In Furfi we trust

Final de Orga II resumen

fuente: cubawiki

Versión en LaTeX:
Camila Gallo

Índice

1. Un poco de historia

1.1. Máquina de Turing

En 1936, Turing publica su paper en donde define lo que es una máquina de Turing. Esto es un modelo matemático basado en autómatas que "traduce"las fórmulas matemáticas en programas. Es la primera vez, que se define un modelo de computación y sienta las bases de lo que hoy entendemos como computación.

1.2. Modelo de Von Neumann

Basado en el modelo teórico de Turing, 9 años después, publica un paper en donde define en detalle el modelo de cómputo desarrollado por Turing años antes, y propone este modelo de computadora.

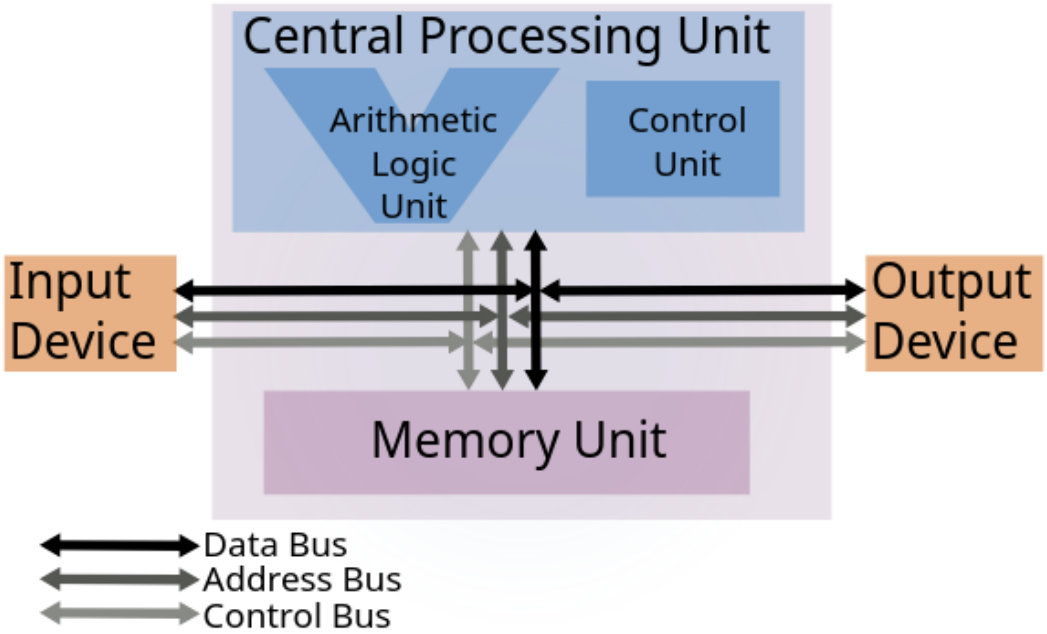

Figura 1: Modelo simplificado

Este modelo fue el predominante dentro de la 1era y 2da generación de computadoras. Los componentes más importantes eran:

- Central aritmética (CA): Una unidad de procesamiento aritmético, de lógica y registros.
- Central de Control (CC): Una implementación de la máquina de estados, con registro de instrucciones y registro Contador de Programa.
- Memoria: Para guardar los datos y el código que se ejecutaban en ese momento. No había espacio para otro programa. Ambos se almacenaban en el mismo lugar, esto daría problemas a un cuello de botella.
- I/O: Mecanismos tanto de entrada como de salida para que se pudieron ingresar datos y comandos a la computadora y para que esta puediera enviar resultados al exterior.

- Memoria Externa: Era otro medio de almacenamiento con mayor capacidad que la Unidad de Memoria. Se guardaban otros programas y datos que luego se copiaban a la Unidad de Memoria. Por ejemplo, cintas perforadas.

Tenemos entonces un modelo en donde hay un programa almacenado, los datos y el programa están en el mismo y único banco de memoria. El cpu es una máquina de estados perpétua.

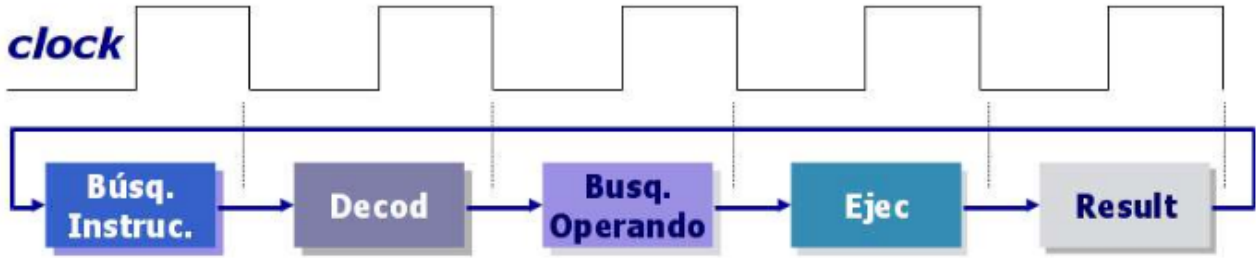

Figura 2: Máquina de estados de la máquina de Von Neumann

1.3. Modelo de Hardvard

Con el pasar de los años, en menos de dos décadas, se necesitó leer **al mismo tiempo**(o al menos que se comporte como así lo fuese) tanto los datos de la operación a realizar como sus operandos. Entonces, claramente había un problema con el modelo de Von Neumann porque al haber un único banco de memoria con buses compartidos, no se podía hacer ambas cosas .ª la vez". Esto se conoce como el cuello de botella de Von Neumann.

Para solucionar el problema del cuello de botella, revisaron la arquitectura de la computadora Mark I, un computador electromecánico desarrollado por IBM y la universidad de Harvard entre 1939 y 1944. Es decir, era anterior a la ENDVAC de Von Neumann.

La Mark I tenía datapaths diferentes para los datos que ingresaban y para las instrucciones que se ejecutaban. Por eso era interesante a pesar de ser un computador en decimal y que tarda entre 0.3 y 10 segundos para cada cálculo.

1.4. Computadoras de 2da Generación

Son computadoras con transistores. Aparecen en la década del 50. Las ventajas que tienen es que tienen menor consumo, menor espacio y mejor desempeño a largo plazo.

También aparecen los lenguajes de alto nivel. En 1957 se escribe el primer compilador de FORTRAN.

Se consolida IBM, nacen XEROX y DEC.

1.5. 3era generación

Consiste de las computadores de circuitos integrados (CI). Es decir, chips. Aparecen en la época de los 60.

Se produjeron los minicomputadores y nace la programación estructurada. Se libera el primer compilador de C y nace UNIX.

Nota: En 1971, Intel presenta el CI 4004, el primer microprocesaor de la historia.

1.6. Problemas a resolver

- Los usuarios querían disminuir el costo de actualización de los equipos.
- Los equipos eran incompatibles con otros modelos de la misma marca. Cambiaba la ISA, las toolchains, los SO, etc. Cada mercado era distinto.
- Cada actualización te hacía cambiar todo.
- Empieza a hacer más ruido la palabra compatibilidad. Al menos para IBM.

1.7. La era del silicio

Hay dos aspectos importantes cuando se diseña un computador. El diseño del:

- El Datapath. Es el conjunto de recursos que se utilizan para que los números se almacenen en registros y las operacion entre ellos se puedan ejecutar.
- El Control. Es la lógica necesaria para controlar el correcto envío por el datapath de las operaciones que se van a realizar.

El control es el item más complejo. Y en IBM, a Maurice Wilkes, se le ocurrió el concepto de **microcódigo**, una lógica de control para un microprocesador. En ese momento como la ROM era más barata, se propuso implementar el microcódigo en una memoria de estas y cada palabra era una microinstrucción(pequeños programas). A mitad de la década del 60, IBM anuncia que sus modelos de mainframes IBM 360 se basarán en microcódigo.

Este control por microcódigo dio a luz al diseño de una ISA con instrucciones de complejidad creciente. La complejidad se cargaba en el microcódigo en la CPU y ocupaba la menor memoria posible porque no se podía minimizar el tamaño de los datos, era lo que era.

Nota: Para el caso del código, menos instrucciones implica menos memoria.

Acá elijo no anotar varias cosas.

1.8. Nuevo paradigma de arquitectura: RISC

Antes de las computadoras personales, lo que predominaban era los mainframes y quienes dominaban el mercado eran IBM, DEC, Amdahl, entre otros.

En los 80, Intel lanza los primeros microprocesadores de 16 bits y arrasa con todo. No olvidemos que el se de instrucciones tiene una complejidad creciente con cada lanzamiento.

Durante la época de los años 60 y 70, IBM, Control Data Corporation y Data General, lanzan lines de investigación para el diseño de chips con pocas pocas instrucciones simples. En los años 80, en la universidad de Berkley y en la Stanford, se publican los primeros papers donde se presenta una arquitectura antagonista a la que dominaba en ese momento llamada RISC. Por sus siglas en inglés: Reduced Instruction Set Computer.

Nota: Lo que motiva a llamar a los procesadores de ese entonces como CISC: Complex Instruction Set Computer.

En RISC lo que importa es lo siguiente

- Hay 32 registros de propósito general(O sea, una banda).

- Las instrucciones se ejecutan en un sólo ciclo de clocl.
- Las instrucciones derivan en códigos de operación de igual tamaño y formato.
- Las instrucciones deben ser sencillas de decodificar.
- No se utiliza microcodigo. O sea, no existen instrucciones complejas. Ni mul ni div.
- Los datos en memoria se accenden mediante LOAD y STORE.

Nota: Por estas razones, verificar una máquina RISC es más simple y es menor la probabilidad de tener un bug en el procesador.

Nota: Es esperable tener programas de mayor tamaño.

1.9. Resumen del capítulo

1.9.1. Arquitectura Harvard

- Las instrucciones y los datos se leen desde memorias físicamente diferentes y por caminos de señal diferentes.
- Se pueden tener tecnologías y organizaciones diferentes para la memoria usada para instrucciones y para datos, además de caminos de señal (buses) independientes.
- Si tenés buses diferentes, tenés dos direcciones `0x0000_0000` de memoria.
- La memoria de instrucciones puede estar organizada en palabras de 4 bytes (32 bits) y la de datos en 1 byte (8 bits), sin que nada se rompa.

1.9.2. Arquitectura Von Neumann

- Las instrucciones y los datos se guardan y leen de la misma unidad de memoria.
- Comparten la misma organización y tecnología.
- No hay separación entre memoria de instrucciones y memoria de datos.

Los microprocesadores actuales presentan una organización siguiendo el modelo de Von Neumann a los usuarios. Pero, desde los 90 que se usa una tecnología **split cache**. Se dividen los primeros niveles de memoria cache en una de datos y otra de instrucciones con dos buses independientes para accederlas en paralelo. Es decir, un modelo Hardvard. Sin embargo, los niveles de memoria más externos almacenan datos e instrucciones juntos, o sea, Von Neumann. El subsistema de memoria cache está diseñado y dimensionado para que se resuelvan en el él la mayoría de los accesos de memoria de parte de las CPUs. Un valor típico de eficiencia de un cache de Nivel 1 es 80 %. O sea, se resuelve en una memoria organizada según el modelo de Hardvard.

Nota:	El nivel 1 de cache no es estrictamente Hardvard porque quien programa ve un solo espacio de direccionamiento en lugar de dos. Esta división la realiza el sistema Cache.

Nota:	Sólo los procesador de señales (DSP) son puramente Hardvard. Los procesadores de propósito general actuales y algún que otro microcontrolador CORTEX-M trabajan con split cache en nivel 1 de su jerarquía de memoria y después juntan código y datos en los niveles restantes.

Nota:	Como todos presentan un mapa de direcciones de memoria único al programador, se denominan çasi Von Neumann".

2. Arquitecura vs. Microarquitectura

2.1. Arquitectura

Es el **conjunto de recursos accesibles para quien programa**. Se suelen mantener a lo largo de distintos modelos de procesadores. Estos recursos incluyen:

- Registros
- Set de instrucciones
- Estructuras de memoria relacionadas(por ej. Descriptores de páginas)

2.2. Microarquitectura

Es la **implementación** de la arquitectura, el diseño lógico detrás del set de registros y del modelo de programación. Cambia de modelo a otro dentro de una misma familia de procesadores. Se trata de definir acá los aspectos más relevantes en la arquitectura de un computador para maximizar el rendimiento del procesador pero sin dejar de lado el costo o consumo de energía, etc.

Incluye el diseño de

- El modo en el que se manejará la memoria y sus modos de direccionamiento
- El set de instrucciones
- Los bloques funciones del CPU: estos son, el File register, el data path y el diseño de la lógica de control
- e implementación del circuito integrado, su encapsulado, su montaje, alimentación y refrigeración

2.3. ISA

Cuando hablamos de ISA, nos referimos al Instruction Set Architecture que son el conjunto de set de instrucciones visibles para el programador. Además es el límite entre el software y el hardware. Y define cosas como:

- La clase de ISA: Con prop general vs Registros dedicados. ISAs registro-memoria vs ISAs Load Store.
- El direccionamiento a memoria. La alineación obligatoria de los datos vs la administración de los bytes.
- El modo de direccionamiento: Cómo se especifican los operandos.
- El tipo y tamaños de los operandos: Los diferentes tamaños y precisiones. Por ejemplo, enteros, floats y doubles no son tratados igual.
- Operaciones: Si es una arquitectura RISC o CISC. O sea, pocas instrucciones simples o muchas y complejas.
- Instrucciones de control de flujo: Que definen los jumps, y los calls.
- La longitud del código: Si las instrucciones que provee la ISA son de tamaño fijo o variable.

Entonces, podemos decir que la microarquitectura comprende la Organización y el Hardware.

Nota: Microarquitectura = Organización + Hardware.

2.4. Organización

Se refiere a los detalles de implementación de la ISA.

- La organización e interconexión de la memoria.
- El diseño de los diferentes bloques de la CPU y la implementación del set de instrucciones.
- La implementación del paralelismo en el nivel de instrucciones y datos.

Podemos entonces encontrar dos ISAs iguales pero con distinta organización.

Nota: Los procesadores AMD FX y los Intel Core i7 tienen la misma ISA pero organizan su caché y ejecución de maneras distintas.

2.5. Hardware

Son los detalles de diseño lógico y la tecnología de fabricación. Entonces, pueden existir procesadores con la misma ISA y la misma organización pero distintos a nivel de hardware y diseño lógico.

Nota: El Pentium 4 diseñado para escritorio es distinto en nivel del hardware al Pentium 4M diseñado para notebooks. Este último tiene una lógica de control para el consumo de energía.

3. Implementando la arquitectura

3.1. El procesador

Sea cual sea el set de instrucciones a implementar, **hay dos pasos fundamentales** que cualquier instrucción **debe** realizar.

1. Aumentar el Program Counter(PC) para que apunte a la siguiente instrucción.
2. Leer uno o dos registros(depende de la instrucción) contenidos en la palabra de la instrucción.

Y las acciones posteriores depende de la instrucción. En general, el siguiente paso es utilizar la ALU para acceder a memoria (hay que calcular la dirección de memoria). Además, las instrucción aritméticas y lógicas usarán la ALU para ejecutar la operación solicitada y/o los saltos condicionales para evaluar la condición de verdad del salto.

3.2. Implementación del procesador

3.2.1. El datapath, las unidades de control y el control

El data path es el camino que recorren las instrucciones y los datos por los buses internos y los bloques circuitales necesarios para su procesamiento. Inicia en la memoria cuando se busca una instrucción o dato. **Afecta al PC**, que se debe ajustar a la siguiente instrucción.

Por ejemplo, para el siguiente data path, se suma 4 el PC porque asumimos una CPU RISC con tamaño de instrucción de 32 bits (4 bytes).

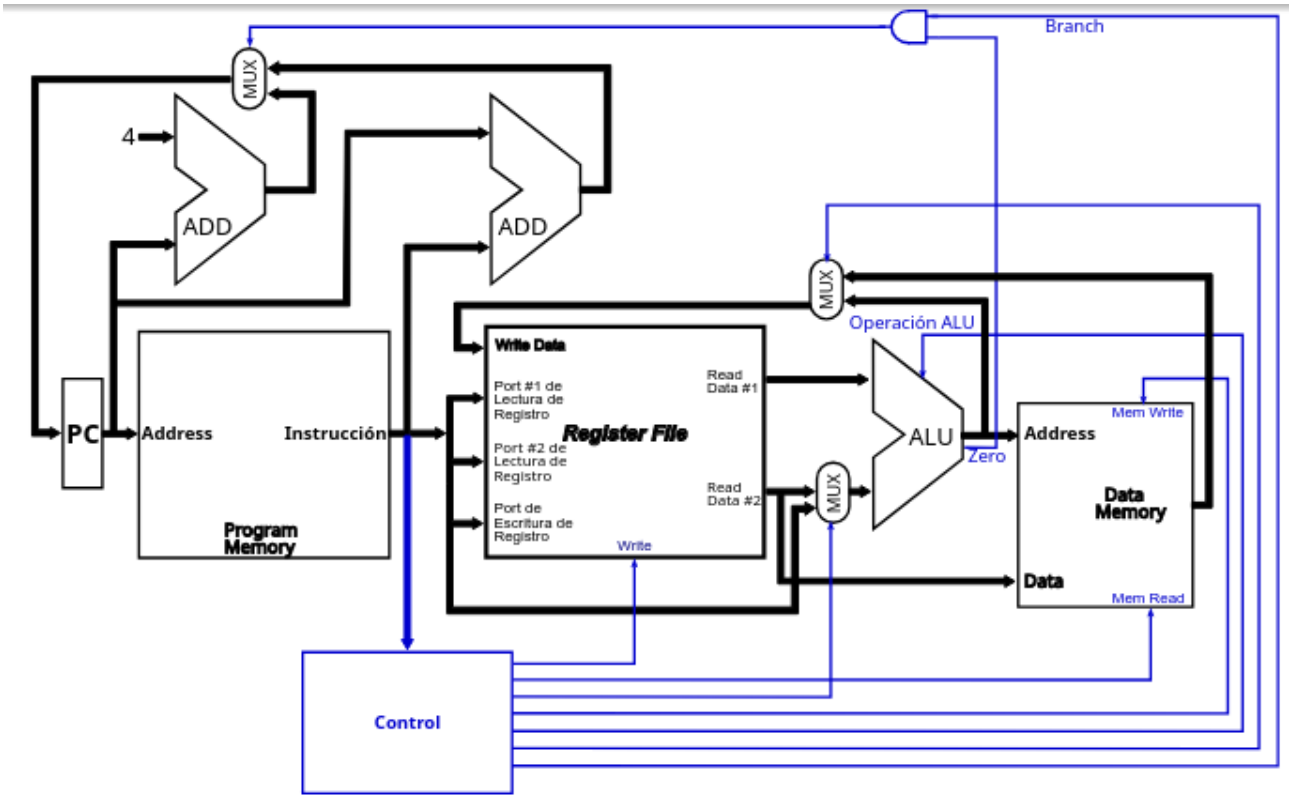

Figura 3: datapath de un CPU RISC de tamaño 4 bytes

3.2.2. El register file

Es parte fundamental en el data path y consiste de un arreglo de tamaño N, o sea N registros, a los cuales se pueden acceder para lectura o escritura. Están numerados del 0 al N - 1.

Nota:	Se puede implementar con un decodificador para cada puerto de lectura/escritura y un arreglo de registros construido a partir de flips fliops D.

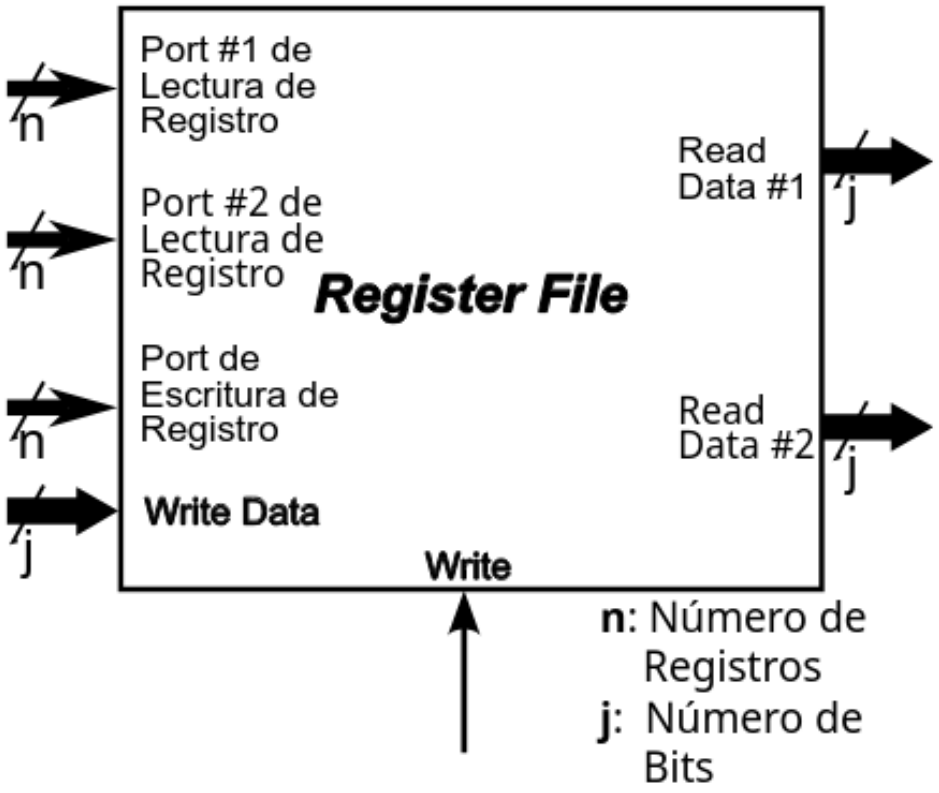

Figura 4: Representación de un registerfile de NxJ (N registros de J bits cada uno)

La lectura de un registro no cambia ningún estado y la salida Q de un FF está disponible. Sólo necesitamos proporcionar un número de registro como entrada y la única salida serán los datos contenidos en ese registro. Como se muestra en la figura de arriba.

Para escribir en un registro, necesitaremos tres entradas: El número de registro adentro del arrelgo, los datos aescribir y un clock que controla la escritura en el registro.

Se usa un Register File con dos puertos de lectura y uno de escritura.

Como se ve en la figura 6, para evitar atascos y agilizar el procesamiento es conveniente tener al menos dos puertos de lectura. Así, pueden leerse los dos operandos de una instrucción a la vez.

Un puerto es una entrada de N bits al que se le envía el número de registro al que se quiere acceder. Luego, el contenido del registro sale por las J líneas Read data 1 o Read data 2 según el puerto seleccionado.

Hay un sólo puerto de escritura porque imaginate el quilombo que se arma si hay dos procesos que quieren escribir a la vez.

En la siguiente imagen hay un Register File esquematizado en detalle.

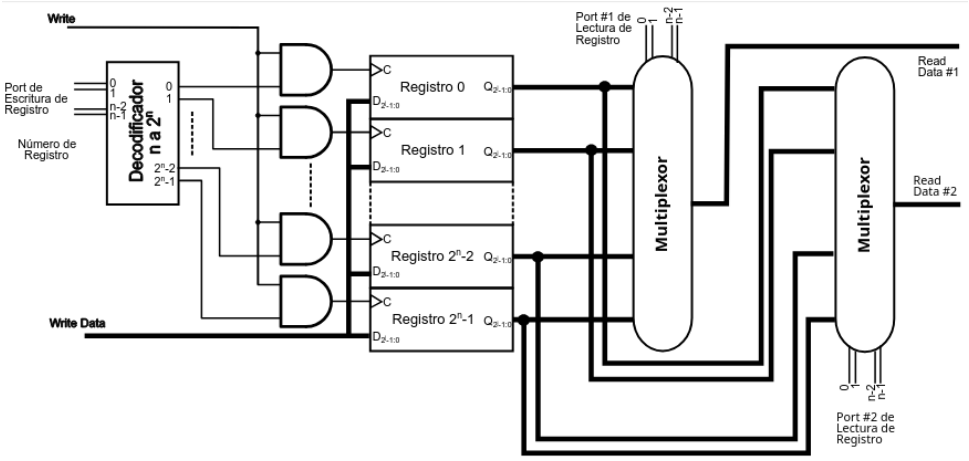

Figura 5: Representación de un registerfile en detalle

3.3. Pipeline

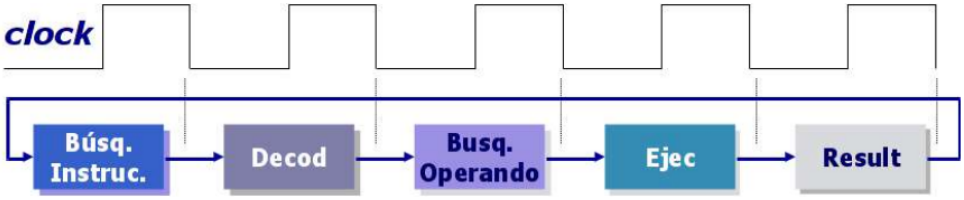

Figura 6: Máquina de estados elemental

Un procesador ejecuta una máquina de estados compuesta por diferentes etapas. En las primeras generaciones de microprocesadores, cada etapa se ejecutaba durante uno o más ciclos de clock, durante los cuales la CPU entera estaba dedicada a ella. Una vez finalizada una etapa, en el siguiente ciclo de clock se pasaba a ejecutar la siguiente. De este modo, ejecutar una instrucción llevaba varios ciclos de clock.

Con los años, surge el pipeline. **El pipeline** permite crear el efecto de superponer en el tiempo la ejecución de varias instrucciones a la vez. Formaliza el concepto de **Instruction Level Paralelism(ILP)** y requiere muy poco o ningún hardware adicional. Solo necesita que los bloques del procesador que resuelven la máquina de estados de ejecución de una instrucción operen de forma simultánea. Esto de logra si **todos los bloques funcionales trabajan en paralelo, pero cada uno en una instrucción diferente.** Cada parte se denomina etapa(stage).

En la figura 7 se ve una representación ideal.

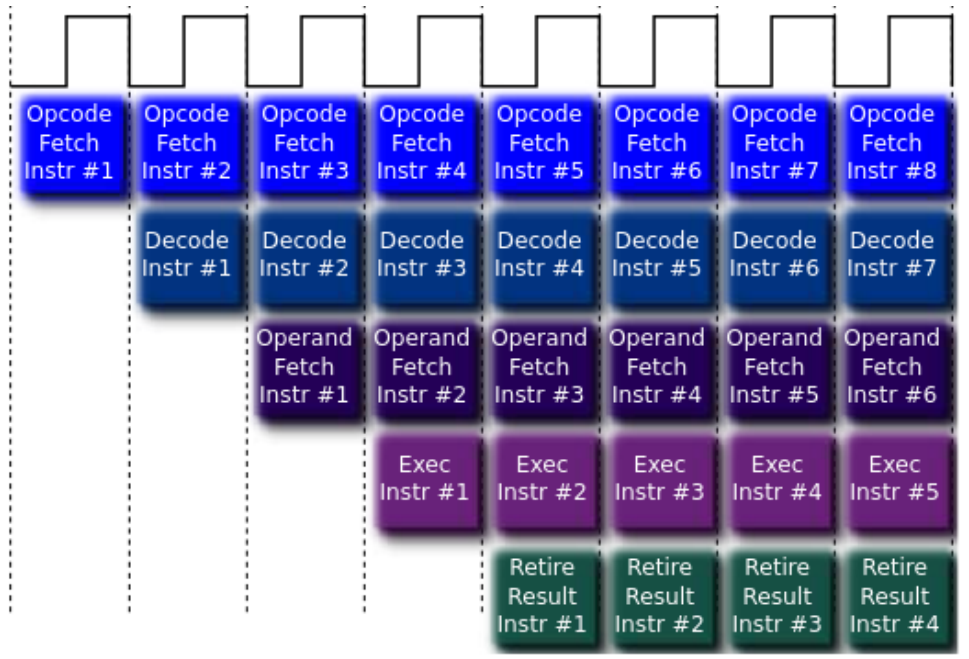

Figura 7: Modelo teórico y representación de una situación ideal

Viendo la figura del modelo ideal, se pueden llegar a conclusione preliminares:

1. El pipeline llega a su **condición de régimen** luego de tantos ciclos de clock como etapas tenga.

2. En un pipeline de 5 etapas, y en el **caso ideal** en que cada etapa consuma solamente un ciclo de clock, una arquitectura pipeline provee un resultado de instrucción por cada ciclo de clock, a partir del ciclo de clock en el cual se llega a resolver la primer instrucción.

3. Este escenario permanente es puramente teórico, en la práctica no se cumple todo el tiempo.

A primera vista, una tendería a pensar que agregar más etapas simples mejoraría la performance, pero antes de agregarlas, deberíamos preguntarnos "¿Por qué? ¿Vale la pena?"

Si sabemos el tiempo de procesamiento interno de una instrucción para una arquitectura no pipeline, al saber las etapas que tiene, si lo pasamos a una estructura pipeline, el tiempo de ejecución se reduciría proporcionalmente a la cantidad de etapas porque ahora estamos procesando las instrucciones en paralelo.

Si llamamos TPI, Time Per Instruction, al tiempo medido entre el resultado de dos instrucciones consecutivas, la siguiente expresión resume todo el análisis.

$$\text{TPI} = \frac{\text{Tiempo por instrucción en la CPU no pipeline}}{\text{Cantidad de etapas}}$$

La realidad es que como la situación ideal es justamente ideal, en la práctica existen **overheads introducidos por el hardware** de implementación del pipeline, que suman **pequeñas demoras** en la interfaz entre cada una de sus etapas. Aunque resultan lo suficientemente pequeñaz como para aceptar calcular el TPI ideal así.

El resultado es que el efecto del pipeline en la práctica se percibe una cantidad de ciclos de clock significativamente menor para ejecutar cada instrucción. **El pipeline no reduce el tiempo de ejecución de cada instrucción individual**, sino que al aplicarse en paralelo al flujo de instrucciones, incrementa el número de instrucciones completadas por unidad de tiempo.

Nota:

Más aún, el overhead del pipeline perjudica el tiempo de ejecución individual, aunque de manera poco significativa al agregar un

$$\delta$$

t a cada instrucción.

El rendimiento (throughput) del procesador mejora notablemente ya que los programas se ejecutan notablemente más rápido.

A pesar de la mejora en el performance que ofrece un pipeline, también tiene sus obstáculos que cospiran contra la eficiencia de este.

1. Obstáculos estructurales.
2. Obstáculos de datos.
3. Obstáculos de control.

Nota:

Al efecto ocasionado por un obstáculo se lo llama **pipeline stall** y su efecto degrada la performance del procesador.

3.3.1. Obstáculos estructurales

Los obstáculos estructurales tienen diversos orígenes, puede ser que **una etapa no esté lo suficientemente atomizada** y para completarse necesita más de un ciclo de clock. Las instrucciones pueden **caer en conflicto de recursos para su ejecución** si dos instrucciones que utilizarán esa etapa están más próximas del tiempo que necesita esta etapa para procesar su parte. Este grupo de instrucciones, entonces, no pueden pasar por esa etapa del pipeline en un ciclo de clock. Y por eso, una de las instrucciones **debe detenerse.** Como consecuencia **el CPI(ciclos por instrucción) se incrementa en 1 o más respecto del valor ideal.**

Ejemplo

Tenemos un procesador que sólo tiene una etapa para acceder a memoria y la comparte para acceso a datos e instrucciones.

En el caso de que se necesite un operando de memoria, el acceso para traer este operando **interferirá con la búsqueda del operando** de una instrucción más adelante del programa.

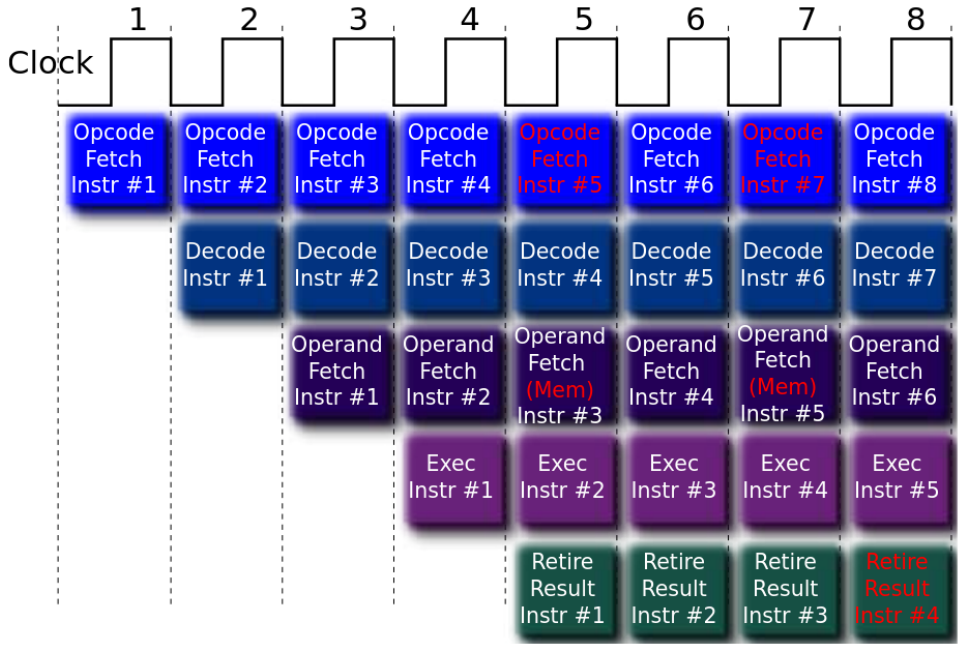

Figura 8: Ejemplo de pipeline de un procesador con datapath compartido

También interferirá con el fetch en la siguiente instrucción. Dicha situación se esquematiza en la figura 8.

Por cada obstáculo, se pospone una operación y entonces nos queda CPI = CPI + 1. En general, la cantidad de CPI que se incrementa es igual a la cantidad de concurrencias menos 1, en el lapso considerado. Para este ejemplo quedaría.

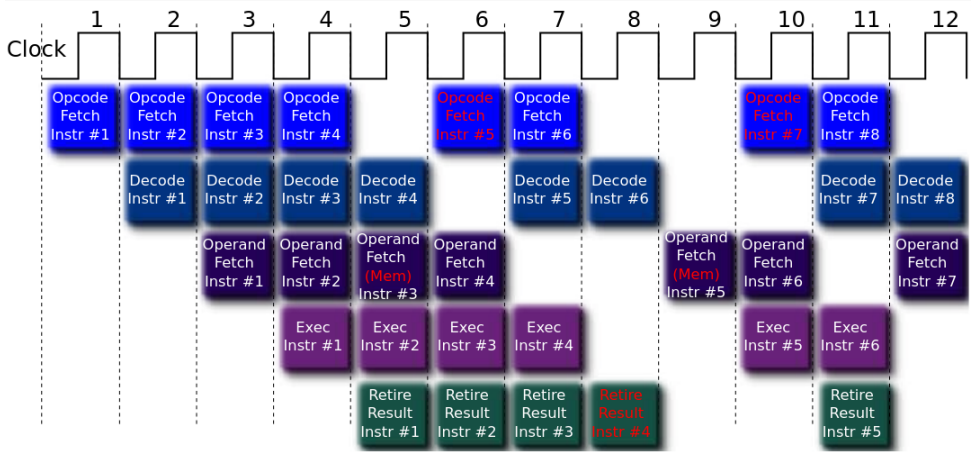

Figura 9: Solución a los obstáculos.

Cualquiera sean las soluciones que deseemos implementar, siempre hay que agregar hardware. Algunas soluciones son:

En el caso de accesos a memoria, se puede resolver mediante las opciones por separado o combinadas.

1. Desdoblamiento del caché L1 en caché de datos y caché de instrucciones.
2. Empleo de buffers de instrucciones implementados como pequeñas colas FIFO.
3. Ensanchamiento de los buses más allá de los anchos de palabra del procesador.

También se puede aumentar la profundidad del pipeline. Esto produce etapas mucho más simples capaces de resolver en un clock su tarea.

3.3.2. Obstáculos de datos

Se producen cuando por **por efecto del pipeline**, una instrucción requiere de un dato antes de que este esté disponible por efecto de la secuencia lógica prevista en el programa.

Consideremos el código que siguiente que corresponde a un procesador ARM.

```
add R1, R2, R3 ; R1 es el destino de esta operación
sub R4, R1, R5 ; R1 es operando pero aún no fue escrito el resultado
and R6, R1, R7
orr R8, R1, R9 ; Se ejecutan finalizada la inst anterior
eor R10, R1, R11
```

Leyendo el código, vemos dependencias para el registro R1. Hasta que la instrucción add no complete su operación, R1 no tiene un valor válido. Por lo tanto, no puede continuar aplicándolo en las restantes que lo utilicen.

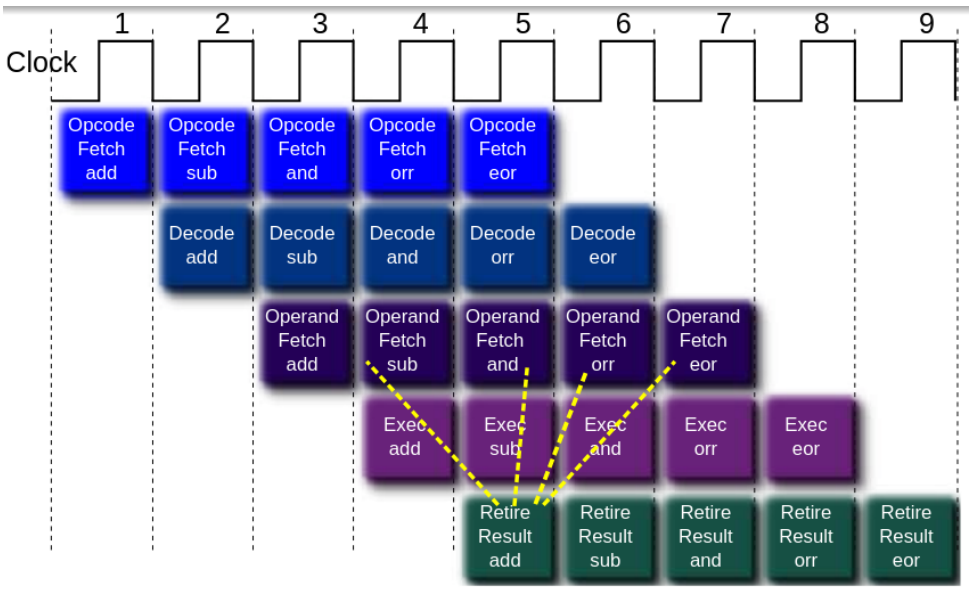

Figura 10: Dependencia de datos

Por cada dependencia, se pospone una operación. Entonces, el CPI aumenta en n siendo n la distancia entre las etapas del pipeline que requieren el mismo dato. CPI = CPI + n.

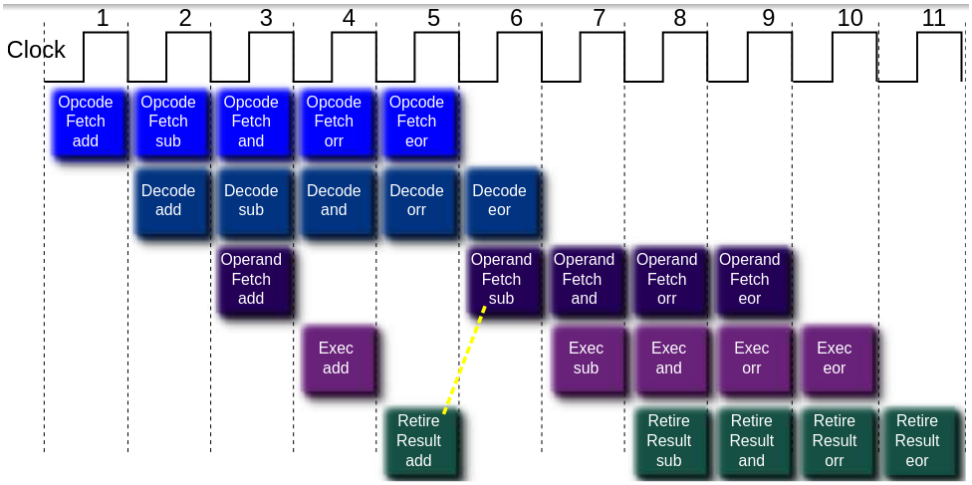

Figura 11: Operaciones pospuestas.

La solución al obstáculo de datos es el **Forwarding**.

Ni bien el resultado está disponible se lo realimenta a la entrada de la Unidad de Ejecución (ALU, FPU, etc).

Así, la etapa que lo requiere lo dispone en el mismo ciclo de reloj en que se escribirá en el operando destino. Esto permite ahorrar el tiempo de escritura en el operando destino (no lo tiene que esperar).

Se aplica solamente a las etapas posteriores que quedarían en estado stall. Aquellas etapas que a pesar de la dependencia de datos no quedarían en estado stall, reciben el resultado cuando este es aplicado en el operando destino.

No resuelve completamente el problema pero reduce el efecto a la mitad.

Figura 12: A wrapped figure going nicely inside the text.

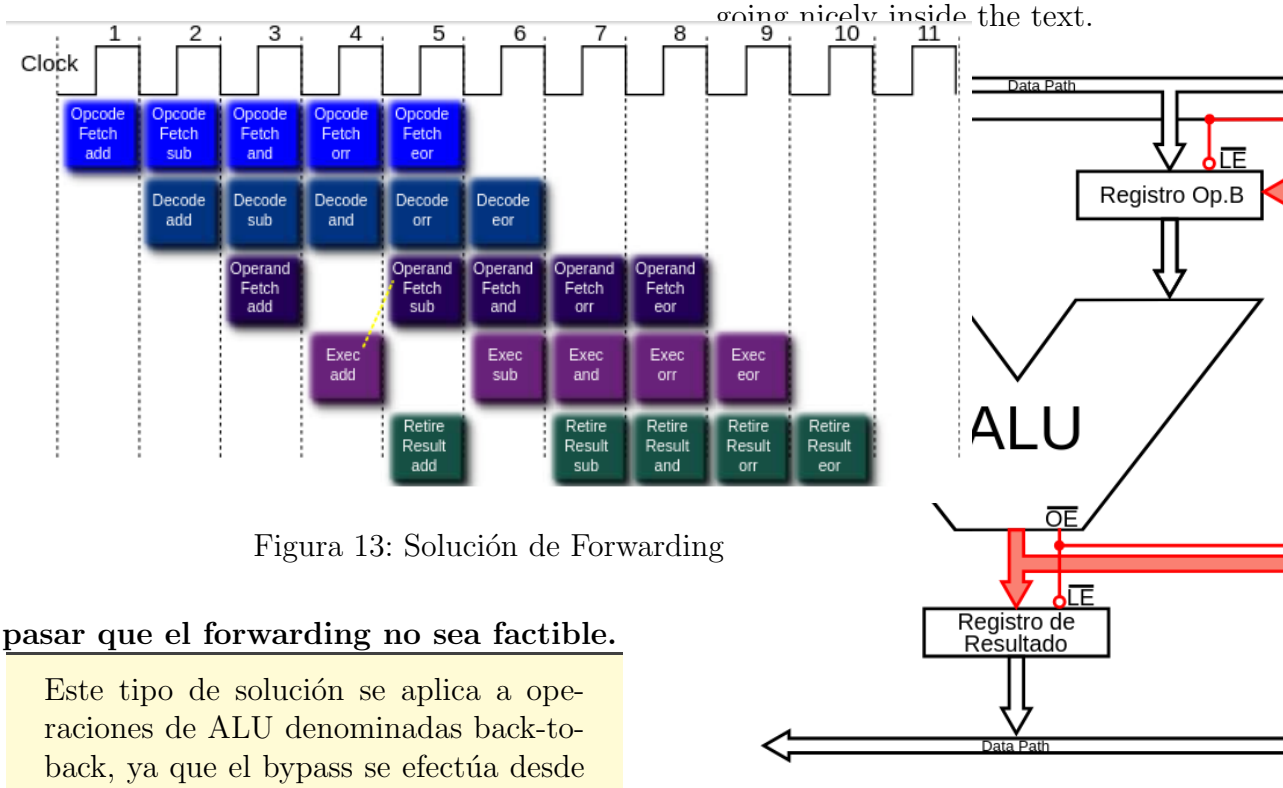

Figura 13: Solución de Forwarding

Puede pasar que el forwarding no sea factible.

Nota: Este tipo de solución se aplica a operaciones de ALU denominadas back-to-back, ya que el bypass se efectúa desde la ALU a un registro entero del procesador.

Consideremos esta variante al código anterior

```
ldr R1, [R2] ; lectura de memoria no pasa por la ALU.
sub R4, R1, R5 ; R1 no puede ser forwardeado al ingresar.
and R6, R1, R7
orr R8, R1, R9
eor R10, R1, R11
```

En este caso, el dato proveniente de memoria es almacenado en elregistro de datos que interface con el bus de datos. Desde este registro es enviado directamente al registro destino, es decir R1. No puede adelantarse el valor a la siguiente etapa hasta que no se complete su escritura en R1.

3.3.3. Obstáculos de control

La máquina de estados de un pipeline busca instrucciones de forma secuencial.

Un branch es una instrucción de control de flujo que provoca una discontinuidad en el flujo de ejecución. Un branch **es la peor** situación en pérdida de performance del pipeline.

El branch hace que todo lo que estaba pre procesado deba descartarse, ya que son las instrucciones sucesoras secuenciales al branch en el programa. Al interrumpirse la secuencia de ejecicón, ya no son necesarias y deben ser reemplazadas por las instrucciones del target del branch.

El pipeline se vacía. Se requieren n-1 ciclos de reloj para obtener el próximo resultado, siendo n la cantidad de etapas del pipeline. Esto se conoce como branch penalty.

En la figura 14 se esquematiza el branch penalty.

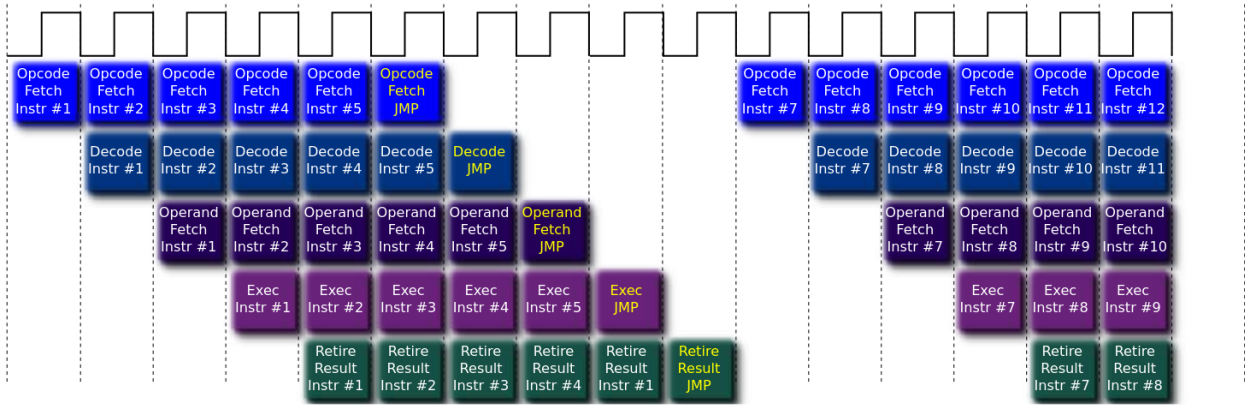

Figura 14: Branch penalty

En las interrupciones la situación es la misma mostrada que en el branch penalty.

En el caso de un branch, el principal incoveniente se tiene cuando éste es condicional ya que es necesario determinar si la condición es True o False.

Si la condición es True, se habla de **branch taken**. Cambia el registro PC a la dirección de salto y limpia el pipeline.

Si el resultado es False, se habla de **branch untaken**. El PC que apunta a la siguiente instrucción al branch(la sucesora secuencial) se mantiene y no se limpia el pipeline.

Esta comprobación requiere llegar hasta la etapa de ejecución del pipeline.

En el caso de un salto incondicional, o llamada a un procedimiento, en la etapa de decodificación ya se tiene la certeza que se producirá el salto, y sólo se pierden menos elmentos del pipeline. Hay un impacto de menor magnitud.

Nota: Para neutralizar los efectos de las ramas, se puede usar forwarding que ayuda a minimizar los efectos de los diferentes casos, pero sólo disminuye algún ciclo de reljo del branch penalty. Aunque su mejora termina siendo insuficiente.

Las soluciones más eficaces son las que tienen que ver con el análisis de los algoritmos empleados y con la frecuencia en la que los compiladores emplean instrucciones de salto. Las

unidades de predicción de saltos pueden ser simples o muy complejas y dependen no sólo de la generación del procesador sino también con el tipo de compilador? que se tome.

3.4. La bendita memoria

El problema de la memoria es el acceso.

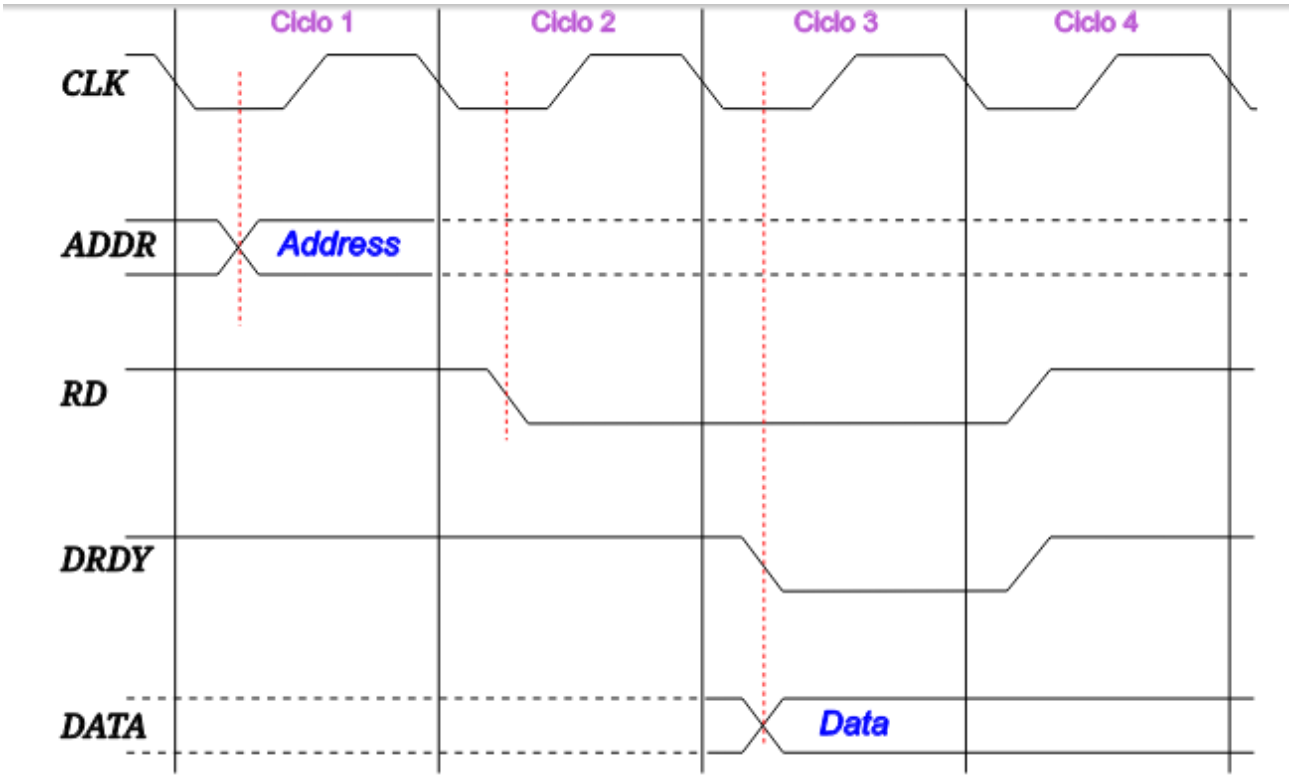

Figura 15: Ciclo de bus de lectura

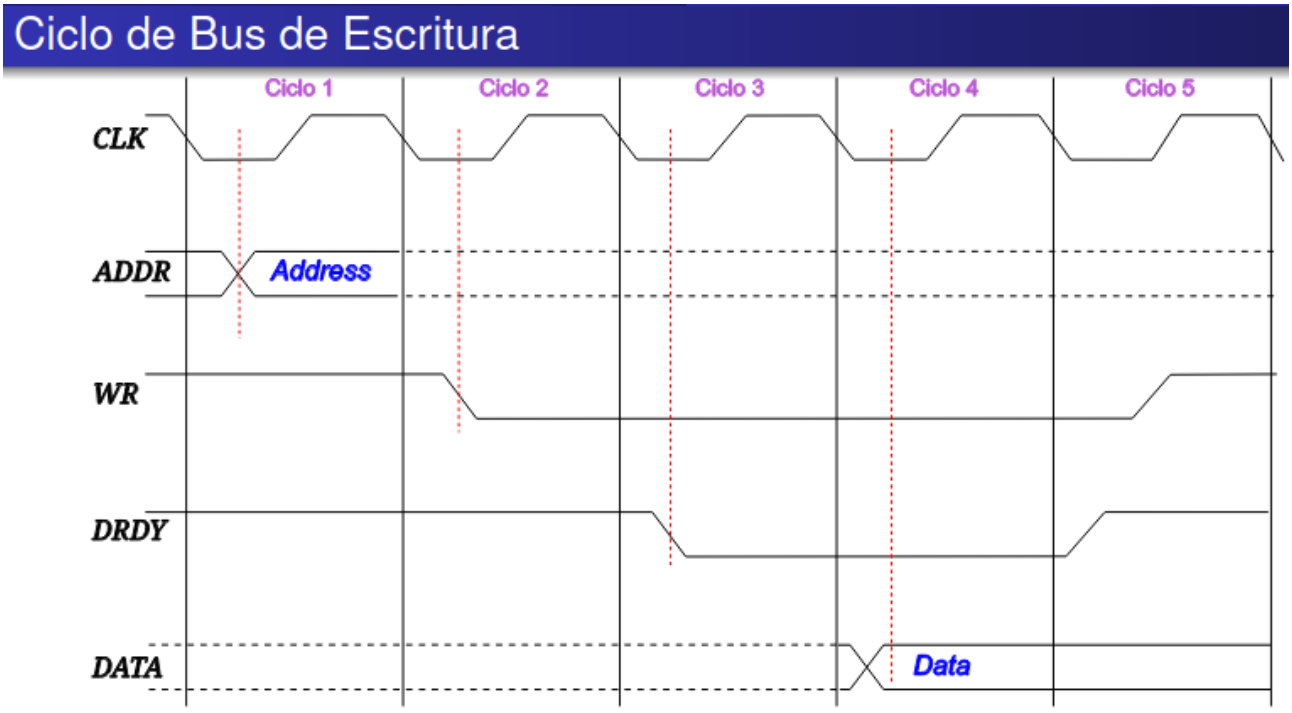

Figura 16: Ciclo de bus de escritura

Podemos ver que es más rápido leer que escribir.

Nota: La eficiencia de los procesador mejoró exponencialmente respecto de la performance de la memoria. Se necesita una estrategia de diseño para al menos minimizar el efecto de este gap.
En 2005 se presentar los chips multicore.