Arquitectura de computadoras

Introducción

La *arquitectura* de una computadora la define el set de instrucciones (ISA). Este set de instrucciones hace 3 descripciones:

- funcional de las operaciones suportadas por el hw.
- cómo invocar las funciones.
- de la ubicación de los operandos (registros y memoria).

Dentro de las distintas arquitecturas podemos tener arquitecturas de distintas cantidades de bits. El procesador opera con palabras (words) de n-bits.

- **Microprocesadores**: 64-bits, 32-bits
- **Sistemas embebidos**: 16-bits, 8-bits

- Registros de n-bits
- Direcciones de (hasta) n-bits

La mínima unidad direccionarle es el byte. Con **direccionamiento al byte** cada byte va a tener su dirección propia de memoria y los words se van a formar accediendo a direcciones consecutivas en la memoria. En cambio el **direccionamiento al Word** cada dirección de memoria va a almacenar más de un byte, esto implica que no vamos a poder acceder a datos más chicos que un Word.

Vamos a considerar **MIPS** (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages):

Implementaciones de MIPS se usan en Sistemas embebidos.

- **RISC** (reduced instruction set computer)
- 32 y 64 bits

Arquitectura MIPS 32 bits

- 32 registros de 32 bits
- Instrucciones 32 bits
- 32 bits de direccionamiento

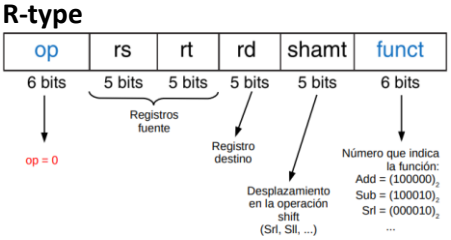
Name	Number	Use
$0	0	the constant value 0
$at	1	assembler temporary
$v0–$v1	2–3	function return value
$a0–$a3	4–7	function arguments
$t0–$t7	8–15	temporary variables
$s0–$s7	16–23	saved variables
$t8–$t9	24–25	temporary variables
$k0–$k1	26–27	operating system (OS) temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	function return address

MIPS incluye:

- **Instrucciones R-type**: instrucciones aritmético-lógicas. Van a ser registro a registro.
- **Instrucciones de escritura y lectura en memoria:** Load y Store.
- **Instrucciones de bifurcación:** saltos condicionales (branch) e incondicionales (jump).

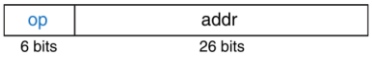
Hay varios tipos de formato de instrucción:

- **R-type**

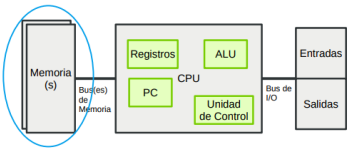


- **I-type**: la I viene de inmediato.

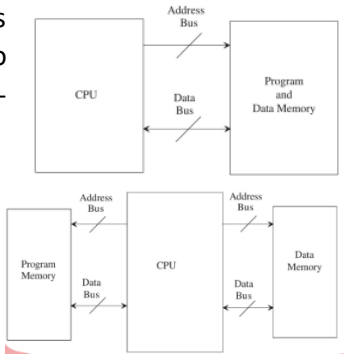
- **J-type**: son jumps incondicionales



La arquitectura no define la implementación del hardware. La *microarquitectura* es la combinación específica del hw (registros, memoria, ALUs, etc.) que implementa las operaciones definidas por una arquitectura. En función de cómo este organizada la memoria, tenemos dos distintos:



- Modelo von Neumann
- Modelo Harvard



- Modelo von Neumann: La memoria almacena instrucciones, datos y resultados intermedios y finales, es decir, los datos y las instrucciones conviven en el mismo espacio de direccionamiento. Se implementa como una jerarquía. Tiene un bus de datos bidireccional y otro de dirección unidireccional.
- Modelo Harvard: Usa memoria y buses separados para instrucciones y datos. Pueden ejecutar instrucciones y acceder a datos simultáneamente. Requiere 4 buses, dos de direcciones unidireccionales y dos de contenido bidireccionales.

¿Qué es un *Microprocesador*? Es un chip que contiene la CPU. En el modelo von Neumann, la memoria es externa al Microprocesador.

¿Qué es un Microcontrolador? Es un único chip que contiene CPU, memoria, entrada/salida y periféricos. Modelo Harvard.

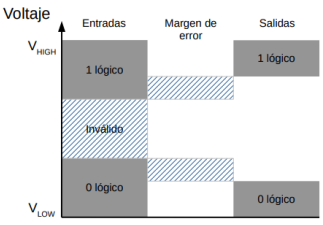
Técnicas digitales

Una **Compuerta** o **gate** es un circuito electrónico que opera sobre una o más señales de entrada para proveer una señal de salida. El voltaje y la corriente son señales analógicas, toman valores de un rango continuo. Los sistemas digitales responden a dos niveles de voltaje bien diferenciados: V_{HIGH} y V_{LOW}. Uno se corresponderá al valor 1 lógico y el otro, al 0 lógico.

Las compuertas AND, OR y NOT entienden tensiones V_{HIGH} y V_{LOW}. El voltaje de salida, en respuesta a los voltajes de entrada, son fijos pero no la correspondencia lógica. Tenemos 3 tipos de lógica para la entrada y salida:

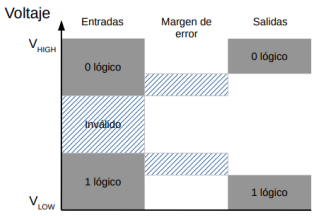
Lógica Positiva (LP): se utiliza para especificar las compuertas.

- $V_H \rightarrow 1$
- $V_L \rightarrow 0$



Lógica Negativa (LN):

- $V_H \rightarrow 0$
- $V_L \rightarrow 1$



A nivel eléctrico no cambia nada, solo cambia la forma de interpretar los voltajes

Lógica mixta: es la mezcla de las 2 lógicas anteriores. Por ejemplo compuertas NAND y NOR.

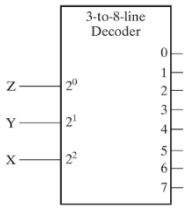
- Entradas lógica positiva y Salidas lógica negativa.
- Entradas lógica negativa y Salidas lógica positiva.

Circuito integrado (IC) o **chip** es una estructura de silicio (material semiconductor) que contiene los componentes electrónicos de compuertas digitales y elementos de almacenamiento. Se monta en un contenedor de plástico o cerámica, que es conocido como **encapsulado** o **packaging** del chip. Los encapsulados pueden ser de tamaños y formas diferentes y con diferentes distribuciones de los pines.

Dentro de los circuitos integrados hay varios niveles de integración, a medida que mejora la tecnología se incrementa la cantidad de compuertas que puede haber en un chip y por lo tanto aumenta el nivel de integración posible:

- Small-scale integration (SSI): Varias compuertas independientes en un chip. Menos de 10 compuertas.
- Medium-scale integration (MSI): Realizan alguna función específica elemental (sumar). Entre 10 y 100 compuertas.
- Large-scale integration (LSI): Incluye procesadores pequeños, memorias pequeñas y módulos programables. Entre 100 y 100.000 compuertas.
- Very-large-scale integration (VLSI): Microprocesadores complejos o para procesamiento digital de señales (DSP). Más de 100.000 compuertas (cientos de millones).

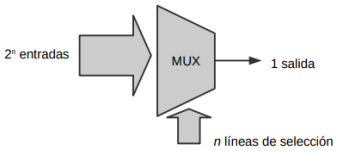
Bloques funcionales:

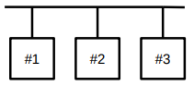


- **Decoder**: Con n bits puede representar 2^n elementos distintos. Decodificador de n a m ($n \leq m \leq 2^n$). La salida D_i es igual a 1 ($D_j = 0\ \forall j \neq i$) si la entrada codifica el valor i.
- **Encoder**: Realiza la función inversa del decoder. Tiene 2^n entradas (o menos) y n salidas. Esta definición de enconder tiene ciertas ambigüedades:
  - Solo una entrada puede tener valor 1. ¿Qué pasa cuando hay más de una entrada en 1? La salida dependerá de la implementación.
  - ¿Qué pasa cuando todas las entradas valen 0?
- **Priority encoder**: surge como solución a las ambigüedades del encoder, donde las entradas tienen diferentes prioridades. Agrega una salida que indica validez de la demás salidas. Agrega una salida adicional V que va a estar en 1 solo si alguna de las entradas esta en 1. Cuando está en 1, la salida va a indicar la entrada con mayor prioridad que se encuentra en 1. Cuando esta salida esta en 0, los valores de A_1 y A_0 son considerados *basura*.

Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

- **Multiplexor (MUX)**: Circuito combinacional que selecciona una de las entradas y la mapea a la (única) salida. Generalmente, vamos a tener n líneas de selección que determinan cuál de las 2^n entradas será la salida.



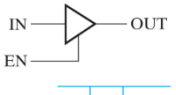


Hay casos (implementación de los buses) en la que es deseable poder unir salidas. Para esto es necesario que uno de los niveles predomine sobre el otro para evitar:

- Niveles de voltajes indefinidos
- Daños en los circuitos

En las compuertas convencionales hay solo dos estados V_H y V_L que se corresponden con el 0 y 1 lógicos. El tercer estado es el estado de alta impedancia (Hi-Z) que se comporta como un circuito abierto (desconectada). Es un estado válido para la salida, no para las entradas.

Conexión directa de salidas:

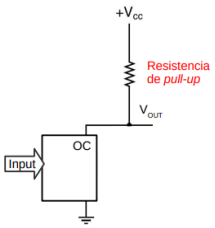


EN	IN	OUT
0	X	Hi-Z
1	0	0
1	1	1

- **Lógica Three state, Tristate, 3 state**: Las compuertas pueden devolver cualquier de los tres valores: V_H, V_L o Hi-Z. Disponible para cualquier tipo de compuerta. Común en buffers. La forma de habilitar ese tercer estado es con una señal adicional llamada enable (habilita o desabilita el circuito).

Los buffers three-state se puede conectar para generar una línea de multiplexado.

- **Open collector**: Es más tolerante a fallas. La salida de una compuerta Open Collector puede tener dos valores: V_L o Hi-Z. La salida de la compuerta OC se une a V_{cc} a través de una resistencia de *pull-up*. Si la salida de la compuerta es:
  - V_L, entonces $V_{OUT} = V_L$.
  - Hi-Z, entonces $V_{OUT} = V_{CC}$ (V_H).



Las salidas de varias compuertas Open Collector se pueden unir a una misma resistencia de pull-up. Si la salida de alguna compuerta es V_L, entonces $V_{OUT} = V_L$. Si la salida de ninguna compuerta es V_L (la salida de todas las compuertas es Hi-Z), $V_{OUT} = V_{CC}$ (V_H).

La función que se está implementando al unir salidas de compuertas Open Collector son:

  - LP → AND
  - LN → OR

Los **dispositivos lógicos programables** no tienen una función lógica preestablecida, a diferencia del Encoder, Decoder y MUX. Sino que es posible controlar las conexiones o almacenar información para definir la lógica a implementar. Para poder ser usados necesitan ser programados.

Hay distintos tipo de dispositivos lógicos programables. Alguno de ellos son las Tecnologías permanentes, que además de ser no volátiles, no pueden programarse. Pueden estar basadas en:

- **Fusibles**: Inicialmente cerrados. Se queman con voltajes superiores a los normales y eso abre la conexión.
- **Antifusibles**: Inicialmente abiertos. Contienen un material no conductor que con voltajes elevados se funde y baja la resistencia cerrando la conexión.
- **Programación por máscara**: La realiza el fabricante durante las últimas fases del proceso de fabricación del chip. Dependiendo de la función a implementar, se realizan o no las conexiones sobre las capas de metal que sirven como conductoras en el chip.

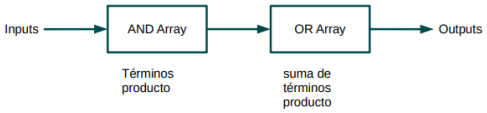
Por otro lado están las tecnologías reconfigurables son un dispositivo de almacenamiento de 1 bit que controla un transistor:

- si el bit está en 1, el transistor cierra el circuito.
- Si el bit está en 0, el transistor abre el circuito.

En función de que tecnología se está usando, pueden ser volátiles o no.

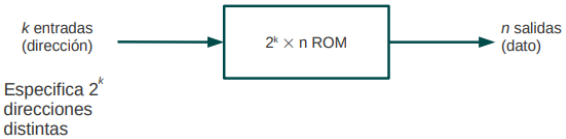
Es fácilmente reprogramable, pero necesita alimentación. Basada en transistores floating-gate (transistor que tiene una puerta flotante aislada en el interior) conectada de forma capacitiva. Como está aislada permite mantener la carga por largos períodos de tiempo.

En toda función lógica puede expresarse en dos niveles de compuertas como una suma de productos.



Hay muchas tecnologías distintas de dispositivos lógicos programables:

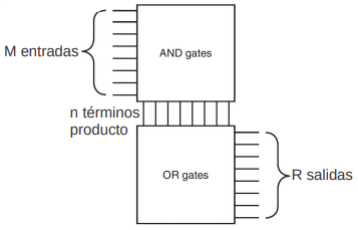
- Simple Programmable Logic Devices (SPLD)
  - **Read Only Memory (ROM)**: permite solo programar el arreglo de ORs porque el de ANDs es fijo. No volátil y permanente. Permite especificar completamente n funciones de k entradas. Tamaño: $2^k \times n$ bits.



Una ROM de 2^k x n tendrá internamente un decoder de k a 2^k y n compuertas OR. Cada OR tendrá 2^k entradas conectadas de manera programable a cada salida del decoder. Es una reprecentacion directa de las tablas de verdad.
Dependiendo de la tecnología de programación:

- *ROM* → programación por máscara. Lo hace el fabricante y es *read only*.
- *PROM* → programación por fusible o antifusible
- *EPROM* → tecnología que usa transistores floatinggate. Programable electrónicamente y borrable con luz ultravioleta.
- *EEPROM* → tecnología que usa transistores floatinggate. Programable y borrable electrónicamente.
- *Memoria FLASH* → Mejora de la EEPROM

- **Programmable Logic Array (PLA)**: Implementa funciones como sumas de productos. Consiste de un arreglo de AND y un arreglo de OR, ambos con conexiones programables. Tiene M entradas, $n < 2^M$ términos productos y R salidas. Programar R funciones de M entradas cada una. No es posible generar todos los minitérminos. Los términos productos se conectan selectivamente en el arreglo de OR para generar las funciones requeridas.



- **Programmable Array Logic (PAL®)**: Similar al PLA pero no tan flexible. El plano de AND es programable. El plano de OR es fijo. Los términos producto no se pueden compartir entre múltiples salidas. En general ya no se utilizan en diseños nuevos porque han ido apareciendo dispositivos con más capacidades y uso energético más eficiente.
- ...

- Complex Programmable Logic Device (CPLD)**:** Varios SPLD en un único chip e interconectados con conexiones programables. Los SPLD simples tienen un conjunto de unas pocas entradas y salidas. ¿Qué pasa si hace falta mayor cantidad?
  - Interconectar varios SPLD → problemas de layout, de energía, de costos…
  - Complex Programmable Logic Device (CPLD)

  Se basan en la misma estrutura que los SPLD.

- Field Programmable Gate Array (FPGA)**:** Interconectan elementos básicos para lograr funcionalidad más avanzada.
  - Bloques de lógica programable
  - Interconexiones programables entre los bloques
  - Pines de entrada/salida programables

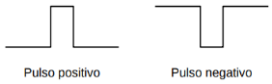
  Adicionalmente, puede tener bloques de lógica dedicada:

  - Memorias
  - Unidades aritmético/lógicas
  - Microprocesadores

  Los FPGA permiten la implementación de circuitos lógicos relativamente complejos (como pequeños procesadores).
  Conceptualmente es un arreglo 2D de celdas con posibles interconexiones entre ellas. Las celdas consisten de tablas de lookup (LUT): una pequeña cantidad de lógica y RAM. Configurar el FPGA requiere configurar los pines de entrada, los bloques de lógica programable, el conexionado entre los bloques, y los pines de salida.

Circuitos secuenciales

¿Qué es un pulso? Cambio de un nivel de voltaje a otro, seguido del retorno al nivel de voltaje inicial. Un pulso puede ser positivo o negativo dependiendo de si el voltaje inicial es V_H o V_L.



En geneal se piensa que los pulsos son ondas cuadras perfectas, que pasa de 0 a 1 en ese instante pero la realidad es que el pulso no es perfecto. El cambio en el voltaje lleva un tiempo y si ese tiempo no es necesario pueden ocurrir valores de voltajes invalidos.

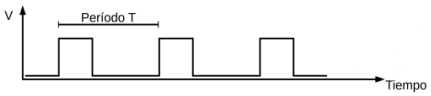
Cuando ocurre el evento se genera un pulso, el cual posee 3 etapas:

- Transcurso del pulso → nivel.
- Flanco ascendente o positivo (inicia el pulso).
- Flanco descendente o negativo (finaliza el pulso).

En general, si queremos capturar la ocurrencia de un evento lo mejor es prestar atención a los flancos.

¿Qué es un reloj? Tren de pulsos correspondiente a una señal periódica:

- T: período (segundos)
- F: frecuencia (ciclos por segundo ó Hertz) 1/T



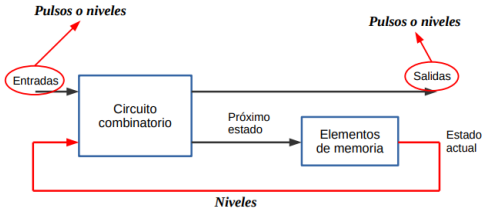
Duty Cycle o ciclo de trabajo es la relación entre el período y el tiempo activo:

$$\text{Duty cycle} = \frac{\text{Tiempo activo}}{\text{Período T}}$$

Porción de periodo en el que el reloj está activo.

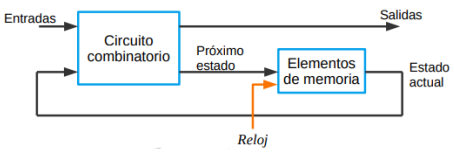
La respuesta de un circuito combinacional depende de las entradas externas. La respuesta de un circuito secuencial depende de:

- Las entradas (externas)
- La historia pasada (estado interno) lo cual se logra a través de elementos en memoria.



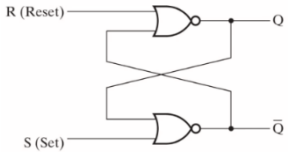
Tipos de circuitos secuenciales:

- **Por nivel (o asincrónicos)**: Opera en función de los niveles (valores: 0–1, V_H–V_L) de las entradas. Los circuitos por nivel puros son difíciles de diseñar porque son muy dependientes del orden en el que cambian las señales de entradas.
- **Por pulso (o sincrónicos)**: Opera en función de los cambios de niveles de las entradas. Opera en función de las señales de entrada en períodos discretos de tiempo (fijados por los pulsos de un reloj). Hay una señal pulso particular (el reloj del circuito) que es la que dispara el cambio en las salidas. Si hay más de una entrada pulso, se asume que no ocurren en simultáneo.



Hay dos tipos de elementos en memoria:

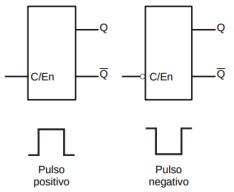
- **Elementos asincrónicos o Latches**: el Latch con el circuito lógico más simple a nivel implementación es el SR. Cambian en función de 3 posibles combinaciones de entradas:
    - Set: $Q^+ \leftarrow 1$
    - Reset: $Q^+ \leftarrow 0$
    - Hold: $Q^+ \leftarrow Q$



S	R	Q	Q^+
0	0	Q	Q
0	1	*	1
1	0	*	0
1	1	*	Inválido

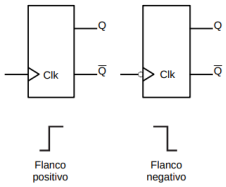
En habilitación por pulso mientras que el pulso se mantenga en el nivel alto, cualquier cambio en las entradas se refleja en las salidas. Cuando se deshabilita, las salidas se bloquean en el último valor que hayan tenido. El pulso no puede ser arbitrariamente angosto.

Habilitados por pulso



- **Elementos sincrónicos o Flip flops**: Los Flip Flops tienen una entrada de control (Control, Enable, Clock) → Sincrónico. Mientras la entrada de control esté deshabilitada, la salida no cambia (independientemente de lo que ocurra con las demás entradas). Las entradas deben estar estables:
    - Un poco antes de que la señal de control de habilite.
    - Durante el período completo en que esté habilitada.
    - Un poco después de que se deshabilite.

Habilitados por flanco



En la habilitación por flanco ascendente o descendente se disparan las acciones sólo durante la transición entre niveles. Se logra mayor frecuencia y mejores resultados. La señal de clock está indicada con un triángulo.

Pueden realizar hasta cuatro operaciones:

- Mantener la salida (hold)
- Poner la salida en 1 (set)
- Poner la salida en 0 (reset)
- Complementar la salida (toggle)

Hay 4 tipos de flip flops:

- SR: dos entradas. Realiza set y reset. Latch SR + una entrada de control para indicar cuando cambian las salidas.

Flip Flop SR				
C	S	R	Q(t+1)	
0	*	*	Q(t)	Hold
1	0	0	Q(t)	Hold
1	0	1	0	Reset
1	1	0	1	Set
1	1	1	?	Indefinido

- D: una entrada. Realiza set y reset. Elimina el estado indefinido del FF SR. Este FF almacena el dato D.

Flip Flop D			
C	D	Q(t+1)	
0	*	Q(t)	Hold
1	0	0	Reset
1	1	1	Set

- JK: dos entradas. Realiza las tres operaciones. Mejora el Flip Flop SR haciendo que la combinación 1 1 sea válida realizando la operación Toggle. Las entradas J y K equivalen a S y R. ambas entradas habilitadas complementan la salida.

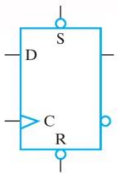
Flip Flop JK				
C	J	K	Q(t+1)	
0	*	*	Q(t)	Hold
1	0	0	Q(t)	Hold
1	0	1	0	Reset
1	1	0	1	Set
1	1	1	Q'(t)	Toggle

- T: una entradas. Realiza toggle (cambia el estado).

Flip Flop T			
C	T	Q(t+1)	
0	*	Q(t)	Hold
1	0	Q(t)	Hold
1	1	Q'(t)	Toggle

Los FF además pueden tener entradas asincrónicas: Direct set (preset) y Direct reset (clear). Modifican la salida independientemente de la señal de control. Sigue manteniendo la misma lógica que un SR.

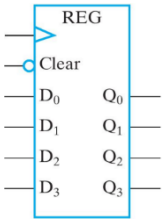
S	R	C	D	Q	$\overline{Q}$
0	1	*	*	1	0
1	0	*	*	0	1
0	0	*	*	Inválido	
1	1	↑	0	0	1
1	1	↑	1	1	0



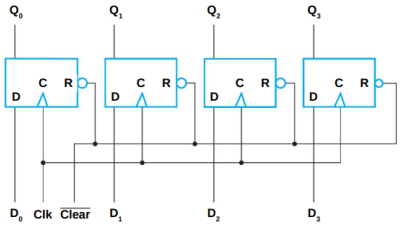
Un **registro** es un grupo de FF que comparten el reloj, cada uno almacena un bit de información. Un registro de n-bits consiste de n FF. Además de los FF, puede haber compuertas que realicen algunas tareas simples sobre los datos. El más simple consiste solo de FF, no tiene compuertas.

Un registro tiene como mínimo:

- N entradas: Las entradas D_i determinan las salidas Q_i con cada flanco positivo del reloj.
- Un reloj.
- Una entrada de clear: resetea las salidas del registro (Pone sus salidas en 0)
- N salidas de datos.

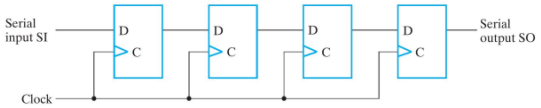


Los registro pueden implementarse con cualquier tipo de FF, aunque la implementación más directa es con FF D. Hay un FF por cada bit de dato y todos comparten la señal de reloj y la señal de clear. Un reloj común dispara todos los FF y el dato binario disponible en las entradas D se transfiere a las salidas Q.



El Registro de desplazamiento (shift) es un registro especial con la capacidad de desplazar, en una dirección seleccionada, la información binaria en cada FF al FF vecino. El más simple de todos opera en serie y solo desplaza los dígitos en un solo sentido.

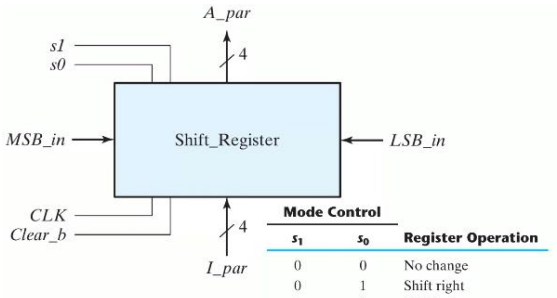
Shift a derecha: $(z_0, z_1, \ldots, z_n) \rightarrow (0, z_0, z_1, \ldots, z_{n-1})$



El registro de desplazamiento más general permite:

Utilidad: conversión serie en paralelo y viceversa.

- Entrada de reloj
- Clear para poner el registro en 0
- Control para desplazar a derecha, con líneas de entrada y salida seriales.
- Control para desplazar a izquierda, con líneas de entrada y salida seriales.
- Control de carga paralela y líneas de entradas asociadas.
- Líneas de salida paralela
- Control para habilitar/deshabilitar cambios en la salida en respuesta al reloj.



Mode Control		
s_1	s_0	Register Operation
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

El **contador** es un grupo de FF que comparte el reloj y pasan por una secuencia predefinida de estados binarios en respuesta a un pulso de entrada. El pulso de entrada puede:

- ser un pulso de reloj u originado por una fuente externa
- ocurrir a intervalos fijos de tiempo o random.

Los contadores pueden diferir en módulo, ser sincrónicos o asincrónicos, contar hacia arriba (up) o hacia abajo (down). También varían en que secuencia de estados ciclan.

Los **contadores asincrónicos** son, en general, contadores *ripple*. El clock dispara los cambios en el primer FF. La salida de ese FF sirve como fuente de disparo de los demás FF.

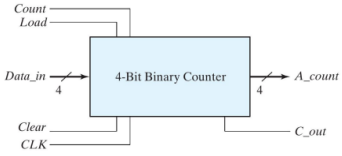
El contador binario en ripple es un contador que cicla a través de la secuencia de números binarios. Un contador binario de n-bits consiste de n FF y puede contar, en binario, desde 0 hasta 2^n-1. Puede implementarse con cualquier FF.

¿Cómo implementar el contador binario en ripple utilizando FF? El bit menos significativo cambia siempre, por lo que el próximo valor será el valor actual negado. El resto de los bits, cambian cuando el bit anterior cambia de 1 a 0. Por lo que el próximo valor será el valor actual negado.

En un **contador sincrónico**, todos los FF reciben la misma entrada de reloj. El reloj común dispara todos los FF simultáneamente. La evolución del contador está dada por las funciones de entrada.

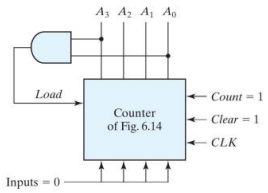
Distintas configuraciones de contadores:

- Contador binario: Un contador binario de n-bits que consiste de n FF y puede contar, en binario, desde 0 hasta 2^n-1. El próximo bit menos significativo es el actual negado. Para el resto de los bits, hay que buscar otra estrategia.
  - Implementado con FF D:
    - $D^+ = D$
    - $C^+ = D \text{ XOR } C$
    - $B^+ = B \text{ XOR } CD$
    - $A^+ = A \text{ XOR } BCD$
  - Pueden implementarse con otros FF
- Contador binario carga paralela: Flexibiliza el uso de los contadores. La carga paralela permite modificar el estado del contador antes de la operación de cuenta. Entonces podemos tener varios tipos de señales y prioridades entre ellas. La señal de Clear es en general asincrónica por lo que resetea el estado del contador sin necesidad de un pulso de reloj. La señal de Load es sincrónica.



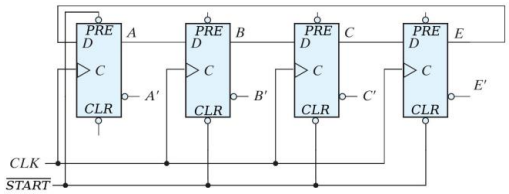
Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

- Contador BCD: es un caso particular de un contador binario. Se puede diseñar un circuito secuencial de 4 FF a través de la tabla de estados. Se puede usar un contador de carga paralela y codificar el estado 9 para cargar un 0 en el próximo pulso de reloj.

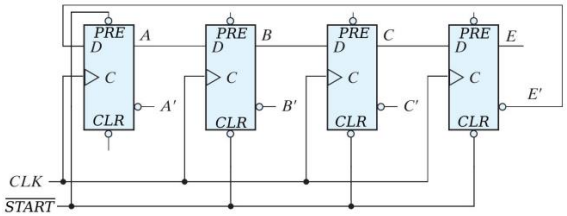


- Contador anillo (ring counter): Define una secuencia de estados con un único FF en 1 en cada momento. Todos los demás están en 0. Ese bit en 1 se desplaza de un FF al siguiente. Es costoso. Para un contador de n estados se necesitan n FF. Define n señales de temporizado. La implementación más simple es con FF con la capacidad de Preset/Clear, encadenados como si fueran un registro de desplazamiento

ABCD	$A^+B^+C^+D^+$
0001	0010
0010	0100
0100	1000
1000	0001



- Contador Möbius (Contador Johnson): Similar al Ring counter. Con n FF logra 2n estados. Duplica la cantidad de estados conectando la salida negada del último FF a la entrada del primero.



Verilog HDL

Los métodos manuales para diseñar circuitos lógicos son posibles y usables, solo en caso de circuitos pequeños. En los demás casos se utilizan herramientas automáticas. Hacer prototipos de circuitos integrados es costoso, tanto en dinero como en tiempo, por lo que los diseños modernos se basan en los lenguajes de descripción de hardware para poder describir, diseñar y testear circuitos en software antes de fabricarlos.

HDL (Hardware Description Languaje – Lenguaje de descripción de Hardware) no es un lenguaje de programación, sino que describe la estructura y el comportamiento del hardware, y representa las operaciones concurrentes que realiza el hardware. Describe la relación entre las señales de entrada y las señales de salida de un circuito. Hay muchos lenguajes de descripción de hardware, los más comunes son VHDL y Verilog.

El proceso de diseñar sistemas puede descomponerse en pasos individuales que se aplican tanto para el diseño manual como para el moderno:

1. Especificación: Especificación de alto nivel del comportamiento del diseño.
2. Diseño funcional: Describe la especificación usando diagramas de bloques para las entradas/salidas, tablas de verdad, diagramas de estado, algoritmos, etc.
3. Síntesis: Crea los esquemáticos a nivel compuertas.
4. Mapeo tecnológico: Elegir la tecnología con que se implementará. Adaptar los esquemáticos anteriores a la tecnología elegida (e.g. solo NAND/NOR).
5. Ubicación y ruteo: Ubicar los componentes para minimizar el área y cablear las conexiones minimizando la longitud y los cruces.
6. Verificación: Elegidos la tecnología y el *layout*, se estiman los retardos de las compuertas y del cableado para analizar si la solución cumple con las especificaciones (en tiempos y consumo).
7. Fabricación: Una vez que el diseño está verificado, puede implementarse (DLP, ASIC, componentes discretas, etc.)

Nosotros usaremos Verilog que tiene una sintaxis basada en C creado en el año 1995. En el año 2001 se realizó una actualización muy grande. En varias bibliografías se puede encontrar código aclarando si es antes del 2001 o después del 2001.

Lo más básico...

- Comentarios:
  - **//** hasta el final de línea.
  - **/* */** múltiples líneas de comentario.
- Comas (**,**) separan elementos en una lista.
- Punto y coma (**;**) termina la sentencia.
- Sensible a mayúsculas y minúsculas.

Verilog se basa en **Módulo** que permiten definir en bloques lógicos tanto el comportamiento como la interfaz.

Module *<nombre>* **(***<lista de señales de entrada y de salida>***);**

input *<lista de señales de entrada>;*

output *<lista de señales de salida>;*

…

endmodule

En función de si estamos definiendo un módulo en Verilog pre o post 2001, la dirección y el tipo de puerto se van a definir dentro de la especificación de un módulo o junto con el nombre del módulo y las señales.

Toda señal debe tener asociado un tipo de dato. Estos pueden ser:

- **De red (net)**: modelan interconexiones entre componentes. El más común es Wire que define señales internas que son salidas / entradas de distintos módulos.
  wire <lista de señales internas>
- **Variables**: Modelan almacenamiento. Mantienen el dato asignado hasta una nueva asignación. El más común es reg que modela una señal de un único bit.

Las señales binarias pueden tomar 4 valores:

- 0
- 1
- x (don't care).
- z (alta impedancia).

Pueden estar organizadas como

- **Vectores**: Arreglo unidimensional de elementos (de red o variables).
  <type> [<Left_bit_index> : <right_bit_index>] vector_name
- **Arreglos**: Es un arreglo multi-dimensional de elementos. Vector de vectores de la misma dimensión.
  <type> [<left_idx> : <right_idx>] array_name [<array_start_idx> : <array_end_idx>];

Cada valor puede

- Tener asociada una potencia.
- Tener múltiples generadores.

En general hay 3 tipos de formas de diseñar los circuitos:

- **Estructural**: Representa el esquemático del circuito, describiendo las conexiones entre las distintas componentes. Se refiere a componer módulos de bajo nivel para crear módulos con

funcionalidades de más alto nivel. Esto crea una jerarquía. Un diseño puramente estructural, no contiene ningún constructor de comportamiento y contiene instanciaciones e interconexiones entre módulos.

Para crear esa jerarquía se van a tener que instanciar módulos básicos e interconectarlos

module_name <instance_identifier> (port mapping …);

- **Del flujo de datos**: Describe el circuito en término de la función y no de la estructura. La salida se especifica en términos de las transformaciones que sufren las entradas. Suele considerarse de comportamiento porque no especifica estructura. Se basa en la asignación continua:
  - Modela lógica combinacional.
  - El orden de las asignaciones no tiene importancia porque opera de manera concurrente o paralela.
  - **assign** <left> = <right>
    - La parte izquierda tiene que ser un tipo de red
    - La parte derecha puede contener tipos de red, registros, constantes y operadores
    - Cualquier cambio en la parte derecha se refleja en la parte izquierda.
- **De comportamiento**: Modela la asignación de señales basada en un evento (transición entre señales). Modela asignaciones secuenciales. La parte derecha de la asignación solo puede ser de tipos variables (reg, integer, real, time). Las asignaciones procedurales se pueden evaluar en el orden listado. Toda asignación procedural debe estar dentro de *bloque procedural*. El bloque procedural puede ser:
  - Bloque *Initial*: Se usa en simulaciones para inicializar señales. Se ejecuta una única vez al inicio de la simulación. No siempre es sintetizable.
  - Bloque *always*: Modela circuitos sintetizables que se ejecutan continuamente. Se le puede agregar una lista de sensibilidad que indica cuándo se disparan las asignaciones dentro del bloque. Cualquier cambio dispara las acciones. Es una lista de señales (equivale a la **or** de señales). Se pueden especificar el flanco (posedge, negedge)

  En un bloque procedural se pueden utilizar 2 tipos de asignaciones:
  - Asignación bloqueante **=**: Modela lógica combinacional. Todas las asignaciones del bloque se evalúan y asignan en paralelo.
  - Asignación no bloqueante **<=**: La asignación de la señal destino se difiere hasta la finalización del bloque procedural. Modela circuito secuencial.

  Esto provoca que una asignación bloqueante y una no bloqueante a veces resulten en el mismo resultado y a veces no.

Asignación bloqueante vs no bloqueante	
Al modelar lógica combinacional, usar asignación bloqueante y listar todas las entradas en la lista de sensibilidad.	Al modelar lógica secuencial, usar asignación no bloqueante y en la lista de sensibilidad incorporar solo los pulsos (clock) y líneas de reset (si hay).

Sumadores

Algoritmo básico para sumar dos números X e Y en la base b: $X_b + Y_b = S_b$

1. $c_0 \leftarrow 0$ (acarreo inicial)
2. For i = 0 step 1 until n-1 do
   a. $tmp \leftarrow x_i + y_i + c_i$
   b. $s_i \leftarrow tmp \bmod b$
   c. $c_{i+1} \leftarrow \lfloor tmp / b \rfloor$
3. $s_n \leftarrow c_n$

$$\begin{array}{cccccc} c_n & c_{n-1} & \cdots & c_2 & c_1 & c_0 \\ + & x_{n-1} & \cdots & x_2 & x_1 & x_0 \\ & y_{n-1} & \cdots & y_2 & y_1 & y_0 \\ \hline s_n & s_{n-1} & \cdots & s_2 & s_1 & s_0 \end{array}$$

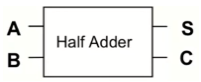
Independientemente de la base, el acarreo es siempre 0 o 1, y si hay acarreo final, significa que hubo overflow.

Si analizamos el algoritmo básico, la complejidad del algoritmo es lineal O(n), con n dependiendo de la cantidad de dígitos donde en el peor de los casos el acarreo se da desde el bit menos significativo al bit más significativo. El tiempo de ejecución es T(n) = kn, donde k va a depender de la tecnología. La complejidad del algoritmo no puede mejorarse, dado que hay que recorrer todos los dígitos. Lo que se mejora es la implementación.

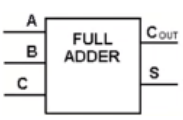
Para b = 2 vamos a buscar las expresiones lógicas que resuelvan la operación de suma de 2/3 dígitos.

- Half adder o semisumador: El sumador más simple. Suma dos operandos de 1 bit. Resultado de 2 bits dentro del rango 0 – 2.
  - S ← A XOR B
  - C ← A ◦ B

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



- Full adder o sumador completo: Realiza la suma de 3 bits. Resultado de 2 bits dentro del rango 0 – 3.
  - S ← A XOR B XOR C
  - C_{OUT} ← A ◦ B + (A + B) ◦ C

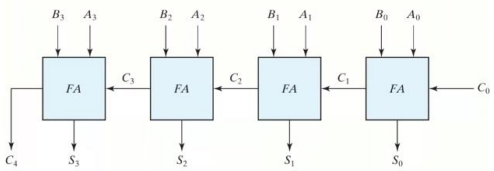


A	B	C	C_{OUT}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Un sumador binario es un circuito digital que realiza la suma de dos números binarios. Hay varias implementaciones distintas:

El principal problema de estos sumadores es como propagar el carry al siguiente

- Sumador serie: si bien no se usa porque tiene un alto tiempo de respuesta, es la que menos hardware necesita y es la más similar al algoritmo de suma. Requiere poco hardware: 2 registros de desplazamiento, 1 Full adder y 1 FF D. Los registros son para almacenar los dos números a sumar. El FF D guarda el carry resultante de la suma de cada par de bits. El resultado de la suma se almacena en uno de los registros de entradas.
- Sumadores paralelos: tiene todos los bits de entrada disponibles al mismo tiempo.
  - *Ripple adder*: Se puede construir un sumador binario paralelo conectando varios full adders en cascada.



Una vez estabilizadas las entradas, el tiempo que un circuito demora en producir la salida es proporcional a la máxima cantidad de niveles de compuertas que debe atravesar. El tiempo exacto es muy dependiente de la tecnología, aunque también influye el tipo de compuerta y la cantidad de entradas que tiene. Lo que vamos a comparar son simplemente los niveles de compuertas y un tiempo estimado de retardo por compuerta. Si Δ_G es el retardo de una compuerta, $\Delta_{FA} = 2\Delta_G$

Sincrónico	Asincrónico
• O(n) • Siempre hay que esperar el peor caso • Tiempo de operación fijo	• O(log n) • Se termina la suma cuando se termina la propagación de carry • Tiempo de operación variable • Más cantidad de hw.

Hablando de arquitectura de computadoras, es preferible un ripple sincrónico que un ripple asincrónico.

  - Carry-look-ahead adder (CLAA): El objetivo es generar todos los carries de entrada en paralelo. Sería posible dado que el carry depende de las entradas $A_{n-1}...A_0$ y $B_{n-1}...B_0$ y esa es información disponible inicialmente. Es impracticable porque requeriría que todas las etapas del sumador tengan todas las entradas. El número de entradas se reduce usando información sobre la generación o propagación del carry en cada etapa. Si analizamos la ecuación del carry tiene dos partes:

$$C_{OUT} \leftarrow \underbrace{A \cdot B}_{\text{Generación}} + \underbrace{(A + B)}_{\text{Propagación}} \cdot C_{IN}$$

- Generación de carry:
  - El par de bits de genera carry (11)
- Propagación de carry:
  - El par de bits no genera carry (10,01) pero propagan el carry de entrada.
  - El par 00 no genera ni propaga el carry.

Se puede calcular el carry de entrada a cualquier posición en función del carry inicial y los carries propagados y generados en las posiciones previas.

$$C_{i+1} = A_i \cdot B_i + (A_i + B_i) \cdot C_i$$

- $G_i = A_i \cdot B_i$ generación de carry
- $P_i = A_i + B_i$ propagación de carry

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

$$C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

$$C_5 = G_4 + P_4 G_3 + P_4 P_3 G_2 + P_4 P_3 P_2 G_1 + P_4 P_3 P_2 P_1 G_0 + P_4 P_3 P_2 P_1 P_0 C_0$$

CLAA	Ripple Adder
$5\Delta_G$	$2n\Delta_G$
O(1)	O(n)

Supongamos que Δ_G es el retardo de una compuerta. El retardo total de la suma será:

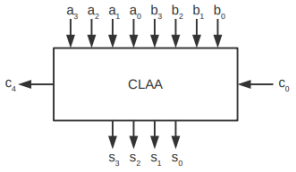
- Δ_G para generar todos los P_i y G_i (se pueden hacer en paralelo)
- $2\Delta_G$ para generar todos los C_{i+1} en función de los P y G anteriores.
- $2\Delta_G$ para generar la suma S_i a partir de C_i, A_i y B_i.

En total $5\Delta_G$ para sumar cualquier cantidad de bits.

En la práctica hay limitaciones en:

- *El número de entradas de las compuertas (fan-in)*: a mayor cantidad de entradas más lenta la compuerta.
- *El número de entradas a los que se pueden conectar una salida (fan-out)*: de manera segura.

La alternativa es aumentar la cantidad de niveles para calcular los P y G. No se sacrifica al carry independiente de la suma.



Niveles de Carry Lookahead:

- **Tecnología MSI**
  - *CLAA en ripple*: es la más simple. Dividir las n etapas en grupos, cada uno con su propio CLAA y conectados en ripple. Grupos de igual tamaño beneficia la modularidad y el diseño de un único CI. Consideramos grupos de 4 bits: 4 es divisor de la mayoría de los tamaños de palabras.
    Para n bits y grupos de 4, se necesitan n/4 grupos. Se necesita:
    - Δ_G para todos P_i y G_i.
    - $(n/4)2\Delta_G = (n/2)\,\Delta_G$ para propagar el carry a través de todos los grupos (una vez conocidos P_i, G_i y $C_{inicial}$, demanda $2\Delta_G$ la propagación de un grupo al siguiente).

CLA hace referencia a la técnica de Carry Lookahead.

CLAA hace referencia a Carry lookahead adder, es decir, sumador que opera usando esta técnica sobre todos los bits

- $2\Delta_G$ para realizar la suma final

Total: $\Delta_G + (n/2)\,\Delta_G + 2\Delta_G = (n/2 + 3)\,\Delta_G$

Sigue siendo O(n), pero es una reducción de casi un 75% frente al ripple adder de $2n\Delta_G$

- *Carry-lookahead generator (CLAG)*: Además del lookahead interno a cada grupo, se puede proveer el carry generado y el carry propagado grupales. G* es el carry generado. P* es el carry propagado. Si G* = 1 el grupo genera (internamente) carry de salida. Si P* = 1 el carry de entrada al grupo se puede propagar para producir un carry de salida del grupo. Para un grupo de 4 bits, teníamos que C_4:

$$C_4 = \underbrace{G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3}_{\text{Generación grupal}} + \underbrace{C_0P_0P_1P_2P_3}_{\text{Propagación grupal}}$$

$$G^* = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3$$

$$P^* = P_0P_1P_2P_3$$

Los P* y G* de varios grupos pueden usarse para generar los carries de entrada a cada grupo (similar a los carries de entrada a un único bit).

Los CLAG también podrían disponerse en varios niveles, es decir, ir formando un árbol de CLAGs.

- **Tecnología VLSI**
  - *Lookahead tree adder*: Generalización de las ecuaciones. Dado el grupo de bits en las posiciones j, j+1, … , i (j ≤ i) notamos:
    - $P_{j:i}$ carry propagado
    - $G_{j:i}$ carry generado

Si $P_{j:i} = 1$, el carry de entrada en la posición j se propaga hasta la posición i+1

Si $G_{j:i} = 1$, el carry se genera en alguna posición entre i y j y se propaga hasta i+1.

Las funciones P y G pueden calcularse de manera recursiva:

$$P_{j:i} = \begin{cases} P_i & \text{Si } i = j \\ P_i \cdot P_{j:i-1} & \text{Si } j < i \end{cases}$$

$$G_{j:i} = \begin{cases} G_i & \text{Si } i = j \\ G_i + P_i \cdot G_{j:i-1} & \text{Si } j < i \end{cases}$$

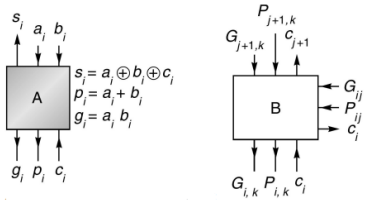
$$P_{j:i} = \begin{cases} P_i & \text{Si } i = j \\ P_{j:m-1} \cdot P_{m:i} & \text{Si } j < m \le i \end{cases}$$

$$G_{j:i} = \begin{cases} G_i & \text{Si } i = j \\ G_{m:i} + G_{j:m-1} \cdot P_{m:i} & \text{Si } j < m \le i \end{cases}$$

Lo que vamos a tener es que el grupo completo va a propagar carry si los dos grupos propagan carry y el grupo completo va a generar carry si el grupo más significativo genera o si el más significativo propaga o el menos significativo genera.

Para crear el Lookahead adder tree tenemos dos tipos de bloques:

- A: a partir de los bits a_i, b_i y c_i calcula: p_i, g_i y s_i
- B: a partir de los P y G de dos grupos distintos consecutivos, calcula los P y G y el acarreo del grupo completo.



Gran mejora en performance:

- Los bits deben atravesar en el orden de $\log_2 n$ niveles lógicos, comparados con los 2n del ripple.

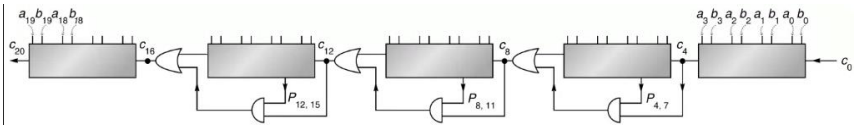
Incremento en el tamaño:

- 2n celdas (A y B) en un espacio n Log_2 n frente a n celdas (FA) en el ripple.

- Carry-skip adder: Divide al número en grupos, no necesariamente iguales, que suman en ripple. Se basa en que calcular P es más simple que G. Los grupos calculan solo P*, el G* se calcula indirectamente. Si un grupo genera, hay que hacer la suma con acarreo inicial 0 para saberlo. Si un grupo propaga el acarreo entonces, ese grupo puede saltearse cuando se va a callar el acarreo de entrada del siguiente bloque.

  El carry-skip adder es práctico solo si puede asegurarse que los acarreos iniciales a todos los bloques sean 0 al comienzo de cada operación.

  Cada bloque de skip adder tiene full adders en configuracion ripple que ademas calculan la propagacion en cada posicion.



¿Cuánto demora el sumador para sumar n bits en bloques iguales de k bits? Cada bloque comienza a calular P* y su acarreo de salida en paralelo. En dos niveles de compuertas ya termina P* pero falta terminar el carculo del G* que demora lo mismo que un ripple de k bits. El peor caso es que el acarreo se prorpague a lo argo de todos los bloques, esto demanda 2 niveles de compuerta por cada grupo que hay que saltear.

$$T = \underbrace{t_{\text{Ripple en k bits}}}_{\text{Primer sumador}} + \underbrace{(n/k - 2)\,(t_{AND} + t_{OR})}_{\text{Tiempo de saltear los sumadores del medio}} + \underbrace{t_{\text{Ripple en k bits}}}_{\text{Último sumador}}$$

Si cada compuerta tiene retardo Δ_G. El tiempo del ripple es 2 Δ_G×cantidad de bits en ripple (k).

$$\mathbf{T = 2k\Delta_G + (n/k - 2)\,(\Delta_G + \Delta_G) + 2k\Delta_G = \Delta_G\,(4k + 2n/k - 4)}$$

¿Cuál sería el tamaño óptimo de los bloques de k bits?

$$\mathbf{T = \Delta_G\,(4k + 2n/k - 4)}$$

Para hallar el óptimo k, se deriva T con respecto a k y se iguala a 0.

$$\mathbf{dT/dk = 4\Delta_G - 2n\Delta_G/k^2 = 0,\ k = \sqrt{n/2}}$$

El tamaño del grupo optimo y el tiempo de propagación del carry son proporcionales a $\sqrt{n}$.

$$\mathbf{T = \Delta_G\,(4\sqrt{n/2} + 2n\sqrt{n/2} - 4) \geq 4\sqrt{2n} - 4}$$

Asumiendo bloques de diferentes tamaños, el tiempo de saltear un sumador no depende de la cantidad de bits a sumar. Se puede optimizar:

- Incrementando el tamaño de los grupos centrales.
- Reduciendo el tamaño del primer y del último grupo.

- Carry-select adder: Cada grupo genera dos sumas y dos carries. Una suma y un carry se corresponden a la suma con carry inicial 0. La otra suma y el otro carry se corresponden a la suma con carry inicial 1. Cuando se tiene el carry de entrada real, se selecciona la suma con su carry correspondiente. Salvo el primero, cada bloque tiene 2 sumadores. Esto va a necesita el doble del hardware. Para bloques de tamaño k:
  - $2k\Delta_G$ para calcular todas las sumas.
  - $2\Delta_G(n/k - 1)$ para seleccionar el acarreo y la suma correctos.

$$\mathbf{T = 2k\Delta_G + 2\Delta_G(n/k - 1) = (2k + 2n/k - 2)\,\Delta_G}$$

El mejor diseño del Carry-select adder también es con bloques de tamaño variable. Si cada bloque de k bits le lleva $2k\Delta_G$ realizar la suma y seleccionar el carry correcto lleva $2\Delta_G$. Entonces, lo ideal es que cada bloque sume un bit más que el anterior. Los grupos deberían seguir la serie 1, 1, 2,3,…, L_{max}. Para n bits se debe satisfacer que:

$$\mathbf{1 + L_{max}\,(L_{max}x - 1)/2 \geq n}$$

Por lo tanto

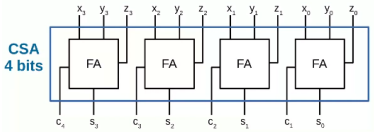
$$\mathbf{L_{max}\,(L_{max} - 1) \geq 2(n - 1)}$$

$$\mathbf{L_{max} \geq (1 + \sqrt{8n - 7})/2}$$

El tamaño del mayor grupo y el tiempo de ejecución son del orden de $\sqrt{n}$

- Semisumador: Carry-save adder (CSA): Sumar más de 2 operandos simultáneamente (como en la multiplicación) usando sumadores de dos operandos es costoso. Para k operandos la propagación del carry debe repetirse k – 1 veces. La estrategia usando Carry-save adder reduce ese costo. Generar sumas parciales y secuencias de carry, Propagar el carry sólo en el último paso. El CSA recibe 3 operandos de n bits y genera dos resultados n bits:
  - La suma parcial de n bits
  - El carry de n bits.

  Reduce la suma de 3 operandos a la suma de 2 operandos. La implementación más simple es con n FA en paralelo y no le agrega ninguna lógica.



Sumar k operandos de n bits requiere k – 2 CSA y un CPA. El tiempo de suma será:

$$(k-2)\,T_{CSA} + T_{CPA}$$

T_{CPA} = retardo del sumador

$T_{CSA} = \Delta_{FA} = 2\Delta_G$

La suma de k operandos de n bits, puede valer hasta, el peor caso, (2n - 1) k. el resultado final puede tener hasta $n + \lceil \log_2 k \rceil$ bits. El tiempo total de suma:

$$(k-2)\,2\Delta_G + T_{CPA}(n + \lceil \log_2 k \rceil)$$

Dependerá del sumador paralelo de la última etapa.

En un árbol de CSA, en cada nivel el número de operandos se reduce en 2/3. Por lo tanto, si L es la cantidad de niveles requeridos $k\,(2/3)^L \geq 2$. Es decir que se necesitaran una cantidad estimada de niveles: $\frac{log\frac{2}{n}}{log\frac{2}{3}}$

El resultado final requiere atravesar O (log(n)) CSA

Number of operands	Number of levels
3	1
4	2
$5 \leq k \leq 6$	3
$7 \leq k \leq 9$	4
$10 \leq k \leq 13$	5
$14 \leq k \leq 19$	6
$20 \leq k \leq 28$	7
$29 \leq k \leq 42$	8
$43 \leq k \leq 63$	9

Restadores

Podría implementarse circuito restador que realice la operación directamente, analizando las funciones lógicas de pedir prestado. Pero, dado que los sumadores pueden sumar números tanto signados como o no signados, con cualquiera de los sumadores vistos puede realizarse la resta usando complemento a la base. Solo hace falta invertir los dígitos del número a restar y sumar 1.

Entonces, para calcular A – B en complemento a la base

Se complementa B: {X ← B' (pero no le sumamos 1 ahora, así nos ahorramos una suma)

Se calcula A + X, forzando el acarreo inicial c_0 a 1 (este es el +1 del complemento la base que no hicimos en el paso anterior)

El problema de usar complemento a la base disminuida, es que si hubo acarreo después de la suma A+X tenemos que sumar 1 al resultado. Y no tenemos forma de evitar que este +1 implique hacer una suma nueva.

Multiplicadores

El multiplicador puede operar sobre enteros:

- No signados: Dos números de n bits

- Multiplicando: $M = m_{n-1}\, m_{n-2} \ldots m_0$
- Multiplicador: $X = x_{n-1}\, x_{n-2} \ldots x_0$

El producto P será

- $P = M \cdot X = M\,(\sum_{i=0}^{n-1} x_i\, 2^i) = \sum_{i=0}^{n-1} M\, x_i\, 2^i,\ xi \in \{0,1\}$

Se resuelve con n sumas de operandos de n bits. Por lo tanto, sería de O (n^2)

El producto de dos enteros no signados de n bits, puede dar como resultado máximo P_{max}, que es el número máximo que se puede codificar con n bits multiplicado por sí mismo.

$$(2^n - 1)(2^n - 1) = 2^{2n} \underbrace{- 2^{2n+1} + 1}_{<0} = 2^{2n-1} + \underbrace{(2^{2n-1} - 2^{n+1} + 1)}_{>0,\ n \geq 3}$$

Luego

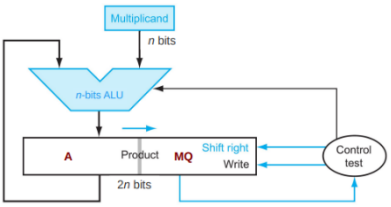
$$2^{2n-1} < P_{max} < 2^{2n}$$

El resultado P tiene como máximo 2n bits

Algoritmo secuencial más sencillo:

- M
- $X = x_{n-1}\, x_{n-2} \ldots x_0$
- P = 0
- Repetir para i = 0 : n – 1
  - $P \leftarrow P + x_i\, M\, 2^i$
- $P = \sum_{i=0}^{n-1} M\, x_i\, 2^i = M \cdot X$

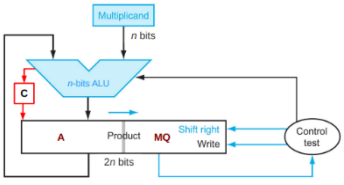
En el hardware del algoritmo secuencial se tiene un registro Producto P que es un registro doble A | MQ, donde A y MQ son registros de n bits, y una ALU que suma los registros de n bits A y Multiplicando



- $X \cdot M =$
- $A \leftarrow 0$
- $MQ \leftarrow$ Multiplicador
- P = A|MQ
- Repetir n veces
  - Si $P_0 = 1$, $A \leftarrow A +$ Multiplicando
  - //Si $P_0 = 0$, A no se modifica
  - Desplazar P 1 bit a derecha.

El registro doble P contiene el resultado

Lo que no se está teniendo en cuenta con este algoritmo es que las sumas parciales pueden generar carry-out. Lo que se hace es agregar el carry en el hw básico



- $X \cdot M =$
- $A \leftarrow 0$
- $MQ \leftarrow$ Multiplicador
- P = A|MQ
- Repetir n veces
  - Si $P_0 = 1$, $A \leftarrow A +$ Multiplicando
  - Si $P_0 = 0$, poner el acarreo en cero
  - Desplazar P 1 bit a derecha ingresando el acarreo.
- El registro doble P contiene el resultado

El acarreo no forma parte del resultado.

- Signados:
  - **Signo-magnitud**:
    - Calcular el producto sin signo

$|p| = |M| \cdot |X|$

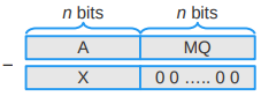
- Calcular el signo de forma separada

  $p_{2n-1} = yn_{-1} \oplus x_{n-1}$

Como el algoritmo anterior es para enteros no signados, es perfecto para multiplicar números en SM agregando la operación XOR para calcular el signo.

- **Complemento a 2 y complemento a 1**:
  - Caso 1: Multiplicando M negativo
    - $M = 2^n - |M|$
    - $M \cdot X = P' = (2^n - |M|) \cdot X = 2^n X - |M| \cdot X$ (realizado con el algoritmo ya visto)
    - $P = 2^{2n} - |M| \cdot X$

    P' no es el resultado P que se busca al hacer la multiplicación. Para esto hay 2 soluciones:
    - Solución 1: ver la diferencia entre el resultado obtenido y el resultado que necesitamos.
      - $P - P' = 2^{2n} - |M| \cdot X - (2^n X - |M| \cdot X)$ Por lo tanto $P - P' = 2^{2n} - 2^n X$
      - Corregir el resultado del algoritmo restando X de la parte más significativa del registro P (esto no requiere ALU de 2n bits)

    - Solución 2:
      - Considerar M de doble precisión: $M = 2^{2n} - |M|$
      - $M \cdot X = P' = (2^{2n} - |M|) \cdot X = 2^{2n} X - |M| \cdot X$
      - $2^{2n} X$ es mayor que la longitud del registro P por lo tanto es carry que se descarta.
      - $P' \equiv 2^{2n} - |M| \cdot X = P$
      - No implica ALU de 2n bits.

    Multiplicando negativo: Solución 2 ✓
    - Si solo el multiplicando es negativo, no hay necesidad de cambiar el algoritmo.
    - Se suma un número negativo y los productos parciales son negativos.

    El hardware debe extenderse de forma tal que provea extensión de signo en el producto parcial.
    - Antes de la primera suma (producto parcial = 0), en la extensión de signo ingresa 0.
    - Luego de la primera suma, en la extensión de signo ingresa m_{n-1}
      - Si $M < 0 \Rightarrow P < 0 \Rightarrow$ ingresa 1
      - Si $M \geq 0 \Rightarrow P \geq 0 \Rightarrow$ ingresa 0
  - Caso 2: Multiplicador X negativo
    - $X = 2^n - |X|$
    - $M \cdot X = P' = M \cdot (2^n - |X|) = 2^n M - M \cdot |X|$
    - $P = - M \cdot |X|$

    Solución 1:
    - La diferencia entre P y P' es $2^n M \therefore P = P' - 2^n M$

- Como P es un registro de 2n bits, -2^n M en 2 complemento es $2^{2n} - 2^n$ M
- Corregir el resultado del algoritmo restando M de la parte más significativa del registro P (esto no requiere ALU de 2n bits).

Solución 2:

- Asumir X de doble precisión: $X = 2^{2n} - |X|$
- $P' = M \cdot X = M \cdot (2^{2n} - |X|) = 22n\ M - M \cdot |X| \equiv 2^{2n} - M \cdot |X|$
- ¿Problema?
  - Multiplicador de doble precisión implica el doble de iteraciones.

Multiplicador Negativo: Solución 1✓

- Caso 3: Ambos negativos

  Se aplican las dos soluciones a la vez. Es decir, asumir el multiplicando de doble precisión y hacer la corrección una vez terminada la operación.

Algoritmo secuencial posee 2 problemas y un punto a favor:

✓ El multiplicando negativo es de solución trivial.
× Requiere corrección en caso de multiplicador negativo.
× Grandes secuencias de 1s en el multiplicador generan sumas sucesivas.

Soluciones:

- Operación en bases mayores a 2: es la recodificación más simple. Examinar de a varios bits a la vez reduce la cantidad de productos parciales, es decir operando en bases mayores. Requiere generar múltiplos del multiplicando. Examinar de a c bits reduce en c los productos parciales. No soluciona los casos particulares en función del signo de los operandos.
- Recodificación de Booth: Reduce la cantidad de productos parciales. Genera secuencias de 1s como la resta entre dos operandos de un solo 1 cada uno.

	1000	(8)		01000000	(64)
−	0001	(1)	−	00001000	(8)
	0111	(7)		00111000	(56)

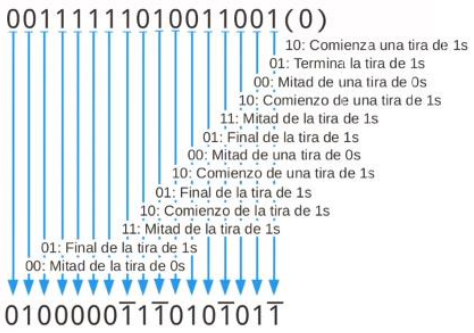
Se basa en que:

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

Recodifica el multiplicador a dígito signado. $X = x_{n-1} \ldots x_2\ x_1\ x_0$, donde $x_i \in \{0, 1, -1\ (o\ \bar{1})\}$. El principal problema del dígito signado es una notación redundante, es decir que un número puede escribirse de varias formas y todas son válidas: **56 = 0111000 = 100$\bar{1}$000 = 10$\bar{1}$1000**
Booth establece reglas para recodificar. Nótese que un número en binario también es un número en digito signado en el que justo no hay ningún digito que sea negativo. Recorrer el multiplicador X de derecha a izquierda (i = 0, 1... n – 1), del menos significativo la más significativo. Para cada par de bits consecutivos $x_i x_{i-1}$ generar el multiplicador recodificado Y, de tal forma que:

Asumir $x_{-1} = 0$

x_i	x_{i-1}	y_i	
0	0	0	En el medio de una tira de ceros
1	1	0	En el medio de una tira de unos
1	0	$\bar{1}$	Comienzo de una tira de unos
0	1	1	Final de la tira de unos



¿Qué modificaciones hay que hacerle al algoritmo secuencial básico para que opere con recodificación de Booth?

Para multiplicar M · X

- Recodificar X y obtener Y
- *//Multiplicar M · Y*
- A ← 0
- MQ ← Y
- Repetir n veces
  - Si $P_0 = 1$, A ← A + Multiplicando
  - Si $P_0 = -1$, A ← A – Multiplicando
  - *//Si $P_0 = 0$, no se modifica A*
  - Desplazar P 1 bit a derecha (extensión de signo).

Pero ahora, ¿Cuál es el signo de P? P tiene el signo del último número que se le haya sumado.

El hardware necesario para codificar un número en binario es de tan solo un registro. En cambio, el hardware necesario para codificar en digito signado son 2 registros: **R (+)** parte positiva y **R (-)** parte negativa. Esto se debe a que no es posible codificar en el hardware el $\overline{1}$. Trabajar en digito signado implica el doble de hardware y el doble de conexiones.

Por eso, en vez de recodificar el multiplicador por adelantado, almacenarlo en un registro y después operar sobre ese registro, lo que vamos a hacer es ir reedificando on the fly para saber cuál es la próxima operación que tenemos que realizar. De esta forma, para actualizar A vamos a mirar 2 bits y en función de si comienza o termina una tira de 1s se va a restar o se va a sumar.

Para multiplicar M · X

- A ← 0
- MQ ← Y
- P_{-1} ← 0 *//bit anterior al menos significativo necesario para recodificar*
- Repetir n veces
  - Si $P_0P_{-1} = 01$ *//Final de la tira de unos*
    - A ← A + Multiplicando
  - Si $P_0P_{-1} = 10$, *//Comienzo de la tira de unos*
    - A ← A – Multiplicando
  - *//Si P_0P_{-1} = 00 o 11, no se modifica A*
  - P_{-1} ← P_0
  - Desplazar P 1 bit a derecha (extensión de signo).

El algoritmo secuencial con recodificación de Booth funciona correctamente con números en complemento a 2. Pero en el caso de números no signados, se debe agregar un cero a la izquierda del multiplicador ($x_n = 0$). El problema que tiene es que se vuelve ineficiente con unos aislados (se duplica la cantidad de operaciones).

Solución, Recodificación de Booth de a más bits: combina Booth con la recodificación de más de a un bit. La recodificación de Booth mirando solamente 2 bits es Booth radix-2 porque recodifica un único bit. Booth raidx-4 recodifica 2 bits mirando 3, Booth radix-8 recodifica 3 bits mirando 4.

La recodificación de Booth de varios bits combina la recodificación de Booth con la recodificación de más de a un bit. Existen dos formas de recodificación:

- Mirando al futuro
- Mirando al pasado

En cualquiera de los dos casos si la recodificación es de c bits, hay que mirar c + 1 bits.

Por ejemplo, en la recodificación de Booth radix-4 se dividen en grupos de 2

$$x_7x_6 \mid x_5x_4 \mid x_3x_2 \mid x_1x_0$$

Se usa un tercer bit para recodificar

$$\underbrace{x_7x_6}_{y_7y_6} \mid \underbrace{x_5x_4}_{y_5y_4} \mid \underbrace{x_3x_2}_{y_3y_2} \mid \underbrace{x_1x_0}_{y_1y_0}$$

Mirando al pasado: Para codificar x_{i+1} y xi, se usa x_{i-1} como referencia (i = 0, 2,4,...)

$$\underbrace{x_7}_{y_7y_6}\ \underbrace{x_6\ x_5}_{y_5y_4}\ \underbrace{x_4\ x_3}_{y_3y_2}\ \underbrace{x_2\ x_1}_{y_1y_0}\ x_0\ (x_{-1})$$

	x_{i+1}	x_i	x_{i-1}	$y_{i+1}y_i$	Múltiplo
Entre dos cadenas	0	0	0	00	0M
Final de cadena de 1s	0	0	1	01	M
1 aislado	0	1	0	01	M
Final de cadena de 1s	0	1	1	10	2M
Principio de cadena de 1s	1	0	0	$\bar{1}0$	– 2M
0 aislado	1	0	1	$0\bar{1}$	– M
Principio de cadena de 1s	1	1	0	$0\bar{1}$	– M
Medio de una cadena	1	1	1	00	0M

Mirando al futuro: Para codificar x_{i+1} y xi, se usa x_{i+1} como referencia (i = 0, 2,4,...)

$$\ldots\ \underbrace{x_7}_{y_7y_6}\ \underbrace{x_6\ x_5}_{y_5y_4}\ \underbrace{x_4\ x_3}_{y_3y_2}\ \underbrace{x_2\ x_1}_{y_1y_0}\ \underbrace{x_0\ (x_{-1})}_{y_0y_{-1}}(x_{-2})$$

	x_{i+2}	x_{i+1}	x_i		Múltiplo
Entre dos cadenas	0	0	0	00	0M
Final de cadena de 1s	0	0	1	10	2M
1 aislado	0	1	0	10	4M – 2M = 2M
Final de cadena de 1s	0	1	1	100	4M
Principio de cadena de 1s	1	0	0	$\bar{1}00$	– 4M
0 aislado	1	0	1	$\bar{1}0$	– 4M + 2M = – 2M
Principio de cadena de 1s	1	1	0	$\bar{1}0$	– 2M
Medio de una cadena	1	1	1	00	0M

Una de las particularidades de esta recodificación es que los múltiplos que obtenemos para la recodificación son 0, 2,4, -2 y -4. Por lo tanto no vamos a poder almacenar el multiplicador recodificado como si lo hacemos en Booth radix-2 o en el caso de Booth radix-4 mirando al pasado. Igualmente como nosotros no

almacenamos los números recodificados, si vamos a poder operar on the fly y viendo cual es el número que tenemos que sumar o restar.

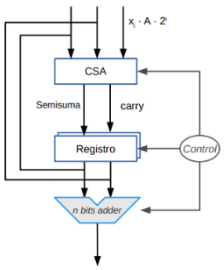
Con la recodificación de 2 bits se necesitan distintos múltiplos de M:

- Sin recodificación de Booth
  - 0M, M, 2M y 3M
- Con recodificación de Booth mirando al pasado
  - 0M, M y 2M
- Con recodificación de Booth mirando al futuro
  - 0M, 2M y 4M

Múltiplos triviales: 0 y 1

Múltiplos fáciles de obtener: 2 y 4

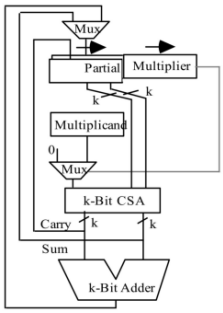
Hasta ahora hemos visto cómo mejorar la multiplicación modificación el algoritmo pero manteniendo el hardware básico de multiplicación secuencial. Con la recodificación de Booth se mejora pero sigue teniendo problemas, por ejemplo propaga carry.



La mejora más simple es evitar propagar acarreo en todas las sumas y para eso se reemplaza el sumador por un CSA, donde se realizan las sumas intermedias hasta obtener una única semisuma y un único carry. En el último paso se realiza la suma propagando carry con un sumador. Realiza n iteraciones para multiplicar operandos de n bits.

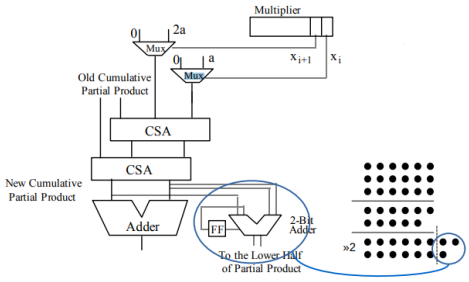
Algoritmo usando CSA

X ◦ M = ?



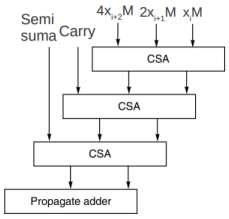
- $[A_S,A_C] \leftarrow [0,0]$
- MQ ←Multiplicador
- $P = A_S|MQ$
- Repetir n veces
  - Si $P_0 = 1$, $[A_S,A_C] \leftarrow CSA(A_S,AC,M)$
  - Si $P_0 = 0$, $[A_S,A_C] \leftarrow CSA(A_S,A_C,0)$
  - Desplazar P 1 bit a la derecha.
- $A_S \leftarrow A_S + A_C$
- El registro doble P contiene el resultado.

También se pueden agregar a más niveles de CSA. En este ejemplo tenemos 2 niveles de CSA con los que podemos sumar 4 números: 2 son la semisuma y el acarreo y los otros 2 son múltiplos del multiplicando. Combina la semisuma y el carry del CSA con los múltiplos x_iA y $2x_{i+1}A$.



Con tres niveles de CSA se puede combinar la semisuma y el carry con tres múltiplos del multiplicando:

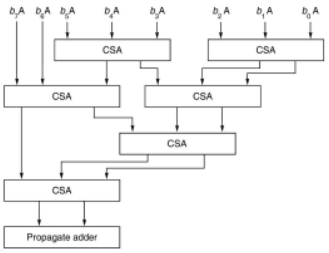
- x_iM
- $2x_{i+1}M$
- y $4x_{i+2}M$



Con 2 niveles de CSA reduce a la mitad la cantidad de iteraciones (n). En cada iteración el camino más largo atraviesa los dos CSA. El retardo por los CSA es de 2n Δ_{CSA}. Ocurre algo análogo para 3 CSA. Con 4 o más CSA se puede lograr paralelismo (Wallace tree).

La cantidad de iteraciones se reduce en 4. En cada iteración se atraviesan hasta 3 CSA. En total el retardo por CSA es de 3/4 n Δ_{CSA}.

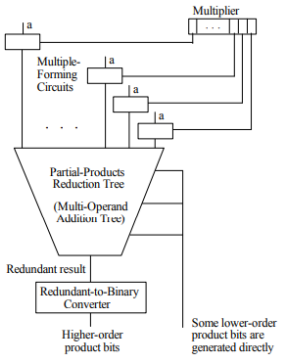
Para maximizar el paralelismo se puede armar un Wallace Tree de n operandos de n bits. El resultado final requiere atravesar O (log(n)) CSA.



Number of operands	Number of levels
3	1
4	2
$5 \leq k \leq 6$	3
$7 \leq k \leq 9$	4
$10 \leq k \leq 13$	5
$14 \leq k \leq 19$	6
$20 \leq k \leq 28$	7
$29 \leq k \leq 42$	8
$43 \leq k \leq 63$	9

En su forma más general, los arboles multiplicadores pueden combinar varios múltiplos de A

- En binario
- Recodificación de varios bits
- Recodificación de Booth

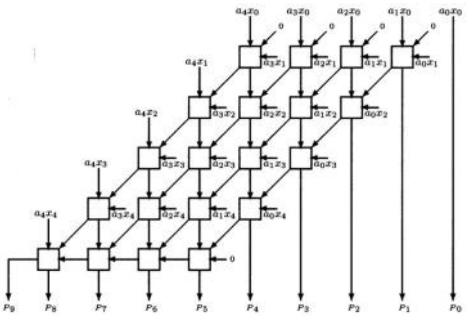


Un árbol de reducción (combinacional) genera los productos parciales. El resultado está en notación redundante y se convierte a binario.

Otra estrategia de implementación muy simple para la multiplicación es el array multiplayer. Esta es una implementación que combina lo más lento de todo lo que vimos para realizar sumas: CSA en cascada y Ripple adder.

¿Por qué es interesante este multiplicador tan lento? Es un layout simple y eficiente para diseño VLSI. Estructura muy regular. Es fácilmente adaptable a complemento a 2. Las conexiones son cortas. Cada FA se conecta con adyacente:

- Vertical
- Horizontal
- Diagonal



Una desventaja de las implementaciones secuenciales es que si bien llevan muchos ciclos resolver la multiplicación, todo el hardware está siendo usado para resolver una única operación. Las implementaciones que no requieren iteraciones, como un Wallace tree completo o un array multiplayer, pueden dividirse en etapas. Cuando una multiplicación termina una etapa y pasa a otra, libera la etapa anterior y esta pueda ser usada para realizar otra operación. En este caso estaríamos hablando de **multiplicadores en pipeline**.

Divisores

Es la más compleja y que consume tiempo de las operaciones básicas. Dados un dividendo X y un divisor D, el resultado de X/D consta de un **cociente Q** y un **resto R**, tales que: $X = Q \cdot D + R$ con $0 \leq R < D$.

En base 2 tenemos

- Dividendo X de 2n bits.
- Divisor D y resto R de n bits.
- Cociente Q de n + 1 bits

Si bien la multiplicación entraba en un registro de 2n bits, los resultados de la división ocupan en total 2n+1 bits.

Si se trabaja con registros de n bits, que el cociente resulte de n + 1 bits genera overflow.

Posibles errores que se pueden generar:

- División por cero (solución: $D \neq 0$)
- Overflow (solución: $X < 2^n \cdot D$)

Hay dos grandes grupos de soluciones:

- <u>Restar (o sumar) y desplazar:</u>
  - **Con restauración (restoring division):** Para cada bit del cociente prueba con valor 1. Si no era la elección correcta porque la resta dio un resultado negativo, restaura el valor anterior y el bit ahora vale 0.

    División secuencial

    La restauración del valor puede hacerse:
    - Manteniendo registros separados para el resto de prueba y el resto real.
    - Con un único registro y realizando la suma para restaurar. (usa un registro menos pero realiza 2 operaciones)

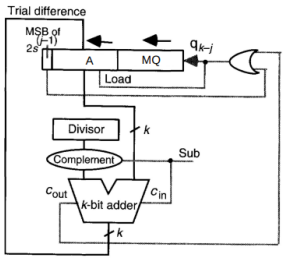
    Algoritmo de división con restauración:

    1) A|MQ ← X *//Si X es de n bits, asignamos* $0_{2n-1\ldots n+1}X$
    2) B ← D
    3) $A_{n+1} \leftarrow A_n$
    4) Desplazar A|MQ un lugar a izquierda
    5) Repetir n veces:
       a. A' ← A – B *//calculamos el resto de prueba.*
       b. Si $c_{out} = 1$ o $A_{n+1} = 1$ *//el resto de prueba es positivo*
          i. A ← A'
          ii. q ← 1 *//próximo bit del resultado*
       c. Si no, q ← 0
       d. Si es el último ciclo, desplazar MQ a izquierda ingresando q
       e. Si no, desplazar A|MQ a izquierda ingresando q
    6) Cociente ← MQ, Resto ← A

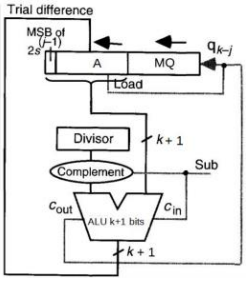
    Porque la parte alta de X debe ser menor a D para que no haya overflow. Ingresa cualquier cosa, en (e) se descarta.

    ¡Ojo! No estamos teniendo en cuenta A_{n+1}

    ¿Esto indica que $A_{n+1}|A - D \geq 0$?



Conceptualmente el algoritmo se puede simplificar asumiendo el registro A y la ALU de n + 1 bits. Con esta simplificación el acarreo de salida de la ALU ya nos va a estar indicando el signo del resultado



Algoritmo de división con restauración simplificado:

1) A|MQ ← 0X *//Si X es de n bits, asignamos* $0_{2n\ldots n+1}X$
2) B ← 0D
3) Desplazar A|MQ un lugar a izquierda
4) Repetir n veces:
   a. A' ← A – B *//calculamos el resto de prueba*.
   b. Si A' ≥ 0 *//el resto de prueba es positivo*
      i. A ← A'
      ii. q ← 1 *//próximo bit del resultado*
   c. Si no, q ← 0
   d. Si es el último ciclo, desplazar MQ a izquierda ingresando q
   e. Si no, desplazar A|MQ a izquierda ingresando q
5) Cociente ← MQ, Resto ← $A_{n-1\ldots0}$

El hardware de división es muy similar al de multiplicación. Por lo que pueden compartir registros y la ALU. La única particularidad es que en este hardware mixto, A|MQ debería poder desplazarse tanto a izquierda como a derecha.
Tiene problemas de temporizado. Cada ciclo de reloj debe ser lo suficientemente largo para permitir:

- Desplazar registros
- Propagar señales al sumador
- Determinar y almacenar el próximo dígito del cociente
- Eventualmente, almacenar el resto de prueba

Los últimos eventos dependen de los primeros.

- **Sin restauración (nonrestoring division)**: Busca evitar esos problemas de temporizado. Siempre asume que $q_i = 1$, resta y almacena el resto parcial aunque sea incorrecto. Se corregirá en el próximo ciclo.
  Algoritmo de división sin restauración versión simplificada:

1) A|MQ ← 0X //Si X es de n bits, asignamos $0_{2n\ldots n+1}X$
2) B ← 0D
3) Desplazar A|MQ un lugar a izquierda
4) q ← 1
5) Repetir n veces:
   a. A ← A + $(-1)^q$B
   b. q ← (A ≥ 0) ? 1 : 0
   c. Si es el último ciclo, desplazar MQ a izquierda ingresando q
   d. Si no, desplazar A|MQ a izquierda ingresando q

Restamos o sumamos B en función de lo que haya pasado antes

6) Si A < 0, A ← A + B

Tenemos en A el resto parcial ya desplazado. Si la resta A – D < 0 hay dos opciones:

- Restaurar el valor previo de A, desplazar a izquierda nuevamente y volver a restar D (como hace el algoritmo con restauración):

$$\mathbf{2A - D}$$

- Mantener el valor negativo, desplazar izquierda y sumar D (como hace el algoritmo sin restauración):

$$\mathbf{2(A - D) + D = 2A - D}$$

Si el resto final queda negativo, hay que corregir ya que no va a haber un próximo paso para corregir

$$\mathbf{R \leftarrow R + D}$$

- **SRT:** Acelera la división sin restauración usando digito signado. Para el cociente usa los dígitos {-1, 0, 1}
  - -1: sumar
  - 0: no operar
  - 1: restar

Busca evitar sumar/restar cuando el resto parcial está entre [–D,D).

- $q_i = 0$ y se desplaza el resto parcial a izquierda [–2D, 2D).

¿Cómo se sabe que el resto parcial está entre [–D,D)? Restando, pero es lo que queremos evitar. La solución se conoce como algoritmo SRT. **S**weeney, **R**obertson y **T**ocher llegaron independientemente al mismo algoritmo.

1) A|MQ ← 0X
2) B ← 0D
3) Si D tiene k ceros adelante, desplazar A|MQ y B k lugares a izquierda.
4) Repetir n veces
   a. Si los tres MSB de A son iguales: q ← 0
   b. Si los tres MSB de A no son iguales y A < 0: q ← –1
   c. Si no: q←1
   d. Desplazar A|MQ 1 bit a izquierda ingresando q
   e. A ← A – q B
5) Si el resto final es negativo, sumar B y restar 1 a MQ.
6) Desplazar A k lugares a derecha.
7) Cociente ← MQ
8) Resto ← $A_{n-1\ldots 0}$

Explicación:

- X/D = Q implica X = D · Q + R (0 ≤ R < D)
- Si asumimos un resto parcial r_i (0 ≤ r_i < D)

$$r_i = 2r_{i-1} - q_i \cdot D$$

  Donde q_i va a ser igual a 0, 1 o -1.
- La selección de los bits del cociente es:

$$q_i = \begin{cases} 1 & \text{if } 2r_{i-1} \geq D \\ 0 & \text{if } -D \leq 2r_{i-1} < D \\ \bar{1} & \text{if } 2r_{i-1} < -D \end{cases}$$

- La dificultad está en comparar $2r_{i-1}$ con D o –D para saber el próximo valor de q
- Si D fuera una fracción normalizada (½ ≤ |D| < 1) se podría determinar parte de la zona en la que $q_i = 0$

- q = 0 cuando

$$-D \le 2r \le D$$

Como ½ ≤ |D|, entonces q= 0 cuando

$$-\tfrac{1}{2} \le 2r < \tfrac{1}{2}$$
$$-\tfrac{1}{4} \le r < \tfrac{1}{4}$$
$$1.110_2 \le r < 0.01_2$$
$$1.110_2 \le r \le 0.00111\ldots_2$$

En este caso los tres primeros bits van a ser iguales. Una simplificación es en vez de analizar los primeros 3 bits alcanza con los primeros 2 excluyendo el signo. Esto permite usar la ALU de n bits, en lugar de ALUs de n+1. Es cierto que con esto no se van a capturar todos los casos en los que no haya que operar pero se van a reducir y se van a realizar menos cantidad de operaciones equivocadas.

- ¿Si D es entero?
  - Para registros de n bits:

$$a = X/2^n \qquad \frac{X}{D} = \frac{a}{b} = \frac{X/2^n}{D/2^n} = Q \qquad r = R/2^n$$
$$b = D/2^n \qquad\qquad\qquad\qquad\qquad\quad a = b \cdot Q + r$$

- ¿Si D no usa fracción normalizada (D ≥ ½)?
  Si D no está normalizado, desplazar X y D k lugares a izquierda para normalizar D.

- Multiplicar
  - **Por convergencia**: Realiza la división a través de multiplicaciones sucesivas. Apta si hay implementados multiplicadores rápidos. Adecuado para división en punto flotante. Calcula solamente el cociente. La división de n bits con m bits de precisión lleva $\log_2 (n/m)$ iteraciones.
    El cociente Q es el resultado de X/D

$$Q = \frac{X}{D}$$

Por lo tanto:

$$\frac{X}{D} = \frac{X \cdot R}{D \cdot R} \to Q \quad \text{si } D \cdot R \to 1$$

$$D \cdot R_0 \cdot R_1 \cdots R_{m-1} \to 1$$

Supongamos D una fracción normalizada 0.1•••

$$\tfrac{1}{2} \le D < 1$$
$$D = 1 - z \text{ donde } z \le \tfrac{1}{2}.$$

Si se selecciona $R_0 = 1 + z$

- $D_1 = D \cdot R_0 = (1 - z)(1 + z) = 1 - z^2$
- Como $z^2 \le \frac{1}{4}$, $D_1 = 0.11\bullet\bullet\bullet \ge \frac{3}{4}$

Si se selecciona $R_1 = 1 + z^2$

- $D_2 = D_1 \cdot R_1 = (1 - z^2)(1 + z^2) = 1 - z^4$
- Como $z^4 \le 1/16$, $D_2 = 0.1111\bullet\bullet\bullet \ge 15/16$

Convergencia

- En general $D_i = 1 - z^{2i}$
- Como $z \le 2^{-1}$ entonces $z^{2i} \le 2^{-2i}$

$$z^{2^i} = 0.\underbrace{0\ldots0}_{i}\ldots$$
$$D_i = 1 - z^{2^i} = 0.\underbrace{1\ldots1}_{i}\ldots$$
$$\lim_{i \to \infty} D_i = 1$$

Cálculo del factor de multiplicación R_i

- Para obtener R_i a partir de D_i

$$D_i = 1 - z^{2^i} \qquad R_i = 1 + z^{2^i}$$
$$R_i = 1 + 1 - D_i = 2 - D_i$$

- R_i es el complemento a 2 de D_i

Procesador Central

La microarquitectura se divide en dos partes que interactúan entre sí:

- Datapath: Componentes de HW que operan sobre los datos. Determina la estructura estática del procesador. Involucra memorias, registros, ALUs y multiplexores.
- Unidad de Control o Lógica de Control: A partir de la instrucción actual, el estado del procesador y los resultados intermedios de la ejecución determina el flujo dinámico de los datos a lo largo del datapath. Produce señales de multiplexado, habilitaciones de registros y señales a la memoria.

Los pasos básicos para ejecutar una instrucción:

1. Se trae de memoria la próxima instrucción a ejecutar (apuntada por el PC)
2. La unidad de control decodifica la instrucción.
3. Se ejecuta la instrucción. Puede ser:
   - Operación de la ALU
   - Cargar un registro con un dato de memoria
   - Almacenar un dato de un registro en memoria
   - Testear la condición de un salto
4. Se actualiza el PC
5. Volver al paso 1

Hay diferentes formas de diseñar una microarquitectura y se la puede dividir en 3 grandes grupos:

- Diseño único ciclo: Ejecuta la instrucción completa en un único ciclo (CPI = 1). Comienza en un flanco ascendente (o descendente) y termina en el próximo flanco ascendente (o descendente). El ciclo tiene que ser lo suficientemente largo para permitir la ejecución de la instrucción más lenta. No es práctico pero es simple de implementar y entender. Además es la base para la microarquitectura en pipeline.

  R-type (add, or, etc):

  - R[d] ← R[s] + R[t]
  - PC ← PC + 4

  1. Traer la instrucción y decodificarla
  2. Obtener los operandos: leer el registro del banco de registros los dos operandos
  3. Realizar la operación de la ALU
  4. Escribir el resultado en el banco de registros.
  5. Actualizar el PC

  Load:

  - R[d] ← Mem[R[s] + #valor]
  - PC ← PC + 4

  1) Traer la instrucción y decodificarla
  2) Obtener los operandos: leer el registro del banco de registros y obtener el valor inmediato
  3) Calcular la dirección de memoria
  4) Leer el dato de memoria y escribirlo en el banco de registros.
  5) Actualizar el PC

Arquitectura MIPS: Instrucciones

Instrucciones aritmético-lógicas

- R-type (add, or, mul…)
- I-type (addi)

Instrucciones de escritura y lectura en memoria

- I-type (load y store)

Instrucciones de bifurcación

- I-type (beq)

Store:

- Mem[R[d] + #valor] ← R[s]
- PC ← PC + 4

1) Traer la instrucción y decodificarla
2) Obtener los operandos: leer los registros s y d del banco de registros y el valor inmediato.
3) Calcular la dirección destino
4) Escribir el valor del registro en memoria.
5) Actualizar el PC

Branch (beq):

- Si R[s] == R[t]
  - Entonces PC ←PC + offset
  - Si no PC ← PC + 4

1) Traer la instrucción y decodificarla
2) Obtener los operandos: leer los registros s y t del banco de registros y el valor inmediato.
3) Evaluar la condición
4) Actualizar el PC en función de la condición

Implica:

✓ Diseño simple

✗ El período del ciclo está limitado por la instrucción más lenta (load).

✗ Necesita 2 sumadores además de la ALU

✗ Fuerza memorias separadas para datos e instrucciones.

✗ La operación del datapath es puramente combinacional. La máquina está en estados estables solamente al principio y al final del ciclo.

- Diseño multi-ciclo: Ejecuta las instrucciones en una serie de ciclos cortos. Reduce el hardware ya que permite reusar bloques de hardware. Agrega hardware para almacenar resultados intermedios. Ejecuta una instrucción por vez y cada instrucción demanda múltiples ciclos.

  ✓Permite mayor frecuencia de reloj (en cada ciclo hay que hacer menor cantidad de cosas).

  ✗ No todos los pasos son de igual duración. El reloj debe permitir el paso más lento.

  ✗ Agrega la sobrecarga de los registros entre los pasos.

  ✗ Ejecuta una instrucción por vez.

- Diseño en pipeline: combina las dos microarquitecturas anteriores. Se divide la microarquitectura único ciclo en etapas. Cada instrucción se ejecuta en varios ciclos, pero todas en la misma cantidad de ciclos. Las etapas se ejecutan en pipeline y diferentes etapas pueden ejecutarse en paralelo. Los procesadores modernos están diseñados en pipeline.

Vamos a tener distintas microarquitecturas para la misma arquitectura ¿Cómo se comparan dos microarquitecturas?

- Tecnología: La velocidad del reloj (GHz, MHz) y el número de transistores indican cómo progresa la tecnología, pero no cuán rápido va a ejecutar un programa. Desde el punto de vista del usuario, lo que importa es cuán rápido el procesador ejecuta la tarea que tiene que hacer.
- Medidas de desempeño: Indican el rendimiento de un procesador y la jerarquía de memoria independientemente del programa en ejecución y de los dispositivos de entrada/salida. Estas son algunas de las medidas de desempeño más usuales:

- **Ciclos por instrucción (CPI)**: Cuántos ciclos de reloj demanda (en promedio) ejecutar una instrucción.
- **Instrucciones por ciclo (IPC)**: Es la recíproca del CPI y determina la cantidad de instrucciones que se ejecutan en un ciclo.

  $$IPC = 1 / CPI$$
- **Tiempo de ejecución (EX_{CPU})**: Indica cuánto tiempo (segundos o fracción de segundos) demora en ejecutar la secuencia de instrucciones (programa).

  $$EX_{CPU} = \#Instrucciones \times CPI \times período\ del\ reloj$$

  $\#Instrucciones \times CPI$: #Total de ciclos; $período\ del\ reloj$: 1/frecuencia
- **Latencia = EX_{CPU}**: Unidades de tiempo (segundos) que demanda realizar un trabajo. En este contexto puede referirse a la latencia de una instrucción, es decir cuánto tiempo demora una instrucción en ejecutar, o a la latencia de un programa, que sería equivalente al Tiempo de ejecución de un programa.
- **Throughput ≈ IPC**: Cantidad de trabajo (instrucciones) por unidad de tiempo (segundo)

  $$\#instrucciones / EX_{CPU}$$
- **Rendimiento (performance)**: Recíproca del tiempo de ejecución, a menor tiempo de ejecución mejor rendimiento

  $$1/EX_{CPU}$$

  Los tres factores que afecta el rendimiento son:
  - #instrucciones → Compilador
  - CPI (o IPC) → Diseño e implementación de la arquitectura
  - Frecuencia del reloj → Tecnología

  La comparación de rendimiento de dos sistemas se realiza en términos relativos (no absolutos). Un sistema A tiene mejor rendimiento que un sistema B si el sistema A tiene menor tiempo de ejecución (para un conjunto de programas) que el sistema B.

  $$\frac{Rendimiento_A}{Rendimiento_B} = \frac{1/EX_{CPU\,A}}{1/EX_{CPU\,B}} = \frac{EX_{CPU\,B}}{EX_{CPU\,A}}$$
- **Speedup**: es la razón entre el rendimiento de un sistema mejorado y el rendimiento de su implementación original

  $$Speedup = \frac{Rendimiento\,Mejorado}{Rendimiento\,Original} = \frac{EX_{CPU\,Original}}{EX_{CPU\,Mejorado}}$$

  Es la ganancia por mejorar un sistema.
- **Eficiencia**: mide la utilización de un recurso. Si $Speedup_n$ es la ganancia por mejorar el sistema con n recursos, la eficiencia mide la utilización de esos recursos.

  $$Eficiencia = \frac{Speedup_n}{n}$$
- **Ley de Amdahl**: El speedup se definió inicialmente para procesadores paralelos. Se buscaba medir la mejora en un sistema al agregarle más procesadores en paralelo. Si T_1 es el tiempo de ejecución en un procesador y T_n es el tiempo de ejecución en n procesadores, lo esperable sería que:

  $$T_n = T_1 / n$$

  Y por lo tanto

  $$Speedup = \frac{T_1}{T_n} = \frac{T_1}{T_1/n} = n$$

  El problema está en que no todo el código es igualmente paralelizable. La ejecución de un programa consiste de una parte secuencial seguido de una parte paralelizable. Si:
  - s es el tiempo de ejecución de la parte secuencial
  - p es el tiempo de ejecución secuencial de la parte paralelizable

  Entonces

  $$T_1 = s + p$$
  $$T_n = s + p / n$$
  $$Speedup = \frac{T_1}{T_n} = \frac{s+p}{s+p/n}$$

Si

$$Speedup = \frac{T_1}{T_n} = \frac{s+p}{s+p/n}$$

En el límite $(n \to \infty)$

$$\lim_{n\to\infty} Speedup = \frac{s+p}{s}$$

Luego, si asumimos $T_1 = 1$ (o 100%) el máximo speedup 1/s está limitado por la fracción de tiempo que se consume en la ejecución de la parte secuencial.

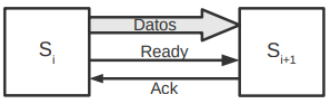
- Benchmarks: Un conjunto de programas que evalúan alguna característica en particular del procesador. Los benchmark que evalúan el procesador y la jerarquía de memoria, deben hacerlo de forma lo más independiente posible de las demás características del sistema.

Diseño en pipeline

La idea del **pipeline** es dividir a una tarea en subtareas menores de forma que cada una pueda realizarse independientemente de lo que esté ocurriendo con las demás subtareas.

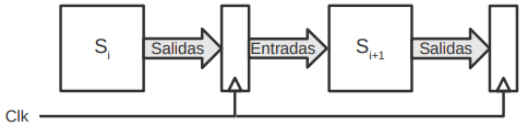
Los pipelines pueden clasificarse de diferentes maneras:

- Nivel de procesamiento:
  - *Pipeline aritmético*: Las operaciones aritméticas se segmentan para permitir el procesamiento en pipeline.
  - *Pipeline de instrucciones*: Las instrucciones se procesan en pipeline, permitiendo el solapamiento.
  - *Pipeline gráfico*
  - *Pipeline de software*
  - *Pipeline de procesadores*
- Configuración y estrategia de control:
  - *Unifuncional vs. Multifuncional*:
    - Un pipeline unifuncional realiza una única tarea fija.
    - Un pipeline multifuncional puede realizar varias tareas diferentes, en simultáneo o no.
  - *Estático vs. Dinámico*:
    - Un pipeline estático asume una única configuración por vez. Un pipeline unifuncional es estático.
    - Un pipeline dinámico permite varias configuraciones en simultáneo. Un pipeline dinámico es multifuncional.
  - *Escalar vs. Vectorial*:
    - Un pipeline escalar procesa operandos escalares (enteros, punto flotante o valor lógico).
    - Un pipeline vectorial procesa instrucciones vectoriales sobre operandos vectoriales.
- Sincronización entre etapas:
  - *Modelo Asincrónico*: El flujo de datos entre etapas adyacentes se realiza a través de un protocolo de sincronización (handshaking). Cuando la etapa S_i está lista para transmitir le envía una señal de ready a la etapa S_{i+1}. Cuando la etapa S_{i+1} recibe el dato le envía a Si una señal de acknowledge.



  - *Modelo Sincrónico*: Las etapas del pipeline están separadas por registros habilitados por pulsos de un reloj. Aíslan las salidas de una etapa con las entradas de la siguiente.

Con la habilitación del pulso de reloj todos los registros transfieren los datos a la próxima etapa.



- Conexión entre etapas
  - *Pipeline lineal*: Las etapas de procesamiento están conectadas en cascada, y los datos fluyen linealmente de un extremo al otro.
    Una tarea se puede dividir en k subtareas. La ejecución de las k subtareas equivale a la ejecución de la tarea original. Cada subtarea se ejecuta con un hardware específico. Cada subtarea puede ejecutar concurrentemente con otras subtareas.
    El pipeline tiene k etapas de procesamiento

    $S_1 \ldots S_k$

    Las entradas externas ingresan al pipeline por la etapa S_1 Los resultados del procesamiento de la etapa S_i es la entrada de la etapa S_{i+1}. El resultado final es la salida de la etapa S_k

    Cada etapa es combinacional. Es deseable que todas las etapas tengan aproximadamente el mismo retardo. El máximo retardo de las etapas determina el período del reloj y por lo tanto la velocidad del pipeline. Esto no quiere decir que las subtareas demoren lo mismo sino que las etapas más rápidas estarán ociosas mientras terminan las más lentas.
    Si τ_i es el retardo de la etapa S_i y d es el retardo de los registros interetapas, entonces el reloj τ es

    $$\tau = \max \{\tau_i\} + d$$
    $$\max \{\tau_i\} >> d$$

    La frecuencia del pipeline es la frecuencia del reloj que sincroniza el pipeline:

    $$f = 1/\tau$$

    Si una tarea demanda un tiempo de $k\tau$, donde τ es el período del reloj. En un procesador no en pipeline, n tareas demandar un tiempo total de $nk\tau$. Idealmente, en un pipeline de k etapas
    - La primera tarea demora k ciclos en completar
    - Cada 1 ciclo termina una tarea nueva.

    El tiempo total requerido para n tareas será

    $$T_k = (k + n–1)\ \tau$$

    Medidas de desempeño para comparar como mejora un procesador secuencial al reorganizarlo en un pipeline de k etapas:
    - El speedup de un pipeline de k etapas, en comparación a un procesamiento lineal es

      $$\text{Speedup}_k = \frac{T_1}{T_k} = \frac{nk\tau}{k\tau + (n-1)\tau} = \frac{nk}{k+(n-1)}$$

      $$\lim_{n\to\infty} \text{Speedup}_k = k$$

      El máximo speedup se logra con n >> k
    - La Eficiencia mide la utilización de los recursos (las k etapas)

      $$\text{Eficiencia} = \frac{\text{Speedup}_k}{k} = \frac{nk}{k(k+(n-1))}$$
      $$= \frac{n}{k+(n-1)}$$

      $$\lim_{n\to\infty} \text{Eficiencia} = 1$$
    - El throughput es la cantidad de tareas que se completan por unidad de tiempo.

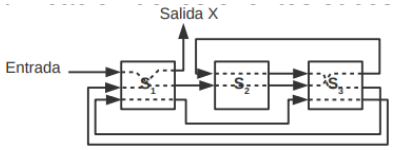
$$\text{Throughput} = \frac{n}{\tau(k+n-1)} = \frac{\text{Eficiencia}}{\tau}$$

Cuando la cantidad de tareas tiende a infinito, la eficiencia es máxima y el throughput iguala la frecuencia del pipeline:

$$\lim_{n\to\infty} \text{Throughput} = \lim_{\text{Eficiencia}\to 1} \frac{\text{Eficiencia}}{\tau} = 1/\tau = f$$

- Pipeline no lineal: permite
  - Conexiones lineales $S_i \rightarrow S_{i+1}$
  - Conexiones hacia adelante $S_i \rightarrow S_j$ tal que $i+1 < j$
  - Conexiones hacia atrás $S_i \rightarrow S_j$ tal que $j < i$

  Cualquier etapa puede generar la salida. La planificación de los eventos sucesivos no es trivial. Cada etapa tiene más de una etapa anterior y más de una siguiente etapa.

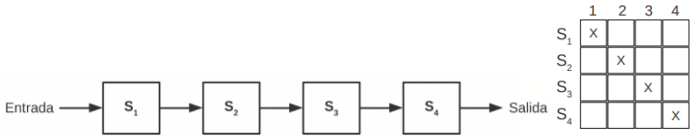


La tabla de reservación representa el flujo de las tareas en el pipeline.

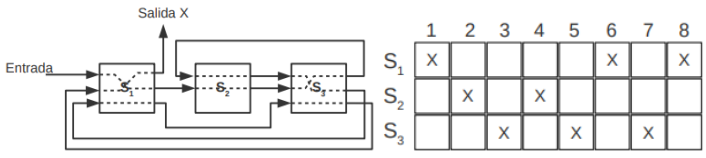
- Las filas representan las etapas del pipeline.
- Las columnas representan los ciclos de operación.
- Se marca en qué ciclo la tarea se encuentra en cada etapa.
  - Más de una marca por fila implica que en más de un ciclo se usa la misma etapa.
  - Más de una marca por columna, implica que en un mismo ciclo se utilizan en paralelo más de una etapa.

Algunos ejemplos de tablas de reservación:

- Tabla de reservación del pipeline lineal. La tarea ingresa por la etapa 1 y el resultado se obtiene en la etapa 4. En cada ciclo se usa una única etapa y cada etapa se usa una única vez.



- Tabla de reservación del pipeline no lineal. Tenemos 3 etapas donde la etapa 1 es la inicial y la final. La etapa 2 se usa 2 veces, la etapa 3 se usa 3 veces y la etapa 1 se usa 3 veces. En este ejemplo no hay ciclos en los que se usen más de 1 etapas a la vez.



Para poder encontrar el secuenciamiento, primero debemos ver algunas definiciones:

- Iniciación: Corresponde al comienzo de una evaluación.
- Latencia: Cantidad de ciclos entre dos iniciaciones.
- Colisión: Hay un conflicto de recursos porque dos iniciaciones intentan usar la misma etapa.
- Latencias prohibidas: Latencias que generan colisión. Si la tabla de reservación tiene n columnas, la máxima latencia prohibida será $m \leq n - 1$.

Latencia 2

	1	2	3	4	5	6	7	8	9	10
S_1	A		B			A		AB		B
S_2		A		AB		B				
S_3			A		AB		AB		B	

Latencia 3

	1	2	3	4	5	6	7	8	9	10	11
S_1	A			B		A		A	B		B
S_2		A		A	B		B				
S_3			A		A	B	A	B		B	

Las latencias prohibidas se detectan controlando las marcas en una misma fila de la tabla de reservación.

Las latencias prohibidas son:

- 2
- 4
- 5
- 7

	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X		X				
S_3			X		X		X	

Ciclo de latencias: Secuencia de latencias (no prohibidas) que se repite indefinidamente. En el ejemplo anterior, 3 es un ejemplo de ciclo de latencia.

- La latencia promedio de un ciclo de latencias es la suma de las latencias del ciclo dividido la cantidad de latencias.
- Un ciclo constante es un ciclo de latencias con un solo valor.

Al planificar los eventos en un pipeline, el objetivo es obtener la menor latencia promedio entre iniciaciones sin ocasionar colisiones. Para ello veremos una estrategia:

- **Vector de colisión**: Si m es la máxima latencia prohibida. El vector binario C = (C_m C_{m-1} … C_2 C_1) tal que:
  - C_i = 1 si i es una latencia prohibida
  - C_i = 0 si i es una latencia permitida

  El bit más a la izquierda se va a corresponder con la máxima latencia prohibida, por lo que siempre vale 1. Y el bit más a la derecha se va a corresponder con la latencia 1 que puede ser prohibida o no.
- **Diagrama de estado**: A partir del vector de colisión se construye un diagrama de estados que especifica las latencias permitidas luego de cada iniciación. El vector de colisión C representa el estado inicial del pipeline. Sea p una latencia permitida (p<m) en el tiempo t y C_P igual a C desplazado p lugares a derecha (ingresando 0 por la izquierda). El próximo estado del pipeline en el tiempo t+p será C or C_P.
- **Ciclos**: A partir del diagrama se pueden detectar los ciclos de latencia. Hay infinitos ciclos de latencia.
  - **Ciclos simples**: Ciclos en los que no se repiten estados.
  - **Ciclo greedy**: Ciclos simples en los cuales la transición entre estados es a través de la mínima latencia.

Planificación de pipeline sin colisiones

1. Identificar el vector de colisiones
2. Generar el diagrama de estados
3. Detectar ciclos greedy
4. El ciclo greedy de mínima latencia promedio determina la secuencia de iniciaciones del pipeline.

Mínima latencia promedio. Se puede demostrar que la mínima latencia promedio está acotada por:

- Cota inferior: máximo número de marcas en cualquier fila de la tabla de reservación.
- Cota superior: mínima latencia promedio de los ciclos greedy, es uno más de la cantidad de 1s en vector de colisión inicial.

La mínima latencia promedio obtenida se puede optimizar aún más agregando etapas de delay al pipeline original.

Pipeline de instrucciones

Para ejecutar una instrucción

- Todas las instrucciones
    1. Leen la instrucción de memoria
    2. Decodifican la instrucción
    3. Leen los operandos del banco de registro
    4. Realizan una operación en la ALU
- Algunas instrucciones
    5. Leen o escriben datos en memoria
    6. Escriben en el banco de registros

Acciones con más retardo

Algunos saltos incondicionales pueden no necesitar realizar las acciones 3 y 4.

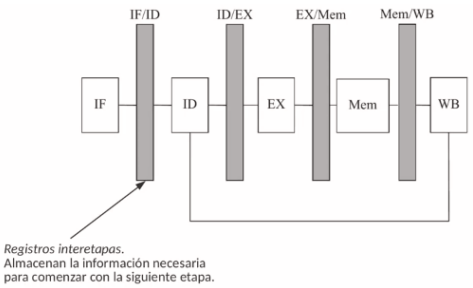
La tarea de ejecutar una instrucción no se va a poder dividir en subtareas de la misma duración. Pero es posible dividirla en 5 subtareas y que el pipeline siga siendo útil. La idea es mantener en etapas las subtareas más costosas. Subtareas:

1. Lectura en memoria
2. Decodificación y lectura de operandos
3. ALU
4. Lectura/escritura en memoria
5. Escritura de registros

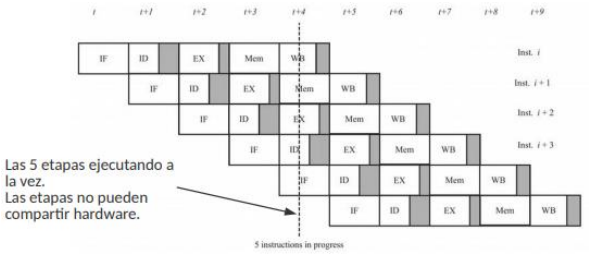
Estas etapas del pipeline van a llamarse:

1. Instruction fetch (**IF**). Se trae de memoria la instrucción en la posición indicada por el PC. Se asume que la instrucción no es un branch o un jump y se incrementa el PC.
2. Instruction decode (**ID**). Se decodifica la instrucción y se reconoce su tipo. Se buscan los operandos.
3. Execution (**EX**). Se ejecuta la instrucción. Difiere si es una operación aritmética, un acceso a memoria o un branch, aunque todas van a utilizar de alguna forma la ALU.
4. Memory access (**Mem**). Se accede efectivamente a la memoria en caso de que la instrucción sea un acceso a memoria.
5. Writeback (WB). Si la instrucción no es un branch o un store, el resultado de la operación (o del load) se almacena en el registro resultado.

Entre etapas vamos a tener un registro que almacene toda la información que haga falta de una etapa a la siguiente para que la ejecución de la instrucción pueda continuar. Hay que tener en cuenta que cada instrucción se va a estar ejecutando en una única etapa, es decir que todas las etapas están ejecutando instrucciones distintas.

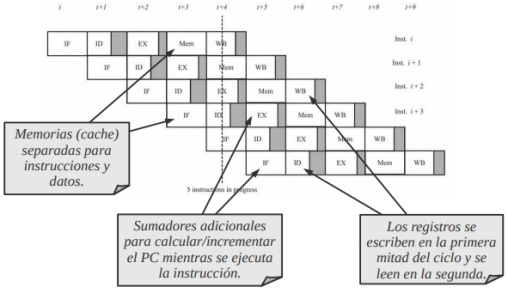


Si se divide la microarquitectura de un único ciclo en 5 etapas. Idealmente, la lógica de cada etapa es 1/5 de la lógica total. La frecuencia del reloj puede ser 5 veces mayor. La latencia de cada instrucción es la aprox. igual, pero el throughput es 5 veces mayor.



Poder ejecutar en pipeline implica poder superponer en el tiempo la ejecución de distintas tareas de distintas instrucciones. Las etapas no demoran exactamente lo mismo, aumenta la latencia de cada instrucción. Con el pipeline se logró aumentar la cantidad de instrucciones ejecutadas por unidad de tiempo aunque también se aumentó la latencia individual de cada instrucción.

Para que las etapas puedan ejecutarse en simultáneo, no pueden compartir hardware. Para que esto no ocurra, se aplican distintas estrategias en las distintas etapas que necesitan el mismo hardware en el mismo momento.



En condiciones ideales, el pipeline termina una instrucción por ciclo. En la realidad, las condiciones no siempre son ideales, porque van a haber conflictos (hazards) entre las instrucciones. Los conflictos se detectan en la etapa de decode. Tipos de conflictos:

- Estructurales: La instrucción necesita de un recurso que está ocupado. Soluciones generales para evitar estos conflictos:
  - Duplicar el recurso
  - Implementar la unidad en pipeline
  - Colas de espera a la unidad funcional

  Dependiendo el tipo de recurso, se pueden realizar otras estrategias.
  La implementación de las operaciones aritméticas esta optimizada para que dure lo menos posible, pero no todas duran lo mismo.
- De datos: La ejecución de una instrucción depende de la ejecución de una instrucción previa y ambas instrucciones están concurrentemente ejecutándose en el pipeline. Conflictos de datos:
  - *Read after Write (RAW)*: La instrucción j lee su dato fuente antes de que una instrucción previa i lo escriba.
  - *Write after Read (WAR) o antidependencia*: La instrucción j escribe el dato antes de que una instrucción previa i lo lea.
  - *Write after Write (WAW)*: Dos instrucciones tienen el mismo registro de salida

  El resultado de la ejecución concurrente debe coincidir con la ejecución secuencial

Gestión de los conflictos de datos	
En tiempo de compilación	En tiempo de ejecución
• Reordenar el código • Insertar instrucción útil • En caso de no encontrar instrucción útil, insertar NOPs en el código	• Frenar el pipeline (stall) • Adelantar los datos (forwarding)

  El forwarding se realiza en el mismo ciclo, por ejemplo si el valor se necesita en el ciclo 3 de una instrucción pero en otra anterior ya se tiene en ese mismo ciclo, ahí se aplica forwarding. A veces el forwarding no es suficiente y se puede realizar combinación de forwarding y stall.
- De control: Dado que el PC se incrementa en cada ciclo, el pipeline funciona bien cuando no hay cambios en el flujo secuencial de las instrucciones. Instrucciones *branch*, *jump*, *call* y *return* interrumpen el flujo secuencial del programa y crean conflictos de control.
  - El PC se incrementa en la etapa IF.
  - Tanto la dirección de salto como la condición (branch) se calculan en EX.
  - Por lo tanto, recién se decide y se actualiza el PC en MEM.

  En el hardware básico no se tiene esquematizado que ocurre con los jumps y con otros saltos incondicionales.
  El jump (call y return) es un salto incondicional. En el pipeline básico, el PC destino no se actualiza hasta la 4ta etapa del pipeline. Tiene los mismos problemas que el branch.
  Penalización del branch / jump es la cantidad de ciclos que se pierden hasta obtener la instrucción correcta que debe seguir al branch / jump.

¿Qué hacer con las instrucciones en el pipeline?	
A nivel compilador	A nivel hardware
• Branch retardado	• Frenar el pipeline • Anular las instrucciones

A nivel sw, en branch retardado el hw ejecuta toda instrucción de la que hace fetch. Los fetch entre el fetch del branch y el fetch del target (i.e. slots de delay) se ejecutan independientemente del resultado del branch. El compilador es responsable de que la ejecución sea siempre correcta independientemente del camino del branch.
La complicación para el compilador esta en ¿Qué instrucción elegir para completar esos slots de delay?

1) Previa al branch
2) Del destino
3) De las consecutivas (no tomado)
4) No-op (se pierden los ciclos)

La penalización promedio de los branch dependerá del código y de la habilidad del compilador para encontrar instrucciones útiles.
A nivel hw, en el stall del pipeline se frena el pipeline hasta que se conoce la dirección destino. La instrucción destino puede ser el target o la inmediata siguiente. Muy simple a nivel hardware. La penalización del branch es fija y no puede mejorarse por software.
A nivel hw, anular instrucciones tiene mejor rendimiento que frenar el pipeline y es apenas más complejo. Si el branch salta, se anula la ejecución de las instrucciones siguientes al branch.

- Flush del pipeline
- Reemplazar las instrucciones a anular por No-ops

Asegurarse de no cambiar el estado del procesador hasta saber el resultado del branch.

- En el pipeline MIPS de 5 etapas es trivial
- En un pipeline más complejo, la complejidad está en saber cuándo una instrucción cambia el estado del procesador y en cómo revertir ese cambio.

¿Cómo reducir la penalización del branch?	
A nivel compilador	A nivel hardware
• Predicción estática	• Determinación temprana • Asumir no tomado • Asumir tomado • Predicción estática o dinámica

La determinación temprana del branch reduce la penalización determinando el branch en la etapa más temprana que sea posible.

- MEM → 3 ciclos de penalización
- EXE → 2 ciclos de penalización
- ¿ID? → Si la evaluación de la condición es lo suficientemente rápida se podría mover a la etapa ID y determinar el PC junto al decode.
    - Branch on equal: un comparador de igualdad es mucho más rápido que restar y detectar resultado cero
- IF???→ como todavía no se sabe que instrucción es, entonces es imposible analizarlo.

Con esta mejora, agregamos otro problema. Adelantar la resolución de los branch al decode implica que en esta etapa vamos a necesitar valores de registros, por lo que se nos agrega un nuevo posible conflicto RAW.
Al asumir no tomado, un pipeline con sólo la capacidad de anular instrucciones, se comporta como si asumiera que el branch no va a saltar.

- El hw continúa con las instrucciones siguientes como si el branch no afectara el PC.
- Si el branch saltaba, se aborta la ejecución de las instrucciones siguientes al branch.

A nivel hw, asumir tomado. Si el resultado de la evaluación del branch se sabe después que la dirección de salto

- Condiciones más lentas de evaluar
- Dirección de salto en ID y condición de salto en EX o MEM

El hw podría asumir que el salto se toma y mientras se evalúa la condición se hace el fetch del target. En el caso de asumir tomado o no tomado, reducen la penalización promedio de los branch. La penalización promedio va a depender de la relación entre la cantidad de branch tomados y no tomados. El compilador puede optimizar el código para que el camino más frecuente coincida con la decisión del hardware.

Predicción de branch	
Predicción estática	Predicción dinámica
• Solamente hay disponible información estática (el código fuente) • Predice siempre lo mismo para el mismo branch.	• Está disponible la historia del branch.

Predicción estática por hw:

1) Predecir siempre no tomado
2) Predecir siempre tomado
3) Predecir tomado para los saltos hacia atrás y no tomado para los saltos hacia adelante
    - Al final de los bucles (for, while) hay un salto para volver al principio y eso se repite varias veces.
    - Los saltos hacia atrás son generalmente tomados.

Son la forma más básica de predicción: el hw predice siempre lo mismo.

A nivel de compilador puede hacerse si el set de instrucciones lo permite.

En la predicción dinámica, usa la historia de ejecución del programa para predecir si el branch será tomado no. Obtiene mejores resultados que la predicción estática, sobre todo en los saltos hacia adelante, pero a costa de mayor complejidad en el hw. La predicción se hace en la etapa IF

- Determinar qué instrucción ejecutar en el próximo ciclo.
- Si es correcta la penalización es cero.

La predicción dinámica implica tener en hardware un Branch Target Buffer que es una tabla con las últimas cien (o mil) instrucciones branch ejecutadas. Incluye:

- PC del branch
- PC del target

En la etapa de fetch, se busca el PC actual en el Branch Target Buffer. Si está ahí, entonces ese PC se corresponde a un branch. Si no está ahí, no se sabe si ese PC se corresponde a un branch o a algún otro tipo de instrucción.

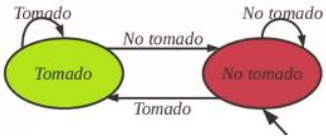
En la versión más simple, que el PC este en el Branch Target Buffer además implica que se asume tomado y el próximo fetch es la dirección de salto que indica el buffer. Si después resultaba que no era tomado, se borrara del buffer.

En versiones más complejas puede además incluir

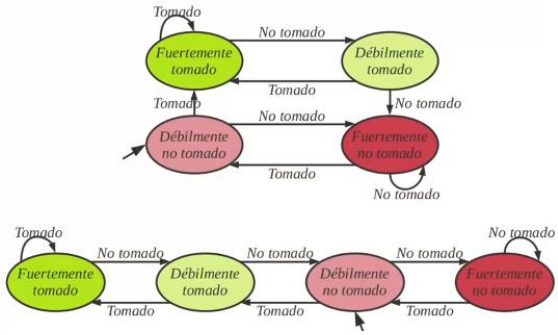
- historia pasada
- instrucciones
- información sobre saltos incondicionales

La exactitud de la predicción va a depender de los bits que se usen para mantener la historia del branch.

- *Predicción dinámica de 1 bit*: Predice lo que ocurrió la última vez.



- *Predicción dinámica de 2 bits*: para realizar la predicción distingue entre 4 estados.



En la implementación simple del pipeline la ejecución es en orden y todas las etapas demoran lo mismo. Por lo tanto evita:

- Conflictos estructurales
- Conflictos de datos
  - WAR
  - WAW

Solamente tenemos conflictos:

- RAW → Dependencia de procedimiento
- De control

¿Cómo afectan los distintos conflictos en el rendimiento del pipeline?

T_1 = Tiempo de procesamiento sin pipeline

T_k = Tiempo de procesamiento en pipeline con k etapas

$$Speedup_{Pipeline} = \frac{T_1}{T_k}$$

Av_1 = Tiempo promedio de procesamiento sin pipeline

Av_k = Tiempo promedio de procesamiento en k etapas

$$Speedup_{Pipeline} = \frac{Av_1}{Av_k}$$

En un pipeline de profundidad k, si se ejecutan n instrucciones, en promedio:

$$\text{1 instrucción demanda: } \frac{k+n-1}{n} \text{ ciclos}$$

Idealmente

$$CPI_{ideal} = \lim_{n \to \infty} \frac{k+n-1}{n} = 1$$

Los stalls e instrucciones anuladas causan que baje el rendimiento del pipeline.

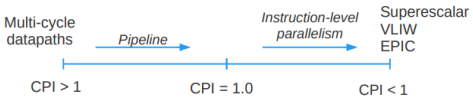
$$\begin{aligned} CPI &= CPI_{ideal} + \text{Ciclos Stall por instrucción} \\ &= 1 + \text{Ciclos Stall por instrucción} \end{aligned}$$

$$Speedup_{Pipeline} = \frac{Av_1}{Av_k} = \frac{k}{CPI_k}$$

$$Speedup_{Pipeline} = \frac{k}{CPI_k} = \frac{k}{1 + \text{Ciclos stall por instrucción}}$$

Si no hay Stalls, el Speedup del pipeline es igual a la profundidad k del pipeline.

Multiple-Issue



Las arquitecturas Multiple-Issue van a aprovechar aún más el Instruction-level parallelism. Si nos olvidamos de la implementación único ciclo porque tiene un periodo de reloj demasiado largo y nos quedamos con aquellas que tienen periodo de reloj comparables, tenemos en un extremo un Multi-cycle datapaths en donde en promedio cada instrucción demora varios ciclos en terminar. Luego pasamos a una implementación en pipeline, en donde si bien cada instrucción demora varios ciclos en ejecutarse en promedio cada instrucción demora un único ciclo. Ahora el objetivo va a ser, en promedio, que cada instrucción demore menos de un ciclo en terminar. Vamos a tener 3 posibles microarquitecturas: superescalares, VLIW (Very Long Instruction Word) y EPIC.

El tiempo de ejecución de un programa es $EX_{CPU} = \#\,Instrucciones \times CPI \times período\,del\,reloj$

Para reducir EX_{CPU} podemos reducir cualquiera de esas 3 variables, cambiando:

- *Cantidad de instrucciones* que depende de la optimización del compilador y, para un mismo nivel de optimización, depende del set de instrucciones disponible. Mejorar eso implica cambiar de arquitectura.
- *Reducir el período del reloj* implica incrementar la frecuencia. Para esto, cada etapa del pipeline debe realizar menos trabajo. Las etapas se dividen en nuevas etapas y el pipeline se vuelve más profundo.
- *Reducir el CPI* implica incrementar el IPC. Modificar la estructura del pipeline para permitir que más de una instrucción ejecute a la vez en la misma etapa. Para ensanchar el pipeline se incrementa la cantidad de unidades funcionales, llamado Grado m (m-way/m-wide)

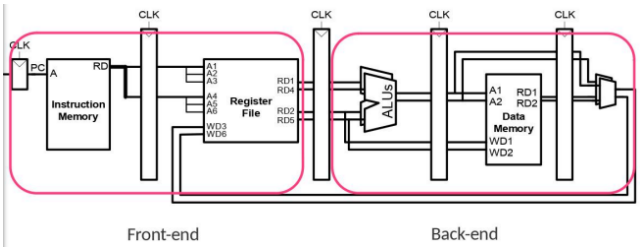
Categorías de procesadores Multiple-issue

	Agrupamiento de instrucciones para ejecución paralela	*Asignación de las instrucciones a las UF en el hardware*	*Determinación del inicio de la ejecución de las instrucciones*	*Ejemplos*
Superescalar	HW	HW	HW	Mayoría de los procesadores
EPIC	Compilador	HW	HW	Itanium
Dynamic VLIW	Compilador	Compilador	HW	–
VLIW	Compilador	Compilador	Compilador	DSP

Superescalares

Las etapas del pipeline se dividen en dos grupos:

- **front-end** que incluye las tareas de las etapas de fetch y decode
- **back-end** que incluye las tareas de las etapas execute, memory y writeback



En función de cuándo comienzan el procesamiento back-end, las microarquitecturas superescales se dividen en dos grandes grupos:

- Superescalar en orden (o estáticos)
- Superescalares fuera de orden (o dinámicos)

	Procesamiento front-end	Comienzo del procesamiento back-end	Finalización del procesamiento back-end (commit)
En orden	Orden definido por el compilador	En el orden definido por el compilador y cuando se resuelvan todas las dependencias de datos (o puedan resolverse con forwarding)	Debe respetar la semántica del programa
Fuera de orden		Ni bien estén disponibles los datos necesarios para operar	

El procesamiento back-end es siempre en el orden especificado por compilador, i.e. en el orden en que se hace fetch de las instrucciones. Varían en cuándo comienzan las instrucciones con el procesamiento back-end.

- En el caso de los superescalares en orden, las instrucciones no solo no comienzan a ejecutar hasta tener resueltas todas las dependencias de datos sino que lo hacen en el orden definido por el compilador.
- En el caso de los superescalares fuera de orden, las instrucciones comienzan a ejecutar ni bien tienen los operandos disponibles. Eso implica que una instrucción puede comenzar a ejecutar antes que sus predecesoras y por lo tanto, terminar antes. En este caso, la etapa de writeback (ahora llamada commit) debe asegurarse de que la escritura de los resultados respete el orden establecido por el programa.

Microarquitecturas superescalares en orden

Las instrucciones dejan el front-end en el orden definido en el programa. Todas las dependencias de datos se resuelven antes de que la instrucción comience el procesamiento back-end (o podrán resolverse más adelante por forwarding).

Implementaciones

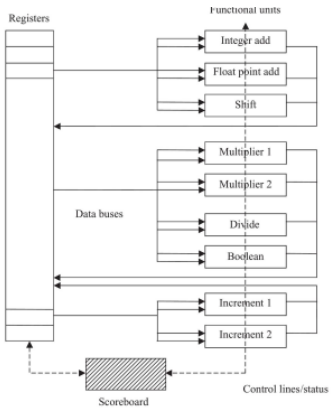
- Extensión directa del pipeline escalar
- Scoreboard: Monitorea la ejecución de las instrucciones (reemplaza la unidad forwarding y stalls)

En el pipeline escalar, las instrucciones pasan a la etapa EX cuando tienen los operandos (porque los leyeron del banco de registros o porque los obtendrán más adelante por forwarding). Si no, se frena la instrucción y el ingreso al pipeline de instrucciones posteriores. Los conflictos WaW no frenan el pipeline.

En la Extensión a superescalar, si la instrucción no puede obtener los operandos, no solo se frena la instrucción y le ingreso de instrucciones nuevas al pipeline, sino también las posteriores que estén en el pipeline.

Scoreboard fue introducida por la CDC6600 (1964) que incorporó (entre otras cosas) unidades funcionales (UF) separadas. Las UF no estaban en pipeline, operaban concurrentemente coordinadas por un Scoreboard (unidad de control).

La CDC 6600 tenía 10 unidades funcionales



- Un shifter, sumador de enteros largos y sumador de punto flotante
- Dos multiplicadores de punto flotante
- Divisor en punto flotante y una unidad lógica
- Dos incrementadores para calcular direcciones
- Una unidad de saltos

La implementación básica de scoreboard divide la etapa de Decode en 2:

- **Issue**: Decodifica la instrucción y reserva la unidad funcional (UF) necesaria. Es parte del font-end.
- **Dispatch**: Lee operandos y envía a operar a la UF. Es parte del back-end.

Una vez decodificada la instrucción, el scoreboard controla los siguientes pasos:

1. **Issue**: Si hay una UF libre y no hay conflictos WAW, se reserva la UF y la instrucción pasa a la siguiente etapa. Si no se cumple alguna condición, se frena la instrucción y las siguientes. Las instrucciones abandonan la etapa Issue en orden.
2. **Dispatch** (o Read Operands): El scoreboard controla que los operandos estén disponibles (ninguna instrucción previa va a escribirlo). Una vez que están disponibles (evita conflictos RAW) le indica a la UF que lea los operandos y comience la ejecución.
3. **Execution**: La UF lee los operandos y ejecuta (uno o más ciclos, dependiendo de la latencia de la UF). Cuando la UF termina, le avisa al scoreboard que está en condiciones de escribir el resultado.
4. **Write result**: Antes de escribir el scoreboard controla por conflictos WAR. Si existe uno, se frena la unidad hasta que se resuelvan. Las instrucciones en progreso ya leyeron sus operandos y por lo tanto no ocasionan conflictos WAR.

El scoreboard necesita tener información de:

- Para cada UF, si está libre u ocupada
- Para cada instrucción
  - Nombre del registro resultado y de los fuentes
  - Nombre de las unidades que produzcan valores para los registros fuentes (puede no haber ninguna).
  - Flags que indiquen si los registros fuentes están disponibles.
  - El estado de la instrucción: en cuál de las 4 etapas está.
- Por cada registros resultado, el nombre de la UF que produce su resultado (es redundante, pero facilita el hw)

Etapa	Condición de *Stall*	Acciones
Issue	Si alguna instrucción previa está esperando en esta etapa o la UF está ocupada o hay conflicto WAW, entonces *Stall* de la instrucción y de las siguientes.	Marcar la UF como ocupada. Actualizar los datos del Scoreboard.
Dispatch	Si alguno de los registros fuente no está disponibles, entonces *Stall* de la instrucción.	Enviar los datos a la UF.
Execute		Al final de la ejecución, pedir permiso para escribir el resultado.
Write result	Si hay una instrucción previa que todavía no leyó los operandos y uno de esos operandos es el destino, entonces *Stall* de la instrucción.	Marcar la UF como libre. Marcar el registro destino como disponible.

Posibles optimizaciones:

- Forwarding
- Buffering en la etapa de Dispatch para que haya varias instrucciones esperando
- Evitar los stalls que se producen cuando dos unidades que comparten el bus de datos quieren escribir el resultado en el mismo ciclo.

Fuera de orden

Superescalar fuera de orden

Las instrucciones comienzan a ejecutar ni bien tienen los operandos disponibles. Las instrucciones pueden terminar el procesamiento front-end y comenzar el back-end antes que instrucciones previas del orden establecido por el programa. Los resultados deben almacenarse en el orden especificado por el programa. La etapa ~~**WB**~~ commit debe respetar la semántica del programa.

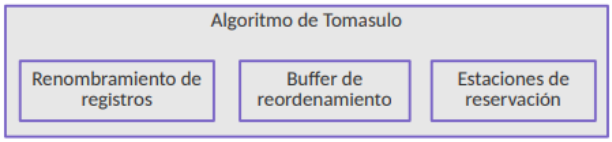
Las tareas del front-end

- Fetch de múltiples instrucciones
- Predicción de branchs
- Decodificación de las instrucciones

Es independiente del procesamiento back-end.

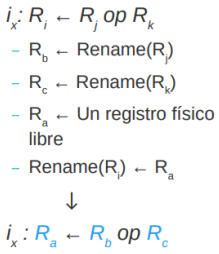
Objetivos de ejecución fuera de orden

- Que las dependencias de nombre (WaW, WaR) no frenen el pipeline. Existen porque los registro son escasos.
- Que los conflictos procedimentales (RaW) no frenen la ejecución de instrucciones siguientes independientes.
- Que los conflictos estructurales no frenen la ejecución de instrucciones siguientes independientes. También ocurren porque los recursos son escasos



Renombramiento de registros

Por un lado tenemos los registros lógicos o de arquitectura que son los registros definidos por el ISA y por otro lado tenemos los registros físicos que son el total de registros reales de almacenamiento que pueden o no corresponder o tener una correspondencia uno a uno con los registros lógicos. Los registros lógicos, en una instrucción, se mapean a un conjunto de registros físicos en la última etapa del front-end.



Etapa de renombramiento

i. Los registros fuentes se reemplazan por los registros físicos a los que ya hayan sido mapeados.
ii. El registro destino se mapea a un registro físico sin usar.

El banco de registros físicos necesita mantener

- Por cada registro:
  - Un flag (*ready bit*) que indica si tiene un valor válido o no
    - Al renombrar un registro se asigna el valor falso ya que un registro físico nunca tiene un valor valido.
    - Cuando se le asigna el resultado de la instrucción, toma el valor verdadero.
  - El valor
- Un mapeo de registros lógicos a físicos.
- Una forma de obtener registros físicos libres.

Resultado de ejecución fuera de orden ≡ Resultado de ejecución secuencial

El buffer de reordenamiento (ROB – reorder buffer)

Reordena las escrituras en los registros lógicos. Va a estar implementado con una cola FIFO circular donde cada elemento almacena:

- Un flag indicando si la instrucción completó la ejecución
- El valor calculado por la instrucción
- El nombre del registro lógico donde debe almacenarse el resultado
- El tipo de la instrucción: aritmética, load, store, branch,…

En el Front end del pipeline, durante el renombramiento de una instrucción, se inserta una entrada al ROB

(false, N/A, R_i, op)

Luego en el Back-end, cuando se completa la instrucción, se modifica la entrada en el ROB con el resultado y se pone en true el flag indicando que terminó. Durante la etapa de *commit*, si la primera instrucción del ROB terminó,

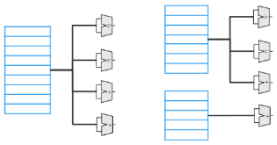
se almacena el resultado en el registro destino. No importa lo que ocurre con las demás instrucciones, lo que importa es que ocurre con la primera.

Estaciones de reservación

Con la ejecución fuera de orden, lo que queríamos lograr era que los conflicto tanto de datos como estructurales no frenaran el pipeline. Si los conflictos RaW y los estructurales no frenan el pipeline

- ¿Dónde están esas instrucciones mientras esperan los operandos?
- ¿Dónde están esas instrucciones mientras esperan la UF?
- ¿Cómo se sabe que una instrucción puede ejecutar?

De eso es a que se van a encargar las estaciones de reservación. Estas pueden ser:



- Centralizadas (*Ventana de instrucción*)
- Descentralizadas: asociadas a una UF (o a un grupo)

La estación de reservación debe contener como mínimo:

- La operación a realizar
- El valor de los operandos o el nombre de los registros físicos (indicado por un flag)
- El nombre del registro físico del resultado

Cuando se envía la instrucción a la estación de reservación, si el *ready bit* del registro físico fuente es

- Verdadero: se le pasa el valor a la estación de reservación.
- Falso: se le pasa el nombre del registro a la estación de reservación.

Cuando todos los registros fuente tienen un valor, la instrucción está lista para pasar a la ejecución en la UF. Si en un dado ciclo más de una instrucción está lista para operar en la misma UF, se debe tomar alguna decisión de planificación:

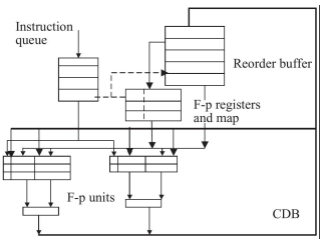
- La instrucción más vieja según el orden del programa
- Operaciones identificadas como críticas

Cuando una instrucción termina hace un broadcast a todas las estaciones de reservación del nombre del registro físico y el valor. Y se escribe el valor en el registro físico destino y se pone el ready bit en verdadero.

Algoritmo de Tomasulo

En el algoritmo de Tomasulo se integran el renombramiento de registros, el buffer de reordenamiento y las estaciones de reservación. El algoritmo de Tomasulo mejorado para soportar predicción de branch incluye:

- ROB para que las instrucciones terminen en orden. Cumple dos funciones:
  - Reordenamiento
  - Banco de registros físicos
- Un mapeo de registros lógicos a físicos



Las estaciones de reservación tienen 6 campos:

- Un bit indicando si la estación de reservación está libre o no
- Dos campos por operando: un flag y el dato. El flag indica si el dato es un valor o un nombre a un registro físico (tag).
- Un campo con un puntero a la entrada en el ROB (tag) donde se debe almacenar el resultado de la instrucción.

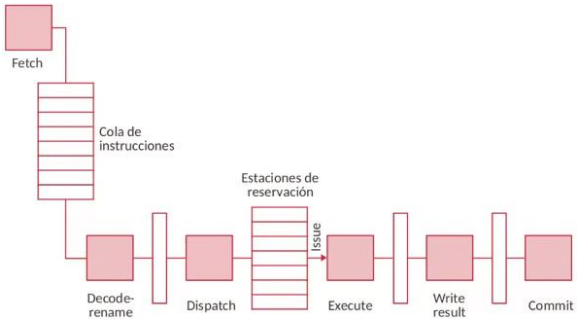
El mapeo entre registros lógicos y físicos es con una tabla indexada por registro lógico. Tiene dos campos:

- Un dato
- Un flag que indica si el dato es el valor del registro lógico o un puntero al banco de registros físicos.

El ROB tiene tres campos:

- Dos campos para el registro físico: flag y dato. El flag (ready-bit) indica si el dato es el valor resultado de la instrucción o es un tag.
- El nombre del registro lógico resultado

Este va a ser el pipeline que vamos a tener en cuenta para el algoritmo de Tomasulo. Donde vamos a tener 6 etapas, entre las etapas Fetch y Decode-rename vamos a tener una Cola de instrucciones y entre las etapas de Dispatch y Execute vamos a tener las Estaciones de reservación.



Luego del fetch las instrucciones se almacenan a una cola de instrucciones. Donde la etapa Decode-rename va a tomar una instrucción de la cola de instrucciones y la Decodifica. Si la estación de reservación necesaria está llena: Conflicto estructural. Si el ROB está lleno: Conflicto estructural. Los conflictos estructurales frenan (stall) el flujo de instrucciones entrantes hasta que se resuelvan. Una vez que se resuelvan los conflictos estructurales, se reserva la estación de reservación y se reserva la última entrada en el ROB.

En la etapa Dispatch se completa la estación de reservación y la entrada del ROB. Para cada operando:

- Si el mapeo indica un registro lógico, el registro contiene un valor válido.
- Si el mapeo indica una entrada en el ROB, se chequea si esa entrada contiene un valor u otra entrada del ROB.
- En cualquier caso, a la estación de reservación se le envía el contenido del registro lógico o de la entrada del ROB (indicado por el flag)

Se mapea el registro resultado a la nueva entrada del ROB y se ingresa en la estación de reservación. Se encola la instrucción en el ROB indicando que lo asociado es un tag.

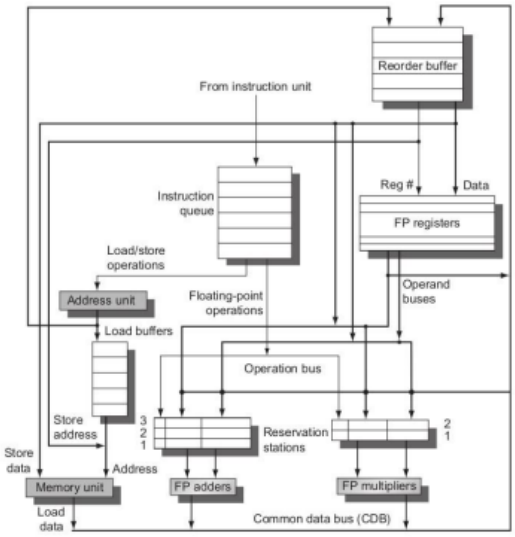
Issue no es una etapa, sino que es una transición de estaciones de reservación a comenzar a ejecutar. Cuando ambos operandos en la estación de reservación están disponibles y la UF no está frenada esperando que el CDB haga broadcast del último resultado, se puede enviar la instrucción a la UF para que comience la ejecución. Si varias estaciones de reservación están listas en el mismo ciclo, algún algoritmo de planificación decidirá cuál enviar primero.

La etapa Execute es la que ejecuta la operación. Al final del último ciclo de la ejecución, la UF pide el escribir en el CDB. Si durante el mismo ciclo hay varias UF pidiendo escribir en el CDB, se resuelve mediante algún esquema de prioridades (cableado). Si el CDB no está disponible, la operación debe esperar.

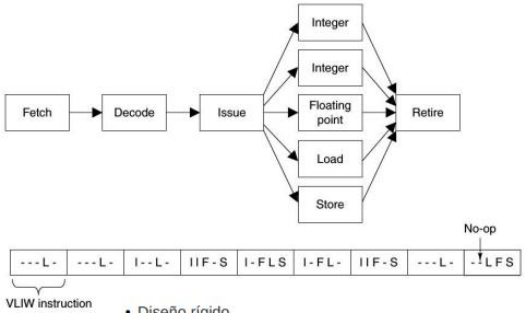
La etapa Write Result lo que va a hacer es una vez que el CDB está disponible, la UF hace broadcast del resultado y del tag asociado a la instrucción. El resultado se almacena en la entrada del ROB que indica el tag y se modifica el ready bit. El resultado se almacena en todas las estaciones de reservación que tengan ese tag como operando y se modifica el flag de disponible.

Finalmente la etapa Commit. En cada ciclo, se controla el ready bit de la primera entrada del ROB. Si está en verdadero, el valor se almacena en el registro lógico indicado en la entrada del ROB y se borra la entrada.

Extensión de hw para incluir accesos a memoria



Very Long Instruction Word (VLIW)



Estamos frente a una planificación estática. El ISA VLIW define patrones para instrucciones largas. Una instrucción larga especifica múltiples operaciones que pueden ejecutarse simultáneamente. Depende fuertemente del compilador.

El compilador debe unificar en una única instrucción operaciones que puedan ejecutarse en paralelo sin conflicto. Si no es posible, se completa con No-op.

Esto lo que va a hacer es que haya una fuerte dependencia del compilador porque este es el que tiene que generar esas instrucciones largas, determinar cuáles son las operaciones que hay que realizar, como pueden agruparse en instrucciones largas para pasárselas al hardware y que las ejecute. El hardware no realiza ningún checkeo de conflictos, directamente ejecuta las instrucciones como las agrupo el compilador

Una secuencia de instrucciones largas define el plan de ejecución para un programa en particular en una implementación en particular.

Problema de compatibilidad entre implementaciones: Un programa es compilado para una microarquitectura en particular (determinado conjunto de unidades funcionales con determinadas latencias)

VLIW depende muy fuertemente de las Técnicas de ILP en tiempo de compilación:

✓ Se puede tener en cuenta un mayor rango de instrucciones.

✓ Se realiza una única vez y por lo tanto puede demandar tanto tiempo como sea necesario.

✗ Solo se cuenta con la información disponible en tiempo de compilación: estrategias conservativas y que asumen el peor caso.

Superescalar	VLIW
Se comienzan a ejecutar varias instrucciones por ciclo.	Se ejecuta una única instrucción larga por vez. Esa instrucción consiste de varias operaciones distintas
El hw es complejo, tiene que controlar varias instrucciones por vez.	El hw es más simple porque debe procesar una única instrucción.
El compilador puede optimizar para mejorar el rendimiento, pero no necesita preocuparse por cómo se ejecutarán las instrucciones.	El compilador necesita controlar las dependencias entre las instrucciones para poder planificarlas
El tamaño del programa depende solo de la cantidad de instrucciones.	El tamaño del programa depende de la cantidad de instrucciones largas. Esta cantidad está dada por las operaciones que hay que realizar y las dependencias entre ellas.

Explicitly Parallel Instruction Computing (EPIC) se basan en el concepto de VLIW. El hw no controla dependencias. El compilador sólo agrupa operaciones independientes siguiendo un conjunto de patrones de agrupamiento. Esos

patrones pueden incluir stops. Un stop entre dos operaciones o al final, indica que una operación debe completarse antes de comenzar con las siguientes (limita el paralelismo pero elimina el control de dependencias). Es compatible para diferentes implementaciones.

En un Dynamic VLIW el compilador agrupa las instrucciones y asigna las unidades funcionales. El hardware determina cuándo comienza a ejecutar cada instrucción. Permitiría responder a eventos que el compilador no puede manejar en tiempo de compilación (fallos en cache).

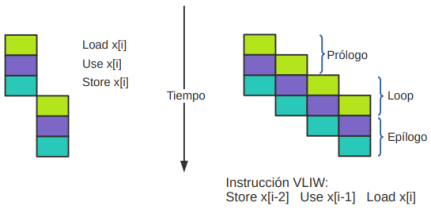
Hay Técnicas para ↑ ILP en compilación que son:

- Genéricas: Aplicables en cualquier hardware. Dependen únicamente del compilador
  - **Pipeline scheduling**: Estrategia más básica para mejorar el ILP. El compilador reordena las instrucciones para mejorar el rendimiento del pipeline

	1	2	3	4	5	6	7	8	9	10
R1 ← Mem[...]	IF	ID	EX	M	WB					
R2 ← R1 + R1		IF	ID	***ID***	EX	M	WB			
R4 ← Mem[...]			IF	***IF***	ID	EX	M	WB		
R3 ← R4 + R4					IF	ID	***ID***	EX	M	WB

	1	2	3	4	5	6	7	8	9	10
R1 ← Mem[...]	IF	ID	EX	M	WB					
R4 ← Mem[...]		IF	ID	EX	M	WB				
R2 ← R1 + R1			IF	ID	EX	M	WB			
R3 ← R4 + R4				IF	ID	EX	M	WB		

  - **Loop Unrolling**: Transformar los ciclos para aumentar el ILP. Esa trasformación la realiza desenrollando los ciclos. Si se renombran los registros, se pueden reordenar las instrucciones.
  - **Software pipelining**: Reordena el cuerpo de un bucle para incrementar el ILP cuando hay restricción de recursos o dependencias entre las iteraciones. Se identifican distintas porciones dentro de esa iteración, que pueden solaparse con otras porciones de otras iteraciones del mismo ciclo.



    El prólogo es la parte donde no completamos todas las etapas de ese pipeline. La parte de ciclo donde tenemos completas todas las etapas del pipeline. Y el epilogo donde nos quedan las partes finales para terminar con el bucle original.
- Soporte del ISA/Hw: Herramientas adicionales que el ISA le brinda al compilador para optimizar.
  - **Registros rotativos**: Renombramiento de registros con una instrucción particular.
    - $R_1 \dots R_{32}$
      - $R_i \rightarrow R_{i+1}$ y $R_{32} \rightarrow R_1$

    Permite que las instrucciones direccionen un registro diferente cada iteración
  - **Instrucciones predicativas**: Se puede reducir el número de branchs a ejecutar usando predicados. Las instrucciones predicativas van acompañadas de un predicado. Si el predicado es verdadero se ejecuta la instrucción. Si es falso, no se ejecuta
  - **Control especulativo**: Equivalente a la predicción en superescalares. Mover una porción de código arriba del branch. Estas instrucciones se vuelen especulativas. No pueden terminar hasta que se sepa la condición del branch.

Orden Original	*Reordenado con especulación*
`Branch label`	`Ld especulativo r1, 0(r9)`
`Ld r1, 0(r9)`	`Ld especulativo r2, 0(r8)`
`Ld r2, 0(r8)`	`add especulativo r3, r1, r2`
`add r3, r1, r2`	`Branch label`

    Estas instrucciones especulativas lo que hacen es que haya que agregar un bit especial para el manejo de las excepciones llamado Poison bit

- Si una instrucción especulativa genera una excepción (por ej. div por cero), la excepción solo debe saltar si efectivamente había que ejecutar esas instrucciones.
- A cada registro se le agrega un bit adicional que indica que el resultado de la instrucción especulativa generó una excepción.
- El poison bit se propaga a las demás instrucciones especulativas que utilicen ese resultado.

Memoria cache

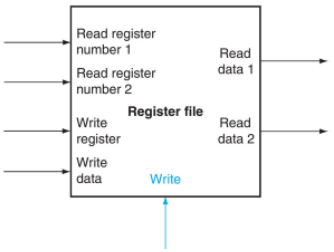
Diferentes tecnologías de implementación de memorias poseen diferentes:

- Costos
- Capacidad
- Velocidad

El objetivo es lograr que la memoria tenga los beneficios de las tecnologías:

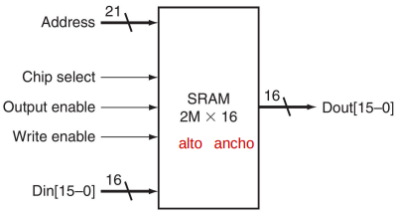
- Máxima velocidad
- Gran capacidad
- Bajo costo

Tipos principales de memoria



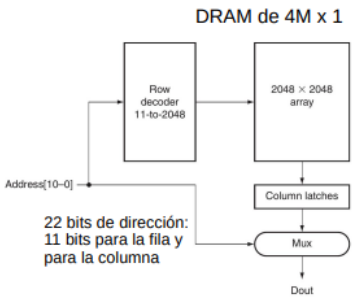
- Registros: conjunto de registros que pueden leerse y escribirse.
  - Es rápido
  - Costoso en la relación $/bit
  - Alto consumo energético

  Los registros se identifican con números.
- SRAM (static random access memory): Circuitos integrados con un arreglo de FF. En general, único puerto de lectura/escritura. Tiempo fijo para acceder a cualquier elemento.

  Es impracticable implementarlo con MUX. Una SRAM de 64K x 1 necesitaría un MUX de 64K a 1. Se tiene una línea de salida común y se usan buffers tristate.
  Una SRAM de 4M x 8 necesitaría un decoder de 22 x 4M y 4M líneas para habilitar los 8 FF de la palabra.
  Las memorias grandes se organizan en un arreglo y la decodificación se hace en 2 pasos.
- DRAM (dynamic random access memory): El valor se almacena en un capacitor. Se utiliza un transistor para acceder al valor cargado (para lectura o escritura). En comparación, las SRAM necesitan 6 transistores por bit. Eso hace a las DRAM más densas y baratas por bit.
  Los capacitores se descargan con el tiempo. Pueden mantener la carga por algunos mseg.
  Las DRAM necesitan refrescarse periódicamente (leer el contenido y volver a escribirlo).
  - Controlador de memoria: incluido en el chip del procesador o separado.

  Decodificación de dos niveles
  1. Row access: Elige una fila y activa la línea correspondiente. El contenido de la fila activa se almacena en un buffer.
  2. Column access: Selecciona el dato de la columna correcta.
  3. Se lee/escribe la fila.
  4. La fila se precarga y vuelve a almacenar en la memoria

  Para ahorrar pines se usan los de dirección tanto para filas como para columnas. Se distingue con las señales RAS (Row Access Strobe) y CAS (Column Access Strobe).
- Flash
- Discos magnéticos

SRAM	DRAM

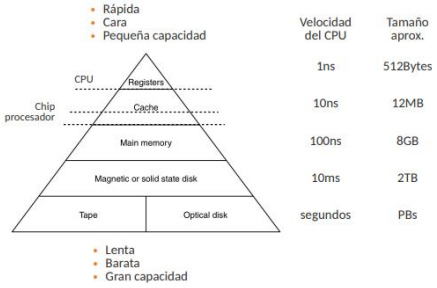
○ son más rápidas que las DRAM. ○ se usa para memoria cache (integrada y externa al chip)	○ Las celdas son más simples y pequeñas (Más densa y menos costosa) ○ Requiere un circuito de refresco. Para grandes memorias, el costo (fijo) adicional del circuito de refresco se compensa con el menor costo de las celdas ○ se usa para memoria principal
Ambas son volátiles	

Synchronous SRAM y Synchronous DRAM

- Mejora la velocidad.
- Elimina el tiempo de sincronización entre memoria y procesador.
- Permite transferir una ráfaga de datos a una serie de direcciones consecutivas.
- Inicialmente: SDR SDRAM (Single Data Rate SDRAM)

Double Data Rate RAM es una SDRAM que transfiere datos tanto en el flanco ascendente como en el descendente del reloj.

Jerarquía de memoria



En el principio de localidad la información no se accede de manera aleatoria

- Localidad temporal: Los datos y códigos usados en el pasado es probable que se usen en el futuro cercano (datos en la pila, bucles)
- Localidad espacial: En el futuro cercano es probable usar datos y código cercanos (en término de direcciones de memoria) a los datos y código actuales (recorrer arreglos, código secuencial).

Cada nivel de la jerarquía tiene menor capacidad que el nivel inferior. Cuando se genera una referencia a memoria puede ocurrir que:

- Se encuentre entre el contenido del nivel accedido → hit (éxito)
- No se encuentre en el contenido del nivel accedido → miss (fallo o faltante)

En el caso de un fallo en el nivel L_i, la referencia se busca en el nivel inmediatamente inferior L_{i+1}.

Tiempo de acceso promedio

- La tasa de aciertos (hit ratio) es la fracción (porcentaje) de accesos exitosos.
- La tasa de fallos (miss ratio) es la fracción (porcentaje) de accesos fallidos.

Miss ratio = 1 – hit ratio

Hit time	Miss penalty
Es el tiempo requerido para obtener un bloque en el nivel L_i, incluyendo el tiempo necesario para determinar si fue acceso exitoso o fallido.	es el tiempo que demanda obtener el bloque del nivel L_{i+1}, incluye el tiempo para acceder al bloque en L_{i+1}, transmitirlo e insertarlo en el nivel L_i

Ambos incluyen el tiempo de transmisión a quien lo requirió.

El tiempo promedio de acceso a un nivel L_i será:

$$T_{promedio\ Li} = \\ = T_{acceso\ Li} + miss\ ratio_{Li} * T_{acceso\ Li+1} \\ = hit\ time_{Li} + miss\ ratio_{Li} * miss\ penalty_{Li}$$

El tiempo de acceso es la disparidad en los tiempos de acceso entre el nivel L_i y el nivel L_{i+1} influye fuertemente en el tiempo de acceso efectivo al nivel L_i. A mayor disparidad entre niveles consecutivos, mayor es la tasa de aciertos que se requiere.

En la penalización de un faltante (miss penalty) influye

- tiempo de acceso
- tiempo de transferencia → depende del tamaño del bloque.

Mayor tamaño de bloque reduce los faltantes por localidad espacial. Mayor cantidad de bloques reduce los faltantes por localidad temporal. El tamaño del bloque tiene que ser balanceado.

Velocidad

- DDRz-xxx: nombre del chip de memoria
- PCz-yyyy: nombre del módulo DIMM
    - z: generación (DDR, DDR2, DDR3… PC, PC2, PC3...)
    - xxx: Indica la máxima cantidad de (millones de) transferencias por segundo.
        - La frecuencia del reloj real de la memoria es la mitad.
        - DDR400 realiza 400 MT/s con un reloj de 200 MHz
        - DDR2-800 realiza 800 MT/s con un reloj de 400 MHz
    - yyyy: Máxima tasa de transferencia que la memoria puede alcanzar (MB/s).
        - Las DDR transfieren 8 bytes por vez (ancho de banda de 64 bits).
        - DDR400 realiza 400 MT de 8 bytes en 1 segundo. En total transfiere 3200 MB/s → PC3200
        - DDR2-800 realiza 800 MT de 8 bytes en 1 segundo. En total transfiere 6400 MB/s → PC2-6400
        - DDR3-1333 realiza 1333MT de 8 bytes en 1 segundo. En total transfiere 10644 MB/s (aprox 10700) → PC3-10700

Latencias (4 números separados por guión)

- **CL** (CAS Latency): Número **exacto** de ciclos entre que se le envía a la memoria la dirección de la columna (previamente se le envío la fila correcta) y ésta lee el primer bit.
- **RCD** (RAS to CAS Delay): Número de ciclos de espera requeridos entre que la activación de fila y la activación de la columna (escritura en los latches).
- **RP** (RAS Precharge): Número de ciclos de espera antes de cambiar la fila activa.
- **RAS** (Active to Precharge Delay): Mínimo número de ciclos que la fila debe estar activa. Son los ciclos necesarios para refrescar la fila y se superpone con RCD.

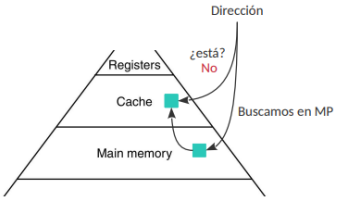
1. ¿Cuántos ciclos demanda leer el primer bit sin una fila activa?
2. ¿Cuántos ciclos demanda leer el primer bit con la fila incorrecta activa?

Dentro de la jerarquía de memoria, la memoria cache sirve como buffer de alta velocidad entre memoria principal (MP) y el CPU.

Diseño de la caché

A la hora de diseñar la cache tenemos 5 puntos principales que debemos responder:

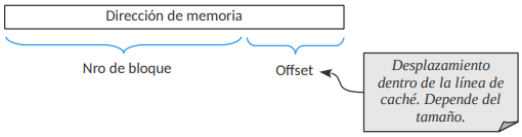
1. ¿Cuándo llevar el contenido de MP a caché?
    - **Básico**: lleva los datos de memoria principal a caché bajo de manda, es decir, cuando hay un faltante.



- **Avanzado**: prefetching

2. ¿Dónde se pone ese contenido?

La caché se divide en líneas o entradas organizadas en conjuntos (set). La organización de la caché determina el mapeo de posiciones de memoria a entradas en la caché. La dirección en memoria se va a dividir en 2 partes:



La organización de la caché es un mapeo entre Bloques de Memoria → Líneas de caché

La geometría completa de la caché se define por 3 cosas:

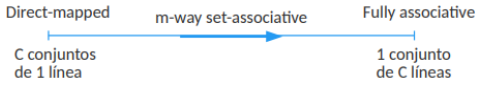
- El tamaño S de la caché destinado para datos (no es el tamaño real de la caché)
- Tamaño de la línea L
- Grado de asociatividad m

El grado de asociatividad define la organización de la caché. La cantidad de líneas C = S/L. Esas líneas se organizan en conjuntos de m líneas. En total hay C/m conjuntos.

$$1 \leq C/m \leq C$$

Tenemos 3 posibles organizaciones de la caché:

- **Direct-mapped**: Un bloque de memoria puede mapearse a una única línea de la caché. Si C (= 2^n) es la cantidad de líneas de la caché y B es el número de bloque correspondiente a la dirección de memoria, entonces al bloque B le corresponde la línea B mod C
- **m-way Set-Associative**: Un bloque de memoria puede mapearse a un conjunto de m líneas de la caché. Si C (= 2^n) es la cantidad de líneas de la caché, m es la cantidad de líneas de cada conjunto y B es el número de bloque correspondiente a la dirección de memoria, entonces al bloque B le corresponde cualquier línea dentro del conjunto B mod C/m
- **Fully Associative**: Un bloque de memoria puede mapearse a cualquier línea de la caché. Si C (= 2^n) es la cantidad de líneas de la caché y B es el número de bloque correspondiente a la dirección de memoria, entonces al bloque B le corresponde cualquiera de las C líneas de la caché.



Los faltantes en la caché se pueden clasificar, principalmente, en 3 grandes grupos:

- **Compulsivo**: Ocurre la primera vez que un bloque es accedido.
- **Conflictivo**: Ocurre cuando los bloques compiten por las misma líneas de caché, habiendo otras vacías. Son conflictos que no ocurrirían si la caché fuera fully associative con reemplazo LRU.
- **Capacitivo**: Ocurre cuando la caché no puede contener todos los bloques que necesita el programa para ejecutar. Hay que volver a llevar a caché un bloque que fue reemplazado.

3. ¿Cómo sabemos si la referencia está en caché?

Cada entrada de la caché contiene (al menos) un identificador (tag), un bloque de información o datos (varios bytes) y un bit de válido.

La ALU genera una dirección de memoria pero antes de acceder a memoria principal vamos a ver si el contenido de esa dirección de memoria no se encuentra en la caché, esta dirección se descompone en 2 campos:

- Número de bloque
- Offset.

A su vez, el número bloque se descompone en otros dos campos:

- Tag
- Index.

En total, la dirección de memoria se descompone en 3 campos.

Tag	Index	Offset
t bits	*i bits*	*d bits*

- Offset: indica el byte dentro de la línea al cual se está direccionando. Se divide en 2 (dependiendo longitud de la línea): offset a la palabra y offset al byte (el direccionamiento es al byte, pero los registros son de más bytes).
- Index: identificador del conjunto (set) al que pertenece la línea.
- Tag: identificador de la línea de caché.

Si la línea tiene L bytes, el offset debe tener d bits

$$d = Log_2 L$$

Si hay C líneas en total, organizadas en conjuntos de m líneas, hay C/m conjuntos (set). El index identifica los C/m conjuntos, por lo tanto necesita i bits

$$i = Log_2 C/m$$

Los bits restantes son los t bits del Tag

Cuando hablábamos del tamaño de la caché estamos hablando para datos, pero en realidad hay más cosas para almacenar en la caché. El tamaño real de la caché incluye:

- Bits de tag
- Bits adicionales (por ejemplo, bit de válido)
- Datos

Por lo tanto, a medida que incrementamos el grado de asociatividad vamos a necesitar más bits para el tag y en principio vamos a estar necesitando un tamaño real de caché mayor.

En general, solo se hace referencia a la capacidad para datos (S)

4. ¿Qué pasa si hay que llevar nuevo contenido a caché pero su lugar está ocupado o lleno?

En el caso de Direct-mapped caché, a cada dirección de memoria le corresponde una única línea de caché. Si hay que cargar un nuevo bloque y la línea que le corresponde está ocupada, se reemplaza con la línea nueva.

En cambio, en el caso de Set associative y fully associative cachés, a cada bloque de memoria le corresponde cualquier línea de caché dentro de un conjunto. Si en el conjunto no hay ninguna línea libre, hay que elegir un bloque para reemplazar.

Algoritmo de reemplazo: resuelve qué entrada reemplazar cuando hay un faltante en la caché y la nueva entrada colisiona con alguna existente. La elección puede ser:

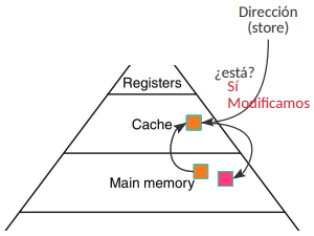
- Random
- Elegir inteligentemente por localidad temporal, la mejor opción sería elegir la línea que hace más tiempo que no se usa:
    - *LRU*: (el algoritmo optimo) Se lleva el control de cuál es la línea del conjunto que hace más tiempo que no se usa. Es simple de implementar en cachés con poco grado de asociatividad ($m \leq 4$). Para grados mayores de asociatividad es más costoso. Se utilizan esquemas alternativos.
    - *Pseudo LRU*: Llevar la cuenta de cuál es la línea menos usada en cachés con grados de asociatividad altos es costoso. El problema se simplifica dividiendo los conjuntos en 2 grupos. Un bit indica cuál de los dos grupos es el menos recientemente usado. Es decir, no contiene a la última línea accedida. Dentro de ese grupo se reemplaza uno al aza

5. ¿Qué ocurre en una escritura?

Una escritura (store) implica modificar la jerarquía de memoria. Las futuras referencias a esa dirección deben retornar el último valor.

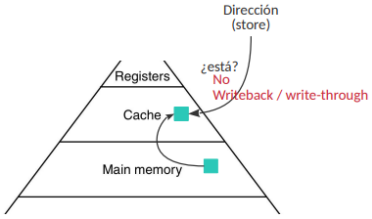
Política de escritura: indica cuándo se actualizan los niveles de la jerarquía en función de si la dirección de memoria está o no en la caché. Pueden ocurrir dos escenarios:

- La dirección de memoria a modificar está en la caché
  - *Writeback caché*: Se escribe sólo en la caché. La información en la caché ya no es la misma que la del siguiente nivel. A cada línea de caché se le asocia un dirty bit. 1 indica que la línea fue modificada, 0 que no. Al reemplazar una línea, se controla el dirty bit
    - Si es 0, la línea puede descartarse.
    - Si es 1, la línea se copia en el siguiente nivel de la jerarquía (memoria principal)
  - *Write-through caché*: Escribe en la caché y en el siguiente nivel de la jerarquía.

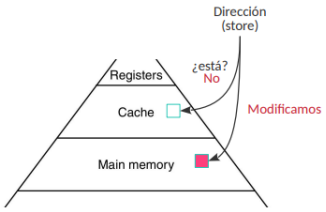


Writeback	Write-through
✓El procesador escribe las palabras individuales a la velocidad de la caché. ✓Múltiples escrituras en el mismo bloque requieren una única escritura en el nivel inferior. ✓El sistema puede transferir eficientemente bloques completos. ✗ Mayor tamaño real de la caché	✓Mantiene la consistencia entre ambos niveles. ✓No necesita dirty bit. ✓La resolución de los faltantes de caché es más simples y económica. ✓Es más simple de implementar. ✗ Hay mayor tráfico entre memoria y caché en comparación con writeback.

- La dirección de memoria a modificar no está en la caché
  - *Write-allocate*: Primero resuelve el faltante por la lectura y luego realiza la escritura.



  - *Write-around*: Se escribe solo en el siguiente nivel de la jerarquía, salteando la caché.



Posibles mejoras

- **Write buffer**: En una escritura el procesador debe esperar a que se escriba el dato en memoria. Esa espera puede ser del orden de los 100 ciclos. Las estrategias de escritura pueden incluir un write buffer entre caché y memoria principal, que almacene temporalmente el dato a escribir en memoria y la dirección destino.
  Una vez que el dato es escrito en el buffer, el procesador puede continuar ejecutando sin esperar la latencia completa de la escritura en memoria. Las escrituras se planifican para cuando el bus de memoria esté libre y, una vez completada la escritura en memoria, se libera la entrada del buffer.
  El write buffer es un nuevo punto de posibles conflictos estructurales. Almacena temporariamente la palabra a escribir y la dirección destino. Mejora importante el caso de write-through caché. Si el write buffer está lleno, el procesador debe frenarse (stall) hasta que se libere un lugar. Si el procesador escribe a una tasa mayor que la tasa a la que se liberan entradas en el write buffer, entonces potencialmente ocurrirá un conflicto estructural. Cuando hay un faltante en la caché hay que asegurarse que no haya escrituras en ese bloque pendientes en el write buffer.
  Cuando hay que reemplazar una línea modificada no se escribe directamente en memoria, se escribe la línea y la dirección en un write buffer. Cuando hay un faltante de caché, debe revisarse primero el write buffer. Se puede transformar en una pequeña caché fully associative. La información a escribir puede sobrescribir (o fusionarse con) la línea que exista en el buffer.
- **Caché multinivel**: Para mejorar el rendimiento de la caché
  - Disminuir la penalización de los faltantes
  - Disminuir la tasa de fallos.

  Velocidad de procesador >> velocidad de MP. Capacidad de procesador << Capacidad de MP. El objetivo es lograr una caché rápida y grande. Varios niveles
  1. Un primer nivel rápido con poca capacidad
  2. Un segundo nivel de mayor capacidad

  - Tasa de fallos local: Es el número de faltantes en la caché dividido la cantidad de accesos a esa caché.
    - Miss rate L_i = cantidad de fallos en L_i / cantidad de accesos a L_i
    - Miss rate L_1 = cantidad de fallos en L_1 / cantidad de accesos a L_1 = cantidad de fallos en L_1 / cantidad total de accesos
    - Miss rate L_2 = cantidad de fallos en L_2 / cantidad de accesos a L_2 = cantidad de fallos en L_2 / cantidad de fallos en L_1
  - Tasa de fallos global: El número de faltantes en la caché dividido el total de accesos a memoria generados por el procesador.
    - Miss rate global L_i = cantidad de fallos en L_i / cantidad total de accesos
    - Miss rate global L_1 = cantidad de fallos en L_1 / cantidad total de accesos
    - Miss rate global L_2 = cantidad de fallos en L_2 / cantidad total de accesos = Miss rate L_1 * Miss rate L_2

  La velocidad del primer nivel de caché afecta la velocidad del procesador. La velocidad del segundo nivel de caché solo afecta el miss penalty del primer nivel.
  Diseño del segundo nivel de caché

- *Tamaño*: Por lo general, el segundo nivel de caché contiene al primer nivel. El tamaño del segundo nivel debe ser mucho mayor que el primero (sino se incrementaría el miss rate local).
- *Grado de asociatividad:* Reemplazar directmapped por algún grado de asociatividad disminuye el miss rate. Se puede reducir la penalización de un faltante en primer nivel, disminuyendo el miss rate del segundo nivel.
- *Inclusión/Exclusión*: ¿Los datos de L_1 están en L_2?
  - Multilevel inclusión
    - Los datos de L_1 están siempre en L_2
    - Vista consistente de la jerarquía
    - L_2 debe ser mucho mayor que L_1
  - Multilevel exclusión
    - Los datos en L_1 nunca están en L_2
    - L_2 puede ser apenas mayor que L_1
    - Un faltante en L_1 implica hacer un swap y no un reemplazo

- **Prefetching**: La implementación básica de la caché lleva los datos a caché bajo demanda. Pre-fetching es una técnica predictiva que trata de llevar datos e instrucciones a la caché antes de que vayan a ser usados.
  - Disminuye el tiempo de acceso a los datos
  - Puede hacerse con la asistencia de instrucciones especiales o por hardware.
  - Es aplicable a cualquier nivel de la jerarquía

  La predicción del prefetching se puede evaluar en función de tres criterios:
  - *Precisión*: número de prefetches útiles / número de prefetches generados
  - *Cobertura*: número de prefetches útiles / número de misses sin prefetch
  - *Oportunidad*: Mide cuán oportuno o a tiempo fue el momento en que se hizo el prefetch. Si el prefetch es muy temprano puede desplazar datos que sean necesarios antes que el del prefetch (polución de la caché). Si el prefetch es muy tarde, el dato se necesita antes de concretar el prefetch (ciclos de hit-wait, igual se considera útil).
- **Caché no bloqueante**: En una microarquitectura en pipeline que permite ejecución fuera de orden las instrucciones que siguen a un load que ocasiona un fallo en la caché, podrían continuar a las estaciones de reservación y ejecutar si tienen todos los operandos. Permitir accesos a la caché mientras se está esperando que se resuelva un faltante no agrega mucha complejidad al diseño. Esta optimización de hit-under-miss reduce la penalización. La caché no bloqueante (lockup-free o non-blocking cache) permiten varios faltantes concurrentes.

Memoria virtual

Espacio de direccionamiento lógico

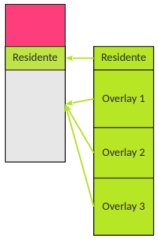
- Arquitectura 32 bits:
  - Registros de 32 bits
  - Referencia de memoria de 32 bits
  - Espacio direccionable de 232 bytes = 4GB
- Arquitectura 64 bits:
  - Registros de 64 bits
  - Referencias de memoria de 64 bits
  - Espacio direccionable de 264 bytes = 234 GB

Problemas

- Gran desproporción entre el espacio direccionable y la capacidad de memoria principal.
- Multiprogramación:
  - Varios programas residentes en memoria a la vez.

- Cuando un programa realiza una operación de I/O, otro programa toma el control de la CPU.

Administración de memoria



- **Overlay**: Programa muy grande para la memoria física. El programador divide al programa en porciones mutuamente excluyentes (overlays). El programa controla en ejecución qué overlays se cargan o sacan de la memoria física. Responsabilidad del programador que el programa no acceda a memoria no cargada en memoria física. Obliga a revisar los overlays si cambia el tamaño de la memoria.
- **Memoria Virtual (MV)**: Administra automáticamente los niveles de la jerarquía representados por memoria principal y secundaria. Permite compartir pequeñas porciones de memoria física entre varios procesos. Permite que algunos programas corran en cualquier posición de memoria física (relocation)

Los programas se compilan y enlazan (linking) como si pudieran acceder a todo el espacio direccionable.

- Dirección virtual (lógica): Dirección generada por la CPU.
- Dirección física: dirección virtual traducida a una dirección real de memoria física.
- Memory Management Unit (MMU): dispositivo que mapea direcciones virtuales a físicas.

Memoria virtual

Memoria caché	Memoria Virtual
La memoria física se divide en pequeños bloques (del mismo tamaño que las líneas de caché). Los datos se buscan primero en la caché (que tiene mucha menor capacidad pero es más rápida que la memoria RAM).	Es la administración de memoria física y disco, para contener al espacio de direccionamiento lógico. El espacio de direccionamiento lógico y la memoria física se dividen en bloques más grandes.

Reemplazos

- Los reemplazos en caché los controla el hardware.
- Los reemplazos en MV los controla el sistema operativo (SO). La penalización por un faltante es mayor, por lo tanto el SO puede tomarse más tiempo para decidir qué reemplazar.

Tamaño de la dirección

- El direccionamiento del procesador determina el tamaño de la memoria virtual.
- El tamaño de la caché es independiente.

La MV se puede dividir en dos categorías de organización en función de cómo se divide la memoria:

- Paginación: Bloques de tamaño fijo (páginas)
- Segmentación: Bloques de tamaño variable (segmentos)

La decisión afecta al procesador.

Estrategias híbridas

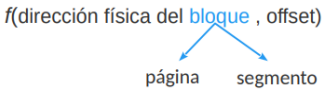
- Segmentación con paginado: Cada segmento se divide en un número entero de páginas. Busca obtener lo mejor de ambos.
- Paginación con tamaños variables: Múltiples tamaños de páginas. El tamaño mayor es un múltiplo potencia de 2 del tamaño menor.

¿Dónde ubicar un bloque en MP?

La penalización de un faltante en MV es muy alta. Se elige menor tasa de fallos sobre simplicidad del algoritmo. Estrategia fully associative: Los bloques se pueden ubicar en cualquier lugar de la memoria.

¿Cómo encontrar un bloque en MP?

Estructura de datos indexada por el número de página (paginación) o el número de segmento (segmentación). Una tabla que contiene la dirección física del bloque (página o segmento). La dirección física del dato buscado:



Paginación

- Frame: bloques en los que se divide la memoria principal.
- Página: bloque en los que se divide la memoria virtual

tamaño frame ≡ tamaño página

- Tabla de páginas: tabla indexada por número de página que contiene la dirección base del frame que contiene la página.

Acceso a memoria

1. La CPU genera la dirección lógica para acceder a memoria (load o store)
2. Con el número de página se indexa en la tabla de páginas.
3. Si se encontró un frame **f** para esa página quiere decir que está en memoria y se puede acceder. Si no está en memoria, ocurre un page fault y el sistema operativo toma el control (deberá traer la página a memoria).
4. Asumiendo que ya tenemos la página en memoria en el frame **f**, se concatena el offset al número de frame y se obtiene la dirección física.
5. Finalmente, obtuvimos la dirección en memoria física de la palabra al a que queríamos acceder.

Las direcciones generada por el CPU se divide en dos:

número de página (p)	offset (d)

El hw determina

- El tamaño del bloque → offset
- La longitud de la dirección virtual
  - → cantidad de páginas posibles.
  - → cantidad de frames en función de la RAM disponible

La Implementación de la Tabla de páginas es con registros de alta velocidad que realizan de manera eficiente la traducción de dirección virtual a física. Es útil cuando la tabla de páginas es pequeña. DEC PDP-11 (1970/80)

- 16 bits de dirección lógica
- 8K de página → 13 bits de offset
- 3 bits de número de página → tabla de páginas de 8 entradas.

✗ No es aplicable en computadoras modernas.

Tabla de páginas en MP: El hw tiene un registro PTBR (Page-Table Base Register) que apunta a la dirección física base de la tabla de páginas. Cambiar de tabla de páginas requiere solo cambiar el registro.

✗ Cada acceso a memoria demanda dos accesos a memoria

1. Acceder a la tabla de páginas para realizar la traducción
2. Acceder al byte buscado

Translation look-aside buffer (TLB): Un conjunto de registros de alta velocidad fully associative. Búsqueda rápida pero hw caro. Pocas entradas (entre 8 y 2048). Cada entrada en el TLB tiene:

- TAG (key)

- Valor
    - Número de frame físico
    - Bit de válidos, bits de protección, use bit, dirty bit (corresponde a la página.)

Acceso a memoria con TLB

1. Con la dirección lógica se indexa en la tabla de páginas (implica un acceso a memoria principal). Mientras se resuelve ese acceso, se busca (en paralelo) en el TLB.
2. Si hubo un TLB hit, la página está en memoria principal y ya tenemos el número de frame. Si no está en el TLB, hay que esperar el resultado de la tabla de páginas.
3. En cualquiera de los dos casos, una vez que tenemos el número de frame, formamos la dirección física y accedemos a memoria.

Cuando el número de página no se encuentra en el TLB hay que hacer una referencia a memoria para acceder a la tabla de páginas y obtener el frame correspondiente. Además, se actualiza el TLB para futuros accesos. Si el TLB está lleno, la política de reemplazo puede ser desde LRU hasta random y puede ser una decisión completamente del HW o puede intervenir el SO. En la implementación básica del TLB, cuando se cambia de tabla de páginas hay que limpiarlo.

Paginación Multinivel

Supongamos

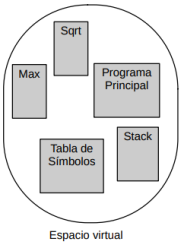
- Dirección lógica de 32 bits
- Página de 4KB (12 bits de offset)
- Hay 220 entradas en la tabla de páginas
- Si cada entrada tiene 4 bytes → 4MB de tabla de páginas

La tabla de páginas se divide en niveles. El primer nivel siempre se mantiene en memoria principal.

Tabla de páginas invertida. La cantidad de frames es mucho menor que la cantidad de páginas virtuales. Tienen una entrada por cada frame en memoria principal. Reduce el tamaño de la tabla de páginas.

Segmentación

La vista del usuario de la memoria no es complemente lineal. Varios módulos con un propósito propio que interactúan entre sí. No hay un orden definido entre los módulos. Los módulos son de tamaño variable.



Segmentación refleja el punto de vista del usuario en la administración de la memoria. El espacio direccionable se divide en un conjunto de segmentos. Cada segmento tiene un nombre y una longitud. Consiste de una secuencia lineal de direcciones, de 0 a un máximo. Una dirección especifica el nombre del segmento (número) y el offset dentro del segmento.

número de segmento (s)	offset (d)

Normalmente, el compilador crea los segmentos automáticamente. Un compilador de C puede crear segmentos para:

1. Código
2. Variables globales
3. El heap
4. La pila
5. Standard C library

El loader le asigna números a los segmentos.

Implementación de la Tabla de Segmentos

- Registros
  ✓ Rápido
  ✗ Muy grande
- Memoria. Requiere un registro STBR (segment-table base register) con la dirección base del segmento y otro registro STLR (segment-table length register) para la longitud.
  ✗ El mapeo de dirección lógica a dirección física requiere dos accesos a memoria.
- Tabla de segmentos en memoria + TLB

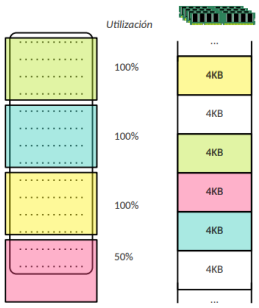
Segmentación con Paginación. Los segmentos se dividen en páginas de tamaño fijo. Algunas de las páginas del segmento están en MP y otras en memoria secundaria. Al paginar los segmentos (que son un espacio lineal), cada segmento necesita su propia tabla de páginas.

Paginación vs. Segmentación

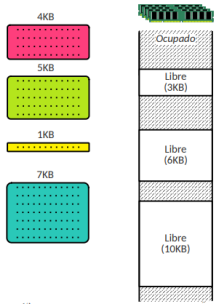
	Página	Segmento
Palabras por dirección	1	2 (segmento y offset)
Visible al programador?	No	Sí
Reemplazo de bloques	Trivial (son todos iguales)	Difícil (hay que encontrar una porción contigua de memoria libre y de tamaño suficiente)
Ineficiencia en el uso de la memoria	Fragmentación interna	Fragmentación externa
Tráfico eficiente con disco	Sí (se especifica un tamaño de la página que balancee los tiempos de acceso y de transferencia)	No siempre (dependerá del tamaño del segmento a transferir)

Fragmentación

- Fragmentación interna: queda una porción de la página sin usar.
  - Páginas de 4KB
  - Un bloque de información de 14KB
    - Datos
    - Código



- Fragmentación externa: porciones libre de la memoria pero que no pueden aprovecharse para contener un segmento.



Memoria Virtual + Memoria Cache

Al buscar en la caché se pueden distinguir dos tareas:

1. Indexar la caché
2. Compara tags

En ambos se puede usar tanto la dirección física como la virtual.

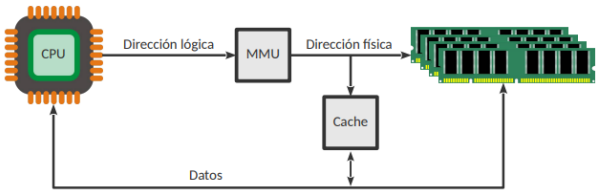
La caché se pude ubicar en dos lugares:

- entre el procesador y la MMU
- entre la MMU y la memoria física.

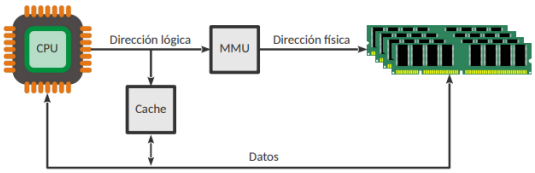
Hay tres posibles formas de acceder a la caché:

- Index y tag físicos
- Index y tag virtuales
- Index virtual y tag físico

Caché física: Los datos se guardan usando la dirección física. El procesador accede a través de la MMU. Index y tag físicos.



Caché virtual (o lógica): Los datos se almacenan usando la dirección virtual. El procesador accede directamente a la caché. Index y tag virtuales.



✓La caché virtual responde más rápido que la caché física. No requiere siempre mapear dirección lógicas a físicas. Solamente ante un faltante en la caché.

✗ La misma dirección virtual en dos aplicaciones diferentes se refiere a direcciones físicas distintas. Las aplicaciones tienen todas el mismo espacio de direcciones virtuales. La caché debe limpiarse al cambiar de aplicación o debe contener información adicional para identificar el espacio virtual al que se corresponde la línea.

✗ La misma dirección física puede corresponderse con varias direcciones virtuales. Problema de sinónimos o alias: Dos aplicaciones comparten el mismo segmento de código o datos. La posición de ese segmento compartido no es la misma en todos los espacios de direcciones virtuales. En caché virtual puede haber dos líneas ocupadas por sinónimos. Si una se modifica, la otra queda inconsistente. Solución

- Por hardware → I-cache AMD Opteron
- Por software → UNIX en la UltraSPARC Sun Microsystem

No es aplicable a caché de datos: Los dispositivos de I/O usan direcciones físicas

Index virtual + tag físico: Buena combinación para L1. La traducción comienza junto con el indexado.

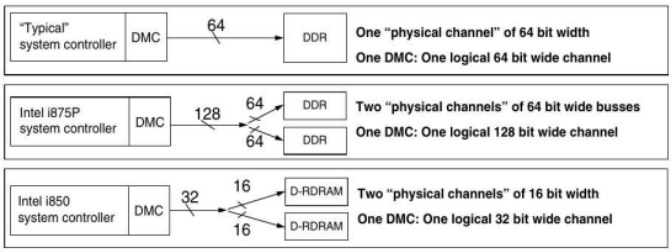
- Index virtual: El offset dentro de la página no varía. Parte del offset puede usarse para indexar.
- Tag físico: Asegura que cada bloque de caché se corresponda con una única dirección física.

Mejorar el ancho de banda

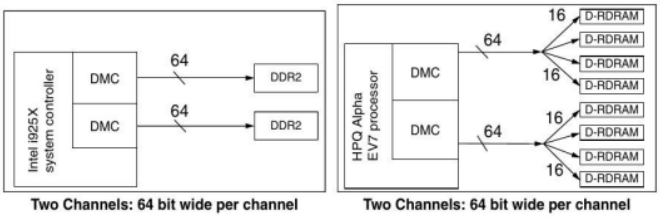
El bus transfiere la mayor cantidad de información posible por ciclo. Acelera la comunicación entre el Procesador y la MP. Dado un tamaño de ancho del bus, los chips de DRAM se organizan para, por cada requerimiento, proveer datos del ancho del bus.

Idealmente, el número de bits que se obtienen por requerimiento de lectura debería ser igual a la cantidad de bits que pueden transmitirse por el bus de datos. Si se leen menos bits, el controlador de memoria deberá incluir un buffer para concatenar los resultados.

Único controlador de memoria, único canal de acceso a memoria.

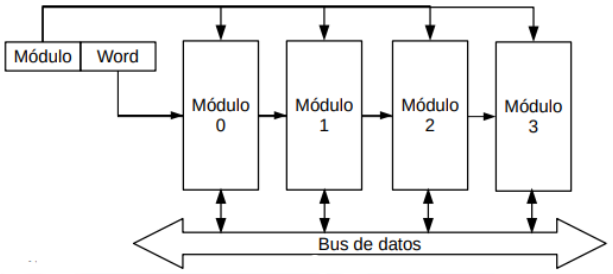


Múltiples controladores de memoria, múltiples canales lógicos de acceso a memoria.

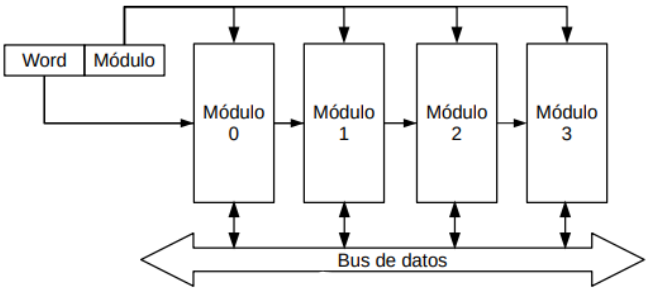


Interleaving de memoria. La memoria se divide en bancos (o módulos) que reciben comandos independientes desde el controlador de memoria. Palabras consecutivas se almacenan en bancos diferentes de memoria. Permite paralelizar las referencias a memoria consecutiva. Esos bancos pueden intercalar la información de diferentes maneras.

High-order interleaving ✗. Los bits más significativos seleccionan el módulo y los bits menos significativos indican el desplazamiento dentro del módulo. Posiciones contiguas de memoria se almacenan en el mismo módulo. No permite acceder a bloques.



Low-order interleaving ✓. Posiciones contiguas de memoria se distribuyen lo largo de los m módulos. Los bits menos significativos seleccionan el módulo y los bits más significativos indican el desplazamiento dentro del módulo.



Clasificación de Flynn

Clasificación de 1966. En función del flujo de instrucciones y datos en un solo procesador:

{instrucción única, múltiples instrucciones} x {dato único, múltiples datos}

	Single Data	*Multiple Data*
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

Está basada en 2 conceptos:

- Flujo de instrucciones: Se corresponde con el program counter. Un sistema con n CPUs tiene n program counters y por lo tanto n flujos de instrucciones.
- Flujo de datos: Se corresponde con el conjunto de operandos.

Flujo de instrucciones	Flujo de datos	Nombre	Ejemplos
1	1	SISD	Máquina Von Neumann Clásica
1	muchos	SIMD	Supercomputadora vectorial. Arreglo de procesadores.
muchos	1	MISD	?
muchos	muchos	MIMD	Multiprocesadores, multicomputadoras

SISD. Un único procesador Von Neumann clásico. Hay un único flujo de instrucciones y un único flujo de datos. Realiza una única operación a la vez.

- Ejecución concurrente en pipeline
- Procesadores superescalar con múltiples unidades funcionales.

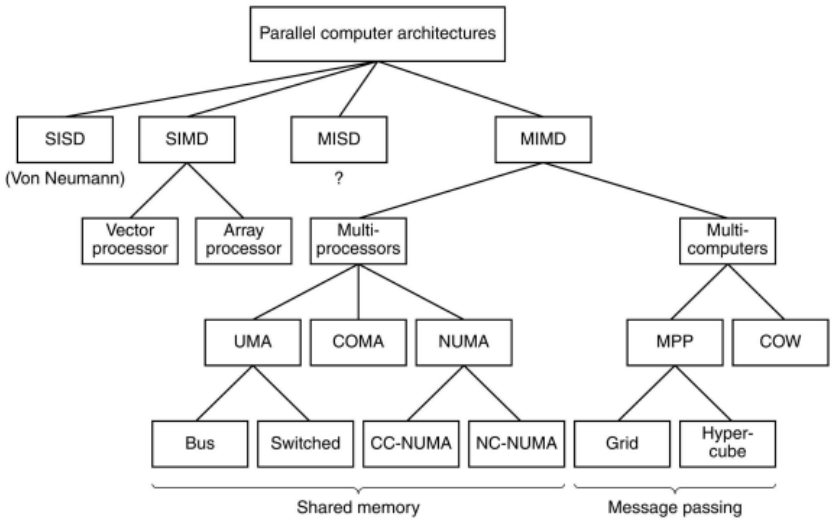
SIMD. Tienen una única unidad de control que procesa una única instrucción a la vez. Tienen múltiples ALUs para operar sobre múltiples conjuntos de datos en simultáneo. Extensiones multimedia:

- MMX (enteros, 1997 Pentium)
- 3DNow! (pf, 1998 K6-2), SSE (1999, Pentium III), SSE2,3,4.x..., AVX (2010)

GPU

MIMD. Varios procesadores operan en paralelo de manera asincrónica. Se distinguen en:

- Cómo se comunican los procesadores
  - Memoria compartida
  - Pasaje de mensajes
- El acceso a memoria principal
  - Uniforme
  - Distribuido entre los procesadores
- Número de procesadores, homogeneidad, interconexión procesador-procesador y procesador-memoria, coherencia de caché, sincronización, etc...



Multi-processors: Los procesadores paralelos comparten un mismo espacio de direcciones de memoria. La comunicación es a través de instrucciones load-store.

Acceso uniforme a memoria (UMA): Todos los procesadores tienen el mismo tiempo de acceso a cualquier módulo de memoria. Todas las palabras de memoria se acceden el mismo tiempo. Si técnicamente no se puede, se demora la referencia más rápida. Hace que el rendimiento sea predecible.

Acceso no uniforme a memoria (NUMA): Memoria compartida distribuida. Hay un módulo de memoria cercano a cada procesador (al cual es más rápido acceder). La ubicación del código y datos afecta el rendimiento.

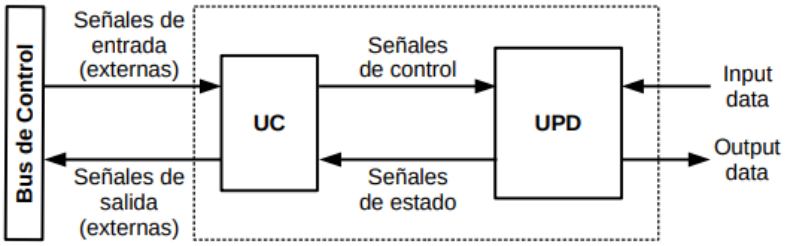
Multi-Computers: Las multicomputadoras no tienen memoria compartida a nivel arquitectura (el SO en una CPU no puede acceder con load/store a memoria asociada a otra CPU). La comunicación es a través de mensajes. No tienen acceso directo a memoria remota.

Massively Parallel Processors (MPP): Supercomputadoras fuertemente acopladas por una red de interconexión rápida y propietaria.

Cluster, Cluster of Workstations (COW): PC regulares, workstations, servidores conectados en una LAN. Nodos más económicos.

Unidad de Control

Provee las señales de control para la operación y coordinación de las componentes del procesador. El principal propósito de la Unidad de Control es decodificar las instrucciones y generar las señales que permitan realizar la operación deseada.



La Unidad de Control (UC) indica la Unidad de Procesamiento de Datos (UPD) una secuencia de comandos. La UPD manipula los datos de acuerdo a lo indicado por la UC.

Tareas de la UC

1. **Secuenciamiento de las instrucciones (FETCH)**:
    - Secuenciamiento lineal
        - Program counter

- Secuenciamiento no lineal
    - Branch – jump
    - Transferencia de control entre (sub)programas
        - Llamada a subrutina: Transferencia temporaria del control de A a B, iniciado por A
        - Interrupción: Transferencia temporaria del control de A a B, iniciado por B o un dispositivo asociado a B
        - Almacenar
            - La dirección de retorno
            - Posibles parámetros de A a B
            - Variables locales a B

2. **Interpretación de la instrucción (DECODE)**: La UC interpreta la instrucción para determinar qué señales de control debe transmitir. Estas señales de control se transmiten a través de líneas de control. Se distinguen 4 grupos de señales de control:
    - Señales de control: Objetivo principal de la UC. Señales que controlan la operación de la UPD.
    - Señales de estado: Permiten que la UC tome decisiones basadas en los datos. Le permiten a la UPD indicar errores en el procesamiento de los datos (x ej. overflow).
    - Señales de entrada (externas): Señales que recibe la UC a través del bus de control (señales de start, stop y temporizado).

      Las señales externas se usan para sincronizar con otros controladores.
    - Señales de salida (externas): Incluye señales de control a la memoria y señales de control a los módulos de I/O
3. **Ejecutar la instrucción (EXECUTE)**: Generar todas señales de control necesarias para indicarle a la UPD qué procesamiento aritméticológico debe realizar.

Implementación de la Unidad de Control

La UC tiene típicamente dos partes:

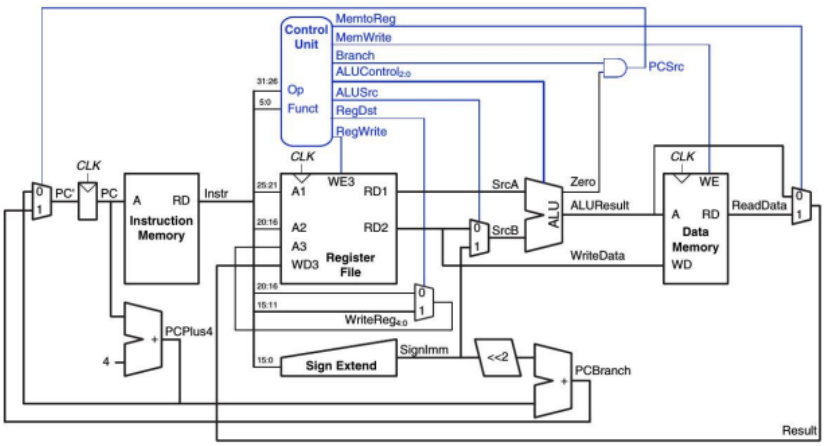
- Una parte combinacional que carece de estado (por ej. para las decodificaciones, diseño único-ciclo)
- Otra parte secuencial para el secuenciamiento de las instrucciones y para el control principal de un diseño múltiple-ciclo.
    - ROM
    - PLA
    - Contador

    Control Cableado
    - Microprogramación

    Control Microprogramado

Control cableado

Procesador único-ciclo.



UC Combinacional. La UC computa las señales de control basándose en el opcode y en el campo funct de la instrucción. La mayoría de la información de control proviene del opcode. Implementa la tabla de verdad.

- R-type

Opcode	RS	RT	RD	...	*Funct*
6 bits	5 bits	5 bits	5 bits	...	6 bits

- J-type

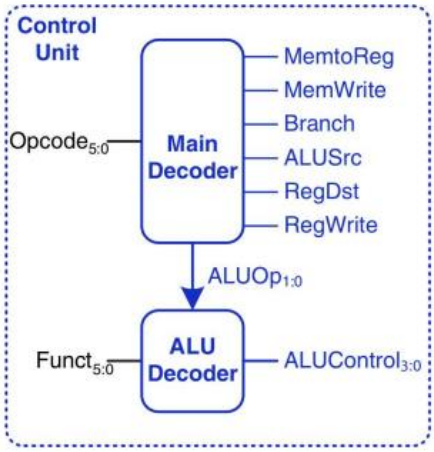
Opcode	Address
6 bits	26 bits

- I-type

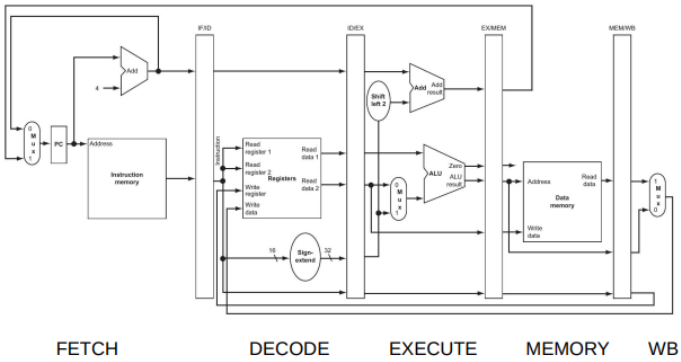
Opcode	RS	RT	Inmediato
6 bits	5 bits	5 bits	16 bits

- UC

Opcode (6 bits)	Func (6 bits)	Señales de control (10 bits)

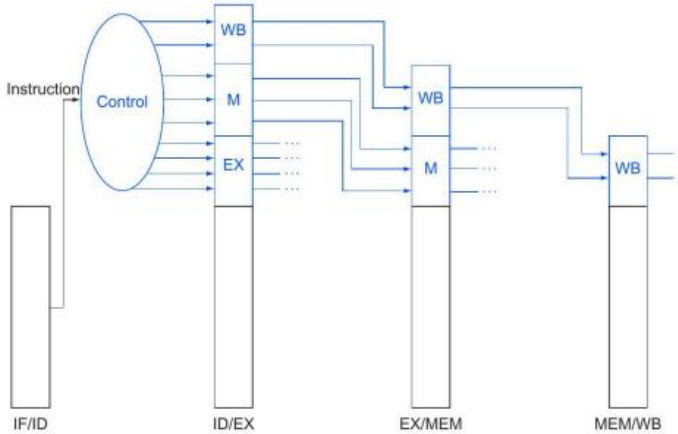


Procesador en Pipeline

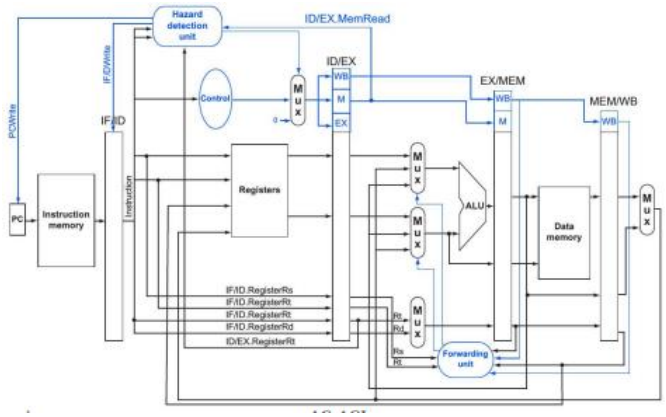


Control del Pipeline. Tiene las mismas señales de control que el procesador único-ciclo. Utiliza la misma UC Las señales de control se activan en la etapa DECODE pero se utilizan en las etapas siguientes.

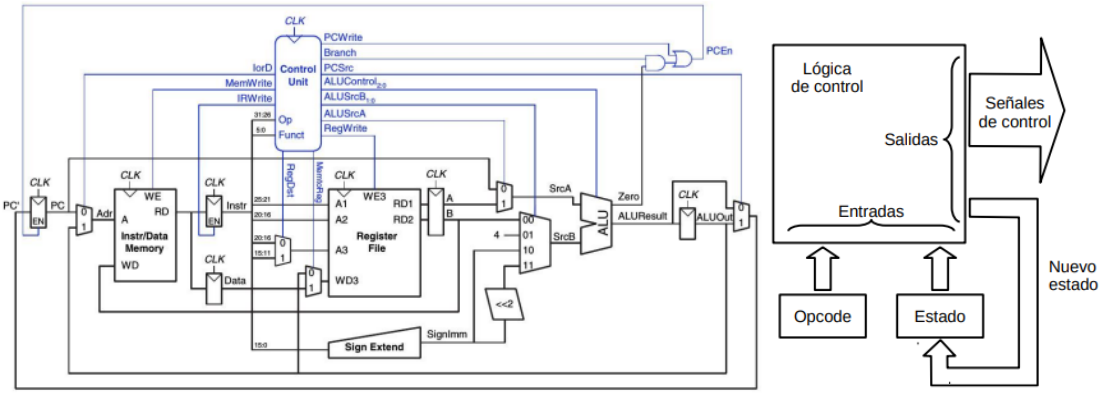
Se extienden los registros interetapas para que incluyan la información de control.



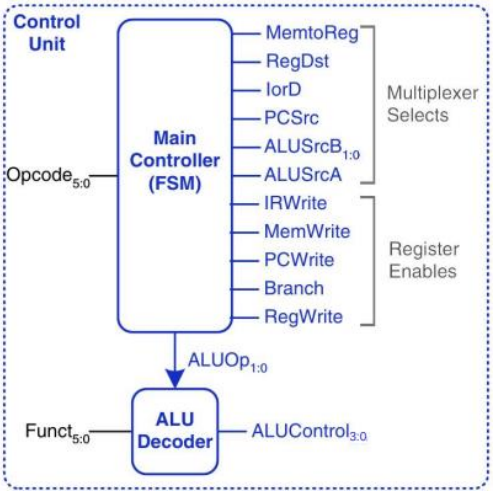
Además de registros interetapas, agrega multiplexores y lógica de control para resolver hazards y forwarding.



UC con una Máquina de estados. Procesador múltiple-ciclo



Estructura interna de la UC combinada para un Procesador múltiple-ciclo



Con un contador

- Usa un contador para calcular el próximo estado
- Debe contemplan saltos a próximos estados no consecutivos:
  - Basándose en el opcode
  - Inicio próxima instrucció

Microprogramación

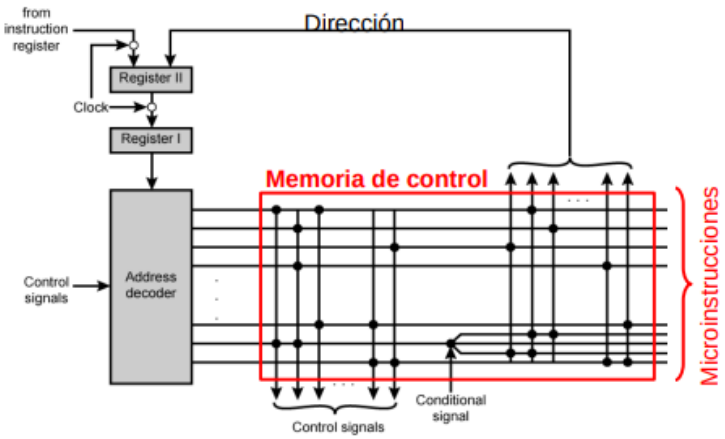
La implementación de la UC como una máquina de estados usando un contador podría verse como una computadora. El concepto de microprogramación fue propuesto por M.V. Wilkes en 1951 y utilizaba un decoder y una ROM compuesta por una matriz de diodos.

Microoperación: Pasos elementales en los que se puede dividir la ejecución de una instrucción:

- Incrementar el PC
- Transferencia registro a registro
- Suma en la ALU
- ...

Microinstrucción: conjunto de microoperaciones que ocurren en un momento de tiempo.

Microprograma (o firmware): Secuencia de microinstrucciones



Durante un ciclo, se activa una fila de la matriz con un pulso. Genera las señales donde haya un diodo presente. La parte izquierda genera las señales de control. La parte derecha genera la próxima dirección. La próxima dirección será el opcode del IR o la dirección proveniente de la matriz (dependiendo de las señales de control)

Por cada microoperación, la UC debe generar un conjunto predefinido de señales de control. Las señales de control pueden valer 1 (activada) o 0 (desactivada). Cada microoperación puede representarse como un patrón diferente de 1s y 0s. Este patrón se llama palabra de control (control word). Las microinstrucciones se organizan en una memoria.

Todas las microinstrucciones tiene tres campos importantes:

- Palabra de control
- Dirección de la (posible) próxima microinstrucción
- Bits de condición

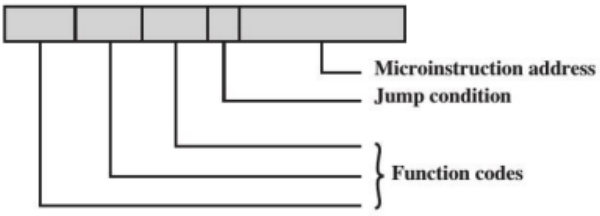
Palabra de control	Bits de condición	Dirección

La longitud de la microinstrucción está relacionada a:

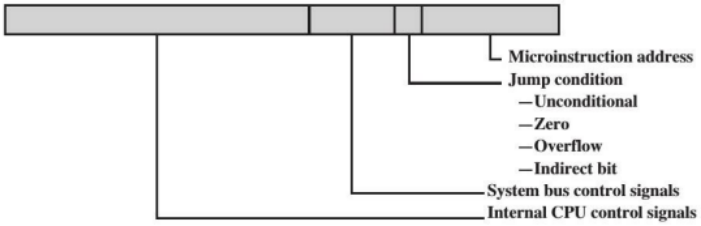
1. El número de microoperaciones que activa simultáneamente (grado de paralelismo)
2. El grado de codificación de la información de control.

En función del tamaño de la microinstrucción:

- *Microprogramación vertical*: Formato corto de microinstrucción. En cada ciclo puede activarse un número muy limitado de microoperaciones (x ej. 1). Alto grado de codificación de la información de control.



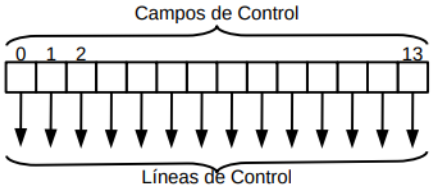
- *Microprogramación horizontal*: Formato largo de microinstrucción. En cada ciclo puede activarse un gran número de microoperaciones. Poca codificación de la información de control.



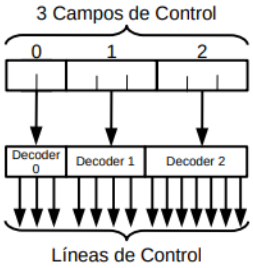
El concepto de VLIW desciende de la microprogramación horizontal

Codificación de la información de control

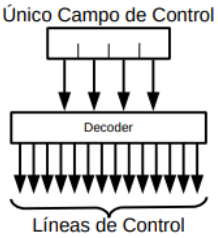
- Sin codificación



- Codificación media



- Codificación completa



Interpretación de la microinstrucción

1. Activar o desactivar las líneas de control siguiendo el patrón especificado por la palabra de control. Esto ejecuta una o más microoperaciones.
2. Si la condición indicada por los bits de condición es:
    a. Falsa: La próxima microinstrucción a ejecutar es la microinstrucción consecutiva.
    b. Verdadera: La próxima microinstrucción a ejecutar es la indicada en el campo de dirección.

Funcionamiento

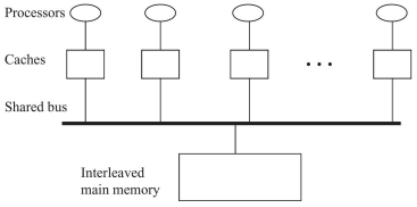
1. La lógica de secuenciamiento dispara una operación de lectura en la memoria de control.
2. La palabra cuya dirección se especifica en el Control Address Register (CAR) se almacena en el Control Buffer Register (CBR)
3. El contenido del CBR genera señales de control e información sobre la próxima dirección.
4. A partir de la información sobre la próxima instrucción del CBR y flags de la ALU, la lógica de control carga la próxima dirección en el CAR.
    a. Cargar la instrucción consecutiva (+1 al CAR)
    b. Carga el campo de dirección del CBR en el CAR, basándose en la condición de salto de la microinstrucción.
    c. Carga el CAR en función de del opcode del IR.

Control cableado	Control microprogramado
– Lógica compleja para el secuenciamiento de las micro-operaciones – Costoso de mantener y actualizar + Es más rápido Dominante en arquitecturas RISC por el formato más simple de instrucciones.	+ Diseño simplificado + Más económico + Menos propenso a errores – A tecnología comparable, es más lento. Dominante en arquitecturas CISC.

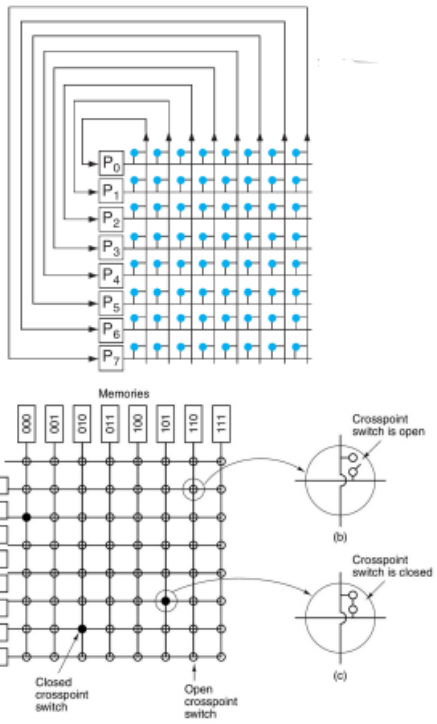
Coherencia de cache

Interconexión memoria/procesador (y procesador/procesador)

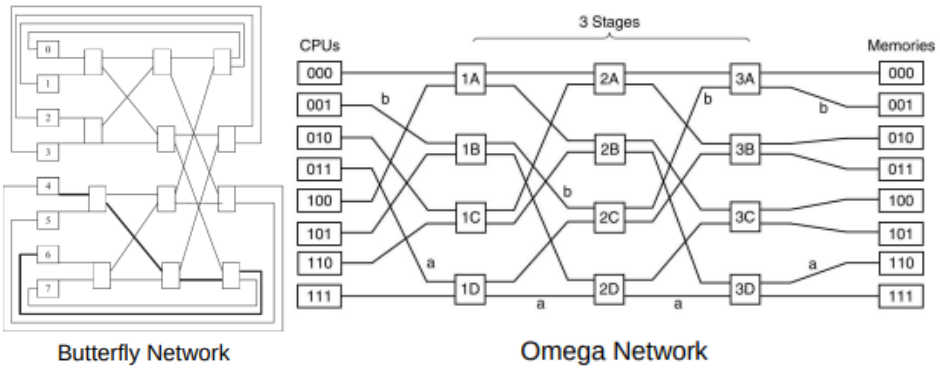
Principales esquemas de conexionado:



- **Shared bus**:
  - ✓Forma más simple de conexión
  - ✓Bajo costo
  - ✓Facilidad de uso
  - ✓Capacidad de hacer broadcast
  - ✗ Restricciones físicas (longitud, carga eléctrica) que limitan el número de dispositivos que pueden conectarse.
  - ✗ A mayor cantidad de dispositivos se incrementa la pelea por el bus.
- **Crossbar**:
  - ✓Provee máxima concurrencia
    - Entre n procesadores y m memorias (UMA)
    - Entre n procesadores (NUMA)
  - ✗ El costo crece con
    - n × m
    - n^2



- **Redes directas**: Se basan en el crossbar de 2x2. Múltiples etapas de interconexión. Cada procesador está a la misma distancia de cada memoria o de los demás procesadores.

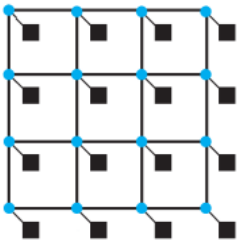


Butterfly Network

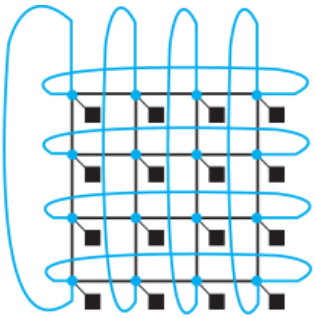
Omega Network

- **Meshes**: Interconexión indirecta entre todos los nodos. Cada nodo está conectado con un enlace de longitud 1 con un subconjunto de nodos vecinos (dimensión). La distancia entre procesadores y memorias/procesadores varía.

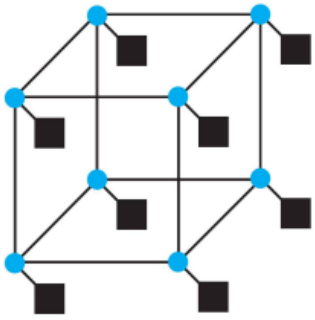
Anillo
Cada nodo tiene conexión directa con 2 vecinos.
Conexión unidimensional.



Mesh 2D
Cada nodo interior tiene conexión directa con sus 4 vecinos.



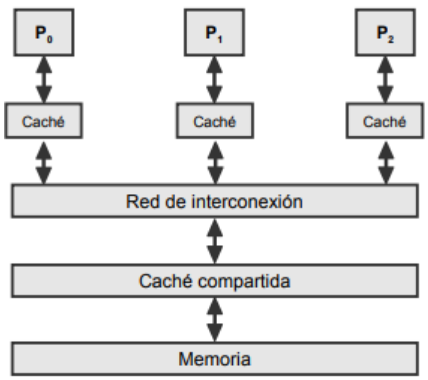
Torus 2D
Todos los nodos están conectado con un enlace de longitud 1 con 4 vecinos.



Hipercubo de *m* dimensiones
Cada nodo está conectado con un enlace de longitud 1 con *m* vecinos.

Multiprocesadores y la coherencia de caché

Memoria caché en multiprocesadores. Varios procesadores. Cada procesador con su propia memoria caché. Todos acceden al mismo espacio de direcciones. Se comunican por loads y store. Pueden tener compartir niveles de memoria caché.



Cómo se sabe el estado de un bloque. Fuertemente dependiente del tipo de interconexión

- Snooping
  - Cada caché mantiene el estado del bloque compartido.
  - Shared bus (fácil hacer broadcast).
  - Los controladores de caché controlan (o espían) el bus por si hay un requerimiento sobre un bloque que tengan.
- Basados en directorios
  - Se almacena un directorio con información del estado de cada bloque compartido de memoria física.
    - UMA: centralizado en algún punto común (por ej. último nivel de caché)
    - NUMA: directorios distribuidos.

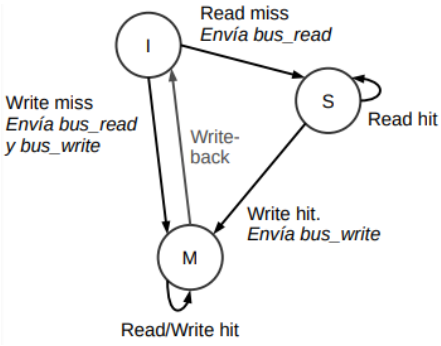
Complejidad del protocolo

- Write-invalidate protocol: Cuando un procesador escribe en un bloque se invalidan todas las demás copias.
  - **MSI**: Un bloque tiene 3 estados posibles:
    - Modified (M): La línea es la única copia en el sistema.
    - Shared (S): En el sistema hay varias copias de la línea y son todas iguales.
    - IInvalid (I): La línea estuvo en la caché pero fue invalidada.
  - MESI (MSI + Exclusive)
  - MESIF (MESI + Forward), MOESI (MESI + Owned)
- Write-update protocol: Cuando un procesador escribe en un bloque se actualizan las demás copias.

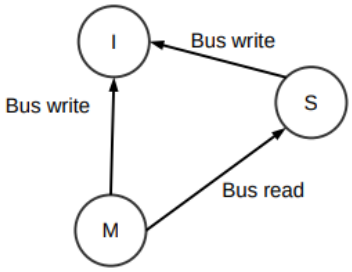
MSI + Snooping

Estado de una línea en la caché local

- Según acciones del procesador



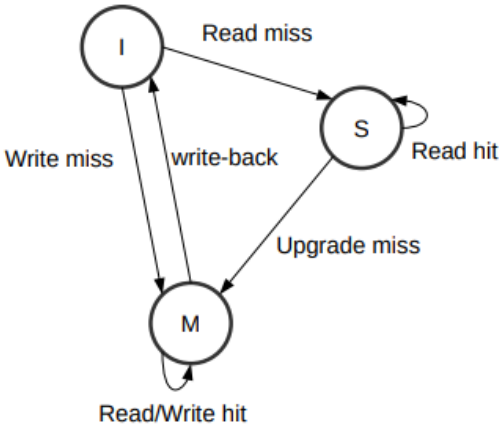
- Según señales externas



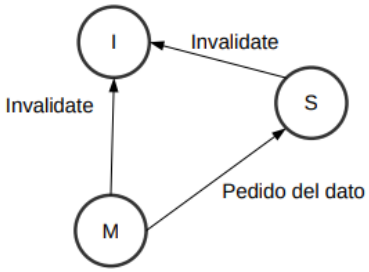
MSI + Directorio

Estados de una línea en la caché local

- Según acciones del procesador local



- Según señales externas



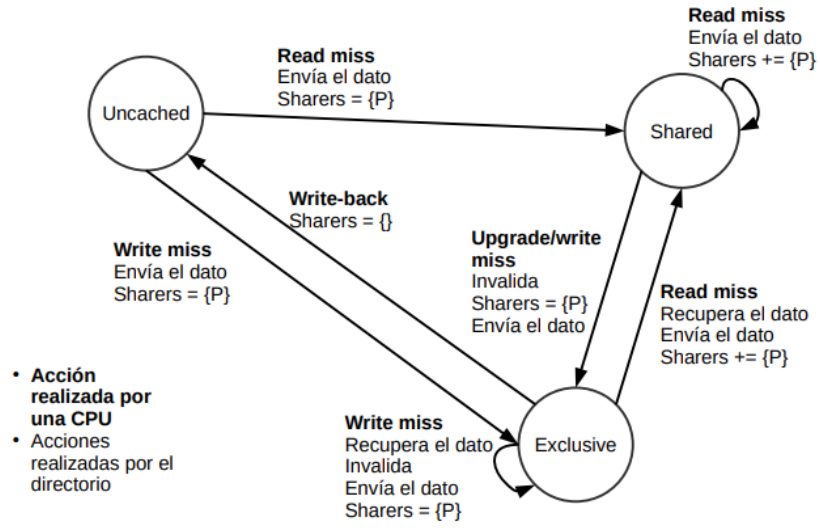
Hay tres posibles estados:

- Uncached: ninguna caché tiene la línea
- Shared: la línea está en varias caché como sólo lectura
- Exclusive: hay una caché que tiene la única copia válida de la línea.

Pueden ocurrir cuatro eventos:

- Read/write miss: un procesador quiere leer/escribir una línea que no tiene en caché.
- Upgrade miss: un procesador quiere escribir una línea que tiene en caché pero como sólo lectura.
- Write-back: un procesador escribe una línea en el nivel inferior antes de reemplazarla.

¿En qué estados puede ocurrir cada evento?



Tipos de fallos en caché

- Capacitivos
- Compulsivos
- Conflictivos

- De coherencia (Coherence miss): Resultado de la comunicación entre procesadores. Motivos:
    - Real existencia de datos compartidos.
    - Falsos datos compartidos. Distintos procesadores acceden a distintos datos pero que están en el mismo bloque. Dependientes del tamaño del bloque.