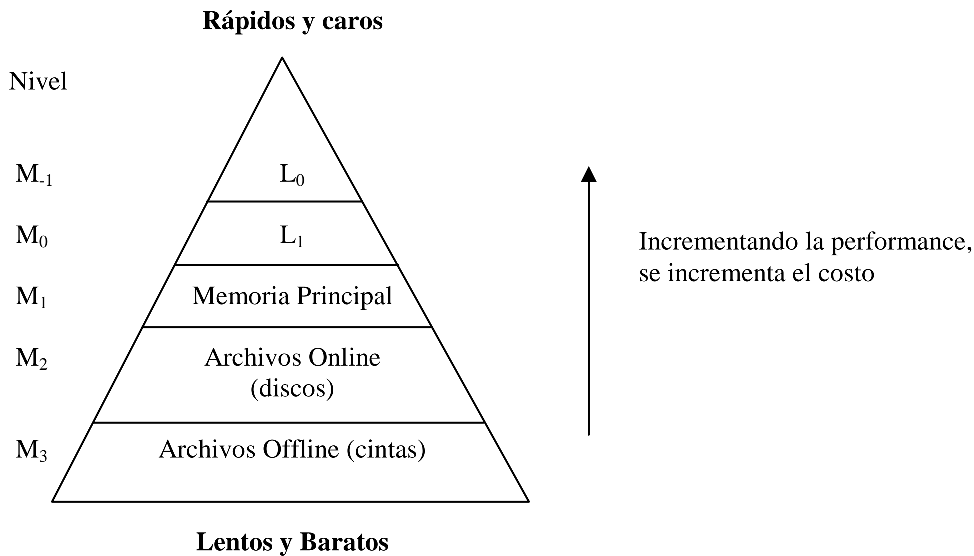
- Reducir el "gap" creciente que hay entre la velocidad del CPU y la memoria. Para esto se incluyen uno (o más) niveles de cache.
- Lograr una performance adecuada a un costo razonable. Equilibrar la relación costo / performance. El costo puede ser descompuesto en dos factores:
    - *factor estático*, el cual limita el tamaño de los componentes más rápidos, y por lo tanto los más caros;
    - *factor dinámico,* alude a los recursos que se consumen para administrar la jerarquía (ej. procesador central, buses).

El mismo se reduce limitando el tamaño de los niveles más costosos (caros).
En lo que a la performance respecta, procurando que la mayoría de los accesos se resuelvan en los niveles más altos de la jerarquía (los más próximos al CPU). Son los más rápidos, un faltante en un nivel superior implicaría una penalización en el tiempo de acceso en tanto se deberá acceder a un nivel próximo inferior, que como fue dicho es más lento.
En resumen el objetivo propuesto es alcanzar una velocidad efectiva próxima al nivel más alto con un costo por bit próximo al del nivel más bajo.
Es decir, se quiere lograr la velocidad de la memoria más rápida, al costo de la memoria más lenta.
¿Qué es lo que da sustento a una jerarquía de memoria? ¿Cómo es que puede llegar a cumplimentar el objetivo de performance?
Los programas no referencian a memoria al azar, sino que se comportan de manera predecible. Esta propiedad se la conoce como Localidad de Referencia. Esta está compuesta por:

- **Localidad Temporal**. La posibilidad cierta que una locación vuelva a ser referenciada en un futuro inmediato. (loops, variables temporales).
- **Localidad Espacial**. La tendencia de un programa a referenciar áreas acotadas de memoria vecinas a la última referencia. (procedimientos).
- **Localidad Secuencial**. Es la tendencia a que referencias sucesivas se resuelvan en locaciones consecutivas de memoria (instrucciones secuenciales, arreglos)

Cada tipo de localidad ayuda o influencia la caracterización de una jerarquía de memoria eficiente. El principio de localidad *espacial* permite determinar el tamaño de bloque a ser transferido entre niveles. El principio de localidad *temporal* ayuda a determinar el número de bloques a ser contenidos en cada nivel. La localidad *secuencial* permite la distribución de identificadores únicos para dispositivos que operan concurrentemente en ciertos niveles de la jerarquía para posibilitar accesos concurrentes.

Los niveles encontrados en las computadoras actuales son mostrados en la siguiente figura:



- **Cache.** Este es el nivel entre el procesador y la memoria principal. Puede haber varios niveles de cache. Forma una memoria de alta velocidad, que eleva la performance del sistema. Debido al alto costo por bit que estas tienen, sus tamaños son muchos más pequeños que la memoria principal.
- **Memoria Principal.** Esta se encuentra entre la cache y el almacenamiento secundario. El acceso a la misma es aleatorio.
- **Almacenamiento Secundario o Archivos Online.** Este corresponde al almacenamiento permanente en archivos. Está soportado por uno o más discos de brazo móvil (discos duros). Los tiempos de acceso son muchos grandes que los de la Memoria Principal, debido al tiempo de búsqueda que se necesita para mover el brazo al cilindro adecuado. Su capacidad es mucho más grande que la de la Memoria Principal y su costo por bit es mucho más pequeño. El acceso es secuencial.
- **Archivos Offline.** Este es el último nivel. Aquí aparecen el almacenamiento de archivos en discos removibles y cintas magnéticas. Estos últimos son más baratos y lentos y tienen un acceso directo y secuencial. Pueden guardar alrededor de cientos de MB. El tiempo de transferencia es menor que los discos y el tiempo de acceso depende del último dato leído.

Principios generales de la jerarquía de Memoria

Si bien algunos incluyen a los registros en el tope de la jerarquía, estos pertenecen al procesador, son parte de él.
La forma piramidal alude al tamaño relativo de cada nivel.
La incidencia de esto es costo; se limita el tamaño de los más costosos y rápidos. Es un costo estático. Podemos pensar en un costo dinámico, esto es cuanto recurso demanda gestionar la jerarquía (Por ejemplo: memoria virtual en un faltante implica tiempo de CPU, discos, canales).

Costo promedio = (costo bit cache * tamaño cache + costo bit memoria * tamaño memoria principal + costo bit memoria secundaria * tamaño memoria secundaria) / tamaño datos

Mientras que la cache utiliza tecnología MOS estática, la memoria principal utiliza tecnología MOS dinámica.
MOS proviene de metal oxido semiconductor.
Antes, en los 60's y 70's y parte de los 80's la tecnología que predominaba era bipolar (transistor de doble juntura).
MOS que nació como un dispositivo muy lento tiene la ventaja que se adapta, todo le resulta viable y beneficioso, al escalamiento, cosa que en caso de bipolar no es así.
Los dispositivos más pequeños en MOS se traducen en:
- Mayor cantidad por unidad de área
- Los dispositivos individuales consumen menos potencia
- Es más rápido

Cuando se diseña una celda de memoria hay dos alternativas con la tecnología MOS:
- Estática: aquí una celda es un biestable
- Dinámica: aquí una celda es un pequeño condensador que se carga a una tensión alta o baja (V_H ó V_L).

La descarga del condensador en el tiempo obliga a continuos "refrescos".
Dentro de MOS lo que se emplea hoy en día es CMOS (Complementary MOS).
Estos dispositivos consumen básicamente cuando hay cambio de estado de 0 a 1 ó de 1 a 0. En reposo el consumo es mínimo (teóricamente 0).

La jerarquía de memoria consta de varios niveles, pero es manejada entre dos niveles adyacentes al mismo tiempo. El nivel superior es aquel que, de los dos, está más cerca del procesador, y el nivel inferior es el que está más lejos. La mínima unidad de información que puede estar presente, o no, en los niveles de la jerarquía se llama bloque. El CPU hace un acceso a una locación de memoria y si no está el bloque en el nivel superior, el faltante se resuelve trayendo un bloque desde el nivel inferior "que lo contiene".
Ese bloque podrá ser en algún caso de tamaño fijo y en otros variable. Si es fijo, el tamaño de la memoria es múltiplo del tamaño de bloque.

¿Cómo administrar la interfaz memoria cache – memoria principal?

Los tiempos involucrados en las transferencias corresponderían a traer de memoria principal algunas palabras (8-16-32 bytes) operando en tamaño fijo. Luego serán unos ciclos de memoria. Para manejar esta interface es claro debe hacerse por hardware.
Entre memoria principal y memoria secundaria los tiempos son mucho mayores. En este tiempo se podrán, por ejemplo:

Disco 10 milis, 10^{-2} seg
2 GHZ da un periodo de $0{,}5x10^{-9}$ con dos instrucciones por ciclo
Luego, $10^{-2} / 0{,}25x10^{-9} = 4x10^{7}$ instrucciones

Entre archivos online y archivos offline los tiempos podrán ser aún mucho mayores (Por ejemplo una cinta varios segundos hasta producir el acceso). Se puede recurrir para su manejo a mecanismos robóticos o hasta intervención humana.
El objetivo de la velocidad se cumplirá en la medida que el tiempo efectivo de acceso se aproxme al del nivel superior y no al del nivel inferior.
Cuando se encuentra la información en el nivel más alto se habla de "**hit**", caso contrario hablamos de "**miss**".

Tiempo de acceso a un dado nivel = Hit Time + Miss Rate * Miss Penalty

- Hit time = es igual al tiempo de acceso del nivel más alto, incluido el chequeo de la búsqueda.
- Miss rate = contempla la transferencia del bloque desde el nivel más bajo y el propio acceso de la información del miss.
- Miss penalty = tiempo que tarda en buscar el bloque y llevarlo al nivel correspondiente.

Una adecuada tasa de aciertos dependerá de la disparidad de velocidad entre niveles. Cuanto más dispar sean los tiempos del nivel superior e inferior mayor deberá ser la tasa de acierto.

La tasa de aciertos (hit ratio) típicamente se define como la fracción de accesos exitosos o como porcentaje.

Miss Rate = 1 – Hit Ratio

Por caso, entre la memoria cache y la memoria principal la relación es de 10-30. Entre la memoria principal y la memoria secundaria 30-15, esto es: $\frac{15 \,.\, 10^{-3}}{15 \,.\, 10^{-9}} = 0{,}5.10^{6}$
Luego se tendría una tasa de acierto en cache del 90%.

Por lo tanto $T_{acceso} = 1 + 0{,}1*10 = 2$ (más parecido a 1 que a 10)

¿Para acceso a memoria principal que tasa de acierto demandaría?
Se necesita 99,999 de acierto para que esto sea sensato.

Luego $T_{acceso} = 1 + 0{,}1*6 * 0{,}001 = 1 + 500 = 501$

Memoria Cache

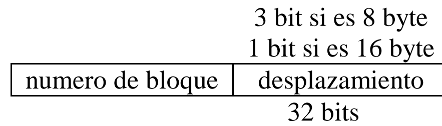
Es una memoria relativamente rápida que se introduce entre el procesador y la memoria principal. Cuando nos hablan del tamaño de la cache nos hacen referencia al tamaño de los datos aunque en la misma también se guarda información adicional. Podemos estudiarla siguiendo varios ejes:
- Donde ubicar en cache un bloque de memoria principal. Como organizarla
- Como actuar ante una escritura en cache
- Algoritmos de reemplazo

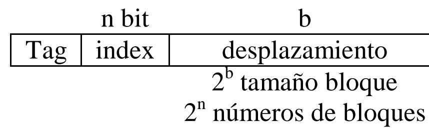
El bloque tiene un tamaño fijo, definido por el diseñador de la arquitectura. La memoria y la cache la tendremos divida en bloques de iguales tamaño.

Locación de un bloque en Cache

Las restricciones de en dónde colocar un bloque, crean 3 categorías de organizaciones de cache:

- **Mapeo Directo**. Aquí cada bloque de memoria principal puede ir a único lugar predeterminado en cache. Es el más simple. El mapeo es generalmente [numero de bloque] MOD [*n*º de bloques en cache].





Cuando realizo una búsqueda:

V	TAG	DATO

Supongamos 2^n=N. Luego los bloques en memoria K, K+N, K+2N van al mismo lugar. La posibilidad de que se estén expulsando continuamente se puede dar si dos direcciones tienen el mismo index de memoria.
Lo que alienta a direccionar al bloque (index) con los bits inferiores es la localidad secuencial de las referencias. "Si tengo varias referencias con el mismo index esto será ineficiente.
La localidad temporal es la que le da sustento a la cache sin prejuicio de que también podrá tomar ventaja de la espacial.
Ante un miss se trae a cache un puñado de palabras y en la medida que se tienen en el futuro próximo acceso a estos lograrse tasas de acierto pretendidas.

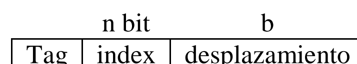
- **Mapeo Full Asociativo**. Un bloque de memoria principal podrá ir a cualquier bloque de cache. Es más complicado.

V	#BLOQUE	DATO

Aquí desaparecen, junto con el índex el acceso directo. La búsqueda en toda la cache debe ser por contenido. Luego a nivel de directorio necesito una memoria asociativa.
Hay dos tipos de memoria asociativa, ambas con cierto nivel de paralelismo.
1. Bit Serie -> Todas las palabras a la vez, pero de a un bit por vez
2. Bit paralelo –> Todas la palabras a la vez, todos los bits de la palabra a la vez.

Para una memoria cache full asociativa se usa una memoria asociativa con bit paralelo. La reservamos en general no a nivel de datos.

- **Mapeo Set Asociativo**. Aquí un bloque de memoria principal podrá ir a un conjunto, set, predeterminado de bloques de cache. Un conjunto es un grupo de dos o más bloques en la memoria. Si es de dos bloque se lo denomina (two way set associative). Si es de cuatro bloques (four way set associative). Bajo esta política un bloque es mapeado a un set y luego es puesto en cualquier lugar dentro de él. El mapeo es generalmente [numero de bloque] MOD [número de set en el cache]. Si hay *n* bloques en un set, se dice que la organización del cache es *n*-associative. En el límite aumentando el tamaño del set, arribaría a full associative.



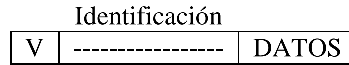
n tal que 2^n es el numero de sectores.

Con el index, de forma directa accedo al sector, luego dentro del sector el bloque de memoria podrá estar en cualquier bloque de ese sector.
En el límite, un solo sector de tamaño total de la cache, el campo index desaparece.

Esta política tiene inconvenientes, por un lado multiplico los comparadores, para un n-way se requieren n comparadores. Luego habrá que multiplexar y utilizar algoritmos de reemplazo.
Como cifra una cache de tamaño N con mapeo directo es equivalente a una cache de tamaño N/2 con two way set associatve.

Encuentro de un bloque si este está en el cache

Cada bloque de cache no solo contiene datos sino también información que identifica al bloque además de un bit de valido.



Este bit se le agrega, para saber si un bloque de cache tiene información válida. Si el bit no está activado, no se puede hacer un match en esa dirección. Por lo que se lo utiliza en dos escenarios: para inicializar la cache en el powerup y para invalidar cache.
Los caches incluyen una dirección tag (etiqueta) en cada bloque que se la da la dirección de block-frame. El tag debe contener la información deseada y que es chequeada para ver si coincide con el block-frame del CPU. Si el tag no coincide y el bit de valido es 0 hablamos de un miss.
Debido a que la velocidad es la esencia, todos los posibles tag son buscados en paralelo.
El tag tiene un costo en memoria. Se necesita uno para cada bloque. Una ventaja de incrementar el tamaño de bloque, es que el overhead del tag por cada entrada en cache se convierte en una fracción más pequeña del costo total del cache.

Reemplazo de bloques en un miss de cache

La decisión entre un bloque que contiene datos válidos y otro que no, es fácil. El problema ocurre cuando ambos bloques tienen datos válidos.
El uso de locación de mapeo directo hace que la decisión del hardware sea simple, es más, no hay opción. Solo un bloque es chequeado para un hit y solo un bloque puede ser reemplazado. En el reemplazo en full associative y set-associative hay varios bloques para elegir en un miss. Hay dos estrategias primarias para la elección del bloque a ser reemplazado:

- **Random**. Los bloques son elegidos al azar.
- **Least-Recently-Used (LRU)**. Para reducir la chance de sacar información que se necesitará pronto, los accesos a los bloques son recordados. El bloque reemplazado es el que ha sido inutilizado por más tiempo. Se hace uso de la localidad temporal.

Una virtud del random es que es simple de construir en hardware. A medida que el número de bloques a mantener aumenta, LRU se convierte incrementalmente en caro y es frecuentemente solo aproximado. Por caso, Pentium con 4 way associative realiza una “aproximación” con 4 bits.
Un hecho paradójico es que en su momento la VAX 780 para two way asociative adoptaba Random. Observar que para LRU hubiese bastado 1 bit.
Las políticas de reemplazo juegan un papel más importante en caches chicos, que en caches grandes donde hay más opciones para el reemplazo.

Escritura en Cache

Las lecturas dominan el acceso al cache. Todas las instrucciones son leídas y la mayoría de estas no escriben en memoria. Afortunadamente, el caso más común es también el caso más fácil de hacer rápido. El bloque puede ser leído al mismo tiempo que se compara el tag, así que la lectura del bloque comienza tan pronto como la dirección del block-frame está disponible. Si la lectura es un hit, el bloque es pasado inmediatamente a la CPU. Si es un miss, no hay beneficio, pero tampoco daño.
En la escritura no ocurre esto. El procesador especifica el tamaño de la lectura (usualmente 1 a 8 bytes); pero solo una parte del bloque se cambia. Esto significa que una secuencia de operaciones lectura-modificación-escritura se realizan sobre el bloque. Leer el bloque, modificar una porción y escribir el nuevo valor del bloque. Además, la modificación del bloque no comienza hasta que el tag se chequea para ver si es un hit. Debido a que el chequeo del tag no ocurre en paralelo, la escritura demora más tiempo que la lectura.
Hay dos opciones básicas cuando se escribe en el cache:

- **Write through**, la información es escrita tanto en el bloque del superior como en el bloque del nivel inferior de memoria (Por ejemplo: en cache y en memoria principal respectivamente).
- **Write back**, la información es escrita solo en el bloque del cache. El bloque de cache modificado es escrito en la memoria principal solo cuando es reemplazado. Claramente esta política de escritura

produce incoherencia entre cache y memoria principal. Un bloque de cache con write back es llamado clean o dirty, dependiendo si la información del cache difiere de la del nivel inferior de memoria. Para reducir la frecuencia de escritura en el reemplazamiento write back, se usa un bit dirty, que indica si el bloque fue o no modificado en el cache. Si no lo fue, el bloque no es escrito, ya que el nivel inferior tiene la misma información que el cache.

Ambos métodos tienen sus ventajas. Con write back, la escritura ocurre a la velocidad de la memoria cache y múltiples escrituras en el bloque requiere solo una escritura en el nivel inferior. Y como cada escritura no va a memoria, write back usa menos ancho de banda, haciendo la escritura más atractiva a multiprocesadores. Sin embargo, es más compleja, en caso de un miss motive un reemplazo puede ocurrir que haya que llevar esa información a memoria.

Con write through, los miss de lecturas no resultan en una escritura en el nivel inferior y es más fácil de implementar que el write back. También se tiene la ventaja de que la memoria principal siempre tiene la copia actual del dato. Por lo cual mantiene la coherencia entre las memorias.

Por esto, los multiprocesadores quieren write back para reducir el tráfico de memoria por procesador y quieren write through para mantener el cache y la memoria consistente.

Cuando el CPU debe esperar para que se realice la escritura, con write through, se dice que está write stall (escribiendo parado). Una optimización para reducir este write stall es usar un write buffer, que permite al procesador continuar mientras la memoria es actualizada.

Hay dos opciones ante un miss de escritura:

- Write Allocate (fetch en la escritura). Se trae el bloque a cache y se procede como en un write-hit.
- Write no Allocate (escritura around). El bloque es modificado en el nivel inferior y no es cargado en el cache.

Cualquier alternativa es licita, es decir cualquier combinación de write back y write through es posible, pero solo dos tienen sentido.

Generalmente caches con write back usan write allocate (esperando que escrituras subsecuentes a un bloque sean capturadas por el cache) y caches con write through usan write no allocate (ya escrituras subsecuentes a este bloque todavía tendrán que ir a memoria). Write through con Write allocate no tiene sentido, nada gano porque distintas escrituran se hacen en ambos niveles y más aún, me expongo a un reemplazo de un bloque eventualmente este usando la cache.

Con la presencia de la cache se origina un problema de coherencia entre la memoria principal y la cache. Este tiene que ver con las operaciones de E/S con DMA, esto es, sin intervención del CPU.

Entrada

El problema se origina si se modifica un bloque en memoria que esta mapeado en cache. Hay dos políticas: Update (Actualizar) o Invalidate (Invalidar). Esta última es la más simple y la que más conviene.

Salida

El problema solo lo tiene WB. De alguna manera la cache tiene que intervenir. Una posibilidad es que de tener el dato en estad dirty frene la transmisión hasta la actualización de la memoria.

Tipificación de los miss

Se pueden atribuir a unas de las siguientes fuentes:

- **Coesitivo o de la primera vez**. Es el primer acceso a un bloque que no está en el cache, entonces el bloque debe ser traído al cache. Es también llamado cold start (arranque en frío).
- **Capacidad**. Se produce dada la imposibilidad de que toda la aplicación entre en cache. Un bloque fue desplazado de cache por problemas de espacio y luego vuelve a ser referenciado, por lo que se produce un miss.
- **Conflictivo**. También llamado miss de colisión. Se podrán dar tanto con mapeo directo como con set asociativo. Nunca con full asociativo. Un bloque abandono la cache no por falta de espacio sino porque competía por una posición especifica, y luego al volver a ser referenciado produce el miss.
- **Coherencia**. En una arquitectura multiprocesador con cache privada en cada nodo podrá tener lugar este tipo de miss.

Si un CPU escribe en cache podrá ser el caso de que el bloque modificado se encuentre en la cache de otro nodo. Se presenta luego un problema de coherencia entre caches. Dos políticas: Update y Invalidate. Un miss de coherencia será posible con la policita de invalídate. La alternativa para evitar el problema de

coherencia dista en cuanto a determinar áreas no cacheables, cuestión altamente inconveniente que aquí no tendrá aplicación.

La arquitectura multiprocesador con memoria compartida encuentra una severa limitación en cuanto a escalamiento (esto es, aumento del número de procesadores). Lo que da impulso a las mismas, habida cuenta del cuello de botella que representa el uso de los recursos compartidos, esto es, buses, memoria (recordemos que el acceso a memoria para el caso de un único procesador es todo un tema, con mucha más razón sin son varios) es el empleo de memoria cache por su efecto de "aislar" la actividad individual.

Tamaño del bloque

Aumentar el tamaño del bloque redundara en un principio en mejoras de hit-ratio.
Dada la localidad espacial de las referencias, las que no desconocemos sin prejuicio de haber analizado que la temporal es la que da sustento a la cache.
Traigo un bloque ante un miss y accesos a distintas palabras del mismo bloque se resuelven con hit.

Esto manteniendo un tamaño fijo de cache.
Con el aumento del tamaño del bloque reduzco su numero y a la larga incrementare los miss de conflicto además de no terminar aprovechando todo el bloque.
Al principio del uso del cache ciertamente de tamaño reducido, tecnología bipolar, el tamaño del bloque era ni más ni menos de lo que se traía en un acceso a memoria. Aumentar el tamaño tendrá incidencia en el tiempo de transferencia.

Luego el tiempo de acceso promedio será = Hit Time + Miss Rate * Miss Penalty

Afortunadamente como se vera conviene acceder a memoria principal por ráfagas y no accesos a palabras individuales.
Caches pequeñas y simples redundan en:
+ Hit time por tamaño y por ejemplo mapeo directo
- Hit ratio
Como atender estas cuestiones, esto es, en orden a mejorar un aspecto (hit time no dañar otro hit ratio). Para el primero "cache multinivel".
Se tiene un doble objetivo cuando dimensionamos cache: tamaño y velocidad que como se dijo son antagónicos.
Se organiza a la cache en varios niveles (2 a 3 niveles).
El primer nivel, el que interactúa directamente con el CPU, el que condiciona fuertemente su temporizado, de tamaño y asociatividad acotada en orden a proponer tiempos de acceso y tasas de acierto atinadas.
Los niveles más bajos, léase L_1 y L_2 podrán ser más grandes con mayor asociatividad sin que esto, vale decir el mayor tiempo de acceso, tenga incidencia negativa directa en la performance.
Esto se comporta como una gran cache (podrá ser que el nivel L_0 este incluido en el L_1 o no).
La disipación de potencia tiene dos orígenes:

- Potencia dinámica, la que disipa cuando conmuta. Aumentar la frecuencia del reloj implica que esta aumenta.
- Potencia estática, de perdida.

Luego los niveles más alejados de cache se podrán alimentar con tensiones más bajas reduciendo esa potencia de perdida haciéndonos cargo de una respuesta más lenta que como se dijo quedara disimulada en la jerarquía.

$$\text{Tiempo ACCESO AL PRIMER NIVEL(1N)} = \text{Hit Time}_{1N} + \text{Miss Rate}_{1N} * \text{Miss Penalty}_{1N} =$$
$$\text{Hit Time}_{1N} + \text{Miss Rate}_{1N} (\text{Hit Time}_{2N} + \text{Miss Rate}_{2N} * \text{Miss Penalty}_{2N})$$

Luego el miss local es el miss de cada nivel. Y el miss global es el compuesto. Indica las veces que se debe ir a memoria.

$$\text{Miss Global} = \text{Miss Rate}_{1N} * \text{Miss Rate}_{2N}$$

Se puede implementar una asociatividad lógica (con predictor de way). Esto lo implementa digital en su segunda generación de Alpha. La cache del primer nivel es two way associativa pero la accede como mapeo directo.
Cada entrada a la cache además de los visto, esto es, tag, valid bit tiene: próxima dirección y predictor way.
Luego hit ratio como two way associative, hit time como mapeo directo

Optimizaciones sobre la memoria cache

Es ideal mantener un cache pequeña y simple sobre aquel nivel que se busca hit time, conservando el hit ratio para una cache asociativa con way prediction.
Otra alternativa en ese sentido, esto es tiempos de hit ratio propios de mapeo directo y hit ratio mejorados frente al mapeo directo es utilizar una "victim cache". La idea es asociar a la cache un pequeño buffer (unos pocos bloques) en configuración full asociativa.

La búsqueda se realiza en paralelo. Cuando se produce un reemplazo en la cache el bloque en cuestión se ingresa en la victim cache. De esa forma se da la oportunidad de disponer del bloque como en un hit. En caso de hallar el bloque en victim cache además de suministrar al CPU el dato "intercambia" con la cache los bloques de esa posición. Por ejemplo: con 4 entradas en la victim cache, dependiendo del programa, remueve entre el 20% a 95% de los miss de conflicto.

Miss Penalty

La reducción del miss penalty se puede alcanzar con cache "no bloqueante" o look-up cache. También apunta a mejorar el ancho de banda. Una cache bloqueante ante un miss se bloquea hasta tanto ese miss no se resuelva. Una aproximación simple a no bloqueante es que la cache siga respondiendo mientras resuelve el miss. Hit under miss. ¿A que apunta esto? A que se pueda seguir trabajando (CPU) en ese tiempo de penalización. Oculta la penalización si solapa la búsqueda con computo.
En general no resulta suficiente, el diseño se lleva a que pueda seguir respondiendo aún ante uno o varios miss. Luego se habla de "Miss Under Miss" o "Hit Under Several Miss". Este esquema resulta fundamental en una arquitectura moderna con dynamic scheduling, ejecución especulativa.
Vimos que bloques más grandes posibilitan mejora al hit ratio. Pero esto puede conducir a un miss penalty que neutralice las potenciales mejoras. Existe dos técnicas para mejorar el miss penalty en tal contexto:

- Critical Word first: Esta es más sutil. Se solicita a memoria la palabra del miss y esta se entrega con anticipación mientras se completa el bloque en cache. Debe seguramente implementarse con un solo acceso.
- Early restart: Tan pronto como arribe la palabra dentro del bloque la entrega al CPU sin esperar a completar el bloque.

Ancho de banda de cache

Esta es mejora en el número de accesos en la unidad de tiempo (throughput).
Repasando el superescalar, en la medida que el issue sea de 4 instrucciones o más para que esto funcione y no sea la memoria un cuello de botella, se requiere poder realizar más de un acceso por ciclo.
El problema es que la memoria tiene un único pórtico de lectura/escritura. ¿Qué se hace?
Se lo "simula" al punto de admitir más de un acceso por ciclo.
Tenemos dos alternativas:

- Pipeline de la cache (Alpha): El alpha hace trabajar a la cahe al doblew de velocidad y la organiza en pipeline.
- Múltiples bancos (Opteron): Esta relacionado con "interlieving de memoria". En lugar de disponer en un solo arreglo todos los bits de cache, los distribuimos en varios bancos. En particular en el Opteron en 8 bancos de acceso independiente.
  El opteron maneja dos accesos por ciclo. Si estos son a distintos bancos, ambos puedan progresar en paralelo. Selecciona al banco con los bits inferiores del address, luego aprovecha la localidad secuencial de las referencias.

Prefetching

Es una búsqueda "anticipada". Apunta a reducir el miss ratio. Podrá ser implementada a nivel de hardware o de software.
De hardware: traer más de un bloque del nivel inferior e introducirlo en cache o en un buffer de acceso rápido.
La arquitectura dispone de instrucciones específicas para ir trayendo información a la cache "en avance". El compilador es el responsable de disponer de estas en el código. En principio requerimos una arquitectura superescalar, esto es avanza el programa por un lado y la carga en cache por otro. Además la cache "debe" ser no bloqueante.

Reducción del hit time

Hasta ahora a la cache la accedemos con una dirección física, esto es, disparando un acceso a memoria que también pasara por cache que en caso de hit anula el acceso a memoria, de ser un mis dicho acceso se efectiviza. ¿Que ocurrirá si empleo direcciones virtuales y no físicas?

Al no tener translación mejora el hit time

La interfaz se maneja de dos formas
1) Overlay. Técnica puramente de software
2) Memoria virtual, cuyo manejo es automático transparente al programador.

Trabajar con cache virtuales, vale decir se accede a la cache con direcciones lógicas (virtuales) y no físicas, reduciendo de tal forma el tiempo de acceso. Se obvia el mecanismo de translación.

Memoria Virtual

Un sistema tiene memoria virtual si en tiempo de ejecución existe una función de translación del espacio virtual V al espacio físico M tal que:
F: V→M = X si la dirección lógica de Y mapea a la dirección física X. Vale decir esta en memoria principal. En caso contrario, esto es, NO está en memoria principal la dirección V, se produce un "fault", faltante que desencadena una excepción (interrupción) al sistema. Esa interrupción es la parte de requerimiento de arquitectura. Esta no es cualquier interrupción, es de las más complejas para manejar. ¿Por qué?
1. Porque se atiende en el medio de una instrucción
2. Porque tiene que ser precisa vale decir que se retenga el estado del procesador a la instancia previa.

Ventajas de la memoria virtual

- Posibilita que no todo el programa resida en memoria principal, luego independiza a este del espacio último de memoria disponible.
- Facilita la multiprogramación a partir de que no necesito que los programas estén completos en memoria
- Comienzan a ejecutar más rápido (cargo menos programas a la memoria)
- Se asocia el mecanismo de protección al mecanismo de translación realizando un control de acceso con la granularidad de página o segmento.

Desventajas de la memoria virtual

- Overhead a nivel de CPU y recursos (disco, bus) para soportar la interfaz Memoria principal-Memoria secundaria.
- Retardo para pasar de dirección lógica a dirección física. Halla o no halla memoria virtual

Se producen tres planteos:
1. Donde se ubica el bloque en memoria principal. Dado lo costoso que resulta traer un bloque de memorias secundaria el planteo es full asociativo, en cualquier posición.
2. Como se detecta un hit (mecanismo de translación). Resulta critico este proceso dado que en cada referencia a memoria se debe realizar la translación.
3. Algoritmos de reemplazo.
El manejo de alocación de memoria primaria (PMA) para una tarea toma varias formas dependiendo de las elecciones de diseño. Estas opciones son:

- Si las tareas son particionadas en áreas fijas o variables
- Si los algoritmos de reemplazo son locales o globales. Local si para el reemplazo interviene solo la aplicación del fault. Global si interviene todo el sistema. Fija si la asignación (PMA) no cambia en toda la ejecución del programa. Variable en caso contrario. El mejor escenario será variable y global y no local y fija.

Sea $C_t = \{p_1, p_2, \ldots, p_i\}$ el conjunto de páginas residentes en el tiempo t para una tarea dada. Si $r_i = p_j \notin C_i$, luego debemos cargar p_j en memoria principal.

- **Reemplazo Aleatorio (local, particiones fijas).** Un p_i es elegido en forma aleatoria de C_t cuando se necesita un reemplazo. No hace análisis alguno, valido solo si la aplicación lo alimenta. Para este algoritmo no es necesario hardware extra y el trabajo del SO es mínimo.
- **Reemplazo First In- First Out (FIFO).** Las páginas en C_t son ordenadas al orden de llegada. Cuando se llena C_t, la primera página cargada es la primera en ser reemplazada. No es necesario hardware extra, el SO debe mantener una cola de las páginas cargadas. El overhead de un page fault, que es la inserción y borrado de la cola, es mínimo comparado con el tiempo que toma traer una página de memoria secundaria. Un problema de este es que una página que es referenciada muchas veces puede ser reemplazada con demasiada frecuencia.

- **Reemplazo Clock o First In- Not Used First Out (FINUFO).** Este trata de resolver el problema que tiene el FIFO. Mantenemos una cola FIFO como antes con la adición de un bit de uso y una cola circular. Cuando ocurre un fault nos fijamos en la cola la primer página con el bit de uso apagado. Si no lo está, se apaga y se posiciona en la siguiente en la lista, así hasta encontrar una página que tenga el bit apagado. Para implementar este algoritmo necesitaremos un bit en hardware para cada frame. Este bit es seteado en cada referencia a memoria de este frame. Esto puede ser hecho durante el proceso de mapeo, si el bit es colocado en las tablas de mapeo. Por lo tanto no habrá un overhead de tiempo y el hardware extra es poco. La función del SO es simple. Si bien toma en consideración si fue o no referenciada, no analiza en que tiempo.
- **Reemplazo Last Recient Use (LRU).** El problema del algoritmo anterior es que no reordena las páginas de acuerdo a *n*° de referencias recientes. Podemos lograr esto mediante el uso de una pila, la cual es reordenada en cada referencia. Cuando se referencia una página que se encuentra en memoria, se busca en la pila, se saca, se mueven todas la páginas una posición para abajo y se coloca esta al tope de la pila. Si la página referenciada no se encuentra en la pila, la página del fondo de esta es la elegida para el reemplazo, se elimina de la pila, se mueven todas las restantes una posición hacia abajo y la página traída a memoria se coloca al tope. El algoritmo LRU puro es impráctico, ya que el orden en la pila cambiará con cada referencia a memoria. Cada movimiento en la pila implica un acceso a memoria y el overhead es inaceptable, a menos que la pila sea almacenada en un buffer de memoria rápido. En este caso el corrimiento de la pila puede ser hecho en paralelo con la referencia a memoria.
- **Reemplazo Óptimo en particiones fijas (Mínimo).** Es teórico, no practicable dado que se basa en conocer en futuro. Lo que hay que optimizar en general es el producto espacio*tiempo. Como el PMA es fijo nuestro factor a optimizar es el tiempo, luego la optimalidad es definida como el mínimo *n*° de page fault. Cuando ocurre un page fault y el C_t está completo, hay que elegir una página para reemplazar, ya que usamos particiones fijas para el C_t. Nos valemos de la información del futuro próximo y elegimos aquella página perteneciente a C_t que sea la última en ser referenciada de las demás en este futuro próximo. Tanto este algoritmo como LRU satisfacen la propiedad stack. Asegura que si aumenta el PMA, con estos algoritmos el mínimo de page fault en un dado instante será menor (o igual) del que se tenía previamente nunca superior, cumplen con el principio de inclusión. Naturalmente esto no es aplicable en tiempo real y es de interés para proveer una medida para los otros algoritmos.
- **Reemplazo Working Set.** Ha tenido el mismo impacto sobre las políticas variables como LRU ha tenido sobre las fijas. Sea T el tamaño de una ventana, o sea, un intervalo (virtual) de tiempo expresado en número de referencias. El algoritmo en el tiempo *t* para una ventana T es el conjunto de páginas las cuales han sido referenciadas en el intervalo $(t - T + 1, t)$, que es:

$$WS(t, T) = \{ p_i \mid r_u = p_i, \quad (t - T + 1) \leq u \leq t \}$$

La página es reemplazada (borrada de MP) en el tiempo *t* si no pertenece a $WS(t, T)$. Notar que una página no es necesariamente reemplazada en un fallo de página.

Ejemplo: asumamos una ventana de 4 referencias:

WS					*g*		*g*	*g*	*g*	*g*	*g*	*c*	*b*	*g*
				d	*d*	*g*	*f*	*f*	*f*	*f*	*f*	*g*	*c*	*b*
		b	*b*	*b*	*b*	*d*	*d*	*d*	*d*	*d*	*d*	*f*	*f*	*c*
	a	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*f*
tiempo	1	2	3	4	5	6	7	8	9	10	11	12	13	14
page fault	*	*		*	*		*					*	*	*
secuencia	*a*	*b*	*a*	*d*	*g*	*a*	*f*	*d*	*g*	*a*	*f*	*c*	*b*	*g*
tamaño WS	1	2	2	3	4	3	4	4	4	4	4	4	4	4

- **Reemplazo Frecuencia de Fallo de Página(PFF, local o global, variable).** Esta regla es un intento de tener una alocación de memoria principal que siga las variaciones en las localidades. Está definida como la recíproca de T´, y monitorea el tiempo de fallo entre páginas (T´ expresado en las mismas unidades que la ventana T del algoritmo WS). En un page fault, si la PFF es más grande que un valor predefinido P, no hay reemplazos y el PMA se aumenta en uno para cargar la página faltante. De otra manera, si la PFF es más chica que P, se descargan todas las páginas no referenciadas desde el último page fault.

  La principal diferencia entre PFF y WS está en el lugar en que se toman las acciones cuando ocurre un fallo de página, ya que en esencia T representa el límite inferior del tamaño (variable) de la ventana.

4. Qué hacer ante un write miss. La única política posible es write back, debe considerarse que al disco se lo accede por sectores nunca por palabras individuales.

Hay algunas diferencias entre los caches y la memoria virtual, más allá de las cualitativas:

- El reemplazo en el cache es primariamente controlado por hardware, mientras que en memoria virtual es controlado por el SO. Un miss penalty más grande significa que el SO puede involucrarse y gastar más tiempo decidiendo cuál reemplazar.
- El tamaño de las direcciones del procesador determina el tamaño de la memoria virtual, mientras que en cache es normalmente independiente de las direcciones del procesador.

Organización de la memoria virtual

- Paginado
- Segmentado
- Sementado páginado

Sistemas Paginados

El sistema paginado usa bloques de tamaño fijo, donde la dirección virtual está dividida en un número de página y un desplazamiento.

VPN	OFFSET

V.P.N : Virtual page number.
El dispositivo de mapeo debe trasladar la dirección de entrada (virtual) de bloque, en una dirección real y si este no está en Memoria principal, debe generar un page fault. Hay tres implementaciones para la translación de direcciones:

- **Mapeo Directo**. Aquí el dispositivo de translación consiste en n entradas correspondientes a la cantidad de páginas virtuales posibles ($|V| / p$, donde p es el tamaño de página). La entrada i de la tabla contiene la dirección de entrada de la página real, o frame, en el cual la página virtual i es mapeada, o una indicación de page fault en caso contrario. Si tenemos tamaño de páginas $p = 2^n$, una dirección virtual efectiva de n bits puede ser descompuesta en un número de página (los $n - k$ bits más a la izquierda) y en un desplazamiento (los k bits más a la derecha). El dispositivo traslada los números de página a un número frame, el cual es concatenado al desplazamiento para obtener la dirección real. Dado el requerimiento de tiempo debería estar implementado en hardware, no en memoria. Sera en tal caso de aplicación solo si el número de entradas resulta acotado.

- **Mapeo Asociativo**. Aquí el dispositivo contiene pares (x, y), donde x es el número de páginas (VPN) e y es el número de frame (FP) en la memoria física. La búsqueda es hecha por contenido, es decir, se busca x para encontrar y. Es por esto que se necesita de una memoria asociativa. Se necesitan tantas entradas a la tabla como frames existan en la memoria física (|M| / p, donde p es el tamaño de página).

Debido a las desventajas de estos dos métodos se usa el tercer método **híbrido**. Este dispone una pequeña memoria asociativa (TLB) que contendrá el par frame-page (x, y) más probable a ser referenciado. Y la tabla de páginas en memoria principal contendrá el resto del mapa. Un registro (PTP) apunta al comienzo de la tabla de páginas del proceso actual. La búsqueda se inicia en el TLB. En paralelo también se realiza en la tabla de páginas. Si hay hit en el TLB sacamos el frame (puntero a la dirección física al comienzo de la página) y cancelamos la búsqueda directa (en la tabla). En caso de que no se encuentre en los TLB, se espera por el resultado de la búsqueda directa. Si ambos fallan ocurre un page fault y se le da el control a una nueva tarea.

Tabla de Páginas

Problemas que acarrea la tabla de páginas:

- Muchos recursos de memoria principal empleados por la página de páginas.
- Problema de fragmentación externa a nivel de tabla. En el sentido de que es un área contigua en la cual se indexa con VPN (número virtual de página). Se requiere un espacio contiguo de tamaño suficiente.

Lo primero se maneja permitiendo que la tabla de páginas se encuentre paginada, esto es, parte en memoria y parte en disco.
El otro punto queda resuelto organizando la tabla de páginas en forma jerárquica, no en único nivel.
Otra solución a los temas planteados es el empleo de una tabla de páginas invertida. De lo que se trata es de englobar en una única estructura la información de todas las páginas, sin distinción de procesos, residentes en

memoria. La tabla estará acotada por el tamaño de la memoria principal del sistema y el tamaño de página, no existiendo tampoco el problema de manejar áreas contiguas. Se usa una técnica de almacenamiento a través de una técnica de hash y se realiza una búsqueda en forma asociativa.

Thrashing

Un manejo global de memoria está sujeto a "thrashing", o sobrecompromiso de las locaciones de memoria principal. Si el nivel de multiprogramación es demasiado alto, las páginas serán tomadas del conjunto del programa más recientemente usado. Pero estas páginas pronto tendrán que ser rellamadas puesto que este programa será el siguiente en ser ejecutado. Incluso si la carga está bien controlada, una tarea puede perder algunas de sus páginas durante las transiciones. Por esto se necesitan controles de localidad y de carga.

Sistemas Segmentados

La segmentación es una técnica para manejar la alocación del espacio virtual, mientras que la paginación es un concepto usado para manejar la alocación del espacio físico. Un segmento es un conjunto de elementos de datos contiguo relacionados lógicamente. A los segmentos les está permitido crecer y contraerse arbitrariamente, a diferencia de las páginas las cuales tienen un tamaño fijo.
El sistema segmentado tiene direcciones de la forma (Si, Dj), donde Si es el *n*º de segmento y Dj es el desplazamiento en el segmento. Asociado con cada tarea hay una tabla de segmento que nos da la translación entre el *n*º de segmento y la dirección real. Un registro (STP) contiene la dirección de la tabla de segmento para el proceso actual. El proceso de translación es así: se tiene la dirección virtual (Si, Dj), donde Si es sumado al contenido del STP para apuntar a la entrada correspondiente en la tabla de segmento. Un flag indica si el segmento está en memoria principal. Si es así, se realiza un chequeo para determinar si Dj es más grande que Li (longitud de segmento).Si este es el caso una rutina de error es invocada y el programa es abortado. Si no es así, se chequean los bit de protección. Asumiendo que la operación es válida, luego Dj es sumado (no concatenado) a la dirección real, encontrada en la entrada de la tabla, para conseguir la dirección efectiva.
El mecanismo es un poco más complejo que el de la paginación, debido que involucra una suma.
Consideremos el caso que un segmento no está en memoria principal. La primera tarea es chequear si hay espacio para el segmento, esto es, si existe en memoria principal un área libre de tamaño Li. Se mantiene información del sistema generalmente en dos listas enlazadas, una para los segmentos reservados y otras para los segmentos libres. Cada segmento libre también tiene información respecto a su tamaño. Para determinar si es posible reservar un área de memoria principal utilizamos un algoritmo de reservación (Ej. First Fit). Si esto es posible, el área de memoria es alocada al segmento. Se actualiza la tabla de segmentos y la lista de segmentos reservados. Si la reservación falla tenemos la opción entre "compactación" y reemplazo.

Sistemas Segmentados con Paginación

Aquí los segmentos son divididos en páginas. Una dirección virtual es de la forma (S_i, P_j, D_l), donde S_i es un *n*º de segmento, P_j un *n*º de página en el segmento y D_l un desplazamiento dentro de la página. Un segmento consiste de una o más páginas. El mecanismo de translación consiste en: un registro STP que apunta a la tabla de segmentos actual, si corresponde a la entrada en la tabla. Luego asociado con esta entrada hay un puntero a la tabla de páginas. La *j*-ésima en la tabla de páginas indica si existe el correspondiente frame o si se produce un page fault. En el primer caso la dirección real es finalmente computada concatenando el desplazamiento con la entrada en la tabla de páginas. En el caso de fault, se maneja como en sistemas paginados.
Observemos que la translación de una dirección es hecha a través de dos referencias indirectas en lugar de una, como en sistemas puramente paginados o segmentados. Por lo tanto la penalidad puede ser extraordinariamente severa a menos que usemos algún tipo de mapeo asociativo y/o registros fast buffers.

Se plantean aquí dos situaciones extremas:
1. Segmentación lineal. Básicamente es un sistema paginado y los segmentos al solo efecto de controlar el alcance del acceso (tamaño). La VAX 780 es un ejemplo de esto. En un acceso a nivel del segmento solo se chequea si excede o no del tamaño previsto y la protección se resuelve a nivel de página (4 bits).
2. Name segment. Claramente es un sistema segmentado solo que se paginan los segmentos.

Paginación vs. Segmentación

El paginado fue concebido para alcanzar un buen manejo de memoria física; es efectivamente en dicho manejo donde saca ventajas.

Al ser de tamaño fijo, siempre que existan frames libres en memoria principal podrán ser cargadas las páginas en memoria.
En cambio la segmentación fue pensada para mejorar el manejo del espacio lógico y claramente allí resultara superior. Dado que los segmentos son de tamaño variable darán lugar a "fragmentación externa" a nivel de memoria principal. Esto es espacio de memoria de tamaño suficiente para el segmento libre pero no contigua. Esto torna a la segmentación muy poco dúctil para ser el mecanismo de memoria virtual, si con la tercera alternativa de segmentado páginado.
En un sistema paginado hay fragmentación interna, desperdicio de memoria a nivel de la última página. Si no existe fragmentación externa, es el mejor escenario para asignar espacio fuera de memoria.

¿Qué pasa con el espacio lógico?

- Compartir código/dato
- Expandir dinámica áreas

Son cuestión que se pueden manejar con paginado per no sin ciertas salvaguardas. Vale decir no lo hace de forma natural como el segmentado.

PTP= Page table pointer

Un sistema paginado se lo entiende como lineal o "unidimensional". Tanto A como B para acceder a C lo deben hacer a través de su tabla de páginas.
Luego las tablas de páginas deberán tener "huecos" en esas posiciones. Se dice que deben ser "ralas".

Un área podría llegar a invadir un espacio ya asignado con lo cual la ejecución no podrá proseguir.
Si el sistema es segmentado nada de esto ocurre (con el espacio lógico, dado que el sistema es multidimensional). Cada segmento define un espacio distinto (es por esto que no pueden colisionar))
Puedo desde una tabla de segmentos invocar a C con un número de segmento X (bastaría determinar la dirección de comienzo en memoria del segmento) y por otro lado otra podría invocar a C con un numero de segmento Y funcionando correctamente.

Dado que son espacios lógicos distintos no podrán interferirse.
¿Qué puede pasar en memoria?
Que un segmento variando requiera espacio contiguo.
El proceso de linkeado se verá simplificado con respecto a un sistema segmentado al obviar la realocación del caso del un sistema paginado.
En cuanto a espacio lógico alcanzable, en un sistema paginado la dirección está dada por una palabra; luego la cantidad de bits de esta determina el espacio dispone a direccionar.
Esto marco la "fuerte" limitación de las arquitecturas de 16 bits como la PDP.
En un sistema segmentado la realidad puede ser muy otra dado que una dirección se arma con dos palabras: dirección base del segmento y desplazamiento dentro del segmento.

Memoria Principal

Después del cache, la memoria principal es el próximo nivel bajando en la jerarquía. La memoria principal satisface las demandas del cache, y sirve como interface de E/S, así sea como destinatario de la entrada o como fuente de la salida.
Podemos distinguir dos atributos en cuanto a performance:

- Tiempo de acceso
- Velocidad de transferencia

Además de hablar de un tiempo de acceso, el que media entre el requerimiento y efectivamente hacerse del dato, corresponde hablar de un tiempo de ciclo; este es el que a nivel básico, condiciona el throughput. Este es el tiempo que debe mediar entre dos accesos consecutivos.
Paradójicamente la evolución de estos tiempos en el caso de memoria se parece a la evolución en el caso de los discos, esto es, el tiempo de acceso evoluciona muy lentamente (1/3 cada diez años), en cambio la velocidad de transferencia en ambos casos lo hace mucho más rápido.
En general se hacen esfuerzos para mejorar "lo mejorable", esto es, el throughput asumiendo que la latencia es algo que no se pueda atacar, más aún, alternativas de organización que apuntan a mejora de throughput pueden llegar acompañadas de cierto prejuicio en el tiempo de acceso.
El tiempo de ciclo condiciona el throughput ¿Cómo se puede mejorar?
a) Palabras más anchas
b) Interleaving de memoria

a) **Palabras más anchas**. Los caches son usualmente organizados con un ancho de una palabra debido a que la mayoría de los accesos de CPU son de este tamaño. La memoria principal es de este tamaño para coincidir con el cache. Doblar o cuadriplicar el ancho de memoria doblará o cuadriplicará el ancho de banda de memoria. Existe un costo en un bus más ancho. El CPU seguirá accediendo a cache de a una palabra, entonces se necesita un multiplexor entre el cache y el CPU y este multiplexor puede ser un tiempo crítico en el camino. Otro problema es que ya que la memoria es expansible por las personas, el mínimo incremento es doblarla o cuadriplicarla. Finalmente, memorias con corrección de errores tienen dificultades con las escrituras en una porción del bloque.
b) **Interleaving de memoria**. Si organizamos la memoria en más de un modulo, c/u con su controlador, si logramos que trabajen en forma concurrente mejoraríamos el ancho de banda.
La pregunta es como direccionar a los distintos módulos. Hay tres alternativas:

- High order
- Low order
- Mixto.

-b-	
H.O.	

dirección a memoria

Con 2^b numero de módulos

	-b-
	L.O.

dirección a memoria

Con 2^b numero de módulos

Low order
Es el que en general proporciona mayor throughput. Esto porque dada la localidad secuencial podre alcanzar el objetivo de concurrencia.
Como desventaja es poco modular y no resiste la falla de un modulo.

High order
Contrariamente no posibilita mayor ancho de banda a nivel de un procesador; no obstante en un esquema multiprocesador en el cual cada nodo tenga una porción de la memoria (DSM distributed shared memory) se podrá direccionar al modulo de cada nodo con los bits superiores y en tal caso el objetivo de concurrencia si se hace posible. Tanto más cuanto más predominen accesos locales frente a los remotos.

Al contrario del Low order, es modular y tolerante a la falla de algún modulo. Perfectamente puedo trabajar con módulos de distinta capacidad. Si se cae un modulo si he organizado bien el sistema podre alcanzar continuidad desafectando al modulo en falla.

Mixto
Busca obtener los beneficios de ambos esquemas.

b-n		-n-
H.O.		L.O.

2^{b-n} grupos de 2^n módulos c/u

Este nuevo concepto se podrá aplicar a nivel interno del modulo. Bajo esta política direccionamos al banco con los bits inferiores (low order).

Este esquema multibanco aumenta el ancho de banda y reduce el tiempo de ciclo a la mitad (si hay dos bancos). Además con el aumento del nivel de integración se lleva a un nivel más bajo, esto es, propio del chip de memoria.

Tecnología de memoria

Las hay de dos tipos MOS estática SRAM y MOS dinámica DRAM.
En la SRAM la celda es un biestable con 6 a 8 transistores. La lectura se hace sensando el estado del flip-flop y el tiempo de ciclo es prácticamente el tiempo de acceso. La lectura NO es destructiva.

En la DRAM la celda es meramente un pequeño capacitor que se carga a uno u otro nivel. (V_H – V_L) y un único transistor que lo conecta.

Esto hace que la DRAM pueda alcanzar mucha mayor capacidad, por ende menor costo.
La operación de lectura requiere que previamente se cargue la línea de dato a un potencial intermedio. Luego cuando T conecta se tendrá un escalón de tensión para arriba o hacia debajo de esa tensión de referencia, lo cual será censado.

La lectura resulta destructiva por efectos secundarios. La regeneración aquí es una fracción del tiempo de acceso. Refrescar implica leer y regenerar.

Direccionado

Hay tres esquemas posibles:
1) Direccionamiento lineal
2) Direccionamiento codificado
3) Direccionamiento multidimensional

1) Direccionamiento lineal. Cada lugar recibe una línea de selección que si se activa indica que se refiere a él. Es simple pero la dificultad puede darse a nivel del layout. Cada línea a un solo lugar.
2) Direccionamiento codificado. En este las líneas de dirección viajan codificadas a los distintos lugares. Esto simplifica el layout, n líneas en lugar de 2^n. Luego se necesita decodificar en cada lugar (lo cual es complejo) y otro inconveniente es que pasa por todos los lugares a direccionar (aquí cuestiones de fan out).
Este es el esquema que se emplea en la utilización de un bus. En cambio para direccionar memoria es el lineal el aconsejable. Los SRAM en principio usaban este esquema, con los aumentos de densidad se vio presionado por cuestiones de layout a emplear direccionamiento multidimensional, por cuestiones de refresco.
3) Multidimensional o por coincidencia. Se organizan las celdas en filas y columnas (básicamente con igual dimensión)

Se gana a nivel de layout en qué lugar de 2^n como lineal se tiene $2.2^{n/2}$. Por caso. Si n=20 paso de 1000000 a 2000.
El direccionamiento se produce en la coincidencia de dos líneas. El fan out se potencia $\sqrt{(2^n)}$.
Hoy en día se habla de SDRAM (Sychrounous dynamic RAM). Visto el acceso fila/columna de estas memorias y habida cuenta de lo más costo que resulta es la activación de la fila, comparado con la selección de la columna, es posible tomar ventaja si producido el acceso a la fila se producen diversos accesos que se resuelven sin cambiar de fila, lo que mejora el throughput. Cabe aclarar que esto se puede potenciar con multibanco.
Para aumentar el ancho de banda (las memorias envían/reciben de a 64 bits) dado una cierta frecuencia (del BUS y de la memoria).
Se concibieron las DDR (dual data rate)

En DDR2 se sale al bus al doble de frecuencia de la memoria y claramente trasmitiendo en ambos bancos.

Las DDR usan ambos flancos del reloj, de esta manera la frecuencia efectiva de transmisión es el doble. Del lado de la memoria se emplea paralelismo para abastecer la mayor frecuencia. En este caso, DDR, un dato doble. Esta muy bien se podría haber denominado DDR1.
Este concepto (paralelismo en la memoria para un mayor ancho de banda) se ha ido potenciando a partir de incorporar DDR2 primero, DDR3 luego y actualmente se trabaja en DDR4.
Se logra de tal forma con frecuencias de memoria contenidas (facto potencia) aumentar sucesivamente el ancho de banda.

Para acceder a la DRAM debo "activar" previamente una fila. Esto va precedido de un proceso de "precarga"; luego el censado y posterior "latching" de toda la fila.
El acceso se completa seleccionando una columna. Como se deja no se limitara a una palabra sino que involucra un conjunto de palabras (4-8) para favorecer el throughput.

Ejemplo

2_ not(CAS) latency (CL). El CL es el Tiempo de Acceso si la fila ya esta activa. Si no es el tiempo que demora en activar la fila y la columna y tiempo hasta que obtengo los datos

3_ T_{RCD} not(RAS) to not(CAS) delay

2_ T_{RP} not(RAS) precharge

8_ T_{RAS} el más largo

1_T command rate (1 o 2 ciclos)

Si quiero activar una nueva fila deberé iniciar un proceso de precarga. Para ello deberé respetar un retardo T_{RAS} (active to precharge delay).

Capítulo 5 – Diseño de Control

Introducción

Los sistemas digitales se pueden separar en dos partes: una unidad de procesamiento de datos, que es una red de unidades funcionales capaces de desarrollar ciertas operaciones sobre el dato, y la unidad de control. Esta tiene como propósito mandar señales de control o instrucciones a la parte que procesa el dato. Estas señales de control seleccionan la función a ser realizada en un tiempo específico y rutean el dato a través de la unidad funcional apropiada.
Dado un set de instrucciones y el diseño de un datapath, el próximo paso es definir la unidad de control. Esta le dice al datapath que hacer en cada ciclo de reloj durante la ejecución de instrucciones.
Su función es traer instrucciones de memoria e interpretarlas para determinar las señales de control que deben ser enviadas a las unidades de procesamiento de datos. Dos aspectos centrales de estos procesos pueden identificarse:

- ***Secuenciamiento de la instrucción***, son los métodos por los cuales las instrucciones son elegidas para ser ejecutadas.
- ***Interpretación de la instrucción***, métodos utilizados para activar las señales de control que causan la ejecución de la instrucción en la unidad de procesamiento de datos.

La unidad de control puede ser especificada por un diagrama de estados finitos. Cada estado se corresponde con un ciclo de reloj, las operaciones ha ser realizadas durante el ciclo de reloj son escritas en el estado. El próximo paso es llevar este diagrama de estados al hardware.
Una forma de medir la complejidad del control está dada por:

Estados * Entradas de Control * Salidas de Control

Existen dos enfoques en el diseño de las unidades de control.
El primero ve a la unidad de control como un circuito lógico secuencial que genera específicas secuencias fijas de señales de control. Las metas de este enfoque es el de minimizar el *n*º de componentes usados y maximizar la velocidad de las operaciones. No se pueden realizar cambios en el diseño. Si se quiere rediseñar se debe modificar y volver a cablear el circuito. En este enfoque se dice que es una ***unidad de control cableada (hardwired control)***. Debido a que las unidades de control son los circuitos más complejos en una computadora, se dice que las unidades de control cableadas son costosas de diseñar y depurar.
El otro enfoque (propuesto por Wilkes) es la ***microprogramación***. Las microinstrucciones se almacenan en una memoria especial llamada memoria de control (CM). Microinstrucciones para distinguirlas de las instrucciones maquina. La operación es realizada trayendo las microinstrucciones, una a la vez, de la CM y usándolas para activar las líneas de control directamente. La microprogramación hace el diseño de la unidad de control más sistemática a través de organizar las señales de control en palabras (microinstrucciones) que tiene un formato bien definido. Las señales son implementadas por un tipo de software(firmware) en vez de hardware, por lo tanto el diseño puede ser fácilmente cambiado, planteando una modificación del microcódigo. El lado negativo, las unidades de control microprogramadas son más costosas que las unidades cableadas por la presencia de la CM. También es más lenta debido al tiempo extra requerido para traer la microinstrucción de la CM.

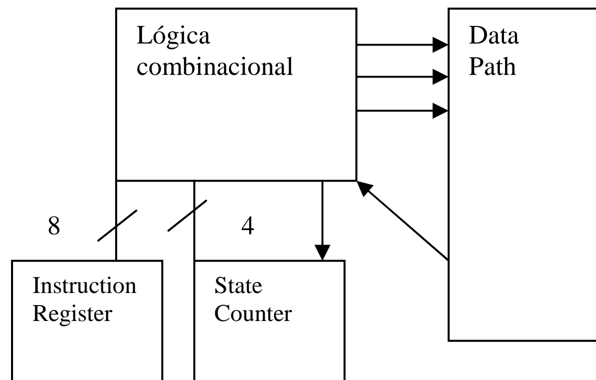
Control Cableado

El diseño de una unidad de control cableada involucra varios compromisos complejos entre la cantidad de HW usado, la velocidad de la operación y el costo del proceso de diseño por si mismo. Los dos métodos más comunes son:

1) Método clásico de diseño de un circuito secuencial
2) Método de contador como generador de secuencias

1) Tabla de estados y un diseño que optimiza hardware más allá del tamaño de dicha tabla. Se dice da lugar a una lógica Random. Random en el sentido de que resulta difícil correlacionar controlado con controlador, tomando la difícil tarea de "debugging" de diseño.
2) En ese sentido el contador como generador de secuencia es superior. Resulta evidente esta relación. Se emplea un contador de modulo tal que permita codificar los distintos estados que podrá atravesar la ejecución de una instrucción. El contador a partir de la conjunción de en qué estado esta y de que

instrucción se esta ejecutando determina a partir de un circuito combinacional, que se hace en el data path y que respecto al próximo estado del contador.



Control cableado usando contador

Como se implementa la lógica combinacional? ROM y mejor que ROM PLA. (Por esas estructuras generales que favorecen la integración).

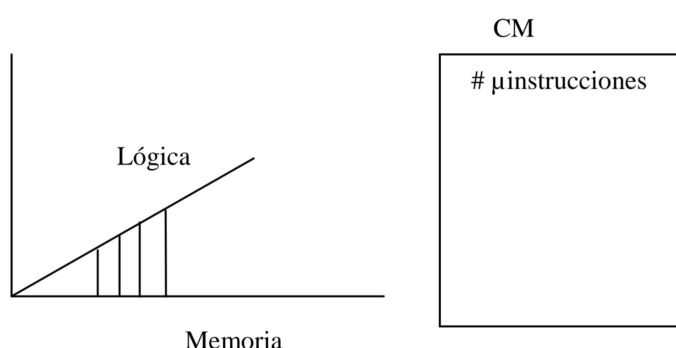
Performance del control Cableado

Al momento de diseñar el control para una máquina se quiere minimizar el:
* CPI promedio,
* El ciclo de reloj
* Y la cantidad de hardware.

El CPI se minimiza reduciendo el número de estados a lo largo del path de ejecución de una instrucción puesto que cada ciclo de reloj corresponde a un estado. Esto se logra típicamente realizando cambios al datapath para combinar o eliminar estados.

Control Microprogramado

Cada instrucción en una CPU es implementada por una secuencia de uno o más conjuntos de microperaciones concurrentes. Cada microperación es asociada con un conjunto específico de líneas de control, las cuales, cuando son activadas, causan que la microperación tome lugar. Ya que el número de instrucciones y líneas de control son a menudo unas cientos, una unidad de control cableada que selecciona y secuencia las señales de control puede ser extremadamente complicada. Como resultado, es muy costoso y dificultoso diseñarla.
La microprogramación es un método de diseño de control en el cual la información del secuenciamiento y de la selección de las señales de control, es almacenada en un ROM o RAM llamada memoria de control (CM). Las señales de control pueden ser activadas en cualquier momento en que son especificadas a través de la microinstrucción, la cual es traída de la CM. Cada microinstrucción también específica explícitamente o implícitamente la próxima microinstrucción que será usada, por lo tanto provee la información necesaria para el secuenciamiento.
En una CPU microprogramada, cada instrucción máquina es ejecutada por un microprograma el cual actúa como un interprete en tiempo real para la instrucción. El conjunto de microprogramas que interpretan un conjunto de instrucciones en particular o lenguaje *L* es llamado un *EMULADOR* de *L*.
La invención de la microprogramación posibilitó que el conjunto de instrucciones pueda ser cambiado alterando el contenido de la CM, sin tocar el hardware.
El control microprogramado es más sistemático, la cuestión es que ha perdido vigencia, no porque el cableado sea ventajoso, sino porque resulta inviable por cuestiones de velocidad.
(A partir de mediados de los 80's comenzó a hacer crisis la microprogramación).
En los 70's los tiempos de acceso de esa memoria de control no desentonaban con los retardos de la lógica.
Luego si así se lo disponía, resusltaba razonable reemplazar el control por hardware por un control por software.



"RISC no es sinónimo de control microprogramado" Cuando surge RISC se buscaba cosas simples y la microprogramación era sofisticada". "Luce como CISC pero adentro es RISC, el control es cableado".

Formato de µinstrucciones (Básico)

En el esquema original, la microinstrucción tenía cuatro partes:

- Condición del salto
- Dirección del salto. Aquí no se especificaba toda la dirección por razones de velocidad. En su lugar, se codifica una serie de bits del µaddress que sustituyen al original si salta.
- Campo de control, el cual indica las líneas de control que serán activadas.
- Próxima µdirección, el cual indica la dirección en la CM de la próxima microinstrucción que será ejecutada.

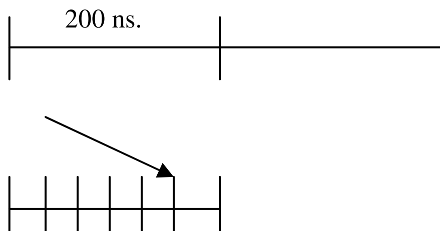
Condición del salto	Dirección del salto	Campo de control	Proxima µaddress

Aquí, cada bit k_i del campo de control corresponde a una línea distinta de control C_i. Cuando $k_i = 1$ en la microinstrucción actual, C_i es activada, sino permanece inactiva.
En este primer esquema, la *CM* estaba organizada como una ROM, compuesto de un PLA (matriz de diodos). La parte izquierda de la matriz contiene los campos de control de cada microinstrucción, mientras que la parte derecha contiene los bits del campo de direcciones. Las filas de la CM representan las microinstrucciones y las columnas representan cada línea de control o línea de dirección. Un registro llamado Control Memory Address Register (CMAR), almacena la dirección de la microinstrucción actual. También hay un switch *S*, que permite responder a señales externas o condiciones. Este *S* permite seleccionar uno de los dos posibles campos de direcciones.

En cuanto al secuenciamiento hay 2 formas:

1) 1 µPC (ejecución secuencial)
2) Que cada instrucción tenga la dirección de la próxima (rutina)

Se opera sincrónicamente con µinstrucciones de tamaño fijo. Esta la posibilidad de simple fase o múltiple fase.
Fijado un periodo si solo hay un cambio de estado al final del ciclo: simple fase (ejecución).
Si por el contrario divido al periodo en fases podre tener múltiples cambios de estado resultantes de la ejecución de una µinstrucción (me estaría secuenciando dentro de ella).



Tarea divida en un ciclo. Con un contador moebius se decidiría un periodo dado en equisparticiones

Ej. 1 µPila
Si tengo simple fase: 1 pop, 1 pop, la operación en sí, 1 push
En cambio si cuento con múltiples fases, en un solo ciclo hago todo involucrando una sola µinstrucción.

Muchas modificaciones se han propuesto a este diseño básico. La mayor área concierne a la longitud de la palabra de la microinstrucción. La longitud de la microinstrucción está determinada por:

- El *n*º máximo de microperaciones simultaneas que se pueden especificar (grado de paralelismo).
- La manera en la cual la información de control es representada o codificada.
- La manera en la cual la dirección de la próxima microinstrucción es especificada.

En una unidad de control microprogramada el costo de hardware se mide por el tamaño de la CM y la performance se mide por el CPI.

Reducir el tamaño de la Control Memory (CM)

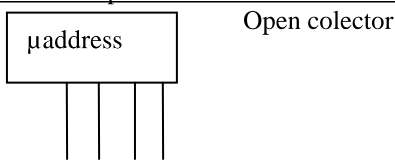
1) Reducir a lo ancho (µinstrucciones de menor tamaño)

2) Reducir a lo alto (menos numero de µinstrucciones)
1) Codificar las líneas de control. En lugar que un bit se asocie a cada línea de control, codificar y luego vía decode implementar las distintas líneas.

Para esto se puede tomar un conjunto de líneas de activación disjuntas, almacenarlos codificados y luego, vía decode implementarlos. No obstante, agrega retardo y condiciona a futuro (flexibilidad)

2) Combinar con control cableado. Por caso, Digital le agrego más bits al formato del µaddress para acotar el alto aunque se incremento el ancho.

VAX 780: Implementación de la dirección de salto



Si el estado es $V_h\,V_h\,V_h\,V_h$
Podría realizar saltos a un máximo número de 16 lugares. {No está fijado el salto, podría ir a cualquier otro lugar ante un mismo microaddress}
Ejemplo
De $V_h\,V_h\,V_h\,V_h$ a $V_l\,V_h\,V_h\,V_h$ o $V_h\,V_l\,V_l\,V_h$
Si dijera $V_l\,V_l\,V_l\,V_l$, no lo puedo subir es open collector. Si lo hago quemo el transistor.
"En la VAX se lo trabajaba por afuera al campo address"

Aplicaciones de la Microprogramación

Además de la interpretación de un dado set de instrucciones, la microprogramación también puede ser aplicada efectivamente a otras tareas de naturaleza interpretativa como la *emulación, ejecución directa de lenguajes de alto nivel* y *mejoras de aspectos específicos* de una dada arquitectura.

Emulación

La microprogramación, tal como ha sido vista, es la interpretación de un conjunto de instrucciones nativo. La emulación sería la interpretación de sets de instrucciones diferentes al set nativo. Esta definición puede no ser precisa para máquinas como Cel Data 100 en la cual no hay set nativo. De hecho es la emulación la que provee la compatibilidad entre tales máquinas.
La interpretación de código no necesita de la microprogramación, puede darse enteramente en SW, en cuyo caso se llama *simulación*. Luego el término emulación lo reservamos a los casos en donde es hecho principalmente a nivel de hardware, con algún soporte de SW. Un emulador microprogramado será luego un microprograma en una máquina ejecutando los 5 pasos del ciclo de ejecución de instrucciones de otra máquina. Para ser eficiente, el PC emulado, el PSW(palabra de estado del programa) y toda variable del estado del procesador emulado deberá estar en registros accedidos por el microcódigo de la máquina original.
Naturalmente, la velocidad de la emulación dependerá de la arquitectura de la máquina original y de la emulada. Si los sets de instrucciones se parecen y los tipos de datos básicos tienen igual longitud, la emulación será rápida; se puede concebir que una versión de una máquina emulada *A* en una máquina *B* puede ser más rápida que la versión original, si *B* es más rápida que *A* (ciclos de MP y CM más rápidos y mayor horizontalidad que *A*).
Lo visto para emulación es relativamente simple. Requiere del entendimiento de la instrucción a ser emulada y de las habilidades de microprogramación de la máquina en la cual se implementará. Cuando se incluyen I/Os e instrucciones privilegiadas en la emulación, se agrega otro nivel de complejidad (deben manejarse conversión de código, buffering, interrupción y excepciones).

Ejecución directa de Lenguajes de Alto Nivel

La traducción y posterior ejecución de un programa escrito en alto nivel, pueden adoptar dos técnicas. En un proceso de compilado, todo el programa fuente es traducido a lenguaje máquina. Luego esta versión o programa objeto, es ejecutado. En el proceso de interpretación, el fuente es ejecutado sin pasar por la fase de programa

objeto. Esto puede llevar a un procesamiento ineficiente, dado que una sentencia será analizada cada vez que se encuentre. Luego el intérprete es dividido en 2 partes:

- *Traducción*: transforma el fuente en un lenguaje intermedio,
- *Ejecución*: que computa directamente las cadenas generadas en el paso anterior.

Veamos que pasa cuando la máquina es microprogramada en el caso de interpretación. La rutina intérprete puede ser escrita en lenguaje máquina (a su vez ejecutada por el microcódigo), o puede ser microprogramada. En este último caso, se tiene lo que se conoce como *ejecución directa de lenguajes de alto nivel*. De esta forma saco ventaja frente a aquellos que ejecutan interpretes por software.

Pregunta de examen: Que dificultad tuvo Digital cuando llevo la VAX a un chip? Se le presento el problema de que las instrucciones de la memoria de control no le entraban en el chip. Resolvió que para aquellas instrucciones menos usadas se proveía un mecanismo de trap handler y las sustituía luego por tuinas de software manteniendo de esta forma la compatibilidad.

Sintonía de Arquitecturas

La microprogramación da la oportunidad al diseñador de modificar y expandir el conjunto de instrucciones clásicos (Von Neumann).

Esto se hace evidente en máquinas como la VAX 780, con instrucciones como CALL y otras.

En esencia, habría una ganancia en la velocidad de ejecución y un aumento de espacio en la memoria de control.

La sintonía del set de instrucciones consiste en monitorear el uso de las mismas, modificando el set nativo, evaluando la ganancia o pérdida en eficiencia, e iterativamente repetir este proceso hasta que alcance una mejora razonable.

Idealmente se podría hacer este proceso dinámicamente con máquinas con memoria de control modificables. El monitoreo podría hacerse en unas pocas microinstrucciones. El análisis del seguimiento de las instrucciones podría revelar cuáles secuencias pueden ser agrupadas.

Un generador de microprograma podrá procesar el microcódigo para tales secuencias e insertar las nuevas microrutinas en el microprograma.

La última acción sería reemplazar las secuencias originales por las nuevas instrucciones, interpretadas por las nuevas microrutinas. Sin llegar a este punto, se puede pensar en optimizar subprocesos. Por ejemplo, para sintonizar una línea orientada a editor de texto para una máquina dada, podríamos comenzar por monitorear los comandos más usados y las rutinas que son llamadas más frecuentemente.

Se deben tomar precauciones (como hace VAX) de que en estas instrucciones de ejecución, alternativamente extensas, se puedan atender interrupciones en otros puntos del ciclo de las mismas, además del fetch.

Capítulo 6 - Comunicaciones

Topologías de la comunicación entre Procesador y Memoria

La red de interconexión entre procesadores y memorias, puede tener varias topologías dependiendo de la concurrencia deseada en la transferencia y de la inversión en HW.

Cross-bar switch

Es el esquema más extenso y caro, ya que suministra rutas directas de procesadores a memorias. Con m procesadores y n módulos de memoria, se obtiene una concurrencia mínima de *mín.*(m, n). Necesita m . n interruptores y cuando m y n son del mismo orden de magnitud, el número de puntos de cruce crece como n^2. Como cada punto de cruce debe tener hardware capaz de intercambiar transmisiones paralelas y de resolver pedidos conflictivos para un dado módulo de memoria, el dispositivo de interconexión puede volverse rápidamente un factor dominante en el costo del sistema completo. Esta tendencia puede acelerarse en el futuro, ya que, con los avances en la tecnología LSI, los costos de las memorias y los procesadores disminuirán más rápidamente que la estructura de los interruptores.
Fueron usados en los multiprocesadores Burroughs. El sistema más representativo actual es el C.mmp (multiminiprocesador).
Aunque este método permite varias conexiones simultáneas, hay un solo un camino posible entre dos unidades. No hay necesidad de elección sobre que camino seguir, no existen caminos alternativos.

Bus de tiempo compartido (time-shared bus)

Es la organización más simple aunque con varios grados de complejidad dependiendo del número de buses. Uno de los más simple es tener todos los procesadores, memorias y unidades de I/O conectadas a un único bus, el cual puede ser totalmente pasivo. Obviamente, la concurrencia es mínima(una transacción a la vez), pero es por la inversión en hardware. La estructura de bus único puede usarse en procesadores(virtuales) compartiendo una memoria común, con alguna circuitería adicional, de una manera multiplexada sincronizada.
Puede ser usado en arquitecturas de funciones distribuidas y redes locales para interconexión entre procesadores. En las arquitecturas MIMD, puede lograrse más paralelismo, al precio de más complejidad, con más buses uni o multidireccionales. Pueden darse prioridades a unidades específicas si se agrega un árbitro de bus para manejarlos.

Buses de Memoria de Múltiple Pórtico

Esta organización concentra el intercambio en los módulos de memoria. Cada procesador tiene acceso, con su propio bus, a todos los módulos de memoria. Los conflictos son resueltos asignando prioridades fijas a los puertos de las memorias. Esta organización es bastante utilizada en monoprocesadores para permitir accesos a memoria concurrentes entre la CPU y el PIO. Una característica interesante de la arquitectura de múltiple puerto es que la CPU puede tener accesos privados a memoria, o equivalentemente negarle el uso de algún módulo de memoria, de una forma muy simple.
La concurrencia en nuevamente *mín.*(m, n). La cantidad necesaria de HW es del mismo orden de magnitud que en el cross-bar(m conexiones por módulo) pero está más localizada. Parece fácil aumentar el número de módulos, pero, por otro lado, el número de puertos en ellos limita el número de procesadores que pueden conectárseles.

Algunos problemas son evidentes para elegir uno de las tres topologías anteriores. La tabla resume estos:

	Concurrencia	*Costo*	*Modularidad*	*Confiabilidad*
Cross-bar	Máxima	Alto	Fácil	Pobre
Tiempo compartido	1	Bajo	Muy fácil	Muy pobre
Memoria multipuerto	Máxima	Mediano	Difícil	Mediana

Clasificación de las Computadoras Paralelas

Clasificación de Flynn:

Flujos de instrucciones	*Flujos de datos*	*Nombre*	*Ejemplos*
1	1	SISD	Máquina Von Neumann clásica
1	muchos	SIMD	Supercomputadora vectorial, arreglo de procesadores
muchos	1	MISD	dudosamente alguna
muchos	muchos	MIMD	Multiprocesadores, multicomputadoras

Está basada en dos conceptos: flujos de instrucciones y de datos. Un flujo de datos corresponde a un contador de programa. Un sistema con *n* CPU tiene *n* contadores de programa, y luego *n* flujos de instrucciones.
Un flujo de datos consiste de un conjunto de operandos.
Los flujos de instrucciones y de datos son, casi siempre, independientes, para las cuatro combinaciones existentes. SISD es propio de las primeras computadoras (minicomputadoras). Tiene un flujo de instrucciones, un flujo de datos, y hace una tarea a la vez. Aritmética escalar (Ej. PDP-8). En las maquinas SIMD una única instrucción opera sobre múltiples datos. Se caracteriza por una unidad de control que busca instrucciones de memoria, las ejecuta perse o transfiere la ejecución a elementos de procesamiento (PE). Todo en operación sincrónica. En esa operación sincrónica radican beneficios y limitaciones. Beneficios: No tengo overhead de sincronización. Pero ese sincronismo resulta limitante, es una limitación a la hora de aprovechar paralelismo. En Array Processing los P.E. son idénticos. Se replican y están conectados entre sí. Normalmente se los utiliza como coprocesador no como una computadora de propósito general.
Las máquinas MISD son una categoría extraña, con varias instrucciones sobre el mismo dato. No está claro si alguna de tales máquinas existe, pero algunos ubican a las máquinas pipeline como MISD.
Finalmente, tenemos MIMD, las cuales son solo varias CPUs independientes operando en forma asincrónica como parte de un gran sistema comunicándose a través de una memoria común: Shared Memory. La mayoría de las computadoras paralelas caen dentro de esta categoría. Los multiprocesadores y las multicomputadoras son máquinas MIMD. En las multicomputadoras cada nodo tiene su propio espacio de direcciones luego la comunicación entre nodos es través de mensajes.
La clasificación de Flynn termina aquí, pero puede extenderse. SIMD puede separarse en tres subgrupos. El primero es para supercomputadoras numéricas y otras máquinas que operan sobre vectores, realizando la misma operación sobre cada elemento del vector. La segunda es para máquinas del tipo paralela, como la ILLIAC IV, en la cual una unidad de control maestro repartía las instrucciones a muchas ALUs independientes. Y otro que incluye a los procesadores asociativos.
En nuestra clasificación, la categoría MIMD puede dividirse en multiprocesadores (máquinas con memoria compartida), y multicomputadoras (máquinas con pasaje de mensajes). Existen tres clases de multiprocesadores, los que se distinguen por como implementan la memoria compartida.

- **UMA** (Accesos a Memoria Uniformes): cada CPU tiene el mismo tiempo de acceso a cada módulo de memoria. Esto hace que el rendimiento sea predecible, un factor importante para la escritura de código eficiente.
- **NUMA** (Accesos a Memoria No Uniformes): no tienen la propiedad de las anteriores y el tiempo de acceso depende de la distancia entre el CPU y el módulo al que se desea acceder.
- **COMA** (Accesos a Memoria Cache Only): son también NUMA, pero de una manera diferente. Trata a toda la memoria principal como cache. Las páginas no están fijas a la máquina que las creó. En su lugar, el espacio de direcciones físicas se divide en líneas de cache, las cuales viajan dentro del sistema a pedido. Los bloques no tienen máquina de residencia. Una memoria que solo llama a las líneas cuando las necesita se dice una **memoria de atracción**. Usar toda la RAM como un gran cache aumenta el hit rate, y por lo tanto, el rendimiento.

La otra categoría principal de máquinas MIMD consiste de las multicomputadoras, las cuales, a diferencia de las anteriores, no tienen una memoria principal compartida a nivel de arquitectura. En otras palabras, el SO no puede acceder a la memoria de otras CPUs solo ejecutando un LOAD. Tiene que enviar un mensaje explícito y esperar por una respuesta. Esta es lo que las distingue de las anteriores. La habilidad de acceder a memoria remota está soportada por el SO y no por el HW. La diferencia es sutil, pero muy importante. A veces son llamadas máquinas NORMA (Accesos a Memoria No Remota), porque no pueden acceder directamente a memoria remota.
Pueden dividirse a su vez en dos subcategorías:

- **MPP**s (Procesadores Paralelos Masivamente), supercomputadoras caras que consisten de muchas CPUs fuertemente acopladas por una red propietaria de interconexión de alta velocidad (Cray T3E, IBM SP/2),
- **NOW** (Red de Estaciones de trabajo) o **COW** (Grupo de estaciones de trabajo), consiste de varias PC normales o estaciones de trabajo conectadas por tecnología comercial. No tienen mucha diferencia, pero se construyen por una fracción de lo que cuesta una MPP y se les dan otros usos.