



# Arquitectura de Procesadores

Alejandro Furfaro

Agosto de 2014

# Contenido

## 1 Paralelismo a Nivel de Instrucción

- Pipeline
- Unidades de Predicción de saltos
- Superscalar
- Scheduling Dinámico

## 2 Ejecución Fueras de Orden

- Prehistoria
- Solución Actual
- Especulación por Hardware
- Casos prácticos que mezclan todo lo visto

## 3 Procesadores Multithread

## 4 Multicore y Manycore

# Contenido

## 1 Paralelismo a Nivel de Instrucción

### ● Pipeline

- Unidades de Predicción de saltos
- Superscalar
- Scheduling Dinámico

## 2 Ejecución Fuerza de Orden

- Prehistoria
- Solución Actual
- Especulación por Hardware
- Casos prácticos que mezclan todo lo visto

## 3 Procesadores Multithread

## 4 Multicore y Manycore

# Máquina de estados elemental

- En la década del 40 Von Newman definió un modelo básico de CPU, que a estas alturas está mas que superado.
- Sin embargo algunos conceptos de ese modelo viven en los mas modernos procesadores.
- Uno de ellos es la máquina de ejecución

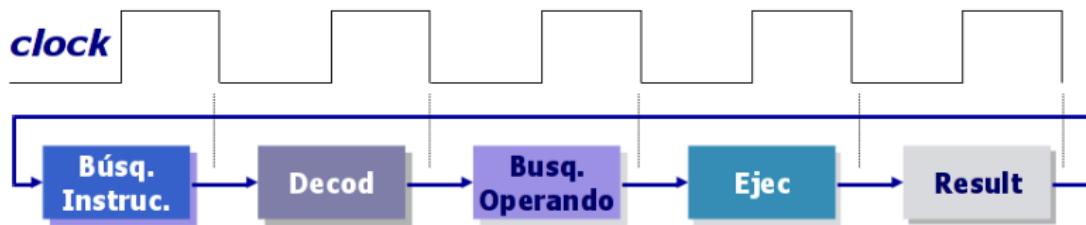


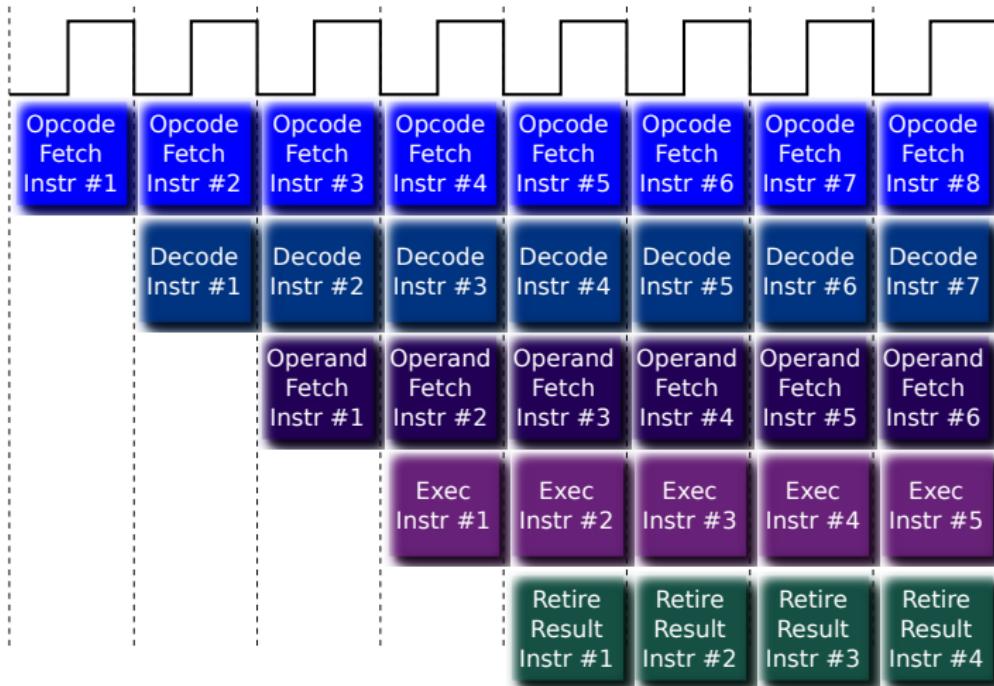
Figura: Etapas mínimas en la ejecución de una instrucción

- En las primeras generaciones de microprocesadores cada etapa de este ciclo se ejecutaba en un ciclo de clock, y la CPU entera estaba dedicada a esa tarea.
- Ejecutar una instrucción insumía de varios ciclos de clock...

# Pipeline

- Arquitectura que permite crear el efecto de superponer en el tiempo, la ejecución de varias instrucciones a la vez.
- Con él se formaliza el concepto de ***Instruction Level Parallelism (ILP)***.
- Requiere muy poco o ningún hardware adicional.
- Solo necesita que los bloques del procesador que resuelven la máquina de estados para la ejecución de una instrucción, operen en forma simultánea.
- Se logra si todos los bloques funcionales trabajan en paralelo pero cada uno en una instrucción diferente.
- Es algo parecido al concepto de una línea de montaje, en donde cada operación se descompone en partes, y se ejecutan en un mismo momento diferentes partes de diferentes operaciones.
- Cada parte se denomina etapa (stage)

# Pipeline

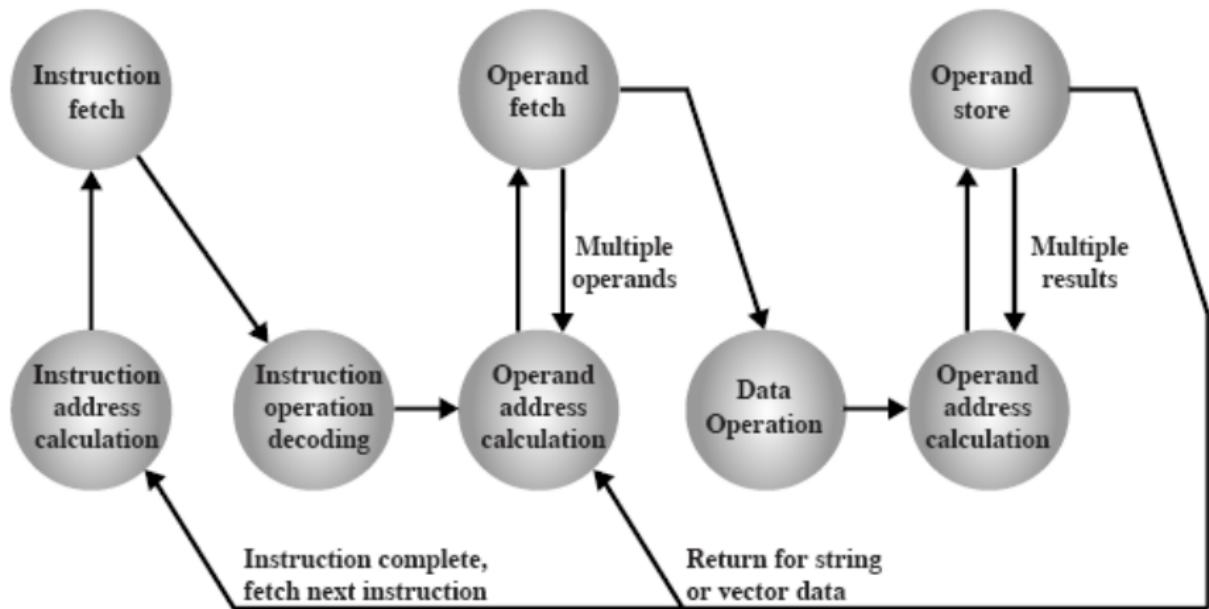


# Pipeline

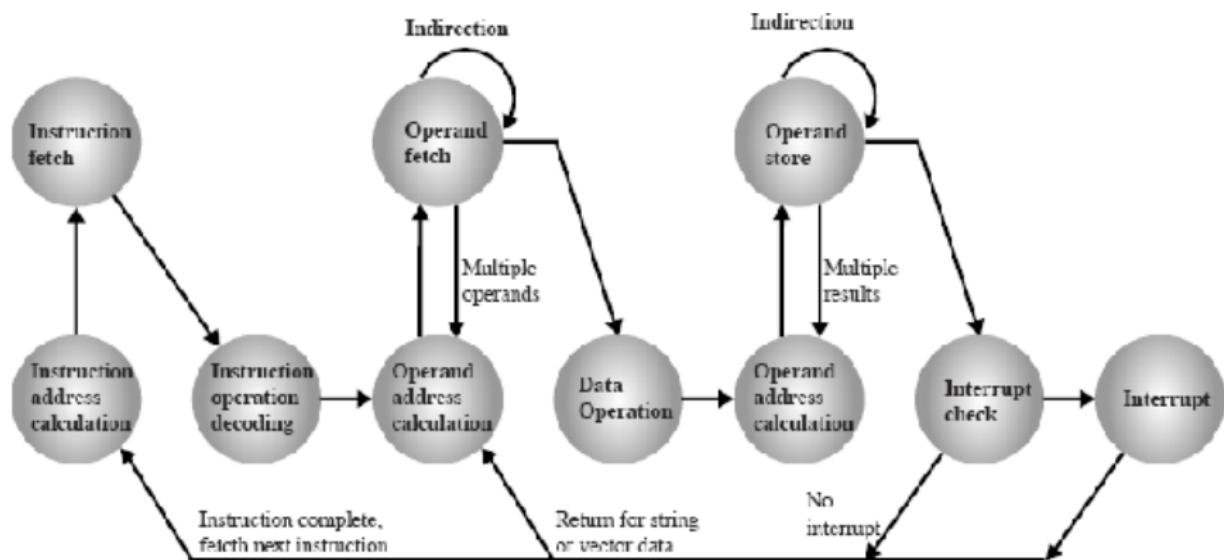
## Resultado

- 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.
- El pipeline tarda en llegar a esa condición de régimen tantos ciclos de clock como etapas tenga.

# Deep Pipeline



# Deeper



# Profundidad de Pipeline. Casos prácticos.

Procesador / uArquitectura	Etapas	Procesador / uArquitectura	Etapas
ARM7TDMI(-S)	3	ARM7EJ-S	5
ARM810	5	ARM9TDMI	5
ARM1020E	6	XScale PXA210/PXA250	7
ARM1136J(F)-S	8	ARM1156T2(F)-S	9
ARM Cortex-A5	8	ARM Cortex-A8	13
AVR32 AP7	7	AVR32 UC3	3
DLX	5	Intel P5 (Pentium)	5
Intel P6 (Pentium Pro)	14	Intel P6 (Pentium III)	10
Intel NetBurst (Willamette)	20	Intel NetBurst (Northwood)	20
Intel NetBurst (Prescott)	31	Intel NetBurst (Cedar Mill)	31
Intel Core	14	Intel Atom	16
LatticeMico32	6	R4000	8
StrongARM SA-110	5	SuperH SH2	5
SuperH SH2A	5	UltraSPARC	9
UltraSPARC T1	6	UltraSPARC T2	8
WinChip	4	LC2200 32 bit	5

# Eficiencia de un pipeline

- ¿Porque querríamos aumentar y aumentar el número de etapas?
- Para un tiempo ya establecido de procesamiento interno de una instrucción para una arquitectura “no pipelinezada”, intuitivamente puede comprenderse que en principio cuantas mas etapas podamos definir para ejecutar esta operación, al ponerlas a trabajar a todas en paralelo en un pipeline, el tiempo de ejecución de la instrucción se reducirá proporcionalmente con la cantidad de etapas.

$$TPI = \frac{\text{Tiempo por instrucción en la CPU "No - Pipeline"}}{\text{Cantidad de etapas}} \quad (1)$$

Donde **TPI** significa Time Per Instruction

# Eficiencia de un pipeline

- ¿Porque queríamos aumentar y aumentar el número de etapas?
- Para un tiempo ya establecido de procesamiento interno de una instrucción para una arquitectura “no pipelinezada”, intuitivamente puede comprenderse que en principio cuantas mas etapas podamos definir para ejecutar esta operación, al ponerlas a trabajar a todas en paralelo en un pipeline, el tiempo de ejecución de la instrucción se reducirá proporcionalmente con la cantidad de etapas.

$$TPI = \frac{\text{Tiempo por instrucción en la CPU "No - Pipeline"}}{\text{Cantidad de etapas}} \quad (1)$$

Donde **TPI** significa Time Per Instruction

# Eficiencia de un pipeline

- ¿Porque queríamos aumentar y aumentar el número de etapas?
- Para un tiempo ya establecido de procesamiento interno de una instrucción para una arquitectura “no pipelinezada“, intuitivamente puede comprenderse que en principio cuantas mas etapas podamos definir para ejecutar esta operación, al ponerlas a trabajar a todas en paralelo en un pipeline, el tiempo de ejecución de la instrucción se reducirá proporcionalmente con la cantidad de etapas.

$$TPI = \frac{\text{Tiempo por instrucción en la CPU "No - Pipeline"}}{\text{Cantidad de etapas}} \quad (1)$$

Donde **TPI** significa Time Per Instruction

# Conclusiones

- Considerar la situación teórica e ideal dada por la ecuación (1) no es 100 % cierto, sin embargo, se aproxima bastante a la situación real.
- En la práctica existen overheads introducidos por el pipeline, que suman pequeñas demoras, pero de todos modos el tiempo se aproxima mucho al ideal.
- El resultado es que la reducción puede apreciarse como si se requiriesen finalmente menos CPI para completar una instrucción.  
¿Porque?

# Conclusiones

- Considerar la situación teórica e ideal dada por la ecuación (1) no es 100 % cierto, sin embargo, se aproxima bastante a la situación real.
- En la práctica existen overheads introducidos por el pipeline, que suman pequeñas demoras, pero de todos modos el tiempo se aproxima mucho al ideal.
- El resultado es que la reducción puede apreciarse como si se requiriesen finalmente menos CPI para completar una instrucción.  
¿Porque?

# Conclusiones

- Considerar la situación teórica e ideal dada por la ecuación (1) no es 100 % cierto, sin embargo, se aproxima bastante a la situación real.
- En la práctica existen overheads introducidos por el pipeline, que suman pequeñas demoras, pero de todos modos el tiempo se aproxima mucho al ideal.
- El resultado es que la reducción puede apreciarse como si se requiriesen finalmente menos CPI para completar una instrucción.  
¿Porque?

# Conclusiones

- Considerar la situación teórica e ideal dada por la ecuación (1) no es 100 % cierto, sin embargo, se aproxima bastante a la situación real.
- En la práctica existen overheads introducidos por el pipeline, que suman pequeñas demoras, pero de todos modos el tiempo se aproxima mucho al ideal.
- El resultado es que la reducción puede apreciarse como si se requiriesen finalmente menos CPI para completar una instrucción.  
¿Porque?

# Conclusiones

- El pipeline no reduce el tiempo de ejecución de cada instrucción individual, sino que al apicarse en paralelo al flujo de instrucciones, incrementa el número de instrucciones completadas por unidad de tiempo.
- De hecho el overhead del pipeline perjudica el tiempo de ejecución individual, de manera poco significativa, pero agrega tiempo a cada instrucción.
- El rendimiento (throughput) del procesador mejora notablemente ya que los programas ejecutan notablemente mas rápido.

# Conclusiones

- El pipeline no reduce el tiempo de ejecución de cada instrucción individual, sino que al apicarse en paralelo al flujo de instrucciones, incrementa el número de instrucciones completadas por unidad de tiempo.
- De hecho el overhead del pipeline perjudica el tiempo de ejecución individual, de manera poco significativa, pero agrega tiempo a cada instrucción.
- El rendimiento (throughput) del procesador mejora notablemente ya que los programas ejecutan notablemente mas rápido.

# Conclusiones

- El pipeline no reduce el tiempo de ejecución de cada instrucción individual, sino que al apicarse en paralelo al flujo de instrucciones, incrementa el número de instrucciones completadas por unidad de tiempo.
- De hecho el overhead del pipeline perjudica el tiempo de ejecución individual, de manera poco significativa, pero agrega tiempo a cada instrucción.
- El rendimiento (throughput) del procesador mejora notablemente ya que los programas ejecutan notablemente mas rápido.

# Hazards (obstáculos)

- Hay obstáculos que conspiran contra la eficiencia de un pipeline.
- Podemos agruparlos en tres categorías:



- En cualquier caso al efecto ocasionado por un obstáculo se lo denomina *pipeline stall*.
- Su efecto degrada la performance del procesador

# Hazards (obstáculos)

- Hay obstáculos que conspiran contra la eficiencia de un pipeline.
- Podemos agruparlos en tres categorías:

## • Obstáculos estructurales.

- Los más comunes son los **hazards**.
- En cualquier caso al efecto ocasionado por un obstáculo se lo denomina ***pipeline stall***.
- Su efecto degrada la performance del procesador

# Hazards (obstáculos)

- Hay obstáculos que conspiran contra la eficiencia de un pipeline.
- Podemos agruparlos en tres categorías:

① *Obstáculos estructurales.*

② *Obstáculos de datos.*

③ *Obstáculos de control.*

- En cualquier caso al efecto ocasionado por un obstáculo se lo denomina *pipeline stall*.
- Su efecto degrada la performance del procesador

# Hazards (obstáculos)

- Hay obstáculos que conspiran contra la eficiencia de un pipeline.
- Podemos agruparlos en tres categorías:

① ***Obstáculos estructurales.***

② ***Obstáculos de datos.***

③ ***Obstáculos de control.***

- En cualquier caso al efecto ocasionado por un obstáculo se lo denomina ***pipeline stall.***
- Su efecto degrada la performance del procesador

# Hazards (obstáculos)

- Hay obstáculos que conspiran contra la eficiencia de un pipeline.
- Podemos agruparlos en tres categorías:

① ***Obstáculos estructurales.***

② ***Obstáculos de datos.***

③ ***Obstáculos de control.***

- En cualquier caso al efecto ocasionado por un obstáculo se lo denomina ***pipeline stall.***
- Su efecto degrada la performance del procesador

# Hazards (obstáculos)

- Hay obstáculos que conspiran contra la eficiencia de un pipeline.
- Podemos agruparlos en tres categorías:

① ***Obstáculos estructurales.***

② ***Obstáculos de datos.***

③ ***Obstáculos de control.***

- En cualquier caso al efecto ocasionado por un obstáculo se lo denomina ***pipeline stall.***
- Su efecto degrada la performance del procesador

# Hazards (obstáculos)

- Hay obstáculos que conspiran contra la eficiencia de un pipeline.
- Podemos agruparlos en tres categorías:
  - ① *Obstáculos estructurales.*
  - ② *Obstáculos de datos.*
  - ③ *Obstáculos de control.*
- En cualquier caso al efecto ocasionado por un obstáculo se lo denomina ***pipeline stall***.
- Su efecto degrada la performance del procesador

# Hazards (obstáculos)

- Hay obstáculos que conspiran contra la eficiencia de un pipeline.
- Podemos agruparlos en tres categorías:
  - ① *Obstáculos estructurales.*
  - ② *Obstáculos de datos.*
  - ③ *Obstáculos de control.*
- En cualquier caso al efecto ocasionado por un obstáculo se lo denomina ***pipeline stall***.
- Su efecto degrada la performance del procesador

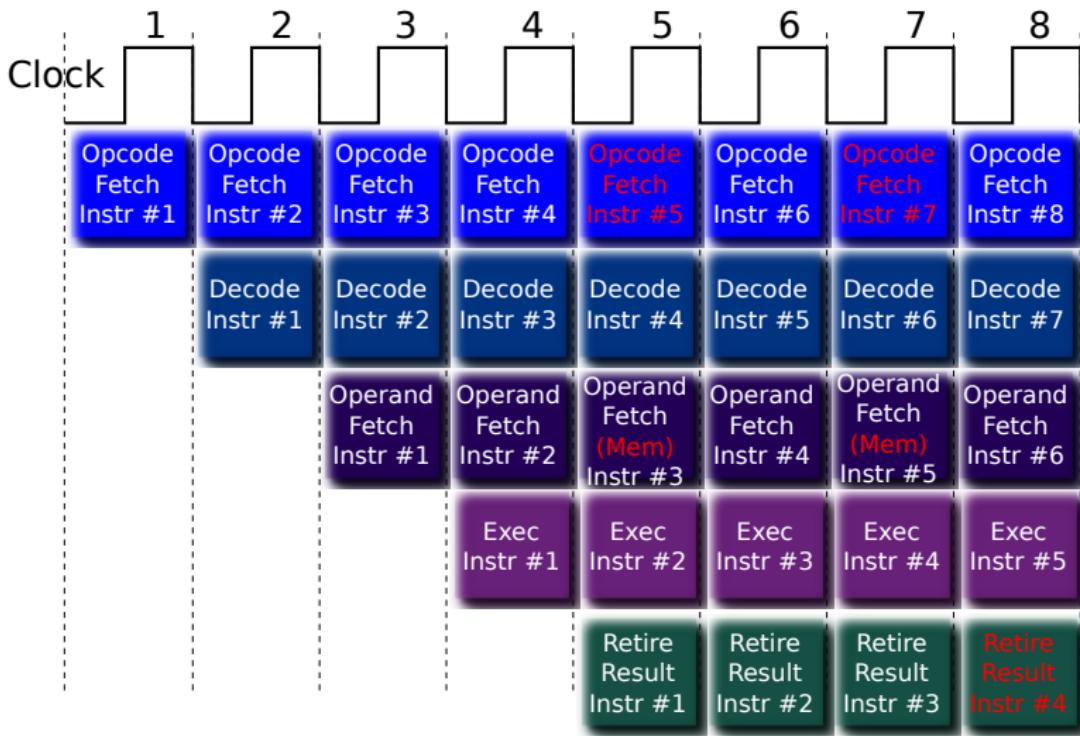
# Obstáculos Estructurales

- Se puede dar, si una etapa no está suficientemente atomizada, y concentra aún demasiadas funciones lo que hace que para completarse requiere mas de un ciclo de clock.
- Si dos instrucciones que utilizarán esta etapa están mas próximas del tiempo que necesita esta etapa paraprocesar su parte, caerán en conflicto de recursos para su ejecución.
- Este grupo de instrucciones entonces, no puede pasar por esa etapa del pipeline en un ciclo de clock.
- En estas circunstancias una de las instrucciones debe detenerse. Como consecuencia CPI se incrementa en 1 o mas respecto del valor ideal.

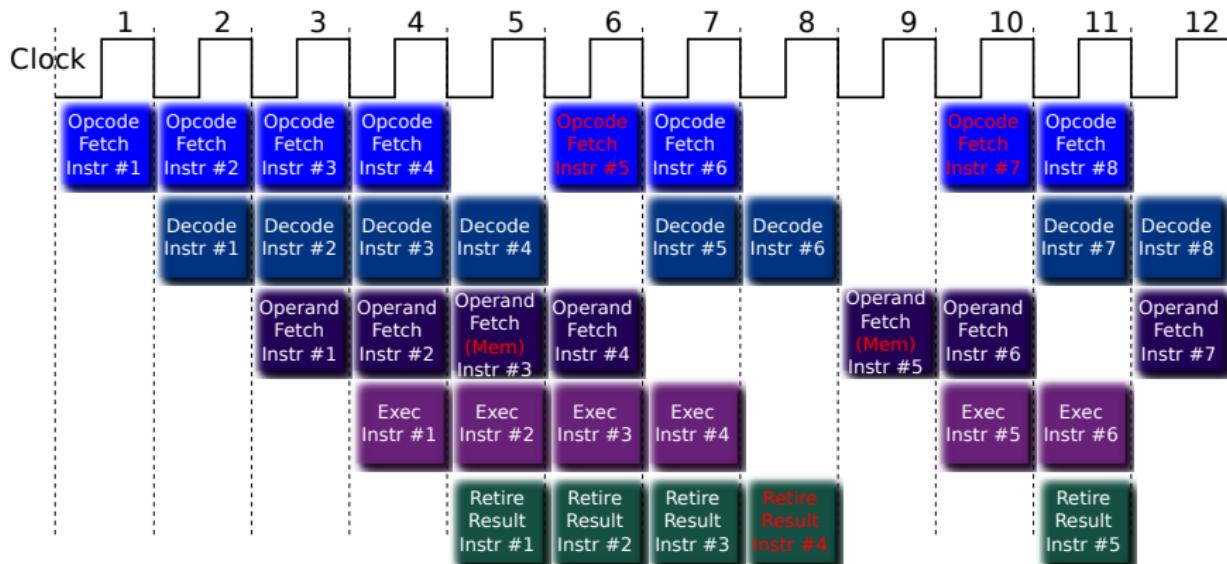
# Obstáculos Estructurales: Ejemplo

- Un procesador solo 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 mas adelante del programa.
- También interferirá con el Fetch de la siguiente instrucción

# Obstáculos Estructurales - Accesos concurrentes a memoria



# Obstáculos Estructurales - Efecto en CPI



- Por cada Obstáculo, pospone una operación.  $CPI = CPI + 1$
- En general la cantidad de CPI que se incrementan es igual a la cantidad de concurrencias menos 1, en el lapso considerado

# Obstáculos Estructurales - Posibles soluciones

- Cualesquiera sean las soluciones que deseemos implementar, es necesario agregar hardware.
- En el caso de accesos a memoria, se puede resolver mediante las siguientes opciones por separado o combinadas:

• Redimensionar la memoria para que sea más grande y tener más bloques de memoria.

• Utilizar una memoria cache que almacene los datos más recientes y que responda más rápidamente a las solicitudes de memoria.

• Utilizar una memoria virtual que permita acceder a memoria que no existe físicamente en el sistema.

- Aumentar la profundidad del pipeline, produce etapas mucho mas simples capaces de resolver en un clock su tarea.

# Obstáculos Estructurales - Posibles soluciones

- Cualesquiera sean las soluciones que deseemos implementar, es necesario agregar hardware.
- En el caso de accesos a memoria, se puede resolver mediante las siguientes opciones por separado o combinadas:

○ Desdoblamiento del cache L1 en Cache de datos y Cache de instrucciones.

○ Utilizar una memoria cache coherente, donde se resuelva la coherencia entre los diferentes procesadores.

○ Utilizar un sistema de memoria cache consistente, donde se resuelva la coherencia entre los diferentes procesadores.

- Aumentar la profundidad del pipeline, produce etapas mucho mas simples capaces de resolver en un clock su tarea.

# Obstáculos Estructurales - Posibles soluciones

- Cualesquiera sean las soluciones que deseemos implementar, es necesario agregar hardware.
- En el caso de accesos a memoria, se puede resolver mediante las siguientes opciones por separado o combinadas:
  - ① Desdoblamiento del cache L1 en Cache de datos y Cache de instrucciones.
  - ② Empleo de buffers de instrucciones implementados como pequeñas colas FIFO.
  - ③ Ensanchamiento de los buses mas allá de los anchos de palabra del procesador.
- Aumentar la profundidad del pipeline, produce etapas mucho mas simples capaces de resolver en un clock su tarea.

# Obstáculos Estructurales - Posibles soluciones

- Cualesquiera sean las soluciones que deseemos implementar, es necesario agregar hardware.
- En el caso de accesos a memoria, se puede resolver mediante las siguientes opciones por separado o combinadas:
  - 1 Desdoblamiento del cache L1 en Cache de datos y Cache de instrucciones.
  - 2 Empleo de buffers de instrucciones implementados como pequeñas colas FIFO.
  - 3 Ensanchamiento de los buses mas allá de los anchos de palabra del procesador.
- Aumentar la profundidad del pipeline, produce etapas mucho mas simples capaces de resolver en un clock su tarea.

# Obstáculos Estructurales - Posibles soluciones

- Cualesquiera sean las soluciones que deseemos implementar, es necesario agregar hardware.
- En el caso de accesos a memoria, se puede resolver mediante las siguientes opciones por separado o combinadas:
  - ① Desdoblamiento del cache L1 en Cache de datos y Cache de instrucciones.
  - ② Empleo de buffers de instrucciones implementados como pequeñas colas FIFO.
  - ③ Ensanchamiento de los buses mas allá de los anchos de palabra del procesador.
- Aumentar la profundidad del pipeline, produce etapas mucho mas simples capaces de resolver en un clock su tarea.

# Obstáculos Estructurales - Posibles soluciones

- Cualesquiera sean las soluciones que deseemos implementar, es necesario agregar hardware.
- En el caso de accesos a memoria, se puede resolver mediante las siguientes opciones por separado o combinadas:
  - ① Desdoblamiento del cache L1 en Cache de datos y Cache de instrucciones.
  - ② Empleo de buffers de instrucciones implementados como pequeñas colas FIFO.
  - ③ Ensanchamiento de los buses mas allá de los anchos de palabra del procesador.
- Aumentar la profundidad del pipeline, produce etapas mucho mas simples capaces de resolver en un clock su tarea.

# Obstáculos Estructurales - Posibles soluciones

- Cualesquiera sean las soluciones que deseemos implementar, es necesario agregar hardware.
- En el caso de accesos a memoria, se puede resolver mediante las siguientes opciones por separado o combinadas:
  - ① Desdoblamiento del cache L1 en Cache de datos y Cache de instrucciones.
  - ② Empleo de buffers de instrucciones implementados como pequeñas colas FIFO.
  - ③ Ensanchamiento de los buses mas allá de los anchos de palabra del procesador.
- Aumentar la profundidad del pipeline, produce etapas mucho mas simples capaces de resolver en un clock su tarea.

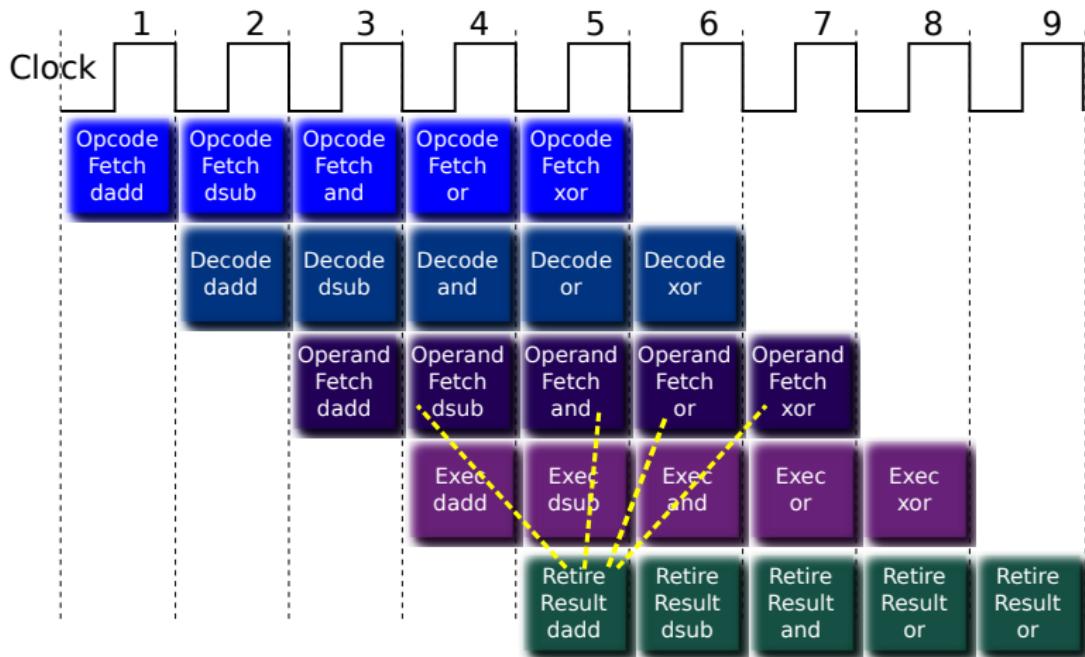
# Obstáculos de datos

- Se producen cuando 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 siguiente código para un procesador MIPS.

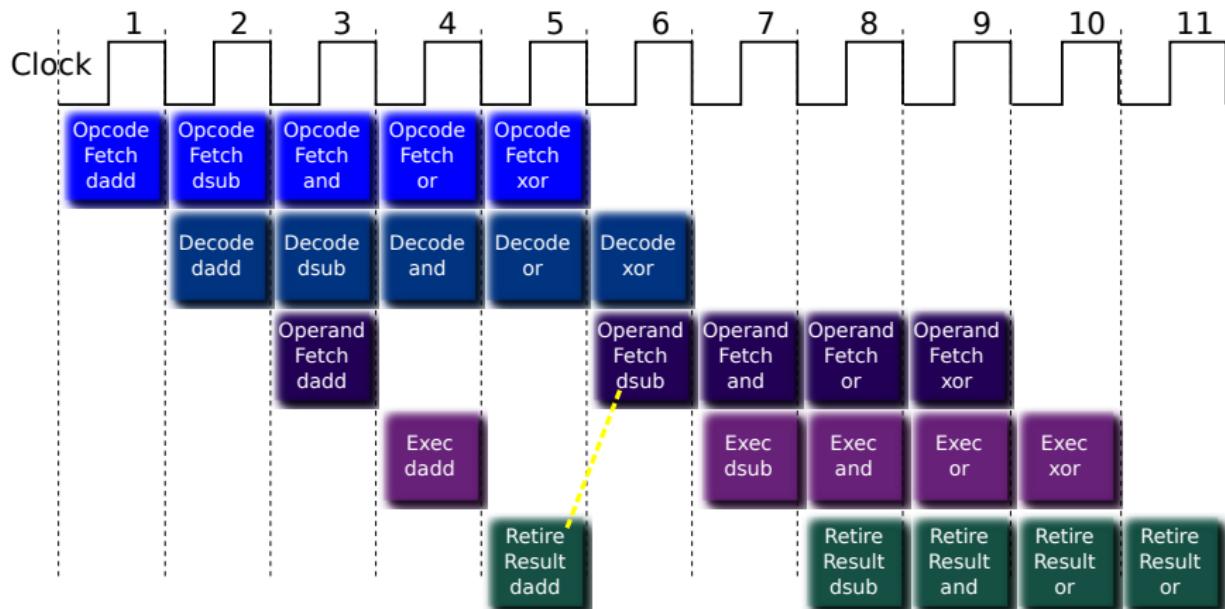
```
1  dadd  R1, R2, R3
2  dsub  R4, R1, R5
3  and   R6, R1, R7
4  or    R8, R1, R9
5  xor   R10, R1, R11
```

- Tenemos dependencias para el Registro R1. Hasta que la instrucción dadd no complete su operación, R1 no tiene un valor válido. Por lo tanto no puede continuar aplicándolo en las restantes que lo utilizan.
- La situación en el pipeline es la siguiente:

# Obstáculos de Datos - Conflicto de recursos



# Obstáculos de Datos - Efecto en CPI



- Por cada Dependencia, pospone una operación.  $CPI = CPI + n$ , siendo  $n$  la distancia entre las etapas del pipeline que requieren el mismo dato.

# Solución a Obstáculos de Datos - Forwarding

- Se extrae el resultado directamente de la salida de la unidad de Ejecución (ALU, Floating Point, o la que corresponda a la instrucción), cuando está disponible y se lo envía a la entrada de la etapa que lo requiere en el mismo ciclo de clock en que se escribe en el operando destino. Esto permite disponer del dato en la siguiente instrucción ahorrando en la espera el tiempo de escritura en el operando destino, sin que ésta deba esperar que se retire el resultado (es decir que se aplique en el operando destino).
- Se aplica solamente a las etapas posteriores que quedarían en estado stall.
- A aquellas etapas que no quedarían en estado stall como consecuencia de la dependencia de datos, se les envía la salida del resultado cuando este es aplicado en el operando destino.

# Solución a Obstáculos de Datos - Forwarding

- Se extrae el resultado directamente de la salida de la unidad de Ejecución (ALU, Floating Point, o la que corresponda a la instrucción), cuando está disponible y se lo envía a la entrada de la etapa que lo requiere en el mismo ciclo de clock en que se escribe en el operando destino. Esto permite disponer del dato en la siguiente instrucción ahorrando en la espera el tiempo de escritura en el operando destino, sin que ésta deba esperar que se retire el resultado (es decir que se aplique en el operando destino).
- Se aplica solamente a las etapas posteriores que quedarían en estado stall.
- A aquellas etapas que no quedarían en estado stall como consecuencia de la dependencia de datos, se les envía la salida del resultado cuando este es aplicado en el operando destino.

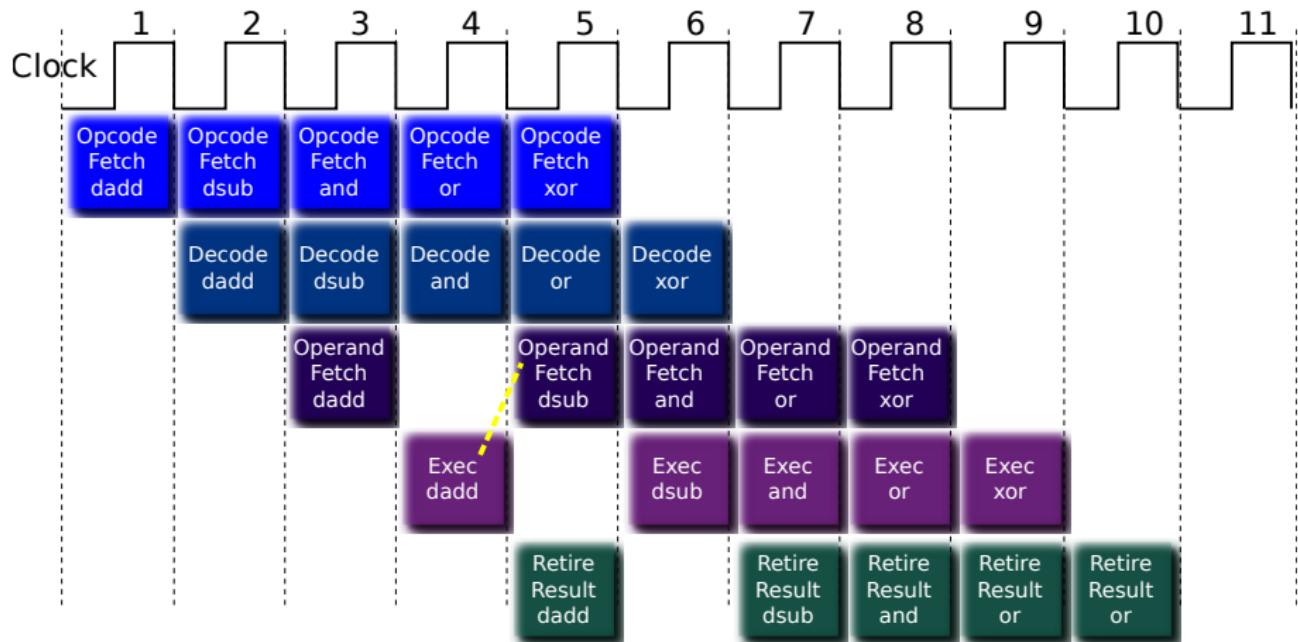
# Solución a Obstáculos de Datos - Forwarding

- Se extrae el resultado directamente de la salida de la unidad de Ejecución (ALU, Floating Point, o la que corresponda a la instrucción), cuando está disponible y se lo envía a la entrada de la etapa que lo requiere en el mismo ciclo de clock en que se escribe en el operando destino. Esto permite disponer del dato en la siguiente instrucción ahorrando en la espera el tiempo de escritura en el operando destino, sin que ésta deba esperar que se retire el resultado (es decir que se aplique en el operando destino).
- Se aplica solamente a las etapas posteriores que quedarían en estado stall.
- A aquellas etapas que no quedarían en estado stall como consecuencia de la dependencia de datos, se les envía la salida del resultado cuando este es aplicado en el operando destino.

# Solución a Obstáculos de Datos - Forwarding

- Se extrae el resultado directamente de la salida de la unidad de Ejecución (ALU, Floating Point, o la que corresponda a la instrucción), cuando está disponible y se lo envía a la entrada de la etapa que lo requiere en el mismo ciclo de clock en que se escribe en el operando destino. Esto permite disponer del dato en la siguiente instrucción ahorrando en la espera el tiempo de escritura en el operando destino, sin que ésta deba esperar que se retire el resultado (es decir que se aplique en el operando destino).
- Se aplica solamente a las etapas posteriores que quedarían en estado stall.
- A aquellas etapas que no quedarían en estado stall como consecuencia de la dependencia de datos, se les envía la salida del resultado cuando este es aplicado en el operando destino.

# Obstáculos de Datos - Forwarding



# Algunas veces forwarding no es factible

- Normalmente forwarding 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 ahora este otro código para un procesador MIPS.

```
1  ld      R1, 0(R2)
2  dsub   R4,  R1,  R5
3  and    R6,  R1,  R7
4  or     R8,  R1,  R9
5  xor    R10, R1,  R11
```

- En este caso el dato no puede adelantarse hasta que no se complete el ciclo de clock en el que se impacta el resultado dentro del procesador.

# Algunas veces forwarding no es factible

- Normalmente forwarding 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 ahora este otro código para un procesador MIPS.

```
1  ld      R1, 0(R2)
2  dsub   R4,  R1,  R5
3  and    R6,  R1,  R7
4  or     R8,  R1,  R9
5  xor    R10, R1,  R11
```

- En este caso el dato no puede adelantarse hasta que no se complete el ciclo de clock en el que se impacta el resultado dentro del procesador.

# Algunas veces forwarding no es factible

- Normalmente forwarding 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 ahora este otro código para un procesador MIPS.

```
1  ld      R1, 0(R2)
2  dsub   R4,  R1,  R5
3  and    R6,  R1,  R7
4  or     R8,  R1,  R9
5  xor    R10, R1,  R11
```

- En este caso el dato no puede adelantarse hasta que no se complete el ciclo de clock en el que se impacta el resultado dentro del procesador.

# Algunas veces forwarding no es factible

- Normalmente forwarding 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 ahora este otro código para un procesador MIPS.

```
1  ld      R1, 0(R2)
2  dsub   R4,  R1,  R5
3  and    R6,  R1,  R7
4  or     R8,  R1,  R9
5  xor     R10, R1, R11
```

- En este caso el dato no puede adelantarse hasta que no se complete el ciclo de clock en el que se impacta el resultado dentro del procesador.

# Obstáculos de Control

- Un branch es la peor situación en pérdida de performance.
- Un branch es una discontinuidad en el flujo de ejecución.
- El pipeline busca instrucciones en secuencia.
- El branch hace que todo lo que estaba pre procesado deba descartarse. Y el pipeline se vacía debiendo transcurrir  $n - 1$  ciclos de clock hasta el próximo resultado. Siendo  $n$  la cantidad de etapas del pipeline.
- Esto se conoce como **branch penalty**.

# Obstáculos de Control

- Un branch es la peor situación en pérdida de performance.
- Un branch es una discontinuidad en el flujo de ejecución.
- El pipeline busca instrucciones en secuencia.
- El branch hace que todo lo que estaba pre procesado deba descartarse. Y el pipeline se vacía debiendo transcurrir  $n - 1$  ciclos de clock hasta el próximo resultado. Siendo  $n$  la cantidad de etapas del pipeline.
- Esto se conoce como **branch penalty**.

# Obstáculos de Control

- Un branch es la peor situación en pérdida de performance.
- Un branch es una discontinuidad en el flujo de ejecución.
- El pipeline busca instrucciones en secuencia.
- El branch hace que todo lo que estaba pre procesado deba descartarse. Y el pipeline se vacía debiendo transcurrir  $n - 1$  ciclos de clock hasta el próximo resultado. Siendo  $n$  la cantidad de etapas del pipeline.
- Esto se conoce como **branch penalty**.

# Obstáculos de Control

- Un branch es la peor situación en pérdida de performance.
- Un branch es una discontinuidad en el flujo de ejecución.
- El pipeline busca instrucciones en secuencia.
- El branch hace que todo lo que estaba pre procesado deba descartarse. Y el pipeline se vacía debiendo transcurrir  $n - 1$  ciclos de clock hasta el próximo resultado. Siendo  $n$  la cantidad de etapas del pipeline.
- Esto se conoce como **branch penalty**.

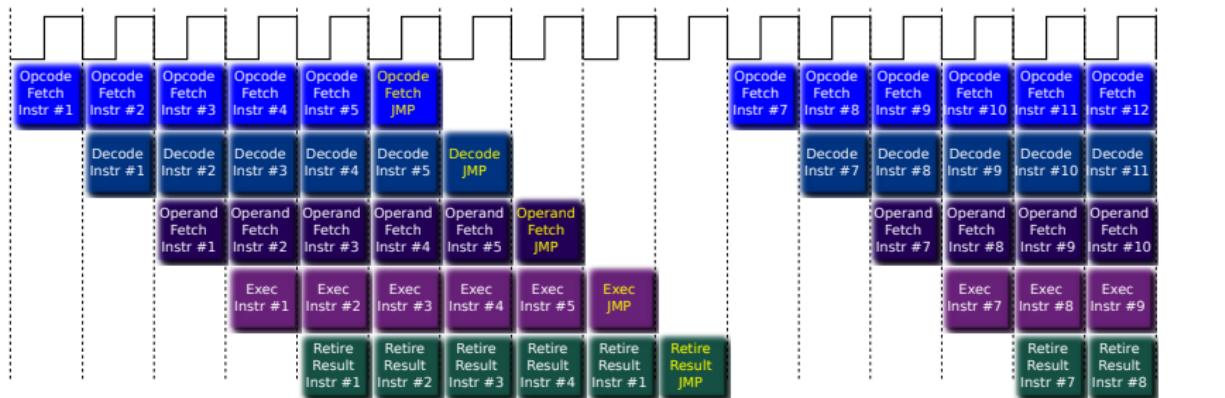
# Obstáculos de Control

- Un branch es la peor situación en pérdida de performance.
- Un branch es una discontinuidad en el flujo de ejecución.
- El pipeline busca instrucciones en secuencia.
- El branch hace que todo lo que estaba pre procesado deba descartarse. Y el pipeline se vacía debiendo transcurrir  **$n - 1$**  ciclos de clock hasta el próximo resultado. Siendo  **$n$**  la cantidad de etapas del pipeline.
- Esto se conoce como **branch penalty**.

# Obstáculos de Control

- Un branch es la peor situación en pérdida de performance.
- Un branch es una discontinuidad en el flujo de ejecución.
- El pipeline busca instrucciones en secuencia.
- El branch hace que todo lo que estaba pre procesado deba descartarse. Y el pipeline se vacía debiendo transcurrir  **$n - 1$**  ciclos de clock hasta el próximo resultado. Siendo  **$n$**  la cantidad de etapas del pipeline.
- Esto se conoce como **branch penalty**.

# “La conspiración de los branches”



# saltos, branches, interrupciones, llamadas...

- En las interrupciones la situación es la del gráfico del slide anterior.
- El principal inconveniente se tiene cuando el salto 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 en el segundo de ***branch untaken*** (PC apunta a la instrucción siguiente al branch).

# saltos, branches, interrupciones, llamadas...

- En las interrupciones la situación es la del gráfico del slide anterior.
- El principal inconveniente se tiene cuando el salto 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 en el segundo de ***branch untaken*** (PC apunta a la instrucción siguiente al branch).

# saltos, branches, interrupciones, llamadas...

- En las interrupciones la situación es la del gráfico del slide anterior.
- El principal inconveniente se tiene cuando el salto 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 en el segundo de *branch untaken* (PC apunta a la instrucción siguiente al branch).

# saltos, branches, interrupciones, llamadas...

- En las interrupciones la situación es la del gráfico del slide anterior.
- El principal inconveniente se tiene cuando el salto 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 en el segundo de **branch untaken** (PC apunta a la instrucción siguiente al branch).

# Saltos: ¿Cuando puede determinarse si es taken?

En el esquema de pipeline clásico que venimos analizando, esta condición se verifica:

- ① en la fase de ejecución, si es un salto condicional.
- ② en la fase de búsqueda de operando si es un salto incondicional o llamada a subrutina, con direccionamiento indirecto (es decir la dirección de salto está en la memoria en la dirección que contiene la instrucción).
- ③ o, en el mejor de los casos en la fase de decodificación si es un salto incondicional o llamada a subrutina con direccionamiento directo (es decir la dirección de salto viene a continuación del código de operación).

# Saltos: ¿Cuando puede determinarse si es taken?

En el esquema de pipeline clásico que venimos analizando, esta condición se verifica:

- ① en la fase de ejecución, si es un salto condicional.
- ② en la fase de búsqueda de operando si es un salto incondicional o llamada a subrutina, con direccionamiento indirecto (es decir la dirección de salto está en la memoria en la dirección que contiene la instrucción).
- ③ o, en el mejor de los casos en la fase de decodificación si es un salto incondicional o llamada a subrutina con direccionamiento directo (es decir la dirección de salto viene a continuación del código de operación).

# Saltos: ¿Cuando puede determinarse si es taken?

En el esquema de pipeline clásico que venimos analizando, esta condición se verifica:

- ① en la fase de ejecución, si es un salto condicional.
- ② en la fase de búsqueda de operando si es un salto incondicional o llamada a subrutina, con direccionamiento indirecto (es decir la dirección de salto está en la memoria en la dirección que contiene la instrucción).
- ③ o, en el mejor de los casos en la fase de decodificación si es un salto incondicional o llamada a subrutina con direccionamiento directo (es decir la dirección de salto viene a continuación del código de operación).

# Saltos: ¿Cuando puede determinarse si es taken?

En el esquema de pipeline clásico que venimos analizando, esta condición se verifica:

- ① en la fase de ejecución, si es un salto condicional.
- ② en la fase de búsqueda de operando si es un salto incondicional o llamada a subrutina, con direccionamiento indirecto (es decir la dirección de salto está en la memoria en la dirección que contiene la instrucción).
- ③ o, en el mejor de los casos en la fase de decodificación si es un salto incondicional o llamada a subrutina con direccionamiento directo (es decir la dirección de salto viene a continuación del código de operación).

# Efectos de los branches y como neutralizarlos

- Forwarding puede ayudar a disminuir el efecto de los diferentes casos expuestos anteriormente. Sin embargo, no es óptima, ya que solo lograremos disminuir algunos ciclos de clock del **branch penalty**, pero no se puede eliminar del todo su efecto.
- Para soluciones mas eficientes es necesario recurrir a análisis mas pormenorizados, que en general tienen en cuenta el comportamiento de los algoritmos y de los saltos.
- Ingresamos al universo de las unidades de predicción de saltos. Las hay desde muy simples a muy sofisticadas, y tiene que ver no solo con las diferentes generaciones de procesadores, sino también con el tipo de procesador bajo análisis.

# Efectos de los branches y como neutralizarlos

- Forwarding puede ayudar a disminuir el efecto de los diferentes casos expuestos anteriormente. Sin embargo, no es óptima, ya que solo lograremos disminuir algunos ciclos de clock del **branch penalty**, pero no se puede eliminar del todo su efecto.
- Para soluciones mas eficientes es necesario recurrir a análisis mas pormenorizados, que en general tienen en cuenta el comportamiento de los algoritmos y de los saltos.
- Ingresamos al universo de las unidades de predicción de saltos. Las hay desde muy simples a muy sofisticadas, y tiene que ver no solo con las diferentes generaciones de procesadores, sino también con el tipo de procesador bajo análisis.

# Efectos de los branches y como neutralizarlos

- Forwarding puede ayudar a disminuir el efecto de los diferentes casos expuestos anteriormente. Sin embargo, no es óptima, ya que solo lograremos disminuir algunos ciclos de clock del **branch penalty**, pero no se puede eliminar del todo su efecto.
- Para soluciones mas eficientes es necesario recurrir a análisis mas pormenorizados, que en general tienen en cuenta el comportamiento de los algoritmos y de los saltos.
- Ingresamos al universo de las unidades de predicción de saltos. Las hay desde muy simples a muy sofisticadas, y tiene que ver no solo con las diferentes generaciones de procesadores, sino también con el tipo de procesador bajo análisis.

# Efectos de los branches y como neutralizarlos

- Forwarding puede ayudar a disminuir el efecto de los diferentes casos expuestos anteriormente. Sin embargo, no es óptima, ya que solo lograremos disminuir algunos ciclos de clock del **branch penalty**, pero no se puede eliminar del todo su efecto.
- Para soluciones mas eficientes es necesario recurrir a análisis mas pormenorizados, que en general tienen en cuenta el comportamiento de los algoritmos y de los saltos.
- Ingresamos al universo de las unidades de predicción de saltos. Las hay desde muy simples a muy sofisticadas, y tiene que ver no solo con las diferentes generaciones de procesadores, sino también con el tipo de procesador bajo análisis.

# Contenido

## 1 Paralelismo a Nivel de Instrucción

- Pipeline
- Unidades de Predicción de saltos
- Superscalar
- Scheduling Dinámico

## 2 Ejecución Fueras de Orden

- Prehistoria
- Solución Actual
- Especulación por Hardware
- Casos prácticos que mezclan todo lo visto

## 3 Procesadores Multithread

## 4 Multicore y Manycore

# Necesidad de la predicción de saltos

## Profundidad de un pipeline vs. penalización de un branch

A medida que aumenta la complejidad de un procesador, aumentan las etapas de un pipeline, ya que muchos diseños consideran paralelizar todas las tareas posibles que un procesador debe realizar para completar la ejecución de una interrupción.

Esto hace que la cantidad de tiempo que demora un pipeline en recuperarse del efecto de un branch sea directamente proporcional a su cantidad de etapas.

Es crucial lograr una correcta predicción de saltos para evitar o minimizar las penalizaciones producto de vaciar el pipeline.

# Necesidad de la predicción de saltos

## Profundidad de un pipeline vs. penalización de un branch

A medida que aumenta la complejidad de un procesador, aumentan las etapas de un pipeline, ya que muchos diseños consideran paralelizar todas las tareas posibles que un procesador debe realizar para completar la ejecución de una interrupción.

Esto hace que la cantidad de tiempo que demora un pipeline en recuperarse del efecto de un branch sea directamente proporcional a su cantidad de etapas.

Es crucial lograr una correcta predicción de saltos para evitar o minimizar las penalizaciones producto de vaciar el pipeline.

# Necesidad de la predicción de saltos

## Profundidad de un pipeline vs. penalización de un branch

A medida que aumenta la complejidad de un procesador, aumentan las etapas de un pipeline, ya que muchos diseños consideran paralelizar todas las tareas posibles que un procesador debe realizar para completar la ejecución de una interrupción.

Esto hace que la cantidad de tiempo que demora un pipeline en recuperarse del efecto de un branch sea directamente proporcional a su cantidad de etapas.

Es crucial lograr una correcta predicción de saltos para evitar o minimizar las penalizaciones producto de vaciar el pipeline.

# predicted-non-taken

- El procesador asume por default que el salto nunca se toma, es decir que continúa la búsqueda del código de operación de las instrucciones siguientes a la de salto como si el salto fuese en realidad una instrucción común y corriente.
- Funciona adecuadamente en estructuras de programa como la siguiente:  
instrucción *branch*  
instrucción sucesora secuencial.  
instrucción destino para *branch taken*.
- Es decir, cuando el salto es “hacia adelante”, o bien cuando la dirección del target es mayor que la de la dirección de memoria que contiene la instrucción de salto.

# predicted-taken

- El procesador asume por default que el salto se toma siempre, es decir que continúa la búsqueda del código de operación de las instrucciones a partir de la dirección target.
- Funciona adecuadamente en estructuras de programa como la siguiente:  
*instrucción destino para branch taken.*  
grupo de instrucciones a ejecutar iterativamente.  
*instrucción branch*
- Resulta de gran utilidad en casos de estructuras de iteración.
- La gran ventaja es que siempre se acierta. Solo falla cuando expira la condición del lazo.

# ¿Que criterio tomar?

- Cual de los dos criterios adoptar, depende del diseño del set de instrucciones.
- El compilador, conociendo el criterio asumido por el procesador, debe organizar el código de la manera mas adecuada para aprovechar el criterio elegido por el diseñador de Hardware.

# delayed branch

- En los primeros modelos de procesadores RISC se introdujo un enfoque superador denominado salto demorado (delayed branch).
- Volviendo a la estructura de programa como la siguiente:  
instrucción *branch*  
instrucción sucesora secuencial.  
instrucción destino para *branch taken*.
- La *instrucción sucesora secuencial* se envía al slot de salto demorado.
- Se ejecuta si o si independientemente del resultado de la evaluación del branch.
- Se aplica o no el resultado dependiendo del resultado de la evaluación de la condición. En este caso no genera demoras. En caso de branch-taken, se descarta la ejecución y se tiene un ciclo de clock para que salga el resultado de la instrucción destino.

# Loop Unrolling en el compilador

- Los compiladores son hasta ahora los responsables de armar los loops de modo de usar las instrucciones en función del branch Prediction.
- Otra posibilidad es desenrollar los branches, técnica conocida como loop unrolling.
- Para implementarla es necesario, que los datos dentro del loop sean paralelizables. Esto es por ejemplo el siguiente código:

```
1   for ( i = 0 ; i < 256 ; i++ )  
2   {  
3       suma = 0.0f ; /* */  
4       for ( j = 0 ; (j <= i && j < 256) ; j++)  
5           suma += v0[i-j] * v1[j];  
6       fAux[ i ] = suma;  
7   }
```

- La ventaja de deshacer el loop y hacer una instrucción detrás de la otra es que se eliminan los branches que componen cualquier loop y desaparece la penalización.

# Predicción de saltos dinámica

- Todos los métodos vistos hasta aquí dependen exclusivamente del compilador. Esto significa que el hardware interno del procesador no realiza ningún análisis del código ni agrega adaptatividad.
- Cuando el procesador comienza a efectuar un análisis del flujo de instrucciones y toma decisiones en función de lo que encuentra se tiene un paso adelante en la predicción de saltos.
- Ingresamos al universo de la predicción dinámica.
- Los métodos subsiguientes son de esta categoría y corresponden a microarquitecturas mas avanzadas, con mayor paralelismo a nivel de instrucciones.

# Predicción de saltos dinámica

- Todos los métodos vistos hasta aquí dependen exclusivamente del compilador. Esto significa que el hardware interno del procesador no realiza ningún análisis del código ni agrega adaptatividad.
- Cuando el procesador comienza a efectuar un análisis del flujo de instrucciones y toma decisiones en función de lo que encuentra se tiene un paso adelante en la predicción de saltos.
- Ingresamos al universo de la predicción dinámica.
- Los métodos subsiguientes son de esta categoría y corresponden a microarquitecturas mas avanzadas, con mayor paralelismo a nivel de instrucciones.

# Predicción de saltos dinámica

- Todos los métodos vistos hasta aquí dependen exclusivamente del compilador. Esto significa que el hardware interno del procesador no realiza ningún análisis del código ni agrega adaptatividad.
- Cuando el procesador comienza a efectuar un análisis del flujo de instrucciones y toma decisiones en función de lo que encuentra se tiene un paso adelante en la predicción de saltos.
- Ingresamos al universo de la predicción dinámica.
- Los métodos subsiguientes son de esta categoría y corresponden a microarquitecturas mas avanzadas, con mayor paralelismo a nivel de instrucciones.

# Predicción de saltos dinámica

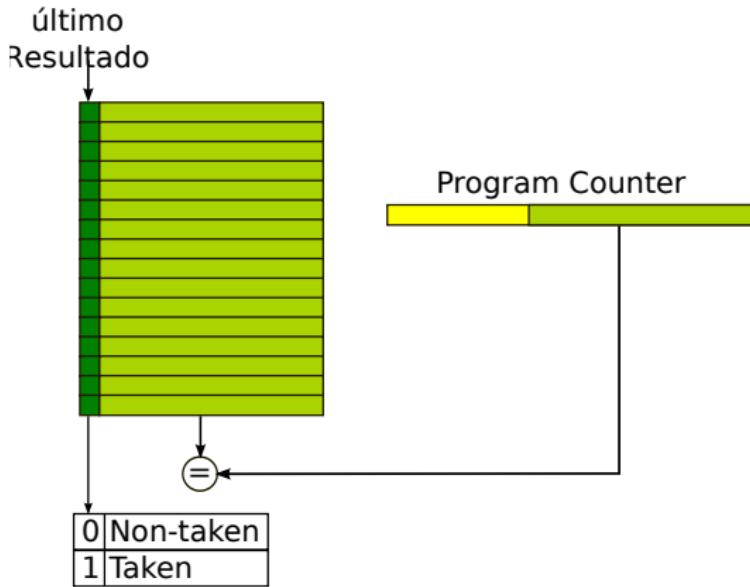
- Todos los métodos vistos hasta aquí dependen exclusivamente del compilador. Esto significa que el hardware interno del procesador no realiza ningún análisis del código ni agrega adaptatividad.
- Cuando el procesador comienza a efectuar un análisis del flujo de instrucciones y toma decisiones en función de lo que encuentra se tiene un paso adelante en la predicción de saltos.
- Ingresamos al universo de la predicción dinámica.
- Los métodos subsiguientes son de esta categoría y corresponden a microarquitecturas mas avanzadas, con mayor paralelismo a nivel de instrucciones.

# Predicción de saltos dinámica

- Todos los métodos vistos hasta aquí dependen exclusivamente del compilador. Esto significa que el hardware interno del procesador no realiza ningún análisis del código ni agrega adaptatividad.
- Cuando el procesador comienza a efectuar un análisis del flujo de instrucciones y toma decisiones en función de lo que encuentra se tiene un paso adelante en la predicción de saltos.
- Ingresamos al universo de la predicción dinámica.
- Los métodos subsiguientes son de esta categoría y corresponden a microarquitecturas mas avanzadas, con mayor paralelismo a nivel de instrucciones.

# Branch Prediction Buffer

- Es una tabla simple indexada por la dirección de memoria de la instrucción de salto (o su campo de bits menos significativos), y un bit que indica simplemente el resultado reciente del salto (*taken* o *non-taken*)



# Branch Prediction Buffer

- En realidad el valor del bit es solo una pista para la Unidad de Ejecución.
- No reviste seguridad en la predicción.
- Hasta incluso puede tratarse de una etiqueta ingresada por otra instrucción de salto cuya dirección finaliza con el mismo patrón de bits que la actual.
- Si el resultado del salto coincide con el valor almacenado en el Branch Prediction Buffer, se mantiene el contenido, y si falla se complementa a 1.
- Este modelo se llama predicción simple de 1 bit. Limitación: cuando un salto siempre resulta *taken*, y falla una vez produce dos predicciones fallidas seguidas, ya que el bit se invierte.
- Para mejorar su rendimiento se implementó un modelo de 2 bits, el cual es mas eficiente en su predicción ya que cumple un diagrama de estados como el siguiente.

# Branch Prediction Buffer

- En realidad el valor del bit es solo una pista para la Unidad de Ejecución.
- No reviste seguridad en la predicción.
- Hasta incluso puede tratarse de una etiqueta ingresada por otra instrucción de salto cuya dirección finaliza con el mismo patrón de bits que la actual.
- Si el resultado del salto coincide con el valor almacenado en el Branch Prediction Buffer, se mantiene el contenido, y si falla se complementa a 1.
- Este modelo se llama predicción simple de 1 bit. Limitación: cuando un salto siempre resulta *taken*, y falla una vez produce dos predicciones fallidas seguidas, ya que el bit se invierte.
- Para mejorar su rendimiento se implementó un modelo de 2 bits, el cual es mas eficiente en su predicción ya que cumple un diagrama de estados como el siguiente.

# Branch Prediction Buffer

- En realidad el valor del bit es solo una pista para la Unidad de Ejecución.
- **No reviste seguridad en la predicción.**
- Hasta incluso puede tratarse de una etiqueta ingresada por otra instrucción de salto cuya dirección finaliza con el mismo patrón de bits que la actual.
- Si el resultado del salto coincide con el valor almacenado en el Branch Prediction Buffer, se mantiene el contenido, y si falla se complementa a 1.
- Este modelo se llama predicción simple de 1 bit. Limitación: cuando un salto siempre resulta *taken*, y falla una vez produce dos predicciones fallidas seguidas, ya que el bit se invierte.
- Para mejorar su rendimiento se implementó un modelo de 2 bits, el cual es mas eficiente en su predicción ya que cumple un diagrama de estados como el siguiente.

# Branch Prediction Buffer

- En realidad el valor del bit es solo una pista para la Unidad de Ejecución.
- No reviste seguridad en la predicción.
- Hasta incluso puede tratarse de una etiqueta ingresada por otra instrucción de salto cuya dirección finaliza con el mismo patrón de bits que la actual.
- Si el resultado del salto coincide con el valor almacenado en el Branch Prediction Buffer, se mantiene el contenido, y si falla se complementa a 1.
- Este modelo se llama predicción simple de 1 bit. Limitación: cuando un salto siempre resulta *taken*, y falla una vez produce dos predicciones fallidas seguidas, ya que el bit se invierte.
- Para mejorar su rendimiento se implementó un modelo de 2 bits, el cual es mas eficiente en su predicción ya que cumple un diagrama de estados como el siguiente.

# Branch Prediction Buffer

- En realidad el valor del bit es solo una pista para la Unidad de Ejecución.
- No reviste seguridad en la predicción.
- Hasta incluso puede tratarse de una etiqueta ingresada por otra instrucción de salto cuya dirección finaliza con el mismo patrón de bits que la actual.
- Si el resultado del salto coincide con el valor almacenado en el Branch Prediction Buffer, se mantiene el contenido, y si falla se complementa a 1.
- Este modelo se llama predicción simple de 1 bit. Limitación: cuando un salto siempre resulta *taken*, y falla una vez produce dos predicciones fallidas seguidas, ya que el bit se invierte.
- Para mejorar su rendimiento se implementó un modelo de 2 bits, el cual es mas eficiente en su predicción ya que cumple un diagrama de estados como el siguiente.

# Branch Prediction Buffer

- En realidad el valor del bit es solo una pista para la Unidad de Ejecución.
- No reviste seguridad en la predicción.
- Hasta incluso puede tratarse de una etiqueta ingresada por otra instrucción de salto cuya dirección finaliza con el mismo patrón de bits que la actual.
- Si el resultado del salto coincide con el valor almacenado en el Branch Prediction Buffer, se mantiene el contenido, y si falla se complementa a 1.
- Este modelo se llama predicción simple de 1 bit. Limitación: cuando un salto siempre resulta *taken*, y falla una vez produce dos predicciones fallidas seguidas, ya que el bit se invierte.
- Para mejorar su rendimiento se implementó un modelo de 2 bits, el cual es mas eficiente en su predicción ya que cumple un diagrama de estados como el siguiente.

# Branch Prediction Buffer

- En realidad el valor del bit es solo una pista para la Unidad de Ejecución.
- No reviste seguridad en la predicción.
- Hasta incluso puede tratarse de una etiqueta ingresada por otra instrucción de salto cuya dirección finaliza con el mismo patrón de bits que la actual.
- Si el resultado del salto coincide con el valor almacenado en el Branch Prediction Buffer, se mantiene el contenido, y si falla se complementa a 1.
- Este modelo se llama predicción simple de 1 bit. Limitación: cuando un salto siempre resulta *taken*, y falla una vez produce dos predicciones fallidas seguidas, ya que el bit se invierte.
- Para mejorar su rendimiento se implementó un modelo de 2 bits, el cual es mas eficiente en su predicción ya que cumple un diagrama de estados como el siguiente.

# Predicción de 2 bits



# Implementación práctica

- En la práctica, este tipo de Branch Predictor, se implementa en la etapa de Instruction Fetch del pipeline, como un pequeño cache de direcciones de salto accesible mediante las direcciones de las instrucciones.
- Otra forma de implementación es agregar un par de bits a cada bloque de líneas en el cache de instrucciones, que se emplean únicamente si ese bloque tiene instrucciones de salto condicional.

En la práctica, el Branch Predictor predice el destino del salto y el Program Counter se actualiza con la dirección destino del salto y su contenido buscando instrucciones a partir de allí.

Si el resultado es correcto, se actualizan las tablas de predicción.

# Implementación práctica

- En la práctica, este tipo de Branch Predictor, se implementa en la etapa de Instruction Fetch del pipeline, como un pequeño cache de direcciones de salto accesible mediante las direcciones de las instrucciones.
- Otra forma de implementación es agregar un par de bits a cada bloque de líneas en el cache de instrucciones, que se emplean únicamente si ese bloque tiene instrucciones de salto condicional.
  - En el caso en que la predicción para la instrucción de salto de ese bloque sea *taken*, el Program Counter se setea con la dirección destino del salto y se continúa buscando instrucciones a partir de allí.
  - Si la predicción es *not taken*, se continúa buscando instrucciones a partir de la dirección original.

# Implementación práctica

- En la práctica, este tipo de Branch Predictor, se implementa en la etapa de Instruction Fetch del pipeline, como un pequeño cache de direcciones de salto accesible mediante las direcciones de las instrucciones.
- Otra forma de implementación es agregar un par de bits a cada bloque de líneas en el cache de instrucciones, que se emplean únicamente si ese bloque tiene instrucciones de salto condicional.
  - En el caso en que la predicción para la instrucción de salto de ese bloque sea *taken*, el Program Counter se setea con la dirección destino del salto y se continúa buscando instrucciones a partir de allí.
  - En otro caso se sigue buscando las instrucciones en secuencia.

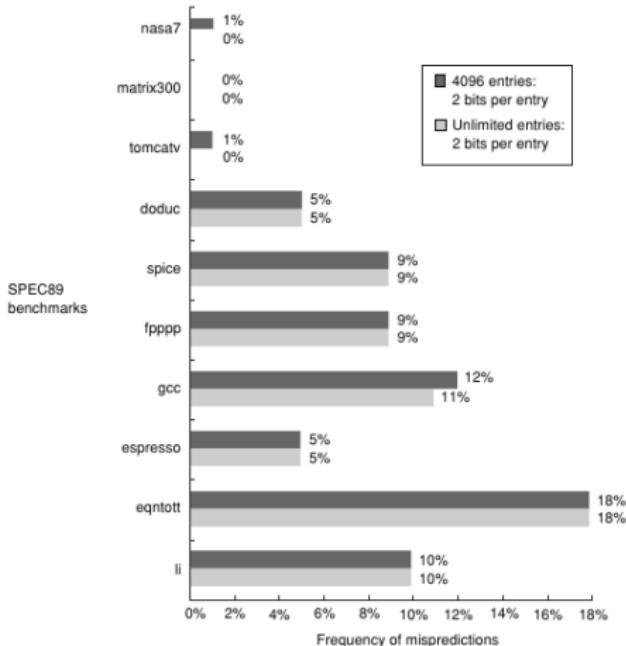
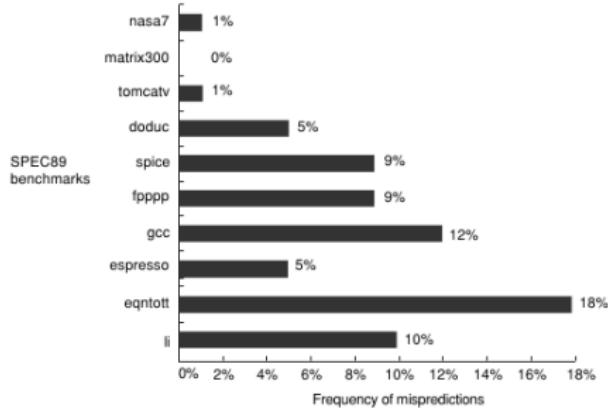
# Implementación práctica

- En la práctica, este tipo de Branch Predictor, se implementa en la etapa de Instruction Fetch del pipeline, como un pequeño cache de direcciones de salto accesible mediante las direcciones de las instrucciones.
- Otra forma de implementación es agregar un par de bits a cada bloque de líneas en el cache de instrucciones, que se emplean únicamente si ese bloque tiene instrucciones de salto condicional.
  - En el caso en que la predicción para la instrucción de salto de ese bloque sea *taken*, el Program Counter se setea con la dirección destino del salto y se continúa buscando instrucciones a partir de allí.
  - En otro caso se sigue buscando las instrucciones en secuencia.

# Implementación práctica

- En la práctica, este tipo de Branch Predictor, se implementa en la etapa de Instruction Fetch del pipeline, como un pequeño cache de direcciones de salto accesible mediante las direcciones de las instrucciones.
- Otra forma de implementación es agregar un par de bits a cada bloque de líneas en el cache de instrucciones, que se emplean únicamente si ese bloque tiene instrucciones de salto condicional.
  - En el caso en que la predicción para la instrucción de salto de ese bloque sea *taken*, el Program Counter se setea con la dirección destino del salto y se continúa buscando instrucciones a partir de allí.
  - En otro caso se sigue buscando las instrucciones en secuencia.

# Eficiencia. Benchmarks SPEC89



# Performance Branch Predictor de 2 bits: Conclusiones

- El método tiene una eficiencia superior al 82 %, para cualquier tipo de programa
- La eficiencia es superior en programas de punto flotante (*missprediction rate* < 4 %) frente a los programas de cálculo entero (4 % < *missprediction rate* < 18 %)
- El tamaño del buffer de predicción no genera efecto en la eficiencia mas allá de los 4Kbytes.
- Tampoco se obtuvieron mejoras aumentando la cantidad de bits de predicción mas allá de 2. En general un branch predictor de n bits tomaría valores entre 0 y  $2^n - 1$ , tomando el salto para los valores contenidos en la mitad mas alta del rango y no lo tomaría para la mitad menos significativa. La complejidad en el diseño no se ve compensada por la mejora en la funcionalidad de predicción.

# Performance Branch Predictor de 2 bits: Conclusiones

- El método tiene una eficiencia superior al 82 %, para cualquier tipo de programa
- La eficiencia es superior en programas de punto flotante (*missprediction rate* < 4 %) frente a los programas de cálculo entero (4 % < *missprediction rate* < 18 %)
- El tamaño del buffer de predicción no genera efecto en la eficiencia mas allá de los 4Kbytes.
- Tampoco se obtuvieron mejoras aumentando la cantidad de bits de predicción mas allá de 2. En general un branch predictor de n bits tomaría valores entre 0 y  $2^n - 1$ , tomando el salto para los valores contenidos en la mitad mas alta del rango y no lo tomaría para la mitad menos significativa. La complejidad en el diseño no se ve compensada por la mejora en la funcionalidad de predicción.

# Performance Branch Predictor de 2 bits: Conclusiones

- El método tiene una eficiencia superior al 82 %, para cualquier tipo de programa
- La eficiencia es superior en programas de punto flotante (*missprediction rate* < 4 %) frente a los programas de cálculo entero (4 % < *missprediction rate* < 18 %)
- El tamaño del buffer de predicción no genera efecto en la eficiencia mas allá de los 4Kbytes.
- Tampoco se obtuvieron mejoras aumentando la cantidad de bits de predicción mas allá de 2. En general un branch predictor de n bits tomaría valores entre 0 y  $2^n - 1$ , tomando el salto para los valores contenidos en la mitad mas alta del rango y no lo tomaría para la mitad menos significativa. La complejidad en el diseño no se ve compensada por la mejora en la funcionalidad de predicción.

# Performance Branch Predictor de 2 bits: Conclusiones

- El método tiene una eficiencia superior al 82 %, para cualquier tipo de programa
- La eficiencia es superior en programas de punto flotante (*missprediction rate* < 4 %) frente a los programas de cálculo entero (4 % < *missprediction rate* < 18 %)
- El tamaño del buffer de predicción no genera efecto en la eficiencia mas allá de los 4Kbytes.
- Tampoco se obtuvieron mejoras aumentando la cantidad de bits de predicción mas allá de 2. En general un branch predictor de n bits tomaría valores entre 0 y  $2^n - 1$ , tomando el salto para los valores contenidos en la mitad mas alta del rango y no lo tomaría para la mitad menos significativa. La complejidad en el diseño no se ve compensada por la mejora en la funcionalidad de predicción.

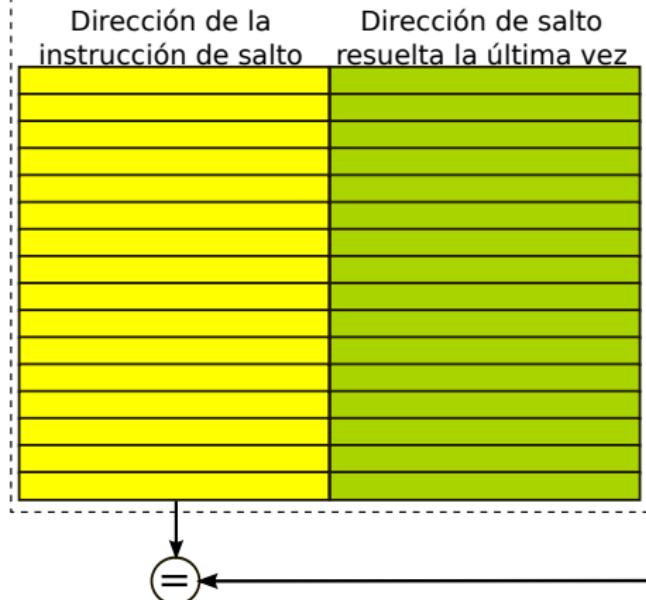
# Performance Branch Predictor de 2 bits: Conclusiones

- El método tiene una eficiencia superior al 82 %, para cualquier tipo de programa
- La eficiencia es superior en programas de punto flotante (*missprediction rate* < 4 %) frente a los programas de cálculo entero (4 % < *missprediction rate* < 18 %)
- El tamaño del buffer de predicción no genera efecto en la eficiencia mas allá de los 4Kbytes.
- Tampoco se obtuvieron mejoras aumentando la cantidad de bits de predicción mas allá de 2. En general un branch predictor de n bits tomaría valores entre 0 y  $2^n - 1$ , tomando el salto para los valores contenidos en la mitad mas alta del rango y no lo tomaría para la mitad menos significativa. La complejidad en el diseño no se ve compensada por la mejora en la funcionalidad de predicción.

# Branch Target Buffer

- Es un cache de instrucciones de salto que contiene para cada entrada el par dirección de la instrucción de salto, y dirección del target resuelta (No los Bits *taken* o *Non-taken*).

## Branch Target Buffer



Program Counter



# Branch Target Buffer

- Se trabaja como una memoria de acceso por contenido.
- Se accede mediante el valor completo del Program Counter.
- Si el valor no se encuentra se asume *taken*.

Si el valor se encuentra en el BTB, se aplica el campo de dirección target almacenado

- Si el valor se encuentra en el BTB, se aplica el campo de dirección target almacenado

Si el valor no se encuentra en el BTB, se aplica el campo de dirección target almacenado

Si el valor no se encuentra en el BTB, se aplica el campo de dirección target almacenado

Si el valor no se encuentra en el BTB, se aplica el campo de dirección target almacenado

Si el valor no se encuentra en el BTB, se aplica el campo de dirección target almacenado

Si el valor no se encuentra en el BTB, se aplica el campo de dirección target almacenado

Si el valor no se encuentra en el BTB, se aplica el campo de dirección target almacenado

# Branch Target Buffer

- Se trabaja como una memoria de acceso por contenido.
- Se accede mediante el valor completo del Program Counter.
- Si el valor no se encuentra se asume *taken*.

Si el PC no se encuentra en el BTB se aplica el delay en el pipeline, y se sigue ejecutando en el PC.

Si se encuentra, se ingresa el valor al BTB.

- Si el valor se encuentra en el BTB, se aplica el campo de dirección target almacenado

Si el PC no se encuentra en el BTB se aplica el delay en el pipeline, y se sigue ejecutando en el PC.

Si se encuentra, se ingresa el valor al BTB.

Si el PC no se encuentra en el BTB se aplica el delay en el pipeline, y se sigue ejecutando en el PC.

Si se encuentra, se ingresa el valor al BTB.

Si el PC no se encuentra en el BTB se aplica el delay en el pipeline, y se sigue ejecutando en el PC.

Si se encuentra, se ingresa el valor al BTB.

# Branch Target Buffer

- Se trabaja como una memoria de acceso por contenido.
- Se accede mediante el valor completo del Program Counter.
- Si el valor no se encuentra se asume *taken*.
  - Si el resultado es *Not-taken* se acepta el *delay* en el pipeline, y no se almacena nada en el BTB.
  - Si el resultado es *taken*, se ingresa el valor al BTB.
- Si el valor se encuentra en el BTB, se aplica el campo de dirección target almacenado

# Branch Target Buffer

- Se trabaja como una memoria de acceso por contenido.
- Se accede mediante el valor completo del Program Counter.
- Si el valor no se encuentra se asume *taken*.
  - Si el resultado es *Non-taken* se acepta el delay en el pipeline, y no se almacena nada en el BTB
  - Si el resultado es *taken*, Se ingresa el valor al BTB.
- Si el valor se encuentra en el BTB, se aplica el campo de dirección target almacenado

# Branch Target Buffer

- Se trabaja como una memoria de acceso por contenido.
- Se accede mediante el valor completo del Program Counter.
- Si el valor no se encuentra se asume *taken*.
  - Si el resultado es *Non-taken* se acepta el delay en el pipeline, y no se almacena nada en el BTB
  - Si el resultado es *taken*, Se ingresa el valor al BTB.
- Si el valor se encuentra en el BTB, se aplica el campo de dirección target almacenado

Si el PC no hay coincidencia, y no se guarda en el BTB ningún nuevo valor ya que el que está almacenado es el que nos interesa.

Si el PC coincide con el que está en el BTB, ya que el que está no sirve, luego de la penalidad correspondiente en el pipeline.

# Branch Target Buffer

- Se trabaja como una memoria de acceso por contenido.
- Se accede mediante el valor completo del Program Counter.
- Si el valor no se encuentra se asume *taken*.
  - Si el resultado es *Non-taken* se acepta el delay en el pipeline, y no se almacena nada en el BTB
  - Si el resultado es *taken*, Se ingresa el valor al BTB.
- Si el valor se encuentra en el BTB, se aplica el campo de dirección target almacenado
  - Si el resultado es *taken*, no hay penalidad, y no se guarda en el BTB ningún nuevo valor ya que el que está almacenado es el que nos sirve.
  - Si el resultado es *Non-taken*, cuando el nuevo valor en el BTB sea que el que está no sirve, luego de la penalidad correspondiente en el pipeline.

# Branch Target Buffer

- Se trabaja como una memoria de acceso por contenido.
- Se accede mediante el valor completo del Program Counter.
- Si el valor no se encuentra se asume *taken*.
  - Si el resultado es *Non-taken* se acepta el delay en el pipeline, y no se almacena nada en el BTB
  - Si el resultado es *taken*, Se ingresa el valor al BTB.
- Si el valor se encuentra en el BTB, se aplica el campo de dirección target almacenado
  - Si el resultado es *taken*, no hay penalidad, y no se guarda en el BTB ningún nuevo valor ya que el que está almacenado es el que nos sirve.
  - Si el resultado es *Non-taken*, guarda el nuevo valor en el BTB ya que el que está no sirvió, luego de la penalidad correspondiente en el pipeline.

# Branch Target Buffer

- Se trabaja como una memoria de acceso por contenido.
- Se accede mediante el valor completo del Program Counter.
- Si el valor no se encuentra se asume *taken*.
  - Si el resultado es *Non-taken* se acepta el delay en el pipeline, y no se almacena nada en el BTB
  - Si el resultado es *taken*, Se ingresa el valor al BTB.
- Si el valor se encuentra en el BTB, se aplica el campo de dirección target almacenado
  - Si el resultado es *taken*, no hay penalidad, y no se guarda en el BTB ningún nuevo valor ya que el que está almacenado es el que nos sirve.
  - Si el resultado es *Non-taken*, guarda el nuevo valor en el BTB ya que el que está no sirvió, luego de la penalidad correspondiente en el pipeline.

# Branch Target Buffer

- Se trabaja como una memoria de acceso por contenido.
- Se accede mediante el valor completo del Program Counter.
- Si el valor no se encuentra se asume *taken*.
  - Si el resultado es *Non-taken* se acepta el delay en el pipeline, y no se almacena nada en el BTB
  - Si el resultado es *taken*, Se ingresa el valor al BTB.
- Si el valor se encuentra en el BTB, se aplica el campo de dirección target almacenado
  - Si el resultado es *taken*, no hay penalidad, y no se guarda en el BTB ningún nuevo valor ya que el que está almacenado es el que nos sirve.
  - Si el resultado es *Non-taken*, guarda el nuevo valor en el BTB ya que el que está no sirvió, luego de la penalidad correspondiente en el pipeline.

# Contenido

## 1 Paralelismo a Nivel de Instrucción

- Pipeline
- Unidades de Predicción de saltos
- **Superscalar**
- Scheduling Dinámico

## 2 Ejecución Fuerza de Orden

- Prehistoria
- Solución Actual
- Especulación por Hardware
- Casos prácticos que mezclan todo lo visto

## 3 Procesadores Multithread

## 4 Multicore y Manycore

# Mas allá de 1 Instrucción por ciclo de clock...

- Podemos trabajar a nivel de Microarquitectura y Hardware, para tratar de aumentar la paralelización jugando con el clock y violar algunas leyes de la física, para llegar a que el pipeline pueda obtener una eficiencia de liberar un resultado por ciclo de clock.
- ¿Como seguimos?
- Simple... tratando de ejecutar mas de una instrucción por ciclo de clock.
- El problema es ¿como?.
- La respuesta es... ILP (Instruction Level Parallelism).
- ¿Como? ¿De que venimos hablando entonces hasta ahora?

# Mas allá de 1 Instrucción por ciclo de clock...

- Podemos trabajar a nivel de Microarquitectura y Hardware, para tratar de aumentar la paralelización jugando con el clock y violar algunas leyes de la física, para llegar a que el pipeline pueda obtener una eficiencia de liberar un resultado por ciclo de clock.
- ¿Como seguimos?
- Simple... tratando de ejecutar mas de una instrucción por ciclo de clock.
- El problema es ¿como?.
- La respuesta es... ILP (Instruction Level Parallelism).
- ¿Como? ¿De que venimos hablando entonces hasta ahora?

# Mas allá de 1 Instrucción por ciclo de clock...

- Podemos trabajar a nivel de Microarquitectura y Hardware, para tratar de aumentar la paralelización jugando con el clock y violar algunas leyes de la física, para llegar a que el pipeline pueda obtener una eficiencia de liberar un resultado por ciclo de clock.
- ¿Como seguimos?
- Simple... tratando de ejecutar mas de una instrucción por ciclo de clock.
- El problema es ¿como?.
- La respuesta es... ILP (Instruction Level Parallelism).
- ¿Como? ¿De que venimos hablando entonces hasta ahora?

# Mas allá de 1 Instrucción por ciclo de clock...

- Podemos trabajar a nivel de Microarquitectura y Hardware, para tratar de aumentar la paralelización jugando con el clock y violar algunas leyes de la física, para llegar a que el pipeline pueda obtener una eficiencia de liberar un resultado por ciclo de clock.
- ¿Como seguimos?
- Simple... tratando de ejecutar mas de una instrucción por ciclo de clock.
- El problema es ¿como?.
- La respuesta es... ILP (Instruction Level Parallelism).
- ¿Como? ¿De que venimos hablando entonces hasta ahora?

# Mas allá de 1 Instrucción por ciclo de clock...

- Podemos trabajar a nivel de Microarquitectura y Hardware, para tratar de aumentar la paralelización jugando con el clock y violar algunas leyes de la física, para llegar a que el pipeline pueda obtener una eficiencia de liberar un resultado por ciclo de clock.
- ¿Como seguimos?
- Simple... tratando de ejecutar mas de una instrucción por ciclo de clock.
- El problema es ¿como?
  - La respuesta es... ILP (Instruction Level Parallelism).
  - ¿Como? ¿De que venimos hablando entonces hasta ahora?

# Mas allá de 1 Instrucción por ciclo de clock...

- Podemos trabajar a nivel de Microarquitectura y Hardware, para tratar de aumentar la paralelización jugando con el clock y violar algunas leyes de la física, para llegar a que el pipeline pueda obtener una eficiencia de liberar un resultado por ciclo de clock.
- ¿Como seguimos?
- Simple... tratando de ejecutar mas de una instrucción por ciclo de clock.
- El problema es ¿como?.
- La respuesta es... ILP (Instruction Level Parallelism).
- ¿Como? ¿De que venimos hablando entonces hasta ahora?

# Mas allá de 1 Instrucción por ciclo de clock...

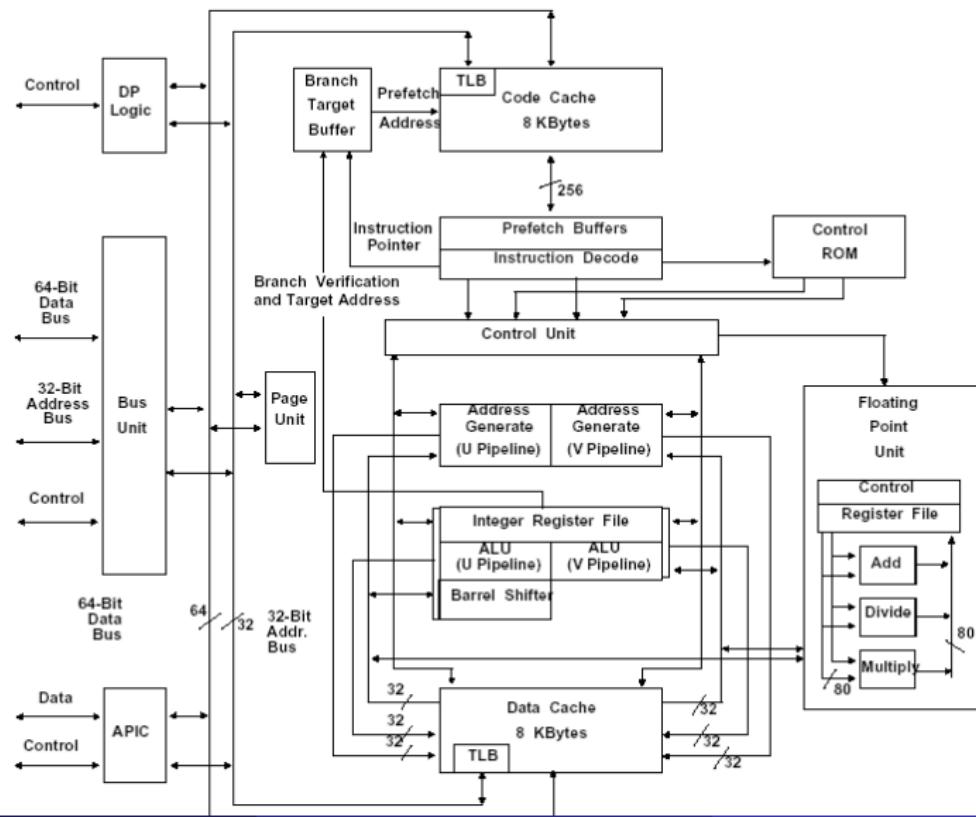
- Podemos trabajar a nivel de Microarquitectura y Hardware, para tratar de aumentar la paralelización jugando con el clock y violar algunas leyes de la física, para llegar a que el pipeline pueda obtener una eficiencia de liberar un resultado por ciclo de clock.
- ¿Como seguimos?
- Simple... tratando de ejecutar mas de una instrucción por ciclo de clock.
- El problema es ¿como?.
- La respuesta es... ILP (Instruction Level Parallelism).
- ¿Como? ¿De que venimos hablando entonces hasta ahora?

# Superscalar de dos vías



# Un Superscalar de dos vías: El pentium

Pentium® Processor (75/90/100/120/133/150/166/200 MHz)



# Mas ILP, aumenta los obstáculos

- Los obstáculos estructurales quedan mas expuestos. Recordando el hazard de acceso simultáneo a memoria, ahora cada etapa debe lidiar con las otras etapas de su propio pipeline que pueden acceder en simultáneo para acceder a instrucciones o datos, y además con las mismas etapas del otro pipeline. La probabilidad de accesos concurrentes aumenta con la cantidad de vías del Superscalar.
- Ahora se pueden ejecutar en dos ALUs dos instrucciones. Si una depende del resultado de la otra, esto no es posible.
- Una falla en el branch prediction... es letal. Limpia ambos pipelines!

# Contenido

## 1 Paralelismo a Nivel de Instrucción

- Pipeline
- Unidades de Predicción de saltos
- Superscalar
- Scheduling Dinámico

## 2 Ejecución Fuerza de Orden

- Prehistoria
- Solución Actual
- Especulación por Hardware
- Casos prácticos que mezclan todo lo visto

## 3 Procesadores Multithread

## 4 Multicore y Manycore

# Obstáculos de Datos. Aumento conforme se avanza

- Hasta aquí estudiamos pipelines con scheduling de instrucciones estático.
- Estos pipelines simplemente buscan una instrucción y la envían a la unidad de ejecución.
- En caso de tener dependencias con otra instrucción en el pipeline, el envío es demorado.
- En caso de superescalares el estudio de dependencias se debe extender a instrucciones en los diferentes pipelines.
- Estos y otros casos que veremos, vuelven insuficiente el método de Forwarding aplicado en los primeros pipelines.
- Hay que ir por mas.

# Obstáculos de Datos. Aumento conforme se avanza

- Hasta aquí estudiamos pipelines con scheduling de instrucciones estático.
- Estos pipelines simplemente buscan una instrucción y la envían a la unidad de ejecución.
- En caso de tener dependencias con otra instrucción en el pipeline, el envío es demorado.
- En caso de superescalares el estudio de dependencias se debe extender a instrucciones en los diferentes pipelines.
- Estos y otros casos que veremos, vuelven insuficiente el método de Forwarding aplicado en los primeros pipelines.
- Hay que ir por mas.

# Obstáculos de Datos. Aumento conforme se avanza

- Hasta aquí estudiamos pipelines con scheduling de instrucciones estático.
- Estos pipelines simplemente buscan una instrucción y la envían a la unidad de ejecución.
- En caso de tener dependencias con otra instrucción en el pipeline, el envío es demorado.
- En caso de superescalares el estudio de dependencias se debe extender a instrucciones en los diferentes pipelines.
- Estos y otros casos que veremos, vuelven insuficiente el método de Forwarding aplicado en los primeros pipelines.
- Hay que ir por mas.

# Obstáculos de Datos. Aumento conforme se avanza

- Hasta aquí estudiamos pipelines con scheduling de instrucciones estático.
- Estos pipelines simplemente buscan una instrucción y la envían a la unidad de ejecución.
- En caso de tener dependencias con otra instrucción en el pipeline, el envío es demorado.
- En caso de superescalares el estudio de dependencias se debe extender a instrucciones en los diferentes pipelines.
- Estos y otros casos que veremos, vuelven insuficiente el método de Forwarding aplicado en los primeros pipelines.
- Hay que ir por mas.

# Obstáculos de Datos. Aumento conforme se avanza

- Hasta aquí estudiamos pipelines con scheduling de instrucciones estático.
- Estos pipelines simplemente buscan una instrucción y la envían a la unidad de ejecución.
- En caso de tener dependencias con otra instrucción en el pipeline, el envío es demorado.
- En caso de superescalares el estudio de dependencias se debe extender a instrucciones en los diferentes pipelines.
- Estos y otros casos que veremos, vuelven insuficiente el método de Forwarding aplicado en los primeros pipelines.
- Hay que ir por mas.

# Obstáculos de Datos. Aumento conforme se avanza

- Hasta aquí estudiamos pipelines con scheduling de instrucciones estático.
- Estos pipelines simplemente buscan una instrucción y la envían a la unidad de ejecución.
- En caso de tener dependencias con otra instrucción en el pipeline, el envío es demorado.
- En caso de superescalares el estudio de dependencias se debe extender a instrucciones en los diferentes pipelines.
- Estos y otros casos que veremos, vuelven insuficiente el método de Forwarding aplicado en los primeros pipelines.
- Hay que ir por mas.

# Obstáculos de Datos. Aumento conforme se avanza

- Hasta aquí estudiamos pipelines con scheduling de instrucciones estático.
- Estos pipelines simplemente buscan una instrucción y la envían a la unidad de ejecución.
- En caso de tener dependencias con otra instrucción en el pipeline, el envío es demorado.
- En caso de superescalares el estudio de dependencias se debe extender a instrucciones en los diferentes pipelines.
- Estos y otros casos que veremos, vuelven insuficiente el método de Forwarding aplicado en los primeros pipelines.
- Hay que ir por mas.

# Idea Fuerza

- Si una instrucción  $j$ , depende una instrucción  $i$ , que para completarse requiere de varios ciclos de clocks adicionales debido a obstáculos de tipo estructural, o de datos, todas las instrucciones que siguen a la instrucción  $j$  no pueden ser ejecutadas.
- Esta situación genera una obstrucción del pipeline, dejando en estado idle a todas las unidades funcionales que compongan la Unidad de Ejecución del pipeline.

# Idea Fuerza

- Si una instrucción  $j$ , depende una instrucción  $i$ , que para completarse requiere de varios ciclos de clocks adicionales debido a obstáculos de tipo estructural, o de datos, todas las instrucciones que siguen a la instrucción  $j$  no pueden ser ejecutadas.
- Esta situación genera una obstrucción del pipeline, dejando en estado idle a todas las unidades funcionales que compongan la Unidad de Ejecución del pipeline.

# Idea Fuerza

- Si una instrucción  $j$ , depende una instrucción  $i$ , que para completarse requiere de varios ciclos de clocks adicionales debido a obstáculos de tipo estructural, o de datos, todas las instrucciones que siguen a la instrucción  $j$  no pueden ser ejecutadas.
- Esta situación genera una obstrucción del pipeline, dejando en estado idle a todas las unidades funcionales que compongan la Unidad de Ejecución del pipeline.

# Idea Fuerza

- Para ilustrarlo consideremos el siguiente código para un procesador MIPS.

```
1  div .d      F0 ,F2 ,F4  
2  add .d      F10 ,F0 ,F8  
3  sub .d      F12 ,F8 ,F14
```

- La instrucción ***div.d***, lleva varios ciclos de clock para completarse, lo cual obstruye la ejecución de la instrucción ***add.d*** ya que utiliza el registro F0, en donde se almacenará el resultado de la instrucción previa.
- La instrucción ***sub.d***, no guarda dependencias de datos con las previas, y queda esperando que se complete la instrucción responsable del atasco.

# Idea Fuerza

- Para ilustrarlo consideremos el siguiente código para un procesador MIPS.

```
1  div .d      F0 ,F2 ,F4  
2  add .d      F10 ,F0 ,F8  
3  sub .d      F12 ,F8 ,F14
```

- La instrucción ***div.d***, lleva varios ciclos de clock para completarse, lo cual obstruye la ejecución de la instrucción ***add.d*** ya que utiliza el registro F0, en donde se almacenará el resultado de la instrucción previa.
- La instrucción ***sub.d***, no guarda dependencias de datos con las previas, y queda esperando que se complete la instrucción responsable del atasco.

# Idea Fuerza

- Para ilustrarlo consideremos el siguiente código para un procesador MIPS.

```
1  div .d      F0 ,F2 ,F4  
2  add .d      F10 ,F0 ,F8  
3  sub .d      F12 ,F8 ,F14
```

- La instrucción ***div.d***, lleva varios ciclos de clock para completarse, lo cual obstruye la ejecución de la instrucción ***add.d*** ya que utiliza el registro F0, en donde se almacenará el resultado de la instrucción previa.
- La instrucción ***sub.d***, no guarda dependencias de datos con las previas, y queda esperando que se complete la instrucción responsable del atasco.

# Ejecución Fuera de Orden: Idea Fuerza

- Ejecutar instrucciones Fuera de Orden consiste, tal como es de esperar, en tratar de enviar las instrucciones a ejecución independientemente del orden en el que están en el código.
- Esta decisión se puede tomar una vez decodificada la instrucción, ya que allí se sabe si hay un atasco de datos o estructural que haga que la instrucción que se comience a ejecutar pueda impedir el envío de sus dependientes próximas.
- Cada vez que lo intentemos pueden aparecer riesgos los cuales deben ser evaluados para no incurrir en errores.

# Ejecución Fuera de Orden: Idea Fuerza

- Ejecutar instrucciones Fuera de Orden consiste, tal como es de esperar, en tratar de enviar las instrucciones a ejecución independientemente del orden en el que están en el código.
- Esta decisión se puede tomar una vez decodificada la instrucción, ya que allí se sabe si hay un atasco de datos o estructural que haga que la instrucción que se comience a ejecutar pueda impedir el envío de sus dependientes próximas.
- Cada vez que lo intentemos pueden aparecer riesgos los cuales deben ser evaluados para no incurrir en errores.

# Ejecución Fuera de Orden: Idea Fuerza

- Ejecutar instrucciones Fuera de Orden consiste, tal como es de esperar, en tratar de enviar las instrucciones a ejecución independientemente del orden en el que están en el código.
- Esta decisión se puede tomar una vez decodificada la instrucción, ya que allí se sabe si hay un atasco de datos o estructural que haga que la instrucción que se comience a ejecutar pueda impida el envío de sus dependientes próximas.
- Cada vez que lo intentemos pueden aparecer riesgos los cuales deben ser evaluados para no incurrir en errores.

# Ejecución Fuera de Orden: Idea Fuerza

- Ejecutar instrucciones Fuera de Orden consiste, tal como es de esperar, en tratar de enviar las instrucciones a ejecución independientemente del orden en el que están en el código.
- Esta decisión se puede tomar una vez decodificada la instrucción, ya que allí se sabe si hay un atasco de datos o estructural que haga que la instrucción que se comience a ejecutar pueda impida el envío de sus dependientes próximas.
- Cada vez que lo intentemos pueden aparecer riesgos los cuales deben ser evaluados para no incurrir en errores.

# Ejecución Fuera de Orden: Riesgos

- Consideremos el siguiente código para un procesador MIPS.

```
1  div .d      F0,F2,F4  
2  add .d      F6,F0,F8  
3  sub .d      F8,F10,F14  
4  mul .d      F6,F10,F8
```

- La instrucción ***div.d*** demora varios ciclos y atasca el envío de la instrucción ***add.d***.
- Es posible enviar las instrucciones ***sub.d*** y ***mul.d***, aunque deben tomarse en cuenta que hay riesgos. ¿Cuales son?

# Ejecución Fuera de Orden: Riesgos

- Consideraremos el siguiente código para un procesador MIPS.

```
1  div .d      F0 ,F2 ,F4
2  add .d      F6 ,F0 ,F8
3  sub .d      F8 ,F10 ,F14
4  mul .d      F6 ,F10 ,F8
```

- La instrucción *div.d* demora varios ciclos y atasca el envío de la instrucción *add.d*.
- Es posible enviar las instrucciones *sub.d* y *mul.d*, aunque deben tomarse en cuenta que hay riesgos. ¿Cuales son?

# Ejecución Fuera de Orden: Riesgos

- Consideraremos el siguiente código para un procesador MIPS.

```
1  div .d      F0 , F2 , F4
2  add .d      F6 , F0 , F8
3  sub .d      F8 , F10 , F14
4  mul .d      F6 , F10 , F8
```

- La instrucción ***div.d*** demora varios ciclos y atasca el envío de la instrucción ***add.d***.
- Es posible enviar las instrucciones ***sub.d*** y ***mul.d***, aunque deben tomarse en cuenta que hay riesgos. ¿Cuales son?

# Ejecución Fuera de Orden: Riesgos

- Consideraremos el siguiente código para un procesador MIPS.

```
1  div .d      F0 , F2 , F4
2  add .d      F6 , F0 , F8
3  sub .d      F8 , F10 , F14
4  mul .d      F6 , F10 , F8
```

- La instrucción ***div.d*** demora varios ciclos y atasca el envío de la instrucción ***add.d***.
- Es posible enviar las instrucciones ***sub.d*** y ***mul.d***, aunque deben tomarse en cuenta que hay riesgos. ¿Cuales son?

# Ejecución Fuera de Orden: Riesgos

- Consideraremos el siguiente código para un procesador MIPS.

```
1  div .d      F0 ,F2 ,F4
2  add .d      F6 ,F0 ,F8
3  sub .d      F8 ,F10 ,F14
4  mul .d      F6 ,F10 ,F8
```

- En el bloque de código anterior, la instrucción 2 se encuentra demorada de envío por dependencia de la instrucción 1.
- Así que se envían las instrucciones 3 y 4, (considerando ya un procesador superescalar).

# Ejecución Fuera de Orden: Riesgos

- Consideraremos el siguiente código para un procesador MIPS.

```
1  div .d      F0 ,F2 ,F4
2  add .d      F6 ,F0 ,F8
3  sub .d      F8 ,F10 ,F14
4  mul .d      F6 ,F10 ,F8
```

- En el bloque de código anterior, la instrucción 2 se encuentra demorada de envío por dependencia de la instrucción 1.
- Así que se envían las instrucciones 3 y 4, (considerando ya un procesador superescalar).

# Ejecución Fuera de Orden: Riesgos

- Consideraremos el siguiente código para un procesador MIPS.

```
1  div .d      F0 ,F2 ,F4  
2  add .d      F6 ,F0 ,F8  
3  sub .d      F8 ,F10 ,F14  
4  mul .d      F6 ,F10 ,F8
```

- En el bloque de código anterior, la instrucción 2 se encuentra demorada de envío por dependencia de la instrucción 1.
- Así que se envían las instrucciones 3 y 4, (considerando ya un procesador superescalar).

# Ejecución Fuera de Orden: Riesgos

- Consideraremos el siguiente código para un procesador MIPS.

```
1  div .d      F0 , F2 , F4
2  add .d      F6 , F0 , F8
3  sub .d      F8 , F10 , F14
4  mul .d      F6 , F10 , F8
```

Esta situación conlleva dos riesgos potenciales:

- La Instrucción 3 escribe su resultado en un operando de la Instrucción 2
- La Instrucción 4 escribe su resultado en el destino de la Instrucción 2

# Ejecución Fuera de Orden: Riesgos

- Consideraremos el siguiente código para un procesador MIPS.

```
1  div .d      F0,F2,F4
2  add .d      F6,F0,F8
3  sub .d      F8,F10,F14
4  mul .d      F6,F10,F8
```

Esta situación conlleva dos riesgos potenciales:

- 1 La Instrucción 3 escribe su resultado en un operando de la Instrucción 2
- 2 La Instrucción 4 escribe su resultado en el destino de la Instrucción 2

# Ejecución Fuera de Orden: Riesgos

- Consideraremos el siguiente código para un procesador MIPS.

```
1  div .d      F0,F2,F4
2  add .d      F6,F0,F8
3  sub .d      F8,F10,F14
4  mul .d      F6,F10,F8
```

Esta situación conlleva dos riesgos potenciales:

- La Instrucción 3 escribe su resultado en un operando de la Instrucción 2
- La Instrucción 4 escribe su resultado en el destino de la Instrucción 2

# Riesgos WAR

## Write, After Read

La situación planteada en 1, se conoce como riesgo **WAR**, por **Write After Read**, y representa el riesgo que como consecuencia de la ejecución de la instrucción 3, ésta escriba (**Write**) el registro F8, y después (**After**) de ello la instrucción 2 lo leerá (**Read**), obteniendo un dato incorrecto y alterando el resultado de la secuencia de código generada originalmente por el programador.

# Riesgos WAW

## Write, After Write

La situación planteada en 2, se conoce como riesgo **WAW**, por **Write After Write**, y representa el riesgo que como consecuencia de la ejecución de la instrucción 4, ésta escriba (**Write**) el registro F6, y después (**After**) de ello la instrucción 2 lo escribirá (**Write**), eliminando el resultado de la instrucción 4 y nuevamente alterando el resultado de lo que el programador determinó.

# Riesgos RAW

## Read, After Write

Resta el que se conoce como **RAW**, **R**ead, **A**fter **W**rite, que si bien no está presente en el ejemplo anterior, es el mas común. Se presenta cada vez que se lee un operando que después es escrito por una instrucción dependiente.

# Manejo de excepciones en Ejecución fuera de orden

- No solo se necesita preservar el comportamiento de las instrucciones de modo que los resultados aparezcan en el mismo orden en el que ocupan las instrucciones en el programa, también en manejo de las excepciones debe preservar el comportamiento de modo que resulte idéntico al que se tendría como resultado de procesar en el orden que establece el programa.
- Al ejecutar fuera de orden, el procesador puede generar lo que se denomina *excepciones imprecisas*, que son aquellas que al producirse, el estado del procesador no es el mismo que debería ser si las instrucciones se hubiesen ejecutado en orden.
- Hay dos posibles motivos:
  - El ordenador ha detectado una excepción que no se ha podido manejar.
  - El ordenador ha detectado una excepción que se ha podido manejar.
- En resumidas cuentas, el procesador debe asegurar que no se levante una excepción hasta no tener completado todo el programa incluida la instrucción que es responsable de la excepción.

# Manejo de excepciones en Ejecución fuera de orden

- No solo se necesita preservar el comportamiento de las instrucciones de modo que los resultados aparezcan en el mismo orden en el que ocupan las instrucciones en el programa, también en manejo de las excepciones debe preservar el comportamiento de modo que resulte idéntico al que se tendría como resultado de procesar en el orden que establece el programa.
- Al ejecutar fuera de orden, el procesador puede generar lo que se denomina *excepciones imprecisas*, que son aquellas que al producirse, el estado del procesador no es el mismo que debería ser si las instrucciones se hubiesen ejecutado en orden.
- Hay dos posibles motivos:
  - El ordenador ha detectado una excepción y ha saltado a la excepción correcta.
  - El ordenador ha detectado una excepción y ha saltado a la excepción incorrecta.
- En resumidas cuentas, el procesador debe asegurar que no se levante una excepción hasta no tener completado todo el programa incluida la instrucción que es responsable de la excepción.

# Manejo de excepciones en Ejecución fuera de orden

- No solo se necesita preservar el comportamiento de las instrucciones de modo que los resultados aparezcan en el mismo orden en el que ocupan las instrucciones en el programa, también en manejo de las excepciones debe preservar el comportamiento de modo que resulte idéntico al que se tendría como resultado de procesar en el orden que establece el programa.
- Al ejecutar fuera de orden, el procesador puede generar lo que se denomina *excepciones imprecisas*, que son aquellas que al producirse, el estado del procesador no es el mismo que debería ser si las instrucciones se hubiesen ejecutado en orden.
  - Hay dos posibles motivos:
    - El pipeline ha completado la ejecución de una o mas instrucciones posteriores a la que produce la excepción.
- En resumidas cuentas, el procesador debe asegurar que no se levante una excepción hasta no tener completado todo el programa incluida la instrucción que es responsable de la excepción.

# Manejo de excepciones en Ejecución fuera de orden

- No solo se necesita preservar el comportamiento de las instrucciones de modo que los resultados aparezcan en el mismo orden en el que ocupan las instrucciones en el programa, también en manejo de las excepciones debe preservar el comportamiento de modo que resulte idéntico al que se tendría como resultado de procesar en el orden que establece el programa.
- Al ejecutar fuera de orden, el procesador puede generar lo que se denomina *excepciones imprecisas*, que son aquellas que al producirse, el estado del procesador no es el mismo que debería ser si las instrucciones se hubiesen ejecutado en orden.
- Hay dos posibles motivos:
  - 1 El pipeline ha completado la ejecución de una o mas instrucciones posteriores a la que produce la excepción.
  - 2 El pipeline no ha completado aún al menos una instrucción previa a la que genera la excepción.
- En resumidas cuentas, el procesador debe asegurar que no se levante una excepción hasta no tener completado todo el programa incluida la instrucción que es responsable de la excepción.

# Manejo de excepciones en Ejecución fuera de orden

- No solo se necesita preservar el comportamiento de las instrucciones de modo que los resultados aparezcan en el mismo orden en el que ocupan las instrucciones en el programa, también en manejo de las excepciones debe preservar el comportamiento de modo que resulte idéntico al que se tendría como resultado de procesar en el orden que establece el programa.
- Al ejecutar fuera de orden, el procesador puede generar lo que se denomina *excepciones imprecisas*, que son aquellas que al producirse, el estado del procesador no es el mismo que debería ser si las instrucciones se hubiesen ejecutado en orden.
- Hay dos posibles motivos:
  - ① El pipeline ha completado la ejecución de una o mas instrucciones posteriores a la que produce la excepción.
  - ② El pipeline no ha completado aún al menos una instrucción previa a la que genera la excepción.
- En resumidas cuentas, el procesador debe asegurar que no se levante una excepción hasta no tener completado todo el programa incluida la instrucción que es responsable de la excepción.

# Manejo de excepciones en Ejecución fuera de orden

- No solo se necesita preservar el comportamiento de las instrucciones de modo que los resultados aparezcan en el mismo orden en el que ocupan las instrucciones en el programa, también en manejo de las excepciones debe preservar el comportamiento de modo que resulte idéntico al que se tendría como resultado de procesar en el orden que establece el programa.
- Al ejecutar fuera de orden, el procesador puede generar lo que se denomina *excepciones imprecisas*, que son aquellas que al producirse, el estado del procesador no es el mismo que debería ser si las instrucciones se hubiesen ejecutado en orden.
- Hay dos posibles motivos:
  - ① El pipeline ha completado la ejecución de una o mas instrucciones posteriores a la que produce la excepción.
  - ② El pipeline no ha completado aún al menos una instrucción previa a la que genera la excepción.
- En resumidas cuentas, el procesador debe asegurar que no se levante una excepción hasta no tener completado todo el programa incluida la instrucción que es responsable de la excepción.

# Manejo de excepciones en Ejecución fuera de orden

- No solo se necesita preservar el comportamiento de las instrucciones de modo que los resultados aparezcan en el mismo orden en el que ocupan las instrucciones en el programa, también en manejo de las excepciones debe preservar el comportamiento de modo que resulte idéntico al que se tendría como resultado de procesar en el orden que establece el programa.
- Al ejecutar fuera de orden, el procesador puede generar lo que se denomina *excepciones imprecisas*, que son aquellas que al producirse, el estado del procesador no es el mismo que debería ser si las instrucciones se hubiesen ejecutado en orden.
- Hay dos posibles motivos:
  - ① El pipeline ha completado la ejecución de una o mas instrucciones posteriores a la que produce la excepción.
  - ② El pipeline no ha completado aún al menos una instrucción previa a la que genera la excepción.
- En resumidas cuentas, el procesador debe asegurar que no se levante una excepción hasta no tener completado todo el programa incluida la instrucción que es responsable de la excepción.

# Ejecución fuera de Orden

## Implementación

Para implementar un procesador con Ejecución Fuera de Orden necesitamos dividir la etapa de Decodificación de instrucciones en dos sub-etapas

- La etapa de **Envío**, trabaja *en orden*, decodificando las instrucciones y enviándolas a la unidad de ejecución. Si encuentra un obstáculo estructural se detiene hasta que se resuelva el obstáculo. La unidad de Prebúsqueda continua igualmente, hasta que se llene el buffer de prebúsqueda. Si esto ocurre se detiene (stall).
- La etapa de **Lectura de Operandos**, inicia la operación *fuera de orden*, ya que es capaz de saltar en el buffer aquellas instrucciones que tienen bloqueos de Datos. Envía a ejecutar aquellas cuyos operandos pueden ser accedidos. Y permanece revisando cuando se resuelven los obstáculos de datos para enviar esa instrucción a ejecución-

# Ejecución fuera de Orden

## Unidad de Prebúsqueda

Se encarga de buscar instrucciones ya sea desde un buffer de instrucciones o desde el cache L1 interno en los procesadores mas actuales. Los coloca en una cola FIFO, mediante un índice para determinar cuando la cola está completa.

## La clave de Ejecución Fuera de Orden

- Para poder implementar este modelo es necesario poder ejecutar varias instrucciones al mismo tiempo (mas allá de un modelo superescalar).
- Si podemos identificar el momento de inicio y finalización de una instrucción podemos definir que la instrucción entre esos momentos se encuentra *en ejecución*.

# Contenido

## 1 Paralelismo a Nivel de Instrucción

- Pipeline
- Unidades de Predicción de saltos
- Superscalar
- Scheduling Dinámico

## 2 Ejecución Fuera de Orden

- Prehistoria
- Solución Actual
- Especulación por Hardware
- Casos prácticos que mezclan todo lo visto

## 3 Procesadores Multithread

## 4 Multicore y Manycore

# Scoreboarding

- Es el método más sencillo (y menos sofisticado) para implementar Ejecución Fuera de Orden evitando los riesgos asociados (**WAR** y **WAW**)
- Su primer implementación fue en la CDC 6600, instalada entre otros laboratorios en la Universidad de Berkeley en 1964. Control Data Corporation, construyó este computador basado en transistores de germanio, con algunas innovaciones interesantes en su diseño:

• Unidad de control de memoria que realizaba el scoreboarding. Se encargaba de la ejecución de las instrucciones y de la validación de los resultados.

# Scoreboarding

- Es el método más sencillo (y menos sofisticado) para implementar Ejecución Fuera de Orden evitando los riesgos asociados (**WAR** y **WAW**)
- Su primer implementación fue en la CDC 6600, instalada entre otros laboratorios en la Universidad de Berkeley en 1964. Control Data Corporation, construyó este computador basado en transistores de germanio, con algunas innovaciones interesantes en su diseño:
  - Un juego de instrucciones muy sencillo en contraposición a los sets complejos de instrucciones que poseían hasta ese momento las restantes CPUs, estableciendo los cimientos de los actuales procesadores RISC.

# Scoreboarding

- Es el método más sencillo (y menos sofisticado) para implementar Ejecución Fuera de Orden evitando los riesgos asociados (**WAR** y **WAW**)
- Su primer implementación fue en la CDC 6600, instalada entre otros laboratorios en la Universidad de Berkeley en 1964. Control Data Corporation, construyó este computador basado en transistores de germanio, con algunas innovaciones interesantes en su diseño:
  - Un juego de instrucciones muy sencillo en contraposición a los sets complejos de instrucciones que poseían hasta ese momento las restantes CPUs, estableciendo los cimientos de los actuales procesadores RISC.
  - Capacidad de ejecutar fuera de orden.
  - Amplia paralelización en la Unidad de Ejecución: 4 unidades de Punto Flotante, 5 Unidades de Referencias a Memoria, y 7 ALUs para enteros.

# Scoreboarding

- Es el método más sencillo (y menos sofisticado) para implementar Ejecución Fuera de Orden evitando los riesgos asociados (**WAR** y **WAW**)
- Su primer implementación fue en la CDC 6600, instalada entre otros laboratorios en la Universidad de Berkeley en 1964. Control Data Corporation, construyó este computador basado en transistores de germanio, con algunas innovaciones interesantes en su diseño:
  - Un juego de instrucciones muy sencillo en contraposición a los sets complejos de instrucciones que poseían hasta ese momento las restantes CPUs, estableciendo los cimientos de los actuales procesadores RISC.
  - Capacidad de ejecutar fuera de orden.
  - Amplia paralelización en la Unidad de Ejecución: 4 unidades de Punto Flotante, 5 Unidades de Referencias a Memoria, y 7 ALUs para enteros.

# Scoreboarding

- Es el método más sencillo (y menos sofisticado) para implementar Ejecución Fuera de Orden evitando los riesgos asociados (**WAR** y **WAW**)
- Su primer implementación fue en la CDC 6600, instalada entre otros laboratorios en la Universidad de Berkeley en 1964. Control Data Corporation, construyó este computador basado en transistores de germanio, con algunas innovaciones interesantes en su diseño:
  - Un juego de instrucciones muy sencillo en contraposición a los sets complejos de instrucciones que poseían hasta ese momento las restantes CPUs, estableciendo los cimientos de los actuales procesadores RISC.
  - Capacidad de ejecutar fuera de orden.
  - Amplia paralelización en la Unidad de Ejecución: 4 unidades de Punto Flotante, 5 Unidades de Referencias a Memoria, y 7 ALUs para enteros.

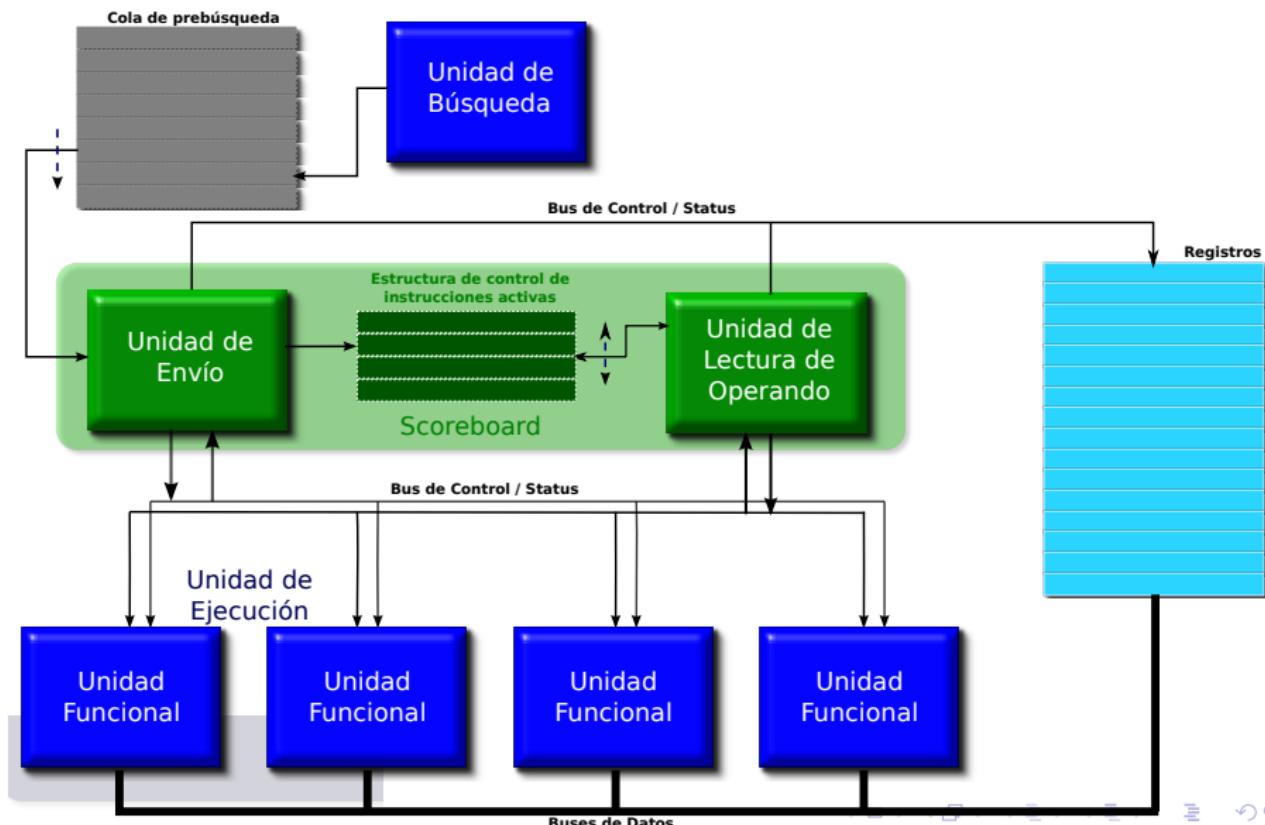
# Scoreboarding

- Es el método más sencillo (y menos sofisticado) para implementar Ejecución Fuera de Orden evitando los riesgos asociados (**WAR** y **WAW**)
- Su primer implementación fue en la CDC 6600, instalada entre otros laboratorios en la Universidad de Berkeley en 1964. Control Data Corporation, construyó este computador basado en transistores de germanio, con algunas innovaciones interesantes en su diseño:
  - Un juego de instrucciones muy sencillo en contraposición a los sets complejos de instrucciones que poseían hasta ese momento las restantes CPUs, estableciendo los cimientos de los actuales procesadores RISC.
  - Capacidad de ejecutar fuera de orden.
  - Amplia paralelización en la Unidad de Ejecución: 4 unidades de Punto Flotante, 5 Unidades de Referencias a Memoria, y 7 ALUs para enteros.

# La CDC 6600... aunque Ud. no lo crea. (©Wikipedia)



# Scoreboarding: Diagrama General



# Limitaciones del Scoreboard

- Aparecen nuevos Obstáculos estructurales debido a que la cantidad de buses es limitada para paralelizar las transferencias entre el scoreboard y los registros.
- Los operandos se leen directamente en los registros, y por lo tanto no se aprovecha la técnica de Forwarding (adelantar el operando desde el registro de salida de la unidad funcional que ejecutó la instrucción).

# Limitaciones del Scoreboard

- Aparecen nuevos Obstáculos estructurales debido a que la cantidad de buses es limitada para paralelizar las transferencias entre el scoreboard y los registros.
- Los operandos se leen directamente en los registros, y por lo tanto no se aprovecha la técnica de Forwarding (adelantar el operando desde el registro de salida de la unidad funcional que ejecutó la instrucción).

# Limitaciones del Scoreboard

- Aparecen nuevos Obstáculos estructurales debido a que la cantidad de buses es limitada para paralelizar las transferencias entre el scoreboard y los registros.
- Los operandos se leen directamente en los registros, y por lo tanto no se aprovecha la técnica de Forwarding (adelantar el operando desde el registro de salida de la unidad funcional que ejecutó la instrucción).

# Contenido

## 1 Paralelismo a Nivel de Instrucción

- Pipeline
- Unidades de Predicción de saltos
- Superscalar
- Scheduling Dinámico

## 2 Ejecución Fuera de Orden

- Prehistoria
- Solución Actual
- Especulación por Hardware
- Casos prácticos que mezclan todo lo visto

## 3 Procesadores Multithread

## 4 Multicore y Manycore

# Approach de Tomasulo

- Proviene de un trabajo de implementación de scheduling dinámico en la unidad de Punto Flotante de la IBM 360/91.
- Su interés es minimizar los riesgos **RAW**, e implementar Register Renaming en los **WAR** y **WAW**.
- Para comprenderlo consideremos el siguiente código para un procesador MIPS.

```
1  div .d      F0,F2,F4
2  add .d      F6,F0,F8    # WAR hazard por F8 con sub.d
3  s .d        F6,0(R1)
4  sub .d      F8,F10,F14
5  mul .d      F6,F10,F8  # WAW hazard por F6 con add.d
```

- Un escenario para la solución sería disponer de dos registros, que denominaremos S y T que permitan tomar datos y utilizarse para su acceso a partir de ese momento.

# Approach de Tomasulo

- Proviene de un trabajo de implementación de scheduling dinámico en la unidad de Punto Flotante de la IBM 360/91.
- Su interés es minimizar los riesgos **RAW**, e implementar Register Renaming en los **WAR** y **WAW**.
- Para comprenderlo consideremos el siguiente código para un procesador MIPS.

```
1  div .d      F0,F2,F4
2  add .d      F6,F0,F8    # WAR hazard por F8 con sub.d
3  s .d        F6,0(R1)
4  sub .d      F8,F10,F14
5  mul .d      F6,F10,F8  # WAW hazard por F6 con add.d
```

- Un escenario para la solución sería disponer de dos registros, que denominaremos S y T que permitan tomar datos y utilizarse para su acceso a partir de ese momento.

# Approach de Tomasulo

- Proviene de un trabajo de implementación de scheduling dinámico en la unidad de Punto Flotante de la IBM 360/91.
- Su interés es minimizar los riesgos **RAW**, e implementar Register Renaming en los **WAR** y **WAW**.
- Para comprenderlo consideremos el siguiente código para un procesador MIPS.

```
1  div .d      F0,F2,F4
2  add .d      F6,F0,F8  # WAR hazard por F8 con sub.d
3  s .d        F6,0(R1)
4  sub .d      F8,F10,F14
5  mul .d      F6,F10,F8  # WAW hazard por F6 con add.d
```

- Un escenario para la solución sería disponer de dos registros, que denominaremos S y T que permitan tomar datos y utilizarse para su acceso a partir de ese momento.

# Approach de Tomasulo

- Proviene de un trabajo de implementación de scheduling dinámico en la unidad de Punto Flotante de la IBM 360/91.
- Su interés es minimizar los riesgos **RAW**, e implementar Register Renaming en los **WAR** y **WAW**.
- Para comprenderlo consideremos el siguiente código para un procesador MIPS.

```
1  div .d      F0,F2,F4
2  add .d      F6,F0,F8    # WAR hazard por F8 con sub.d
3  s.d          F6,0(R1)
4  sub .d      F8,F10,F14
5  mul .d      F6,F10,F8  # WAW hazard por F6 con add.d
```

- Un escenario para la solución sería disponer de dos registros, que denominaremos S y T que permitan tomar datos y utilizarse para su acceso a partir de ese momento.

# Approach de Tomasulo

- Proviene de un trabajo de implementación de scheduling dinámico en la unidad de Punto Flotante de la IBM 360/91.
- Su interés es minimizar los riesgos **RAW**, e implementar Register Renaming en los **WAR** y **WAW**.
- Para comprenderlo consideremos el siguiente código para un procesador MIPS.

```
1  div .d      F0,F2,F4
2  add .d      F6,F0,F8    # WAR hazard por F8 con sub.d
3  s .d        F6,0(R1)
4  sub .d      F8,F10,F14
5  mul .d      F6,F10,F8  # WAW hazard por F6 con add.d
```

- Un escenario para la solución sería disponer de dos registros, que denominaremos S y T que permitan tomar datos y utilizarse para su acceso a partir de ese momento.

# Approach de Tomasulo

- El código anterior con la inclusión de estos dos registros auxiliares queda como se indica a continuación, libre de dependencias:

```
1  div .d      F0 , F2 , F4
2  add .d      S , F0 , F8    # WAR hazard por F8 con sub.d
3  s .d        S , 0(R1)
4  sub .d      T , F10 , F14
5  mul .d      F6 , F10 , T  # WAW hazard por F6 con add.d
```

- Además cualquier uso posterior de F8 es reemplazado por el registro temporal.
- El análisis del código de manera de “mirar mas adelante” de cada instrucción en busca de estos riesgos, demanda gran sofisticación (y consecuentemente una mayor complejidad) en el compilador. Eventualmente también algún soporte de hardware.
- Además el compilador puede recurrir a saltos para resolver estas interdependencias. De modo que puede resultar mas perjudicial la solución que el problema.

# Approach de Tomasulo

- El código anterior con la inclusión de estos dos registros auxiliares queda como se indica a continuación, libre de dependencias:

```
1  div .d      F0 , F2 , F4
2  add .d      S , F0 , F8    # WAR hazard por F8 con sub.d
3  s .d        S , 0(R1)
4  sub .d      T , F10 , F14
5  mul .d      F6 , F10 , T  # WAW hazard por F6 con add.d
```

- Además cualquier uso posterior de F8 es reemplazado por el registro temporal.
- El análisis del código de manera de “mirar mas adelante” de cada instrucción en busca de estos riesgos, demanda gran sofisticación (y consecuentemente una mayor complejidad) en el compilador. Eventualmente también algún soporte de hardware.
- Además el compilador puede recurrir a saltos para resolver estas interdependencias. De modo que puede resultar mas perjudicial la solución que el problema.

# Approach de Tomasulo

- El código anterior con la inclusión de estos dos registros auxiliares queda como se indica a continuación, libre de dependencias:

```
1  div .d      F0 , F2 , F4
2  add .d      S , F0 , F8    # WAR hazard por F8 con sub.d
3  s .d        S , 0(R1)
4  sub .d      T , F10 , F14
5  mul .d      F6 , F10 , T  # WAW hazard por F6 con add.d
```

- Además cualquier uso posterior de F8 es reemplazado por el registro temporal.
- El análisis del código de manera de “mirar mas adelante” de cada instrucción en busca de estos riesgos, demanda gran sofisticación (y consecuentemente una mayor complejidad) en el compilador. Eventualmente también algún soporte de hardware.
- Además el compilador puede recurrir a saltos para resolver estas interdependencias. De modo que puede resultar mas perjudicial la solución que el problema.

# Approach de Tomasulo

- El código anterior con la inclusión de estos dos registros auxiliares queda como se indica a continuación, libre de dependencias:

```
1  div .d      F0 , F2 , F4
2  add .d      S , F0 , F8    # WAR hazard por F8 con sub.d
3  s .d        S , 0(R1)
4  sub .d      T , F10 , F14
5  mul .d      F6 , F10 , T  # WAW hazard por F6 con add.d
```

- Además cualquier uso posterior de F8 es reemplazado por el registro temporal.
- El análisis del código de manera de “mirar mas adelante” de cada instrucción en busca de estos riesgos, demanda gran sofisticación (y consecuentemente una mayor complejidad) en el compilador. Eventualmente también algún soporte de hardware.
- Además el compilador puede recurrir a saltos para resolver estas interdependencias. De modo que puede resultar mas perjudicial la solución que el problema.

# Approach de Tomasulo

- El código anterior con la inclusión de estos dos registros auxiliares queda como se indica a continuación, libre de dependencias:

```
1  div .d      F0 , F2 , F4
2  add .d      S , F0 , F8    # WAR hazard por F8 con sub.d
3  s .d        S , 0(R1)
4  sub .d      T , F10 , F14
5  mul .d      F6 , F10 , T  # WAW hazard por F6 con add.d
```

- Además cualquier uso posterior de F8 es reemplazado por el registro temporal.
- El análisis del código de manera de “mirar mas adelante” de cada instrucción en busca de estos riesgos, demanda gran sofisticación (y consecuentemente una mayor complejidad) en el compilador. Eventualmente también algún soporte de hardware.
- Además el compilador puede recurrir a saltos para resolver estas interdependencias. De modo que puede resultar mas perjudicial la solución que el problema.

# La solución: Renaming de registros y la Reservation Station

- Se utilizan bancos o archivo de registros internos agrupados en uno o mas bloques que se denominan Reservation Station. Cada registro posee un tag para identificarlo.
- La Reservation Station que se encarga de buscar los operandos ni bien estén disponibles almacenándolos en estos registros internos.
- A medida que se emiten instrucciones, por cada operando pendiente se renombra el registro que lo contiene a un registro de la Reservation Station.
- De este modo no hay posibilidad de que una instrucción cuya ejecución se adelanta respecto de otra previa en la secuencia del programa, pueda modificar o utilizar un registro de la instrucción previa y que ésta luego use una copia incorrecta del mismo.
- Una vez disponible el operando, la Reservation Station se encarga de su búsqueda y aplicación en cuanto registro destino lo necesite.
- Si un operando destino recibe múltiples escrituras, a Reservation Station solo aplicará la última.
- Si se incluye en la Reservation Station registros en cantidad suficientemente superior a los registros de la arquitectura se puede potencialmente eliminar los bucles infinitos.

# La solución: Renaming de registros y la Reservation Station

- Se utilizan bancos o archivo de registros internos agrupados en uno o mas bloques que se denominan Reservation Station. Cada registro posee un tag para identificarlo.
- La Reservation Station que se encarga de buscar los operandos ni bien estén disponibles almacenándolos en estos registros internos.
- A medida que se emiten instrucciones, por cada operando pendiente se renombra el registro que lo contiene a un registro de la Reservation Station.
- De este modo no hay posibilidad de que una instrucción cuya ejecución se adelanta respecto de otra previa en la secuencia del programa, pueda modificar o utilizar un registro de la instrucción previa y que ésta luego use una copia incorrecta del mismo.
- Una vez disponible el operando, la Reservation Station se encarga de su búsqueda y aplicación en cuanto registro destino lo necesite.
- Si un operando destino recibe múltiples escrituras, a Reservation Station solo aplicará la última.
- Si se incluye en la Reservation Station registros en cantidad suficientemente superior a los registros de la arquitectura se puede potencialmente eliminar los bucles infinitos.

# La solución: Renaming de registros y la Reservation Station

- Se utilizan bancos o archivo de registros internos agrupados en uno o mas bloques que se denominan Reservation Station. Cada registro posee un tag para identificarlo.
- La Reservation Station que se encarga de buscar los operandos ni bien estén disponibles almacenándolos en estos registros internos.
- A medida que se emiten instrucciones, por cada operando pendiente se renombra el registro que lo contiene a un registro de la Reservation Station.
- De este modo no hay posibilidad de que una instrucción cuya ejecución se adelanta respecto de otra previa en la secuencia del programa, pueda modificar o utilizar un registro de la instrucción previa y que ésta luego use una copia incorrecta del mismo.
- Una vez disponible el operando, la Reservation Station se encarga de su búsqueda y aplicación en cuanto registro destino lo necesite.
- Si un operando destino recibe múltiples escrituras, a Reservation Station solo aplicará la última.
- Si se incluye en la Reservation Station registros en cantidad suficientemente superior a los registros de la arquitectura se puede potencialmente eliminar los bucles infinitos.

# La solución: Renaming de registros y la Reservation Station

- Se utilizan bancos o archivo de registros internos agrupados en uno o mas bloques que se denominan Reservation Station. Cada registro posee un tag para identificarlo.
- La Reservation Station que se encarga de buscar los operandos ni bien estén disponibles almacenándolos en estos registros internos.
- A medida que se emiten instrucciones, por cada operando pendiente se renombra el registro que lo contiene a un registro de la Reservation Station.
- De este modo no hay posibilidad de que una instrucción cuya ejecución se adelanta respecto de otra previa en la secuencia del programa, pueda modificar o utilizar un registro de la instrucción previa y que ésta luego use una copia incorrecta del mismo.
- Una vez disponible el operando, la Reservation Station se encarga de su búsqueda y aplicación en cuanto registro destino lo necesite.
- Si un operando destino recibe múltiples escrituras, a Reservation Station solo aplicará la última.
- Si se incluye en la Reservation Station registros en cantidad suficientemente superior a los registros de la arquitectura se puede potencialmente eliminar los bucles infinitos.

# La solución: Renaming de registros y la Reservation Station

- Se utilizan bancos o archivo de registros internos agrupados en uno o mas bloques que se denominan Reservation Station. Cada registro posee un tag para identificarlo.
- La Reservation Station que se encarga de buscar los operandos ni bien estén disponibles almacenándolos en estos registros internos.
- A medida que se emiten instrucciones, por cada operando pendiente se renombra el registro que lo contiene a un registro de la Reservation Station.
- De este modo no hay posibilidad de que una instrucción cuya ejecución se adelanta respecto de otra previa en la secuencia del programa, pueda modificar o utilizar un registro de la instrucción previa y que ésta luego use una copia incorrecta del mismo.
- Una vez disponible el operando, la Reservation Station se encarga de su búsqueda y aplicación en cuanto registro destino lo necesite.
- Si un operando destino recibe múltiples escrituras, a Reservation Station solo aplicará la última.
- Si se incluye en la Reservation Station registros en cantidad suficientemente superior a los registros de la arquitectura se puede potencialmente eliminar los bucles infinitos.

# La solución: Renaming de registros y la Reservation Station

- Se utilizan bancos o archivo de registros internos agrupados en uno o mas bloques que se denominan Reservation Station. Cada registro posee un tag para identificarlo.
- La Reservation Station que se encarga de buscar los operandos ni bien estén disponibles almacenándolos en estos registros internos.
- A medida que se emiten instrucciones, por cada operando pendiente se renombra el registro que lo contiene a un registro de la Reservation Station.
- De este modo no hay posibilidad de que una instrucción cuya ejecución se adelanta respecto de otra previa en la secuencia del programa, pueda modificar o utilizar un registro de la instrucción previa y que ésta luego use una copia incorrecta del mismo.
- Una vez disponible el operando, la Reservation Station se encarga de su búsqueda y aplicación en cuanto registro destino lo necesite.
- Si un operando destino recibe múltiples escrituras, a Reservation Station solo aplicará la última.
- Si se incluye en la Reservation Station registros en cantidad suficientemente superior a los registros de la arquitectura se puede potencialmente eliminar los bucles infinitos.

# La solución: Renaming de registros y la Reservation Station

- Se utilizan bancos o archivo de registros internos agrupados en uno o mas bloques que se denominan Reservation Station. Cada registro posee un tag para identificarlo.
- La Reservation Station que se encarga de buscar los operandos ni bien estén disponibles almacenándolos en estos registros internos.
- A medida que se emiten instrucciones, por cada operando pendiente se renombra el registro que lo contiene a un registro de la Reservation Station.
- De este modo no hay posibilidad de que una instrucción cuya ejecución se adelanta respecto de otra previa en la secuencia del programa, pueda modificar o utilizar un registro de la instrucción previa y que ésta luego use una copia incorrecta del mismo.
- Una vez disponible el operando, la Reservation Station se encarga de su búsqueda y aplicación en cuanto registro destino lo necesite.
- Si un operando destino recibe múltiples escrituras, a Reservation Station solo aplicará la última.
- Si se incluye en la Reservation Station registros en cantidad suficientemente superior a los registros de la arquitectura se puede potencialmente eliminar los bucles infinitos.

# La solución: Renaming de registros y la Reservation Station

- Se utilizan bancos o archivo de registros internos agrupados en uno o mas bloques que se denominan Reservation Station. Cada registro posee un tag para identificarlo.
- La Reservation Station que se encarga de buscar los operandos ni bien estén disponibles almacenándolos en estos registros internos.
- A medida que se emiten instrucciones, por cada operando pendiente se renombra el registro que lo contiene a un registro de la Reservation Station.
- De este modo no hay posibilidad de que una instrucción cuya ejecución se adelanta respecto de otra previa en la secuencia del programa, pueda modificar o utilizar un registro de la instrucción previa y que ésta luego use una copia incorrecta del mismo.
- Una vez disponible el operando, la Reservation Station se encarga de su búsqueda y aplicación en cuanto registro destino lo necesite.
- Si un operando destino recibe múltiples escrituras, a Reservation Station solo aplicará la última.
- Si se incluye en la Reservation Station registros en cantidad suficientemente superior a los registros de la arquitectura se puede potencialmente eliminar los

# Implementación

## Implementación

Para implementar el modelo de Tomasulo se requiere de dos bloques en el pipeline que trabajen de la siguiente manera:

- La etapa de **Envío**, que se encarga de obtener instrucciones desde una cola de prebúsqueda, tratando de ubicarlas en una Reservation Station vacía. La instrucción se envía junto con los datos correspondientes a sus operandos si estos están disponibles en sus registros.
- Si no hay Reservation Station disponible, se tiene un Obstáculo de tipo Estructural, y la instrucción bloquea (stall).
- Si los operandos no están disponibles en los registros, la etapa de **Envío** rastrea cuales son las unidades de Ejecución que los producirán y actualiza sus estructuras internas con esta información.
- Esto pone fin a potenciales riesgos WAR y WAW, ya que los registros se renombran.
- Hay quienes llaman a esta etapa *Dispatch*.

# Implementación

## Implementación

Para implementar el modelo de Tomasulo se requiere de dos bloques en el pipeline que trabajen de la siguiente manera:

- La etapa de **Envío**, que se encarga de obtener instrucciones desde una cola de prebúsqueda, tratando de ubicarlas en una Reservation Station vacía. La instrucción se envía junto con los datos correspondientes a sus operandos si estos están disponibles en sus registros.
- Si no hay Reservation Station disponible, se tiene un Obstáculo de tipo Estructural, y la instrucción bloquea (stall).
- Si los operandos no están disponibles en los registros, la etapa de **Envío** rastrea cuales son las unidades de Ejecución que los producirán y actualiza sus estructuras internas con esta información.
- Esto pone fin a potenciales riesgos WAR y WAW, ya que los registros se renombran.
- Hay quienes llaman a esta etapa *Dispatch*.

# Implementación

## Implementación

Para implementar el modelo de Tomasulo se requiere de dos bloques en el pipeline que trabajen de la siguiente manera:

- La etapa de **Envío**, que se encarga de obtener instrucciones desde una cola de prebúsqueda, tratando de ubicarlas en una Reservation Station vacía. La instrucción se envía junto con los datos correspondientes a sus operandos si estos están disponibles en sus registros.
- Si no hay Reservation Station disponible, se tiene un Obstáculo de tipo Estructural, y la instrucción bloquea (stall).
- Si los operandos no están disponibles en los registros, la etapa de **Envío** rastrea cuales son las unidades de Ejecución que los producirán y actualiza sus estructuras internas con esta información.
- Esto pone fin a potenciales riesgos WAR y WAW, ya que los registros se renombran.
- Hay quienes llaman a esta etapa *Dispatch*.

# Implementación

## Implementación

Para implementar el modelo de Tomasulo se requiere de dos bloques en el pipeline que trabajen de la siguiente manera:

- La etapa de **Envío**, que se encarga de obtener instrucciones desde una cola de prebúsqueda, tratando de ubicarlas en una Reservation Station vacía. La instrucción se envía junto con los datos correspondientes a sus operandos si estos están disponibles en sus registros.
- Si no hay Reservation Station disponible, se tiene un Obstáculo de tipo Estructural, y la instrucción bloquea (stall).
- Si los operandos no están disponibles en los registros, la etapa de **Envío** rastrea cuales son las unidades de Ejecución que los producirán y actualiza sus estructuras internas con esta información.
- Esto pone fin a potenciales riesgos WAR y WAW, ya que los registros se renombran.
- Hay quienes llaman a esta etapa *Dispatch*.

# Implementación

## Implementación

Para implementar el modelo de Tomasulo se requiere de dos bloques en el pipeline que trabajen de la siguiente manera:

- La etapa de **Envío**, que se encarga de obtener instrucciones desde una cola de prebúsqueda, tratando de ubicarlas en una Reservation Station vacía. La instrucción se envía junto con los datos correspondientes a sus operandos si estos están disponibles en sus registros.
- Si no hay Reservation Station disponible, se tiene un Obstáculo de tipo Estructural, y la instrucción bloquea (stall).
- Si los operandos no están disponibles en los registros, la etapa de **Envío** rastrea cuales son las unidades de Ejecución que los producirán y actualiza sus estructuras internas con esta información.
- Esto pone fin a potenciales riesgos WAR y WAW, ya que los registros se renombran.
- Hay quienes llaman a esta etapa *Dispatch*.

# Implementación

## Implementación

Para implementar el modelo de Tomasulo se requiere de dos bloques en el pipeline que trabajen de la siguiente manera:

- La etapa de **Envío**, que se encarga de obtener instrucciones desde una cola de prebúsqueda, tratando de ubicarlas en una Reservation Station vacía. La instrucción se envía junto con los datos correspondientes a sus operandos si estos están disponibles en sus registros.
- Si no hay Reservation Station disponible, se tiene un Obstáculo de tipo Estructural, y la instrucción bloquea (stall).
- Si los operandos no están disponibles en los registros, la etapa de **Envío** rastrea cuales son las unidades de Ejecución que los producirán y actualiza sus estructuras internas con esta información.
- Esto pone fin a potenciales riesgos WAR y WAW, ya que los registros se renombran.
- Hay quienes llaman a esta etapa **Dispatch**.

# Implementación

## Implementación

- La etapa de **Ejecución**, en caso que los operandos no estén disponibles, queda a la espera de su generación, momento en el cual los copia a los registros internos de la Reservation Station que contiene a la instrucción. Esto permite asignar la instrucción a una de las Unidades de Ejecución internas.
- Puede ocurrir que en un ciclo de clock determinado se tornen disponibles varios operandos que desataban sendas instrucciones.
- Si bien hay múltiples unidades de ejecución puede ocurrir que la cantidad de instrucciones de un tipo determinado superen la cantidad de Unidades de Ejecución disponibles.
- En general el orden en que se ejecutan es arbitrario, ya que los registros están renombrados. Esto es estrictamente cierto para todo tipo de instrucciones excepto por las de carga y almacenamiento en memoria.
- Las instrucciones de carga y almacenamiento se ejecutan siempre en dos pasos: Calculo de la dirección efectiva de memoria y acceso al operando. Este último paso puede demorar varios ciclos de clock de acuerdo a la tecnología de la memoria que se accede.
- Ejecutar en orden estas operaciones permite evitar riesgos de memoria.

# Implementación

## Implementación

- La etapa de **Ejecución**, en caso que los operandos no estén disponibles, queda a la espera de su generación, momento en el cual los copia a los registros internos de la Reservation Station que contiene a la instrucción. Esto permite asignar la instrucción a una de las Unidades de Ejecución internas.
- Puede ocurrir que en un ciclo de clock determinado se tornen disponibles varios operandos que destrababan sendas instrucciones.
- Si bien hay múltiples unidades de ejecución puede ocurrir que la cantidad de instrucciones de un tipo determinado superen la cantidad de Unidades de Ejecución disponibles.
- En general el orden en que se ejecutan es arbitrario, ya que los registros están renombrados. Esto es estrictamente cierto para todo tipo de instrucciones excepto por las de carga y almacenamiento en memoria.
- Las instrucciones de carga y almacenamiento se ejecutan siempre en dos pasos: Calculo de la dirección efectiva de memoria y acceso al operando. Este último paso puede demorar varios ciclos de clock de acuerdo a la tecnología de la memoria que se accede.
- Ejecutar en orden estas operaciones permite evitar riesgos de memoria.

# Implementación

## Implementación

- La etapa de **Ejecución**, en caso que los operandos no estén disponibles, queda a la espera de su generación, momento en el cual los copia a los registros internos de la Reservation Station que contiene a la instrucción. Esto permite asignar la instrucción a una de las Unidades de Ejecución internas.
- Puede ocurrir que en un ciclo de clock determinado se tornen disponibles varios operandos que desataban sendas instrucciones.
- Si bien hay múltiples unidades de ejecución puede ocurrir que la cantidad de instrucciones de un tipo determinado superen la cantidad de Unidades de Ejecución disponibles.
- En general el orden en que se ejecutan es arbitrario, ya que los registros están renombrados. Esto es estrictamente cierto para todo tipo de instrucciones excepto por las de carga y almacenamiento en memoria.
- Las instrucciones de carga y almacenamiento se ejecutan siempre en dos pasos: Calculo de la dirección efectiva de memoria y acceso al operando. Este último paso puede demorar varios ciclos de clock de acuerdo a la tecnología de la memoria que se accede.
- Ejecutar en orden estas operaciones permite evitar riesgos de memoria.

# Implementación

## Implementación

- La etapa de **Ejecución**, en caso que los operandos no estén disponibles, queda a la espera de su generación, momento en el cual los copia a los registros internos de la Reservation Station que contiene a la instrucción. Esto permite asignar la instrucción a una de las Unidades de Ejecución internas.
- Puede ocurrir que en un ciclo de clock determinado se tornen disponibles varios operandos que desataban sendas instrucciones.
- Si bien hay múltiples unidades de ejecución puede ocurrir que la cantidad de instrucciones de un tipo determinado superen la cantidad de Unidades de Ejecución disponibles.
- En general el orden en que se ejecutan es arbitrario, ya que los registros están renombrados. Esto es estrictamente cierto para todo tipo de instrucciones excepto por las de carga y almacenamiento en memoria.
- Las instrucciones de carga y almacenamiento se ejecutan siempre en dos pasos: Calculo de la dirección efectiva de memoria y acceso al operando. Este último paso puede demorar varios ciclos de clock de acuerdo a la tecnología de la memoria que se accede.
- Ejecutar en orden estas operaciones permite evitar riesgos de memoria.

# Implementación

## Implementación

- La etapa de **Ejecución**, en caso que los operandos no estén disponibles, queda a la espera de su generación, momento en el cual los copia a los registros internos de la Reservation Station que contiene a la instrucción. Esto permite asignar la instrucción a una de las Unidades de Ejecución internas.
- Puede ocurrir que en un ciclo de clock determinado se tornen disponibles varios operandos que desataban sendas instrucciones.
- Si bien hay múltiples unidades de ejecución puede ocurrir que la cantidad de instrucciones de un tipo determinado superen la cantidad de Unidades de Ejecución disponibles.
- En general el orden en que se ejecutan es arbitrario, ya que los registros están renombrados. Esto es estrictamente cierto para todo tipo de instrucciones excepto por las de carga y almacenamiento en memoria.
- Las instrucciones de carga y almacenamiento se ejecutan siempre en dos pasos: Calculo de la dirección efectiva de memoria y acceso al operando. Este último paso puede demorar varios ciclos de clock de acuerdo a la tecnología de la memoria que se accede.
- Ejecutar en orden estas operaciones permite evitar riesgos de memoria.

# Implementación

## Implementación

- La etapa de **Ejecución**, en caso que los operandos no estén disponibles, queda a la espera de su generación, momento en el cual los copia a los registros internos de la Reservation Station que contiene a la instrucción. Esto permite asignar la instrucción a una de las Unidades de Ejecución internas.
- Puede ocurrir que en un ciclo de clock determinado se tornen disponibles varios operandos que desataban sendas instrucciones.
- Si bien hay múltiples unidades de ejecución puede ocurrir que la cantidad de instrucciones de un tipo determinado superen la cantidad de Unidades de Ejecución disponibles.
- En general el orden en que se ejecutan es arbitrario, ya que los registros están renombrados. Esto es estrictamente cierto para todo tipo de instrucciones excepto por las de carga y almacenamiento en memoria.
- Las instrucciones de carga y almacenamiento se ejecutan siempre en dos pasos: Calculo de la dirección efectiva de memoria y acceso al operando. Este último paso puede demorar varios ciclos de clock de acuerdo a la tecnología de la memoria que se accede.
- Ejecutar en orden estas operaciones permite evitar riesgos de memoria.

# Implementación

## Implementación

- La etapa de **Ejecución**, en caso que los operandos no estén disponibles, queda a la espera de su generación, momento en el cual los copia a los registros internos de la Reservation Station que contiene a la instrucción. Esto permite asignar la instrucción a una de las Unidades de Ejecución internas.
- Puede ocurrir que en un ciclo de clock determinado se tornen disponibles varios operandos que desataban sendas instrucciones.
- Si bien hay múltiples unidades de ejecución puede ocurrir que la cantidad de instrucciones de un tipo determinado superen la cantidad de Unidades de Ejecución disponibles.
- En general el orden en que se ejecutan es arbitrario, ya que los registros están renombrados. Esto es estrictamente cierto para todo tipo de instrucciones excepto por las de carga y almacenamiento en memoria.
- Las instrucciones de carga y almacenamiento se ejecutan siempre en dos pasos: Calculo de la dirección efectiva de memoria y acceso al operando. Este último paso puede demorar varios ciclos de clock de acuerdo a la tecnología de la memoria que se accede.
- Ejecutar en orden estas operaciones permite evitar riesgos de memoria.

# Implementación

## Implementación

- No se permite ejecutar instrucciones hasta que hayan sido resueltos todos los branches previos.
- La etapa de **Escritura de Resultado**, en caso que los operandos no estén disponibles, queda a la espera de su generación, momento en el cual los copia a los registros internos de la Reservation Station que contiene a la instrucción. Esto permite asignar la instrucción a una de las Unidades de Ejecución internas.
- Si se registra una excepción como resultado de la ejecución de una instrucción, se guarda este registro pero no se genera la excepción sino hasta que se entre en la etapa de escritura del resultado de la instrucción.

# Implementación

## Implementación

- No se permite ejecutar instrucciones hasta que hayan sido resueltos todos los branches previos.
- La etapa de **Escritura de Resultado**, en caso que los operandos no estén disponibles, queda a la espera de su generación, momento en el cual los copia a los registros internos de la Reservation Station que contiene a la instrucción. Esto permite asignar la instrucción a una de las Unidades de Ejecución internas.
- Si se registra una excepción como resultado de la ejecución de una instrucción, se guarda este registro pero no se genera la excepción sino hasta que se entre en la etapa de escritura del resultado de la instrucción.

# Implementación

## Implementación

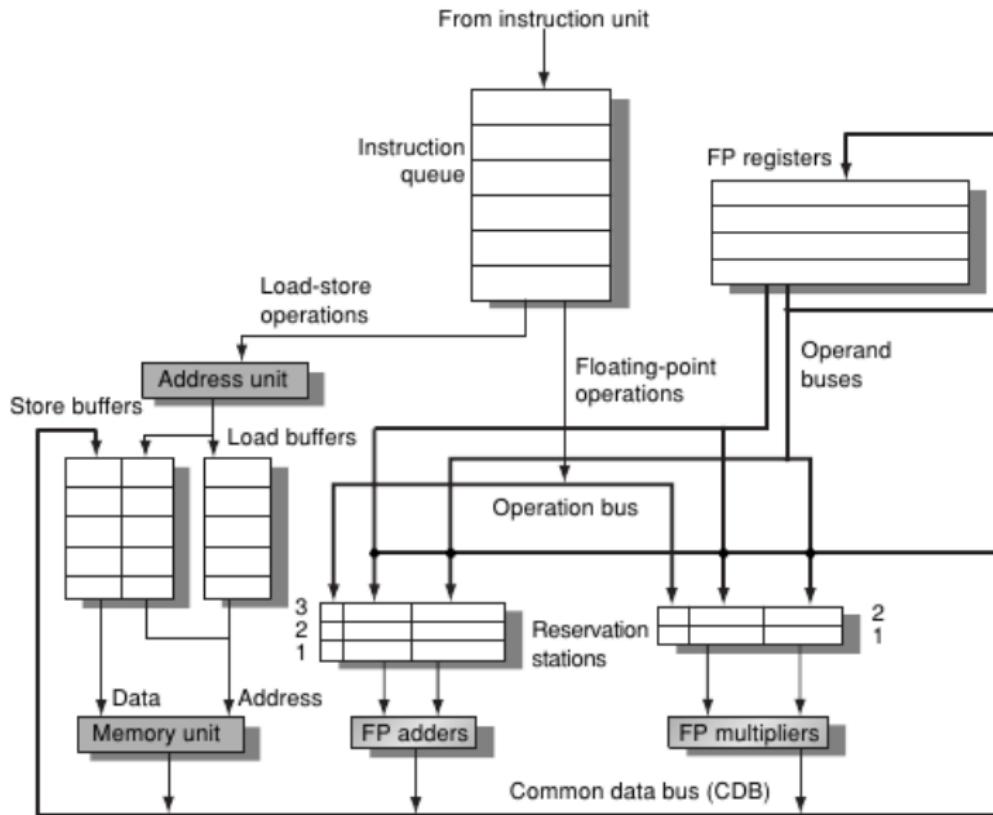
- No se permite ejecutar instrucciones hasta que hayan sido resueltos todos los branches previos.
- La etapa de **Escritura de Resultado**, en caso que los operandos no estén disponibles, queda a la espera de su generación, momento en el cual los copia a los registros internos de la Reservation Station que contiene a la instrucción. Esto permite asignar la instrucción a una de las Unidades de Ejecución internas.
- Si se registra una excepción como resultado de la ejecución de una instrucción, se guarda este registro pero no se genera la excepción sino hasta que se entre en la etapa de escritura del resultado de la instrucción.

# Implementación

## Implementación

- No se permite ejecutar instrucciones hasta que hayan sido resueltos todos los branches previos.
- La etapa de **Escritura de Resultado**, en caso que los operandos no estén disponibles, queda a la espera de su generación, momento en el cual los copia a los registros internos de la Reservation Station que contiene a la instrucción. Esto permite asignar la instrucción a una de las Unidades de Ejecución internas.
- Si se registra una excepción como resultado de la ejecución de una instrucción, se guarda este registro pero no se genera la excepción sino hasta que se entre en la etapa de escritura del resultado de la instrucción.

# Detalle de lo visto



# Contenido

## 1 Paralelismo a Nivel de Instrucción

- Pipeline
- Unidades de Predicción de saltos
- Superscalar
- Scheduling Dinámico

## 2 Ejecución Fuera de Orden

- Prehistoria
- Solución Actual
- **Especulación por Hardware**
- Casos prácticos que mezclan todo lo visto

## 3 Procesadores Multithread

## 4 Multicore y Manycore

# La conspiración de los branches (el regreso)

- A medida que avanzamos en incrementar el paralelismo a Nivel de Instrucciones, el control de las dependencias del programa se complica proporcionalmente.
- Los Branch Target Buffers fueron útiles para evitar stalls en las instrucciones de salto, pero a medida que la ejecución fuera de orden se fue sofisticando considerablemente, es necesario repensar su funcionamiento.
- En general ejecución especulativa es la capacidad de una arquitectura para ejecutar instrucciones sin tener aún los resultados de sus dependencias, sino simplemente asumiendo que el resultado será uno determinado, y lo mas importante, tener la capacidad de deshacer la operación si la especulación no fue correcta.
- Una vez que se tiene el resultado la instrucción deja de ser especulativa y con esta certeza está en condiciones de escribir en el registro destino.

# La conspiración de los branches (el regreso)

- A medida que avanzamos en incrementar el paralelismo a Nivel de Instrucciones, el control de las dependencias del programa se complica proporcionalmente.
- Los Branch Target Buffers fueron útiles para evitar stalls en las instrucciones de salto, pero a medida que la ejecución fuera de orden se fue sofisticando considerablemente, es necesario repensar su funcionamiento.
- En general ejecución especulativa es la capacidad de una arquitectura para ejecutar instrucciones sin tener aún los resultados de sus dependencias, sino simplemente asumiendo que el resultado será uno determinado, y lo mas importante, tener la capacidad de deshacer la operación si la especulación no fue correcta.
- Una vez que se tiene el resultado la instrucción deja de ser especulativa y con esta certeza está en condiciones de escribir en el registro destino.

# La conspiración de los branches (el regreso)

- A medida que avanzamos en incrementar el paralelismo a Nivel de Instrucciones, el control de las dependencias del programa se complica proporcionalmente.
- Los Branch Target Buffers fueron útiles para evitar stalls en las instrucciones de salto, pero a medida que la ejecución fuera de orden se fue sofisticando considerablemente, es necesario repensar su funcionamiento.
- En general ejecución especulativa es la capacidad de una arquitectura para ejecutar instrucciones sin tener aún los resultados de sus dependencias, sino simplemente asumiendo que el resultado será uno determinado, y lo mas importante, tener la capacidad de deshacer la operación si la especulación no fue correcta.
- Una vez que se tiene el resultado la instrucción deja de ser especulativa y con esta certeza está en condiciones de escribir en el registro destino.

# La conspiración de los branches (el regreso)

- A medida que avanzamos en incrementar el paralelismo a Nivel de Instrucciones, el control de las dependencias del programa se complica proporcionalmente.
- Los Branch Target Buffers fueron útiles para evitar stalls en las instrucciones de salto, pero a medida que la ejecución fuera de orden se fue sofisticando considerablemente, es necesario repensar su funcionamiento.
- En general ejecución especulativa es la capacidad de una arquitectura para ejecutar instrucciones sin tener aún los resultados de sus dependencias, sino simplemente asumiendo que el resultado será uno determinado, y lo mas importante, tener la capacidad de deshacer la operación si la especulación no fue correcta.
- Una vez que se tiene el resultado la instrucción deja de ser especulativa y con esta certeza está en condiciones de escribir en el registro destino.

# La conspiración de los branches (el regreso)

- A medida que avanzamos en incrementar el paralelismo a Nivel de Instrucciones, el control de las dependencias del programa se complica proporcionalmente.
- Los Branch Target Buffers fueron útiles para evitar stalls en las instrucciones de salto, pero a medida que la ejecución fuera de orden se fue sofisticando considerablemente, es necesario repensar su funcionamiento.
- En general ejecución especulativa es la capacidad de una arquitectura para ejecutar instrucciones sin tener aún los resultados de sus dependencias, sino simplemente asumiendo que el resultado será uno determinado, y lo mas importante, tener la capacidad de deshacer la operación si la especulación no fue correcta.
- Una vez que se tiene el resultado la instrucción deja de ser especulativa y con esta certeza está en condiciones de escribir en el registro destino.

# Reordenando los resultados

- Este tipo de ejecución especulativa hace que se tengan previamente almacenados resultados de instrucciones posteriores que luego deben impactarse en sus operandos destino.
- El commit de los resultados debe hacerse en orden.
- Esta es la función del ReOrder Buffer (ROB).
- Igual que la Reservation Station de Tomasulo, el ReOrder Buffer agrega registros en los cuales se van almacenando los resultados de las instrucciones ejecutadas en base a especulación por hardware.
- El resultado permanecerá en el ROB desde que se obtenga el resultado hasta que se copie (commit) en el operando destino.
- La diferencia con el algoritmo de Tomasulo, es que éste ponía el resultado en registro directamente, y a partir de entonces el resto de las instrucciones lo tenía disponible. El ROB no lo escribe, sino hasta el commit. Y durante ese lapso que al especular se puede extender, el registro de la arquitectura no tiene el valor.

# Reordenando los resultados

- Este tipo de ejecución especulativa hace que se tengan pre almacenados resultados de instrucciones posteriores que luego deben impactarse en sus operandos destino
- El commit de los resultados debe hacerse en orden.
- Esta es la función del ReOrder Buffer (ROB).
- Igual que la Reservation Station de Tomasulo, el ReOrder Buffer agrega registros en los cuales se van almacenando los resultados de las instrucciones ejecutadas en base a especulación por hardware.
- El resultado permanecerá en el ROB desde que se obtenga el resultado hasta que se copie (commit) en el operando destino.
- La diferencia con el algoritmo de Tomasulo, es que éste ponía el resultado en registro directamente, y a partir de entonces el resto de las instrucciones lo tenía disponible. El ROB no lo escribe, sino hasta el commit. Y durante ese lapso que al especular se puede extender, el registro de la arquitectura no tiene el valor.

# Reordenando los resultados

- Este tipo de ejecución especulativa hace que se tengan pre almacenados resultados de instrucciones posteriores que luego deben impactarse en sus operandos destino
- El commit de los resultados debe hacerse en orden.
- Esta es la función del ReOrder Buffer (ROB).
- Igual que la Reservation Station de Tomasulo, el ReOrder Buffer agrega registros en los cuales se van almacenando los resultados de las instrucciones ejecutadas en base a especulación por hardware.
- El resultado permanecerá en el ROB desde que se obtenga el resultado hasta que se copie (commit) en el operando destino.
- La diferencia con el algoritmo de Tomasulo, es que éste ponía el resultado en registro directamente, y a partir de entonces el resto de las instrucciones lo tenía disponible. El ROB no lo escribe, sino hasta el commit. Y durante ese lapso que al especular se puede extender, el registro de la arquitectura no tiene el valor.

# Reordenando los resultados

- Este tipo de ejecución especulativa hace que se tengan previamente almacenados resultados de instrucciones posteriores que luego deben impactarse en sus operandos destino.
- El commit de los resultados debe hacerse en orden.
- Esta es la función del ReOrder Buffer (ROB).
- Igual que la Reservation Station de Tomasulo, el ReOrder Buffer agrega registros en los cuales se van almacenando los resultados de las instrucciones ejecutadas en base a especulación por hardware.
- El resultado permanecerá en el ROB desde que se obtenga el resultado hasta que se copie (commit) en el operando destino.
- La diferencia con el algoritmo de Tomasulo, es que éste ponía el resultado en registro directamente, y a partir de entonces el resto de las instrucciones lo tenía disponible. El ROB no lo escribe, sino hasta el commit. Y durante ese lapso que al especular se puede extender, el registro de la arquitectura no tiene el valor.

# Reordenando los resultados

- Este tipo de ejecución especulativa hace que se tengan previamente almacenados resultados de instrucciones posteriores que luego deben impactarse en sus operandos destino.
- El commit de los resultados debe hacerse en orden.
- Esta es la función del ReOrder Buffer (ROB).
- Igual que la Reservation Station de Tomasulo, el ReOrder Buffer agrega registros en los cuales se van almacenando los resultados de las instrucciones ejecutadas en base a especulación por hardware.
- El resultado permanecerá en el ROB desde que se obtenga el resultado hasta que se copie (commit) en el operando destino.
- La diferencia con el algoritmo de Tomasulo, es que éste ponía el resultado en registro directamente, y a partir de entonces el resto de las instrucciones lo tenía disponible. El ROB no lo escribe, sino hasta el commit. Y durante ese lapso que al especular se puede extender, el registro de la arquitectura no tiene el valor.

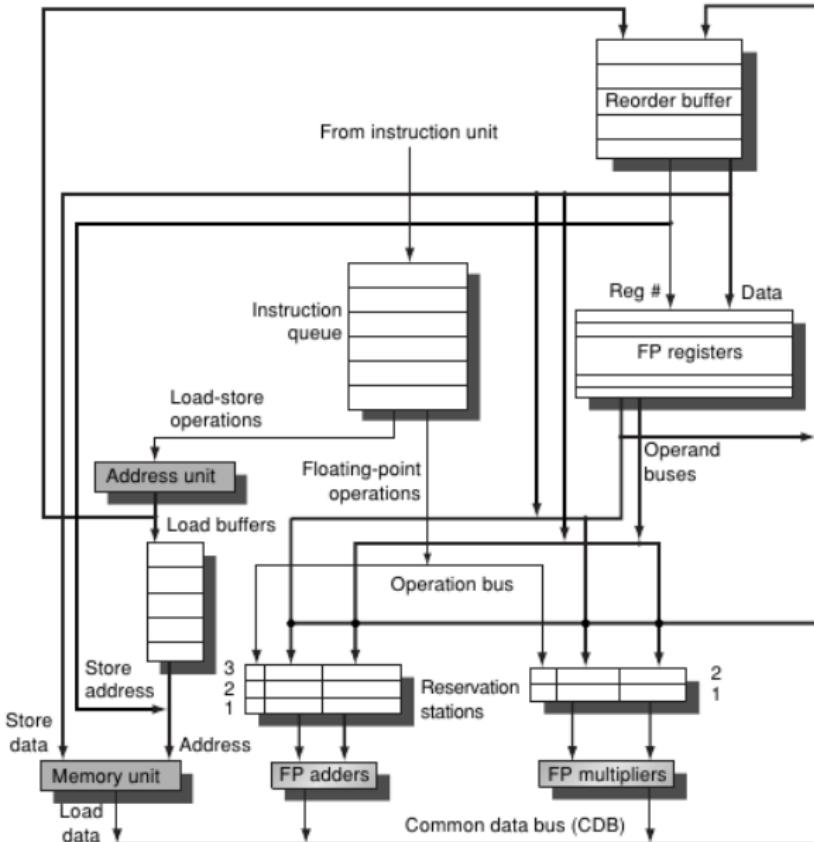
# Reordenando los resultados

- Este tipo de ejecución especulativa hace que se tengan previamente almacenados resultados de instrucciones posteriores que luego deben impactarse en sus operandos destino.
- El commit de los resultados debe hacerse en orden.
- Esta es la función del ReOrder Buffer (ROB).
- Igual que la Reservation Station de Tomasulo, el ReOrder Buffer agrega registros en los cuales se van almacenando los resultados de las instrucciones ejecutadas en base a especulación por hardware.
- El resultado permanecerá en el ROB desde que se obtenga el resultado hasta que se copie (commit) en el operando destino.
- La diferencia con el algoritmo de Tomasulo, es que éste ponía el resultado en registro directamente, y a partir de entonces el resto de las instrucciones lo tenía disponible. El ROB no lo escribe, sino hasta el commit. Y durante ese lapso que al especular se puede extender, el registro de la arquitectura no tiene el valor.

# Reordenando los resultados

- Este tipo de ejecución especulativa hace que se tengan previamente almacenados resultados de instrucciones posteriores que luego deben impactarse en sus operandos destino.
- El commit de los resultados debe hacerse en orden.
- Esta es la función del ReOrder Buffer (ROB).
- Igual que la Reservation Station de Tomasulo, el ReOrder Buffer agrega registros en los cuales se van almacenando los resultados de las instrucciones ejecutadas en base a especulación por hardware.
- El resultado permanecerá en el ROB desde que se obtenga el resultado hasta que se copie (commit) en el operando destino.
- La diferencia con el algoritmo de Tomasulo, es que éste ponía el resultado en registro directamente, y a partir de entonces el resto de las instrucciones lo tenía disponible. El ROB no lo escribe, sino hasta el commit. Y durante ese lapso que al especular se puede extender, el registro de la arquitectura no tiene el valor.

# ReOrder Buffer (ROB)



# Estructuras asociadas al ROB

Cada entrada del ROB contiene la siguiente información.

**Tipo de Instrucción** Indica si la instrucción es un branch (y no tiene resultado destino), un memory store (en cuyo caso se requiere calcular la dirección del operando), o una instrucción cualquiera de la ALU en cuyo caso el resultado irá a parar a un registro de la arquitectura.

**Destino** Contiene el número de registro de la arquitectura en donde se guardará el resultado (En el caso de memory loads u otra operación de ALU cualquiera sea), o la dirección de memoria en caso de ser un memory store

**Valor** Almacena el resultado de la operación hasta el commit.

**Ready** Indica que la instrucción se ha completado y su resultado está disponible.

# Estructuras asociadas al ROB

Cada entrada del ROB contiene la siguiente información.

**Tipo de Instrucción** Indica si la instrucción es un branch (y no tiene resultado destino), un memory store (en cuyo caso se requiere calcular la dirección del operando), o una instrucción cualquiera de la ALU en cuyo caso el resultado irá a parar a un registro de la arquitectura.

**Destino** Contiene el número de registro de la arquitectura en donde se guardará el resultado (En el caso de memory loads u otra operación de ALU cualquiera sea), o la dirección de memoria en caso de ser un memory store

**Valor** Almacena el resultado de la operación hasta el commit.

**Ready** Indica que la instrucción se ha completado y su resultado está disponible.

# Estructuras asociadas al ROB

Cada entrada del ROB contiene la siguiente información.

**Tipo de Instrucción** Indica si la instrucción es un branch (y no tiene resultado destino), un memory store (en cuyo caso se requiere calcular la dirección del operando), o una instrucción cualquiera de la ALU en cuyo caso el resultado irá a parar a un registro de la arquitectura.

**Destino** Contiene el número de registro de la arquitectura en donde se guardará el resultado (En el caso de memory loads u otra operación de ALU cualquiera sea), o la dirección de memoria en caso de ser un memory store

**Valor** Almacena el resultado de la operación hasta el commit.

**Ready** Indica que la instrucción se ha completado y su resultado está disponible.

# Estructuras asociadas al ROB

Cada entrada del ROB contiene la siguiente información.

**Tipo de Instrucción** Indica si la instrucción es un branch (y no tiene resultado destino), un memory store (en cuyo caso se requiere calcular la dirección del operando), o una instrucción cualquiera de la ALU en cuyo caso el resultado irá a parar a un registro de la arquitectura.

**Destino** Contiene el número de registro de la arquitectura en donde se guardará el resultado (En el caso de memory loads u otra operación de ALU cualquiera sea), o la dirección de memoria en caso de ser un memory store

**Valor** Almacena el resultado de la operación hasta el commit.

**Ready** Indica que la instrucción se ha completado y su resultado está disponible.

# Estructuras asociadas al ROB

Cada entrada del ROB contiene la siguiente información.

**Tipo de Instrucción** Indica si la instrucción es un branch (y no tiene resultado destino), un memory store (en cuyo caso se requiere calcular la dirección del operando), o una instrucción cualquiera de la ALU en cuyo caso el resultado irá a parar a un registro de la arquitectura.

**Destino** Contiene el número de registro de la arquitectura en donde se guardará el resultado (En el caso de memory loads u otra operación de ALU cualquiera sea), o la dirección de memoria en caso de ser un memory store

**Valor** Almacena el resultado de la operación hasta el commit.

**Ready** Indica que la instrucción se ha completado y su resultado está disponible.

# Implementación

Para implementar espectación por hardware se requieren de los siguientes bloques en el pipeline:

## Envío

- La etapa de Envío, que se encarga de obtener instrucciones desde una cola de prebúsqueda, tratando de ubicarlas en una Reservation Station vacía, y un slot en el ROB. La instrucción se envía junto con los datos correspondientes a sus operandos si estos están disponibles en sus registros.
- Si no hay Reservation Station disponible, o no hay un slot disponible en el ROB, se tiene un Obstáculo de tipo Estructural, y la instrucción bloquea (stall).
- Se actualizan las estructuras internas con la información pertinente.

# Implementación

Para implementar especulación por hardware se requieren de los siguientes bloques en el pipeline:

## Envío

- La etapa de **Envío**, que se encarga de obtener instrucciones desde una cola de prebúsqueda, tratando de ubicarlas en una Reservation Station vacía, y un slot en el ROB. La instrucción se envía junto con los datos correspondientes a sus operandos si estos están disponibles en sus registros.
- Si no hay Reservation Station disponible, o no hay un slot disponible en el ROB, se tiene un Obstáculo de tipo Estructural, y la instrucción bloquea (stall).
- Se actualizan las estructuras internas con la información pertinente.

# Implementación

Para implementar especulación por hardware se requieren de los siguientes bloques en el pipeline:

## Envío

- La etapa de **Envío**, que se encarga de obtener instrucciones desde una cola de prebúsqueda, tratando de ubicarlas en una Reservation Station vacía, y un slot en el ROB. La instrucción se envía junto con los datos correspondientes a sus operandos si estos están disponibles en sus registros.
- Si no hay Reservation Station disponible, o no hay un slot disponible en el ROB, se tiene un Obstáculo de tipo Estructural, y la instrucción bloquea (stall).
- Se actualizan las estructuras internas con la información pertinente.

# Implementación

Para implementar especulación por hardware se requieren de los siguientes bloques en el pipeline:

## Envío

- La etapa de **Envío**, que se encarga de obtener instrucciones desde una cola de prebúsqueda, tratando de ubicarlas en una Reservation Station vacía, y un slot en el ROB. La instrucción se envía junto con los datos correspondientes a sus operandos si estos están disponibles en sus registros.
- Si no hay Reservation Station disponible, o no hay un slot disponible en el ROB, se tiene un Obstáculo de tipo Estructural, y la instrucción bloquea (stall).
- Se actualizan las estructuras internas con la información pertinente.

# Implementación

## Ejecución

- Si faltan operandos monitorea por el Bus de datos interno del procesador hasta que éstos estén disponibles.
- Chequea riesgos RAW.
- Ejecuta las operaciones en las que tiene los operandos en la RS.
- Hay instrucciones que pueden demorar varios ciclos de clock

## Write Result

# Implementación

## Ejecución

- Si faltan operandos monitorea por el Bus de datos interno del procesador hasta que éstos estén disponibles.
- Chequea riesgos RAW.
- Ejecuta las operaciones en las que tiene los operandos en la RS.
- Hay instrucciones que pueden demorar varios ciclos de clock

## Write Result

# Implementación

## Ejecución

- Si faltan operandos monitorea por el Bus de datos interno del procesador hasta que éstos estén disponibles.
- Chequea riesgos RAW.
- Ejecuta las operaciones en las que tiene los operandos en la RS.
- Hay instrucciones que pueden demorar varios ciclos de clock

## Write Result

# Implementación

## Ejecución

- Si faltan operandos monitorea por el Bus de datos interno del procesador hasta que éstos estén disponibles.
- Chequea riesgos RAW.
- Ejecuta las operaciones en las que tiene los operandos en la RS.
- Hay instrucciones que pueden demorar varios ciclos de clock

## Write Result

● Escribe el resultado en el slot correspondiente del ROB, una vez disponible.

# Implementación

## Ejecución

- Si faltan operandos monitorea por el Bus de datos interno del procesador hasta que éstos estén disponibles.
- Chequea riesgos RAW.
- Ejecuta las operaciones en las que tiene los operandos en la RS.
- Hay instrucciones que pueden demorar varios ciclos de clock

## Write Result

• Escribe el resultado en el slot correspondiente del ROB, una vez disponible.

• Si alguna RS espera el resultado, también lo escribe en el registro correspondiente.

# Implementación

## Ejecución

- Si faltan operandos monitorea por el Bus de datos interno del procesador hasta que éstos estén disponibles.
- Chequea riesgos RAW.
- Ejecuta las operaciones en las que tiene los operandos en la RS.
- Hay instrucciones que pueden demorar varios ciclos de clock

## Write Result

- Escribe el resultado en el slot correspondiente del ROB, una vez disponible.
- Si alguna RS espera el resultado, también lo escribe en el registro correspondiente.
- En el caso de un memory store escribe el resultado en el campo valor del registro correspondiente del ROB

# Implementación

## Ejecución

- Si faltan operandos monitorea por el Bus de datos interno del procesador hasta que éstos estén disponibles.
- Chequea riesgos RAW.
- Ejecuta las operaciones en las que tiene los operandos en la RS.
- Hay instrucciones que pueden demorar varios ciclos de clock

## Write Result

- Escribe el resultado en el slot correspondiente del ROB, una vez disponible.
- Si alguna RS espera el resultado, también lo escribe en el registro correspondiente.
- En el caso de un memory store escribe el resultado en el campo valor del registro correspondiente del ROB

# Implementación

## Ejecución

- Si faltan operandos monitorea por el Bus de datos interno del procesador hasta que éstos estén disponibles.
- Chequea riesgos RAW.
- Ejecuta las operaciones en las que tiene los operandos en la RS.
- Hay instrucciones que pueden demorar varios ciclos de clock

## Write Result

- Escribe el resultado en el slot correspondiente del ROB, una vez disponible.
- Si alguna RS espera el resultado, también lo escribe en el registro correspondiente.
- En el caso de un memory store escribe el resultado en el campo valor del registro correspondiente del ROB

# Implementación

## Commit

- Es la fase final de completamiento de la instrucción. A partir de aquí el resultado solo estará en el operando destino.
- Si se trata de un branch con predicción, incorrecta, flushea la entrada del ROB y recomienza a partir de la dirección sucesora correcta del branch.
- Para cualquier instrucción que finaliza correctamente (excepto memory stores) y para los branches con predicción correcta, se copia el valor del ROB en el operando destino.
- Si la operación es un memory store es similar solo que se escribe en una dirección de memoria.

# Implementación

## Commit

- Es la fase final de completamiento de la instrucción. A partir de aquí el resultado solo estará en el operando destino.
- Si se trata de un branch con predicción, incorrecta, flushea la entrada del ROB y recomienza a partir de la dirección sucesora correcta del branch.
- Para cualquier instrucción que finaliza correctamente (excepto memory stores) y para los branches con predicción correcta, se copia el valor del ROB en el operando destino.
- Si la operación es un memory store es similar solo que se escribe en una dirección de memoria.

# Implementación

## Commit

- Es la fase final de completamiento de la instrucción. A partir de aquí el resultado solo estará en el operando destino.
- Si se trata de un branch con predicción, incorrecta, flushea la entrada del ROB y recomienza a partir de la dirección sucesora correcta del branch.
- Para cualquier instrucción que finaliza correctamente (excepto memory stores) y para los branches con predicción correcta, se copia el valor del ROB en el operando destino.
- Si la operación es un memory store es similar solo que se escribe en una dirección de memoria.

# Implementación

## Commit

- Es la fase final de completamiento de la instrucción. A partir de aquí el resultado solo estará en el operando destino.
- Si se trata de un branch con predicción, incorrecta, flushea la entrada del ROB y recomienza a partir de la dirección sucesora correcta del branch.
- Para cualquier instrucción que finaliza correctamente (excepto memory stores) y para los branches con predicción correcta, se copia el valor del ROB en el operando destino.
- Si la operación es un memory store es similar solo que se escribe en una dirección de memoria.

# Implementación

## Commit

- Es la fase final de completamiento de la instrucción. A partir de aquí el resultado solo estará en el operando destino.
- Si se trata de un branch con predicción, incorrecta, flushea la entrada del ROB y recomienza a partir de la dirección sucesora correcta del branch.
- Para cualquier instrucción que finaliza correctamente (excepto memory stores) y para los branches con predicción correcta, se copia el valor del ROB en el operando destino.
- Si la operación es un memory store es similar solo que se escribe en una dirección de memoria.

# Contenido

## 1 Paralelismo a Nivel de Instrucción

- Pipeline
- Unidades de Predicción de saltos
- Superscalar
- Scheduling Dinámico

## 2 Ejecución Fuera de Orden

- Prehistoria
- Solución Actual
- Especulación por Hardware
- Casos prácticos que mezclan todo lo visto

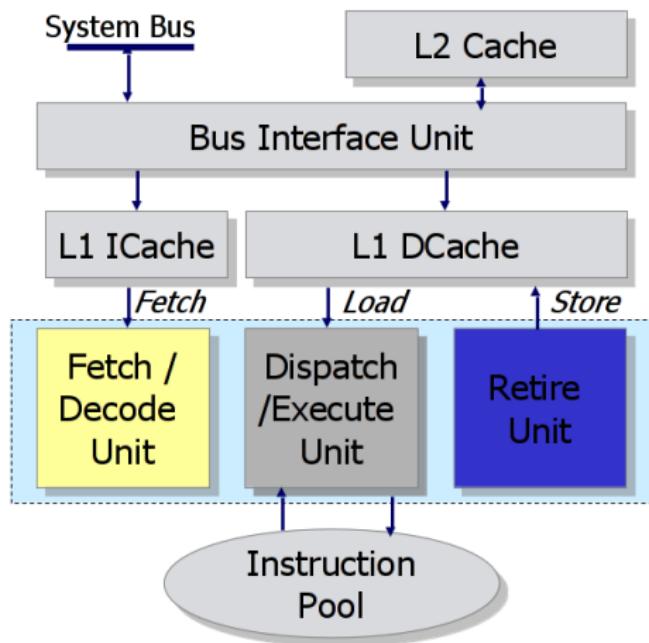
## 3 Procesadores Multithread

## 4 Multicore y Manycore

# Intel - Microarquitectura P6

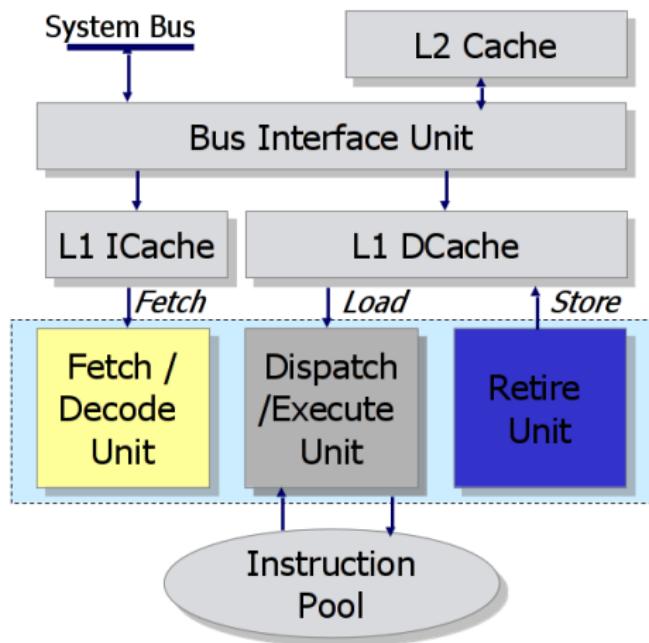
- En 1993 Intel presenta el procesador Pentium Pro que incluye una serie de innovaciones.
- Con este procesador se inaugura la Microarquitectura que se llamó P6
- Los sucesivos procesadores que mejoran este modelo dentro de P6 son:
  - Pentium II, Pentium II Xeon,
  - Celeron
  - Pentium III, Pentium III Xeon

# Intel - Three Cores Engine



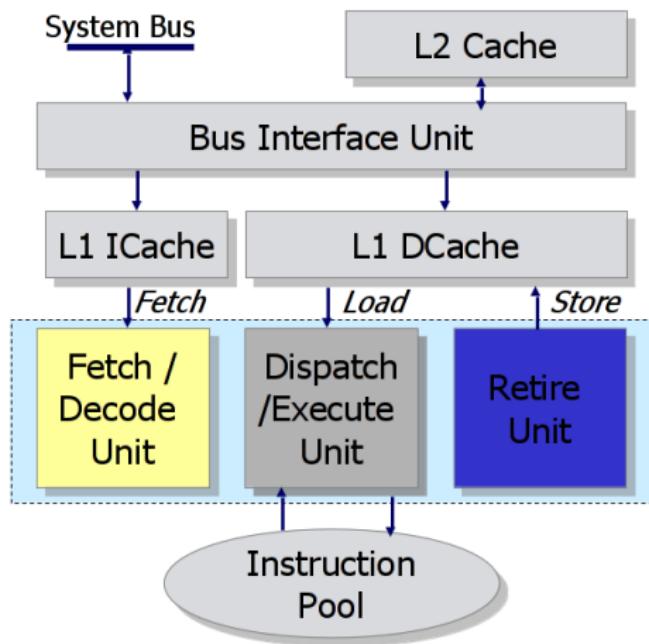
- Emplea Dynamic Instruction Scheduling
- Basado en una ventana de instrucciones y no en un pipeline superescalar.
- Las instrucciones se traducen en micro operaciones básicas (uops)
- Las uops ingresan a un pool (ventana) en donde se mantienen para su ejecución
- Los tres cores tienen plena visibilidad de esa ventana de ejecución
- Implementa ejecución fuera de orden y ejecución especulativa.
- La unidad de despacho y ejecución mantiene el modelo superescalar y lo combina con un súper pipeline de 20 etapas

# Intel - Three Cores Engine



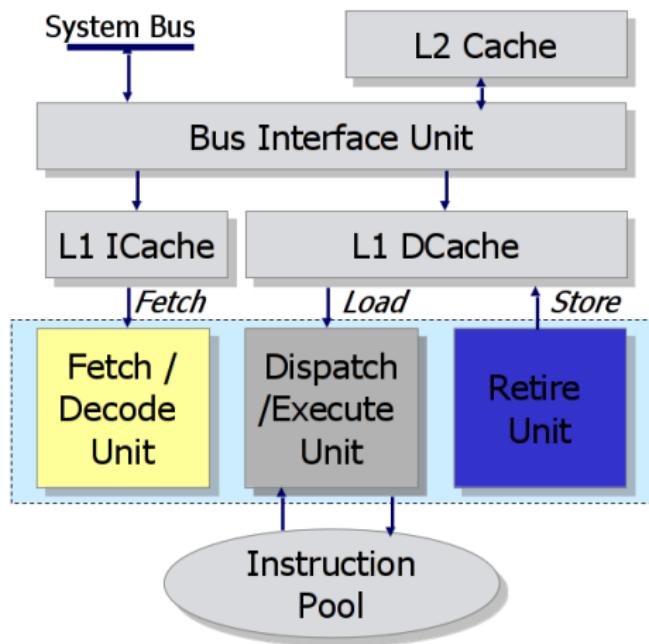
- Emplea Dynamic Instruction Scheduling
- Basado en una ventana de instrucciones y no en un pipeline superescalar.
- Las instrucciones se traducen en micro operaciones básicas (uops)
- Las uops ingresan a un pool (ventana) en donde se mantienen para su ejecución
- Los tres cores tienen plena visibilidad de esa ventana de ejecución
- Implementa ejecución fuera de orden y ejecución especulativa.
- La unidad de despacho y ejecución mantiene el modelo superescalar y lo combina con un super pipeline de 20 etapas

# Intel - Three Cores Engine



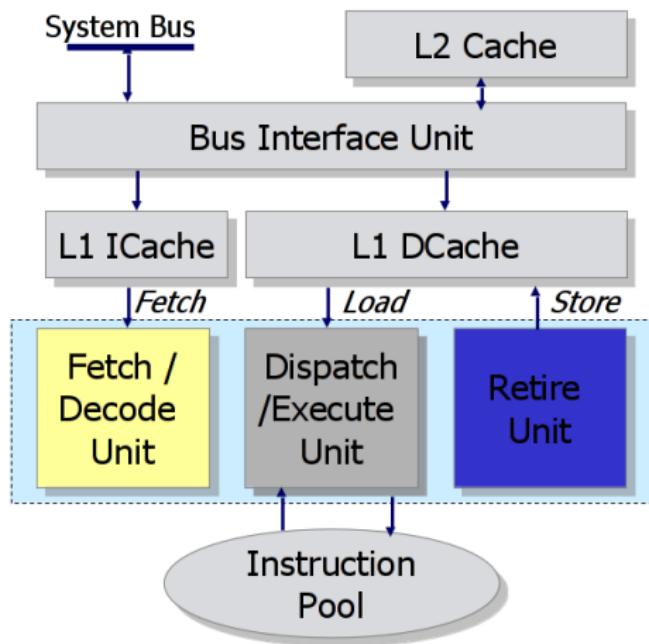
- Emplea Dynamic Instruction Scheduling
- Basado en una ventana de instrucciones y no en un pipeline superescalar.
- Las instrucciones se traducen en micro operaciones básicas (uops)
- Las uops ingresan a un pool (ventana) en donde se mantienen para su ejecución
- Los tres cores tienen plena visibilidad de esa ventana de ejecución
- Implementa ejecución fuera de orden y ejecución especulativa.
- La unidad de despacho y ejecución mantiene el modelo superescalar y lo combina con un super pipeline de 20 etapas

# Intel - Three Cores Engine



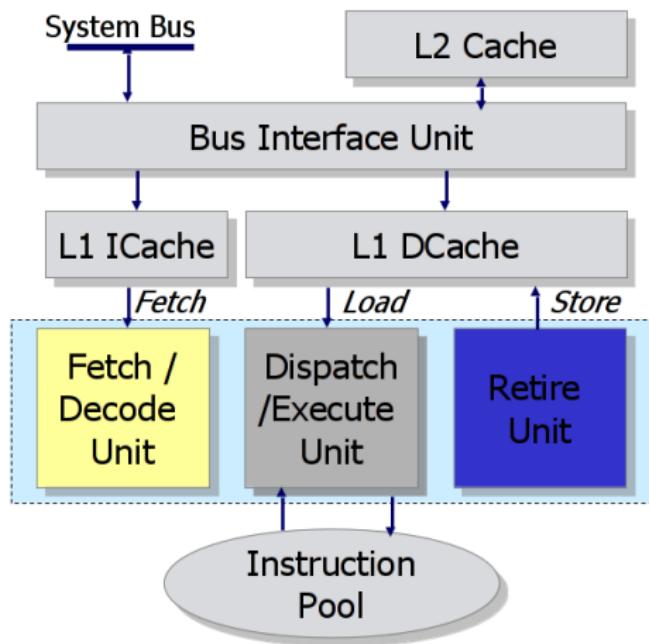
- Emplea Dynamic Instruction Scheduling
- Basado en una ventana de instrucciones y no en un pipeline superescalar.
- Las instrucciones se traducen en micro operaciones básicas (uops)
- Las uops ingresan a un pool (ventana) en donde se mantienen para su ejecución
- Los tres cores tienen plena visibilidad de esa ventana de ejecución
- Implementa ejecución fuera de orden y ejecución especulativa.
- La unidad de despacho y ejecución mantiene el modelo superescalar y lo combina con un super pipeline de 20 etapas

# Intel - Three Cores Engine



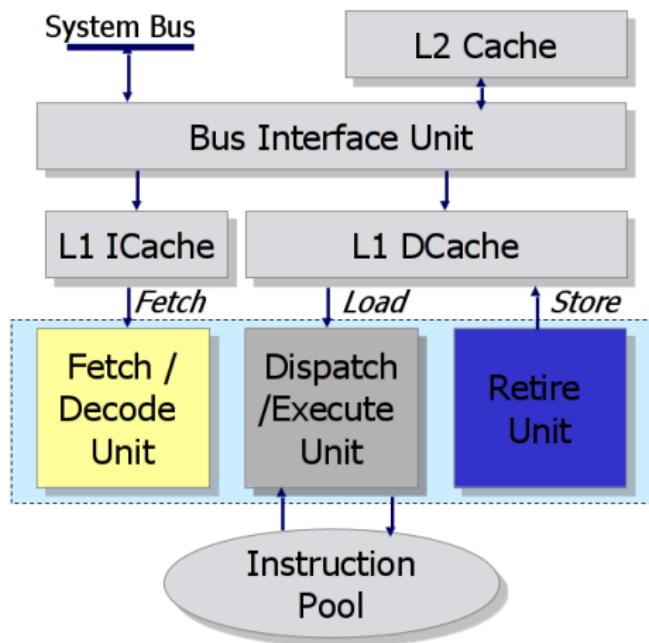
- Emplea Dynamic Instruction Scheduling
- Basado en una ventana de instrucciones y no en un pipeline superescalar.
- Las instrucciones se traducen en micro operaciones básicas (uops)
- Las uops ingresan a un pool (ventana) en donde se mantienen para su ejecución
- Los tres cores tienen plena visibilidad de esa ventana de ejecución
- Implementa ejecución fuera de orden y ejecución especulativa.
- La unidad de despacho y ejecución mantiene el modelo superescalar y lo combina con un super pipeline de 20 etapas

# Intel - Three Cores Engine



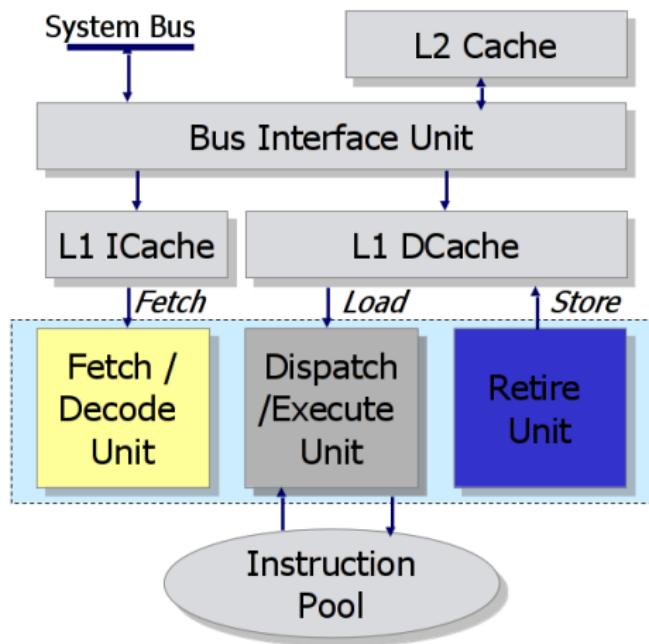
- Emplea Dynamic Instruction Scheduling
- Basado en una ventana de instrucciones y no en un pipeline superescalar.
- Las instrucciones se traducen en micro operaciones básicas (uops)
- Las uops ingresan a un pool (ventana) en donde se mantienen para su ejecución
- Los tres cores tienen plena visibilidad de esa ventana de ejecución
- Implementa ejecución fuera de orden y ejecución especulativa.
- La unidad de despacho y ejecución mantiene el modelo superescalar y lo combina con un súper pipeline de 20 etapas

# Intel - Three Cores Engine



- Emplea Dynamic Instruction Scheduling
- Basado en una ventana de instrucciones y no en un pipeline superescalar.
- Las instrucciones se traducen en micro operaciones básicas (uops)
- Las uops ingresan a un pool (ventana) en donde se mantienen para su ejecución
- Los tres cores tienen plena visibilidad de esa ventana de ejecución
- Implementa ejecución fuera de orden y ejecución especulativa.
- La unidad de despacho y ejecución mantiene el modelo superescalar y lo combina con un súper pipeline de 20 etapas

# Intel - Three Cores Engine



- Emplea Dynamic Instruction Scheduling
- Basado en una ventana de instrucciones y no en un pipeline superescalar.
- Las instrucciones se traducen en micro operaciones básicas (uops)
- Las uops ingresan a un pool (ventana) en donde se mantienen para su ejecución
- Los tres cores tienen plena visibilidad de esa ventana de ejecución
- Implementa ejecución fuera de orden y ejecución especulativa.
- La unidad de despacho y ejecución mantiene el modelo superescalar y lo combina con un súper pipeline de 20 etapas

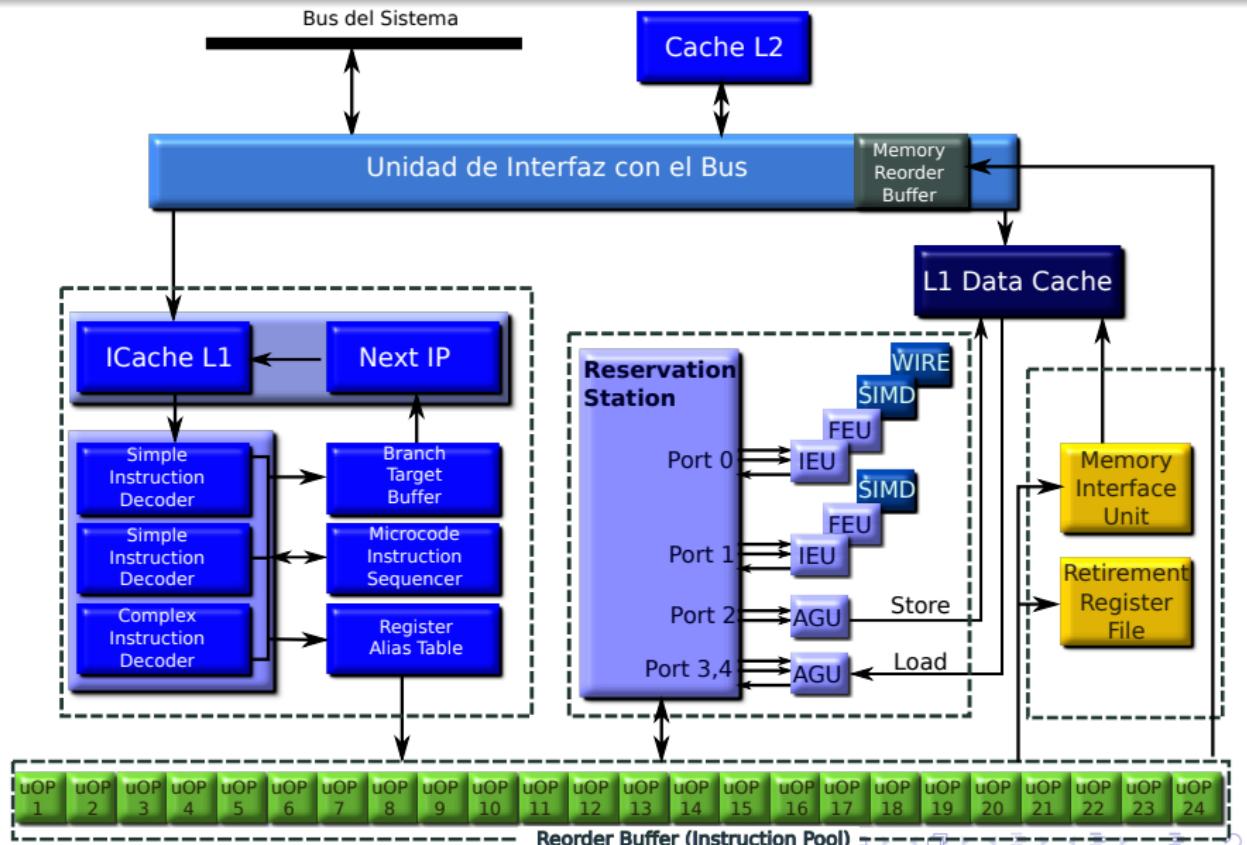
# Ejecución fuera de orden

Se trabaja sobre una ventana de instrucciones

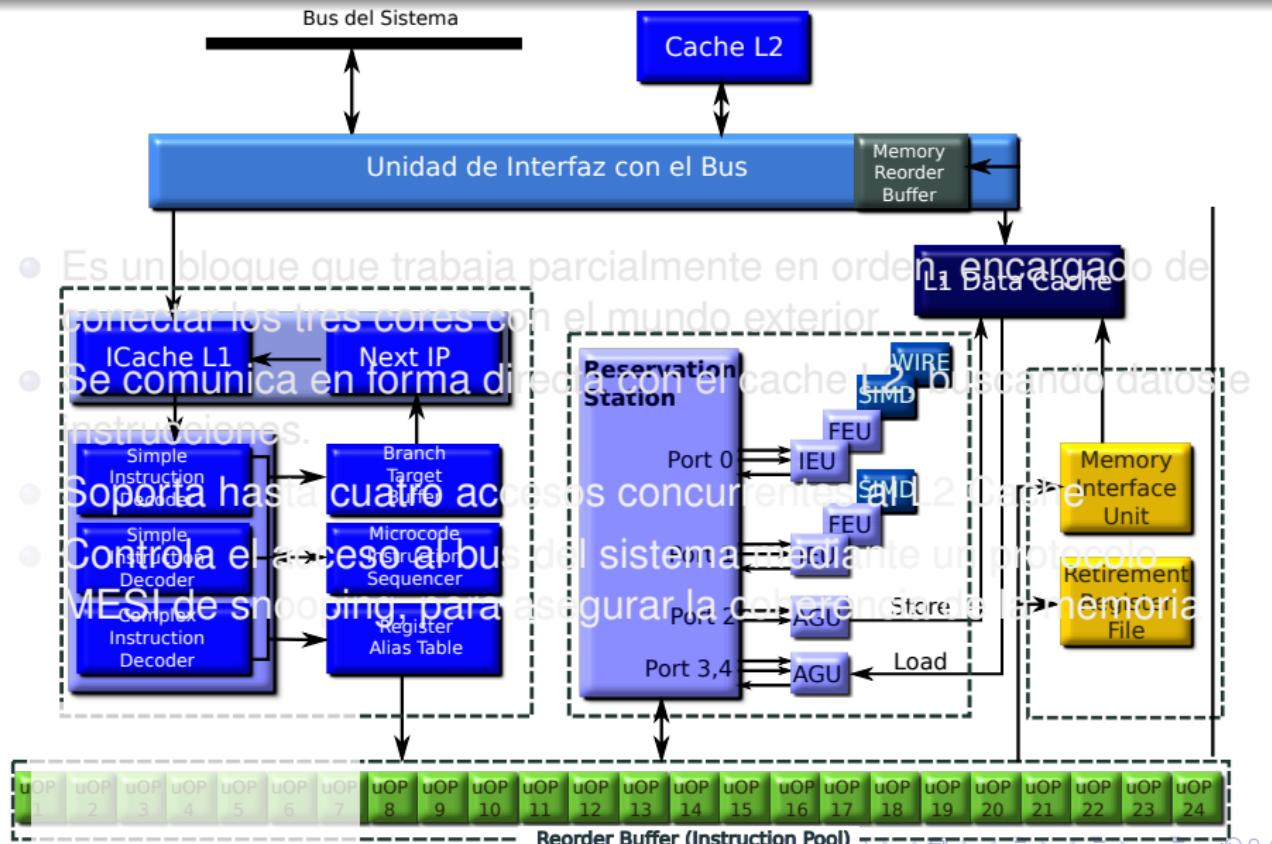
Si se observan entre 20 y 30 instrucciones los compiladores generan en promedio 5 saltos dentro de esa ventana. Estos deben ser correctamente predictos.

Como se ejecutan fuera de orden, se mantienen los resultados en el campo adecuado de las

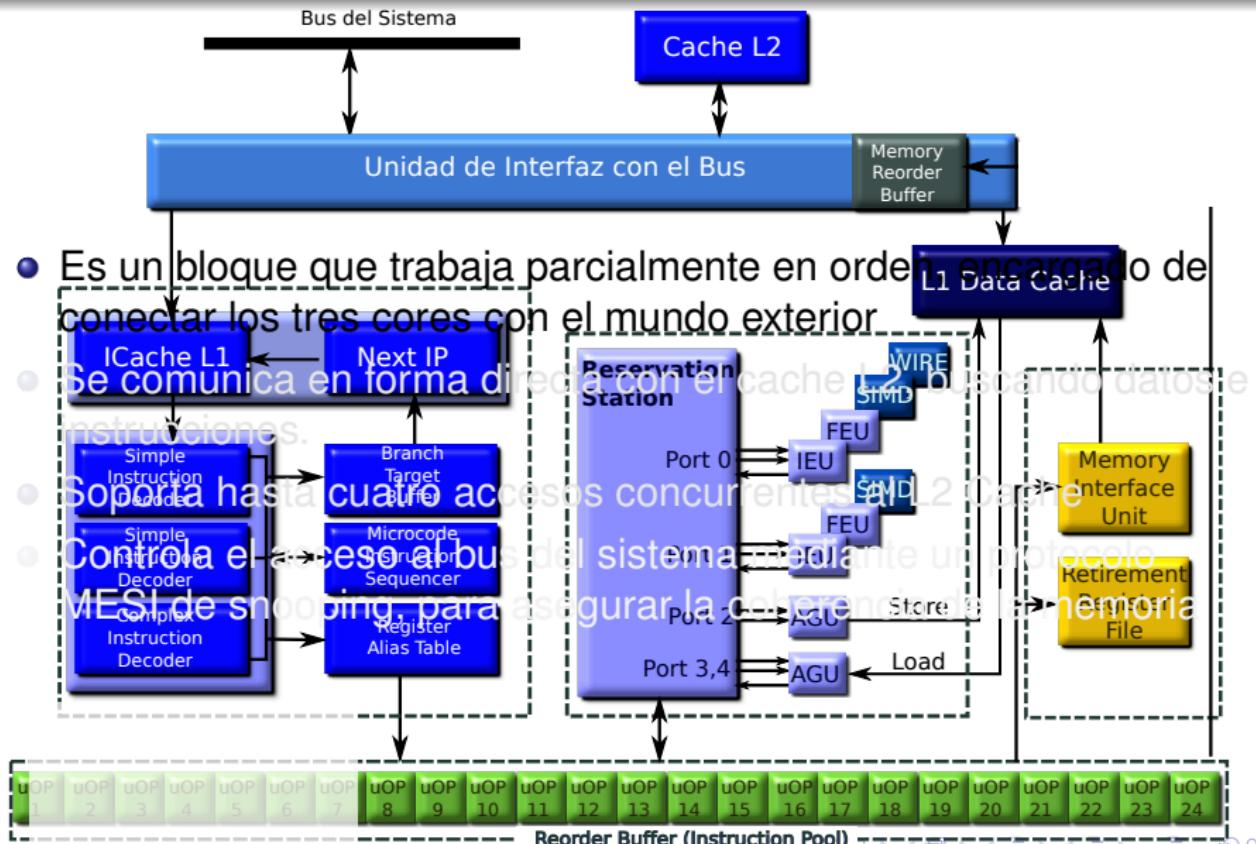
# Three Cores Engine en detalle



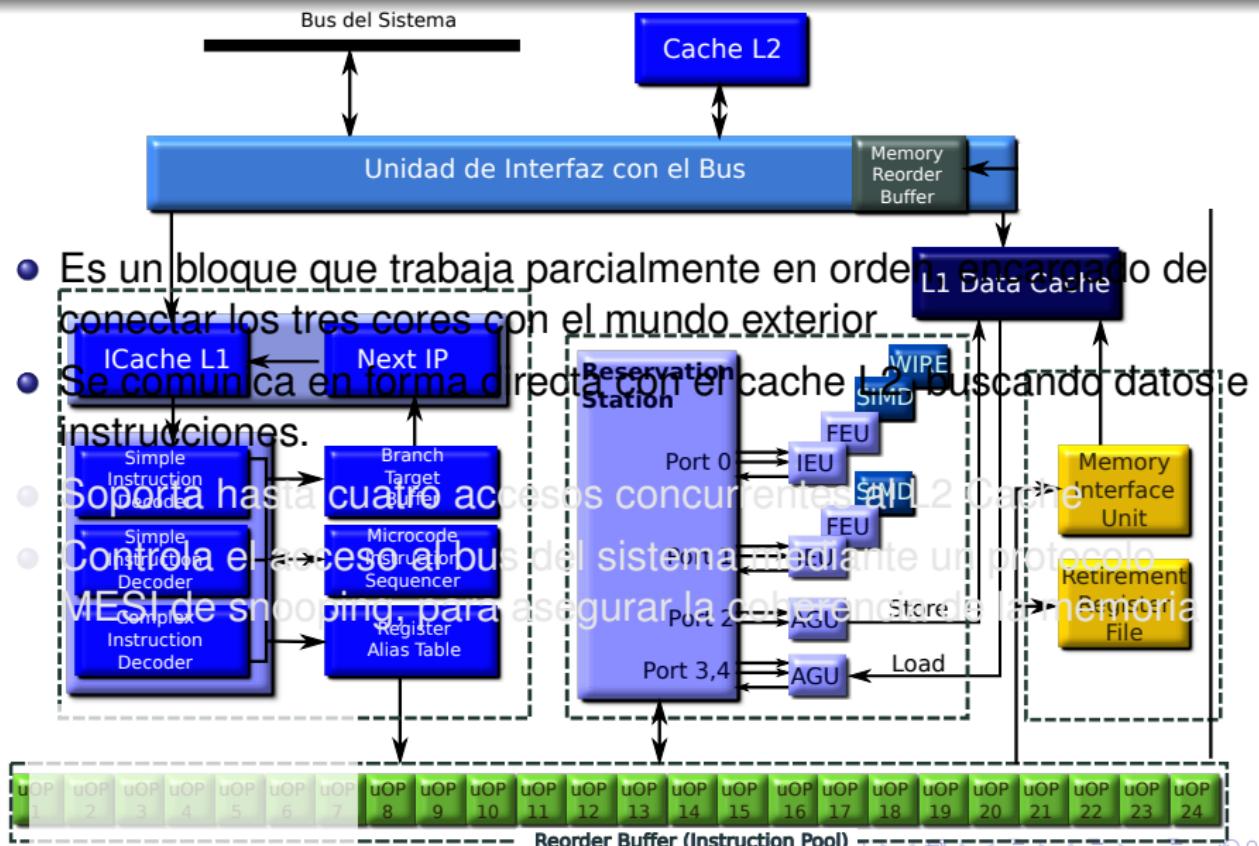
# Unidad de Interfaz con el Bus



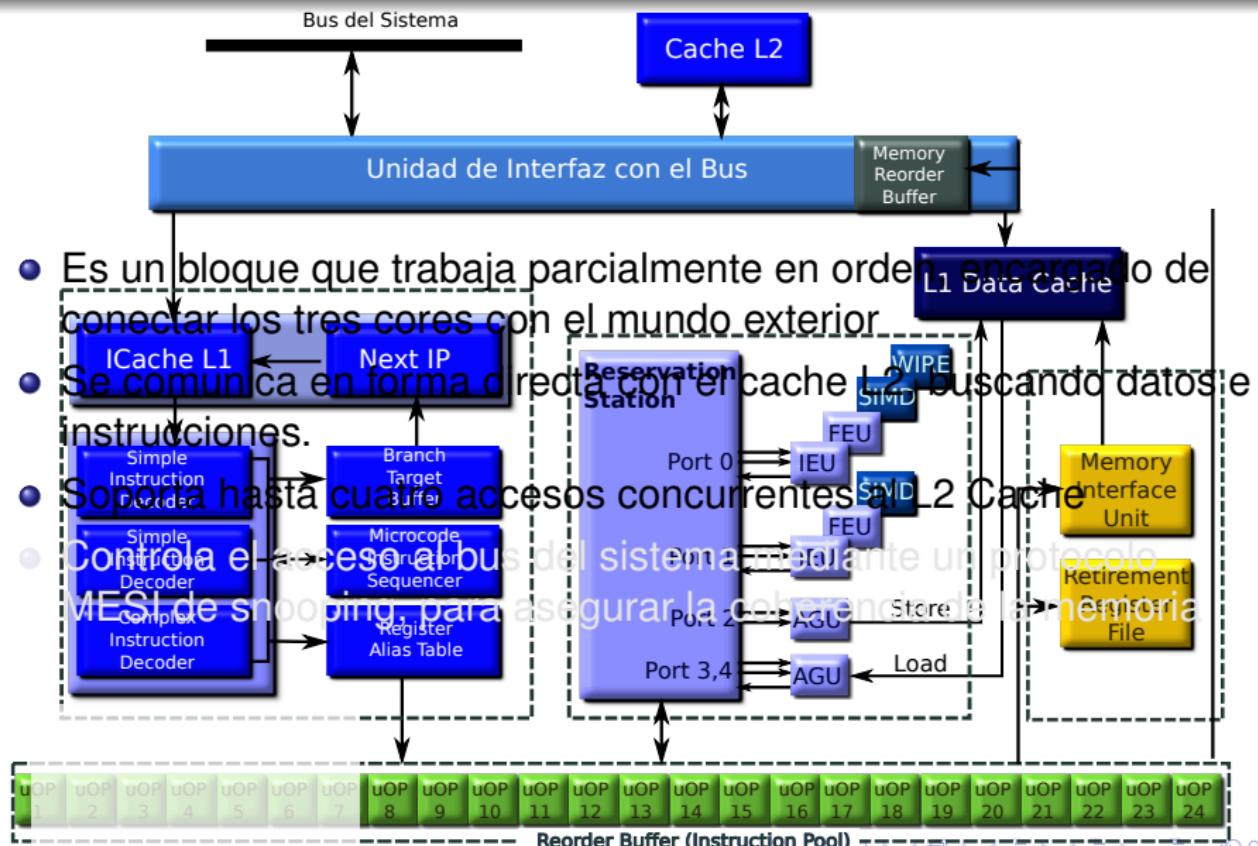
# Unidad de Interfaz con el Bus



# Unidad de Interfaz con el Bus

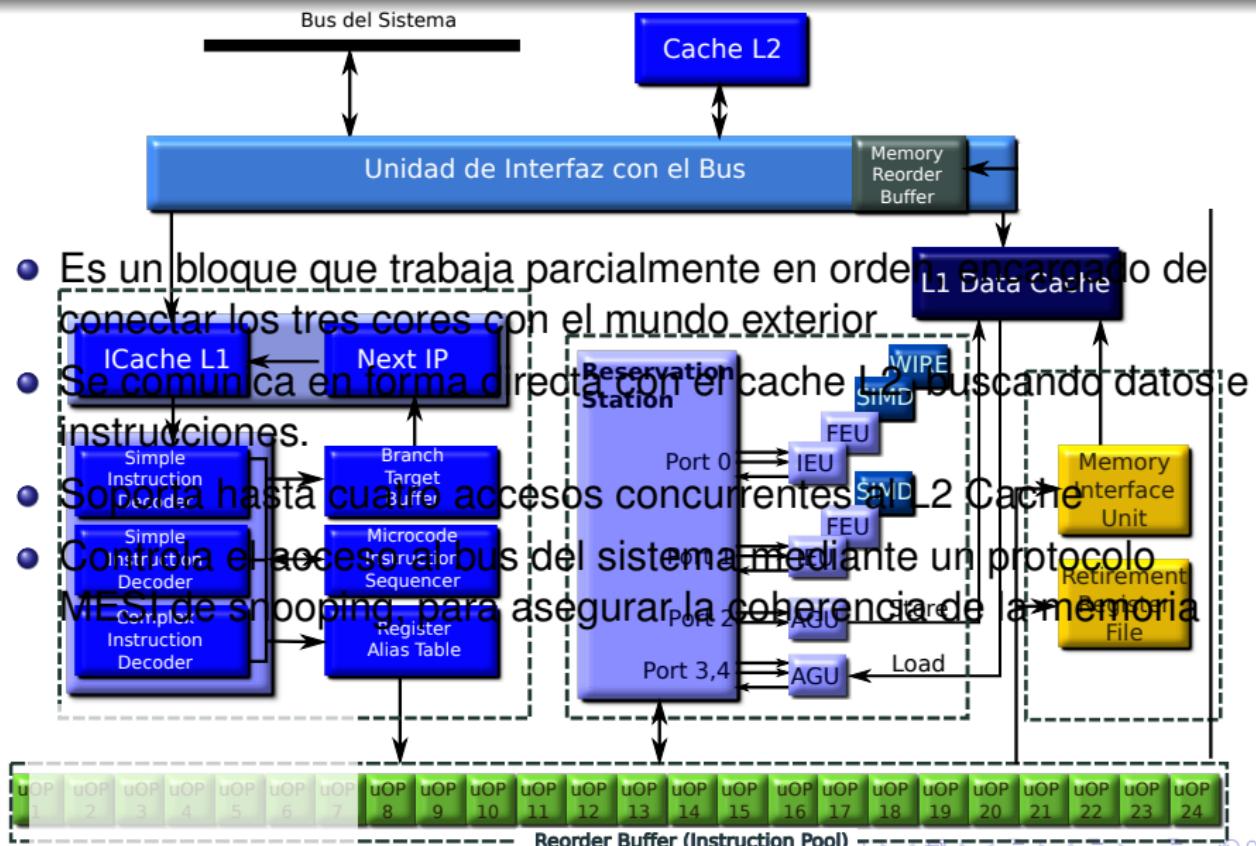


# Unidad de Interfaz con el Bus



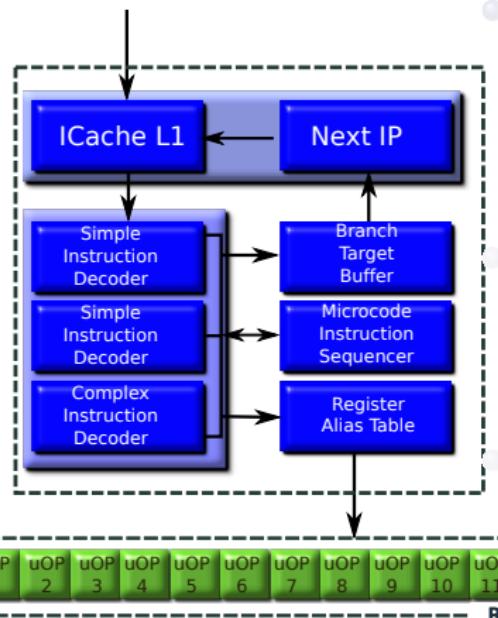
- Es un **bloque** que trabaja parcialmente en orden encargado de conectar los tres cores con el mundo exterior.
- Se comunica en forma directa con el cache L2 Cache buscando datos e instrucciones.
- Soporta hasta cuatro accesos concurrentes al L2 Cache.
- Controla el acceso a bus del sistema mediante un protocolo MESI de snooping, para asegurar la coherencia de la memoria.

# Unidad de Interfaz con el Bus



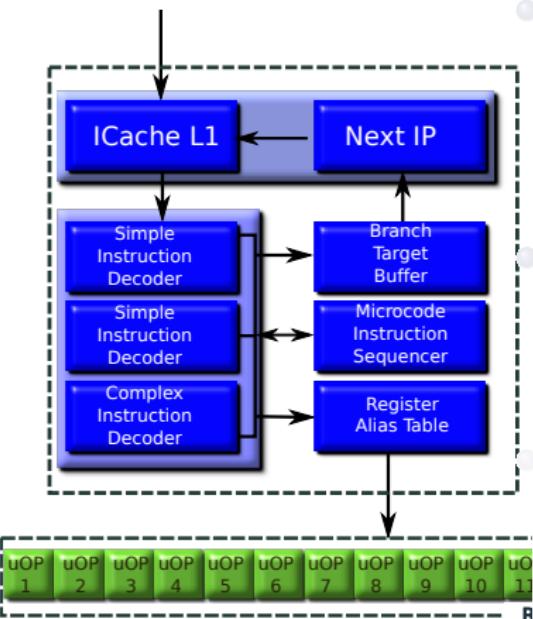
# Three Cores Engine en detalle

- El bloque ***ICache***, es el cache de instrucciones. Su tamaño de línea es 16 bytes. De este modo entrega 16 byte alineados a la Unidad de Decodificación.
- El puntero ***Next IP***, apunta a la línea de la siguiente instrucción o a la de la instrucción target de un salto de acuerdo con la predicción generada en el Branch Target Buffer.
- Hay tres decodificadores que toman en paralelo esta línea de 16 bytes proveniente del ***ICache*** y buscan dentro las instrucciones que hay que decodificar.
- Cada instrucción de la Arquitectura Intel (***IA*** de aquí en mas) se traduce en una o mas uops las cuales tienen dos operandos y un operando destino.



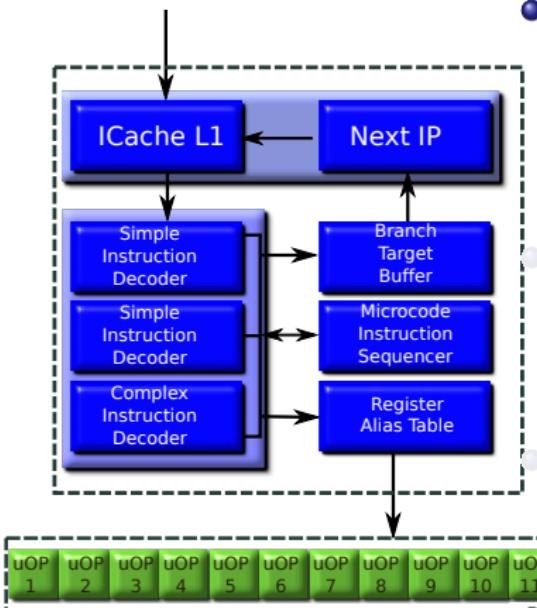
# Three Cores Engine en detalle

- El bloque **ICache**, es el cache de instrucciones. Su tamaño de línea es 16 bytes. De este modo entrega 16 byte alineados a la Unidad de Decodificación.
- El puntero **Next IP**, apunta a la línea de la siguiente instrucción o a la de la instrucción target de un salto de acuerdo con la predicción generada en el Branch Target Buffer.
- Hay tres decodificadores que toman en paralelo esta línea de 16 bytes proveniente del **ICache** y buscan dentro las instrucciones que hay que decodificar.
- Cada instrucción de la Arquitectura Intel (**IA** de aquí en mas) se traduce en una o mas uops las cuales tienen dos operandos y un operando destino.

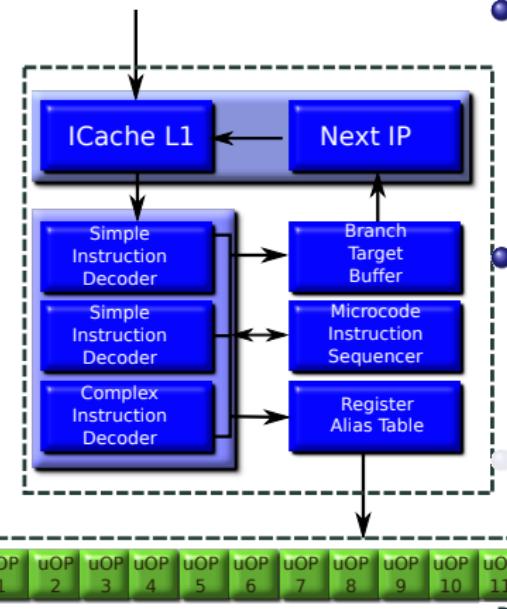


# Three Cores Engine en detalle

- El bloque **ICache**, es el cache de instrucciones. Su tamaño de línea es 16 bytes. De este modo entrega 16 byte alineados a la Unidad de Decodificación.
- El puntero **Next IP**, apunta a la línea de la siguiente instrucción o a la de la instrucción target de un salto de acuerdo con la predicción generada en el Branch Target Buffer.
- Hay tres decodificadores que toman en paralelo esta línea de 16 bytes proveniente del **ICache** y buscan dentro las instrucciones que hay que decodificar.
- Cada instrucción de la Arquitectura Intel (**IA** de aquí en mas) se traduce en una o mas uops las cuales tienen dos operandos fuente y un operando destino.

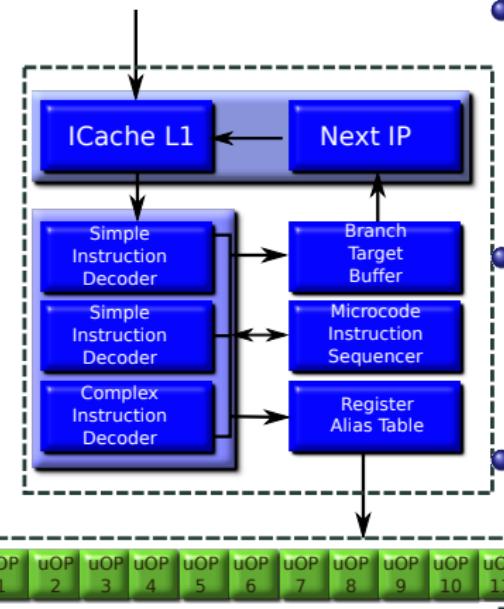


# Three Cores Engine en detalle



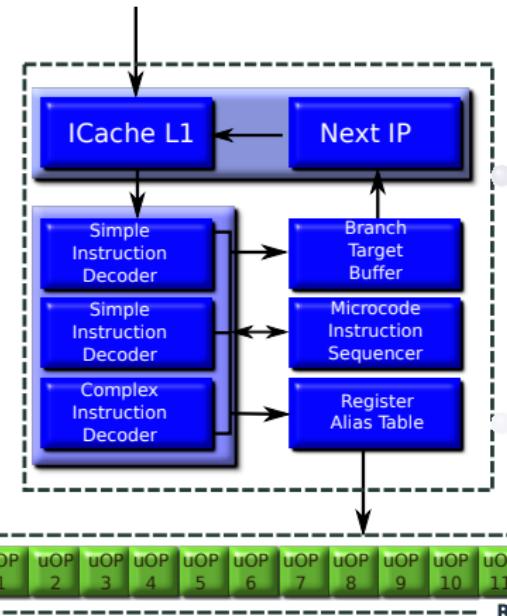
- El bloque **ICache**, es el cache de instrucciones. Su tamaño de línea es 16 bytes. De este modo entrega 16 byte alineados a la Unidad de Decodificación.
- El puntero **Next\_IP**, apunta a la línea de la siguiente instrucción o a la de la instrucción target de un salto de acuerdo con la predicción generada en el Branch Target Buffer.
- Hay tres decodificadores que toman en paralelo esta línea de 16 bytes proveniente del **ICache** y buscan dentro las instrucciones que hay que decodificar.
- Cada instrucción de la Arquitectura Intel (**IA** de aquí en mas) se traduce en una o mas uops las cuales tienen dos operandos fuente y un operando destino.

# Three Cores Engine en detalle



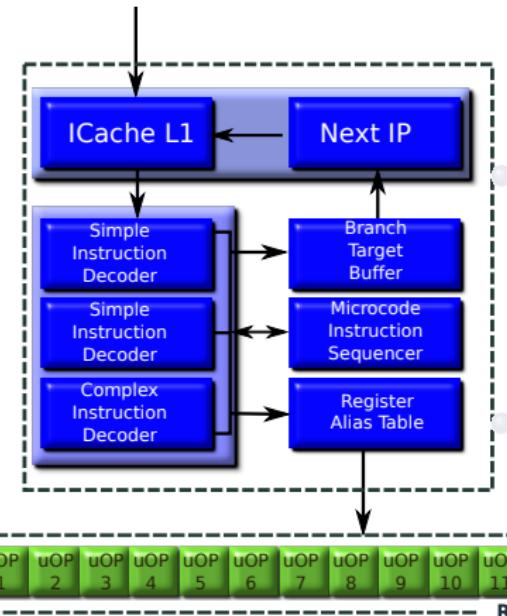
- El bloque **ICache**, es el cache de instrucciones. Su tamaño de línea es 16 bytes. De este modo entrega 16 byte alineados a la Unidad de Decodificación.
- El puntero **Next\_IP**, apunta a la línea de la siguiente instrucción o a la de la instrucción target de un salto de acuerdo con la predicción generada en el Branch Target Buffer.
- Hay tres decodificadores que toman en paralelo esta línea de 16 bytes proveniente del **ICache** y buscan dentro las instrucciones que hay que decodificar.
- Cada instrucción de la Arquitectura Intel (**IA** de aquí en mas) se traduce en una o mas uops las cuales tienen dos operandos fuente y un operando destino.

# Three Cores Engine en detalle



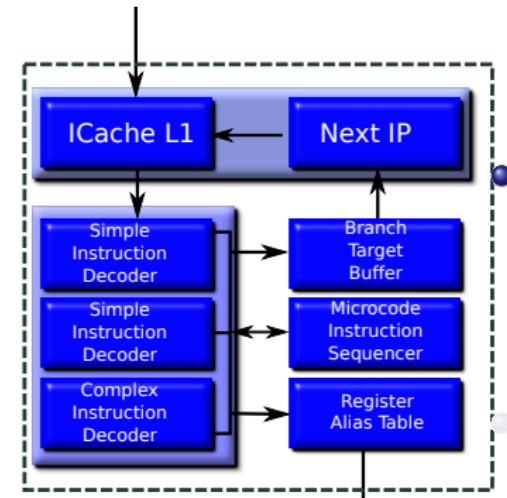
- La mayor parte de las instrucciones **IA** se traducen en una uOP. El resto se convierten 4 uOPs, y las mas complejas entran al Bloque **MicroCode Instruction Sequencer** (de aquí en mas MIS) que genera la cantidad de uOPs necesarias.
- Las uOPs se encolan y se envían al **Register Alias Table**, que se encarga de traducir los registros de la **IA** en registros de un amplio register file interno del procesador. Esta operación se llama Register Renaming.
- Además las uOPs se ingresan al Allocator que les agrega información de estado y las ingresa al pool de instrucciones.

# Three Cores Engine en detalle



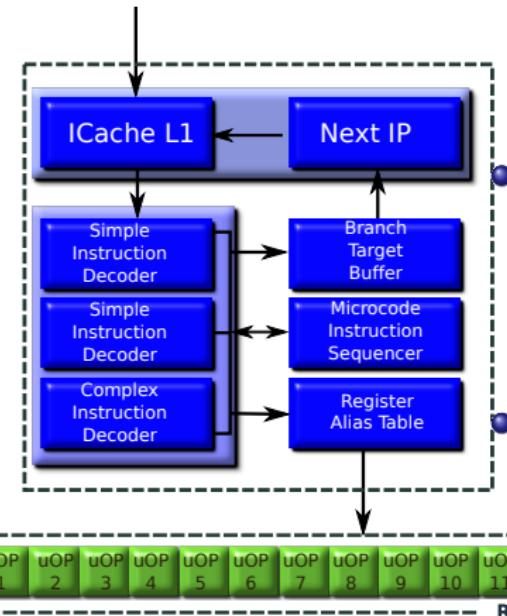
- La mayor parte de las instrucciones **IA** se traducen en una uOP. El resto se convierten 4 uOPs, y las mas complejas entran al Bloque **MicroCode Instruction Sequencer** (de aquí en mas MIS) que genera la cantidad de uOPs necesarias.
- Las uOPs se encolan y se envían al **Register Alias Table**, que se encarga de traducir los registros de la **IA** en registros de un amplio register file interno del procesador. Esta operación se llama Register Renaming.
- Además las uOPs se ingresan al Allocator que les agrega información de estado y las ingresa al pool de instrucciones.

# Three Cores Engine en detalle



- La mayor parte de las instrucciones **IA** se traducen en una uOP. El resto se convierten 4 uOPs, y las mas complejas entran al Bloque **MicroCode Instruction Sequencer** (de aquí en mas MIS) que genera la cantidad de uOPs necesarias.
- Las uOPs se encolan y se envían al **Register Alias Table**, que se encarga de traducir los registros de la **IA** en registros de un amplio register file interno del procesador. Esta operación se llama Register Renaming.
- Además las uOPs se ingresan al Allocator que les agrega información de estado y las ingresa al pool de instrucciones.

# Three Cores Engine en detalle



- La mayor parte de las instrucciones **IA** se traducen en una uOP. El resto se convierten 4 uOPs, y las mas complejas entran al Bloque **MicroCode Instruction Sequencer** (de aquí en mas MIS) que genera la cantidad de uOPs necesarias.
- Las uOPs se encolan y se envían al **Register Alias Table**, que se encarga de traducir los registros de la **IA** en registros de un amplio register file interno del procesador. Esta operación se llama Register Renaming.
- Además las uOPs se ingresan al Allocator que les agrega información de estado y las ingresa al pool de instrucciones.

# Three Cores Engine en detalle

- La Unidad de Despacho selecciona uOPs desde el pool de instrucciones dependiendo de su estado y no de su orden dentro del mismo.
- Si el estado de la uOP indica que sus operandos están disponibles, la **Reservation Station** verifica que esté libre algún recurso para su ejecución, y eventualmente le envía la uOP.
- El resultado será escrito en el pool de uOPs por la Unidad de Ejecución correspondiente una vez finalizada.
- Se dispone de 5 puertos de ejecución. De este modo se pueden schedular a lo sumo 5 uOPs por ciclo de clock. EN la práctica se tiene un promedio de 3.
- Respecto de los saltos, las uOPs se marcan como tales en el **Re Order Buffer** (de ahora en mas **ROB**) y se les pone la dirección target predicta por el **BTB**. Si el salto falla, se limpian del **ROB** las instrucciones subsiguientes así estén finalizadas

# Three Cores Engine en detalle

- La Unidad de Despacho selecciona uOPs desde el pool de instrucciones dependiendo de su estado y no de su orden dentro del mismo.
- Si el estado de la uOP indica que sus operandos están disponibles, la **Reservation Station** verifica que esté libre algún recurso para su ejecución, y eventualmente le envía la uOP.
- El resultado será escrito en el pool de uOPs por la Unidad de Ejecución correspondiente una vez finalizada.
- Se dispone de 5 puertos de ejecución. De este modo se pueden schedular a lo sumo 5 uOPs por ciclo de clock. EN la práctica se tiene un promedio de 3.
- Respecto de los saltos, las uOPs se marcan como tales en el **Re Order Buffer** (de ahora en mas **ROB**) y se les pone la dirección target predicta por el **BTB**. Si el salto falla, se limpian del **ROB** las instrucciones subsiguientes así estén finalizadas

# Three Cores Engine en detalle

- La Unidad de Despacho selecciona uOPs desde el pool de instrucciones dependiendo de su estado y no de su orden dentro del mismo.
- Si el estado de la uOP indica que sus operandos están disponibles, la **Reservation Station** verifica que esté libre algún recurso para su ejecución, y eventualmente le envía la uOP.
- El resultado será escrito en el pool de uOPs por la Unidad de Ejecución correspondiente una vez finalizada.
- Se dispone de 5 puertos de ejecución. De este modo se pueden schedular a lo sumo 5 uOPs por ciclo de clock. EN la práctica se tiene un promedio de 3.
- Respecto de los saltos, las uOPs se marcan como tales en el **Re Order Buffer** (de ahora en mas **ROB**) y se les pone la dirección target predicta por el **BTB**. Si el salto falla, se limpian del **ROB** las instrucciones subsiguientes así estén finalizadas

# Three Cores Engine en detalle

- La Unidad de Despacho selecciona uOPs desde el pool de instrucciones dependiendo de su estado y no de su orden dentro del mismo.
- Si el estado de la uOP indica que sus operandos están disponibles, la **Reservation Station** verifica que esté libre algún recurso para su ejecución, y eventualmente le envía la uOP.
- El resultado será escrito en el pool de uOPs por la Unidad de Ejecución correspondiente una vez finalizada.
- Se dispone de 5 puertos de ejecución. De este modo se pueden schedular a lo sumo 5 uOPs por ciclo de clock. EN la práctica se tiene un promedio de 3.
- Respecto de los saltos, las uOPs se marcan como tales en el **Re Order Buffer** (de ahora en mas **ROB**) y se les pone la dirección target predicta por el **BTB**. Si el salto falla, se limpian del **ROB** las instrucciones subsiguientes así estén finalizadas

# Three Cores Engine en detalle

- La Unidad de Despacho selecciona uOPs desde el pool de instrucciones dependiendo de su estado y no de su orden dentro del mismo.
- Si el estado de la uOP indica que sus operandos están disponibles, la **Reservation Station** verifica que esté libre algún recurso para su ejecución, y eventualmente le envía la uOP.
- El resultado será escrito en el pool de uOPs por la Unidad de Ejecución correspondiente una vez finalizada.
- Se dispone de 5 puertos de ejecución. De este modo se pueden scheduler a lo sumo 5 uOPs por ciclo de clock. EN la práctica se tiene un promedio de 3.
- Respecto de los saltos, las uOPs se marcan como tales en el **Re Order Buffer** (de ahora en mas **ROB**) y se les pone la dirección target predicta por el **BTB**. Si el salto falla, se limpian del **ROB** las instrucciones subsiguientes así estén finalizadas

# Three Cores Engine en detalle

- La Unidad de Despacho selecciona uOPs desde el pool de instrucciones dependiendo de su estado y no de su orden dentro del mismo.
- Si el estado de la uOP indica que sus operandos están disponibles, la **Reservation Station** verifica que esté libre algún recurso para su ejecución, y eventualmente le envía la uOP.
- El resultado será escrito en el pool de uOPs por la Unidad de Ejecución correspondiente una vez finalizada.
- Se dispone de 5 puertos de ejecución. De este modo se pueden scheduler a lo sumo 5 uOPs por ciclo de clock. EN la práctica se tiene un promedio de 3.
- Respecto de los saltos, las uOPs se marcan como tales en el **Re Order Buffer** (de ahora en mas **ROB**) y se les pone la dirección target predicta por el **BTB**. Si el salto falla, se limpian del **ROB** las instrucciones subsiguientes así estén finalizadas

# Three Cores Engine en detalle

- La Unidad de Retiro, se encarga de monitorear el estado de cada instrucción del **ROB**.
- No solo debe chequear su estado, sino que las debe reinsertar en el orden en que ocupan en el programa.
- Esto incluye el procesamiento de Interrupciones, excepciones, trap, breackpoints, y predicciones fallidas.
- El **Retirement Register File** (de aquí en mas **RRF**), se encarga de escribir los operandos resultantes en los registros de la **IA** y la Unidad de Interfaz de memoria, aplica en el cache L1 de Datos el resultado si este corresponde a una dirección de memoria.
- Puede retirar 3 uOPs por ciclo de clock.

# Three Cores Engine en detalle

- La Unidad de Retiro, se encarga de monitorear el estado de cada instrucción del **ROB**.
- No solo debe chequear su estado, sino que las debe reinsertar en el orden en que ocupan en el programa.
- Esto incluye el procesamiento de Interrupciones, excepciones, trap, breackpoints, y predicciones fallidas.
- El **Retirement Register File** (de aquí en mas **RRF**), se encarga de escribir los operandos resultantes en los registros de la **IA** y la Unidad de Interfaz de memoria, aplica en el cache L1 de Datos el resultado si este corresponde a una dirección de memoria.
- Puede retirar 3 uOPs por ciclo de clock.

# Three Cores Engine en detalle

- La Unidad de Retiro, se encarga de monitorear el estado de cada instrucción del **ROB**.
- No solo debe chequear su estado, sino que las debe reinsertar en el orden en que ocupan en el programa.
- Esto incluye el procesamiento de Interrupciones, excepciones, trap, breackpoints, y predicciones fallidas.
- El **Retirement Register File** (de aquí en mas **RRF**), se encarga de escribir los operandos resultantes en los registros de la **IA** y la Unidad de Interfaz de memoria, aplica en el cache L1 de Datos el resultado si este corresponde a una dirección de memoria.
- Puede retirar 3 uOPs por ciclo de clock.

# Three Cores Engine en detalle

- La Unidad de Retiro, se encarga de monitorear el estado de cada instrucción del **ROB**.
- No solo debe chequear su estado, sino que las debe reinsertar en el orden en que ocupan en el programa.
- Esto incluye el procesamiento de Interrupciones, excepciones, trap, breackpoints, y predicciones fallidas.
- El **Retirement Register File** (de aquí en mas **RRF**), se encarga de escribir los operandos resultantes en los registros de la **IA** y la Unidad de Interfaz de memoria, aplica en el cache L1 de Datos el resultado si este corresponde a una dirección de memoria.
- Puede retirar 3 uOPs por ciclo de clock.

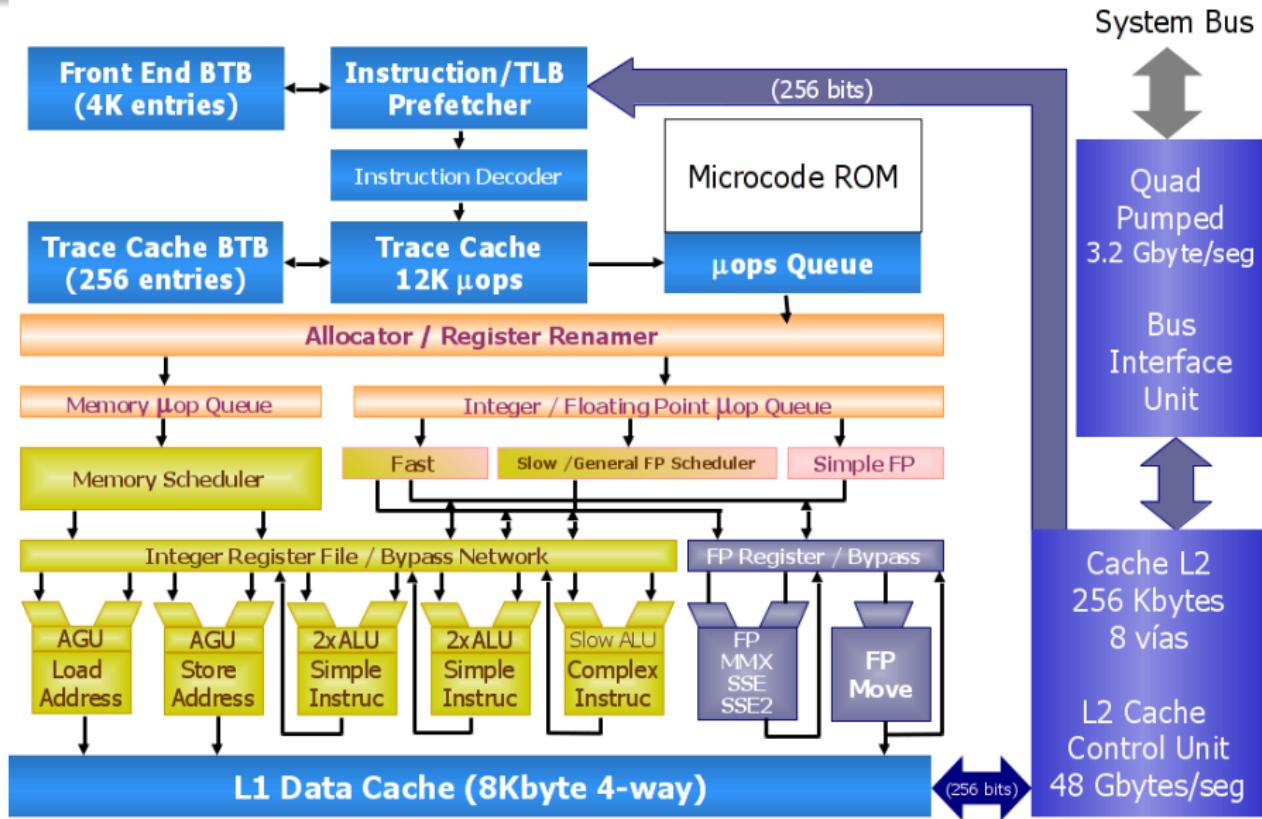
# Three Cores Engine en detalle

- La Unidad de Retiro, se encarga de monitorear el estado de cada instrucción del **ROB**.
- No solo debe chequear su estado, sino que las debe reinsertar en el orden en que ocupan en el programa.
- Esto incluye el procesamiento de Interrupciones, excepciones, trap, breackpoints, y predicciones fallidas.
- El **Retirement Register File** (de aquí en mas **RRF**), se encarga de escribir los operandos resultantes en los registros de la **IA** y la Unidad de Interfaz de memoria, aplica en el cache L1 de Datos el resultado si este corresponde a una dirección de memoria.
- Puede retirar 3 uOPs por ciclo de clock.

# Three Cores Engine en detalle

- La Unidad de Retiro, se encarga de monitorear el estado de cada instrucción del **ROB**.
- No solo debe chequear su estado, sino que las debe reinsertar en el orden en que ocupan en el programa.
- Esto incluye el procesamiento de Interrupciones, excepciones, trap, breackpoints, y predicciones fallidas.
- El **Retirement Register File** (de aquí en mas **RRF**), se encarga de escribir los operandos resultantes en los registros de la **IA** y la Unidad de Interfaz de memoria, aplica en el cache L1 de Datos el resultado si este corresponde a una dirección de memoria.
- Puede retirar 3 uOPs por ciclo de clock.

# Intel Netburst



# Procesadores Multithred

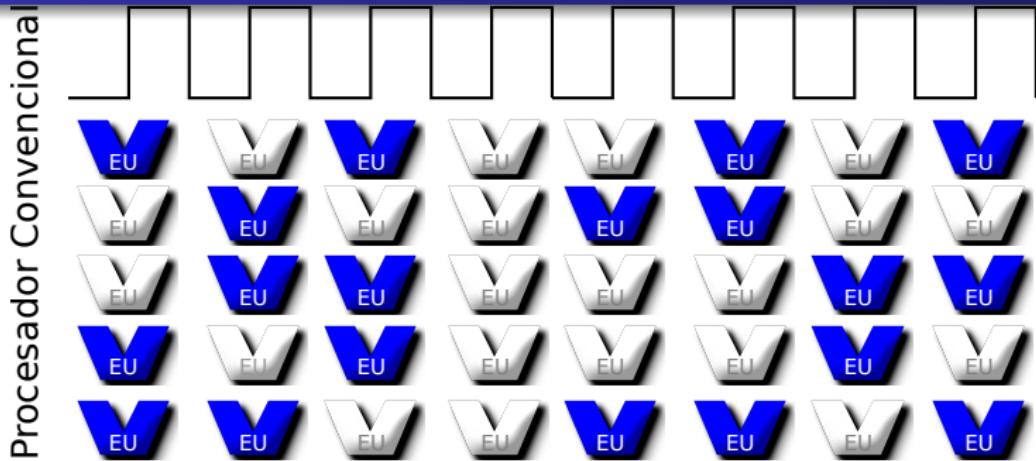
Procesador "Convencional"



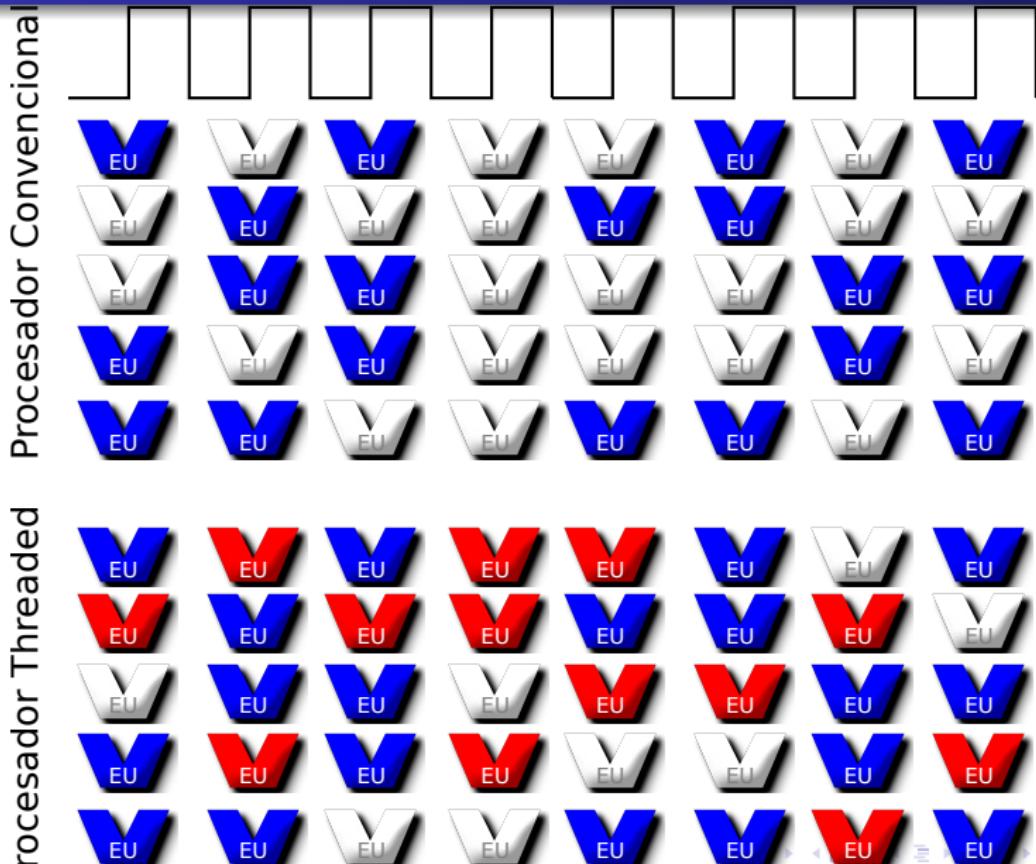
Procesador Multithread



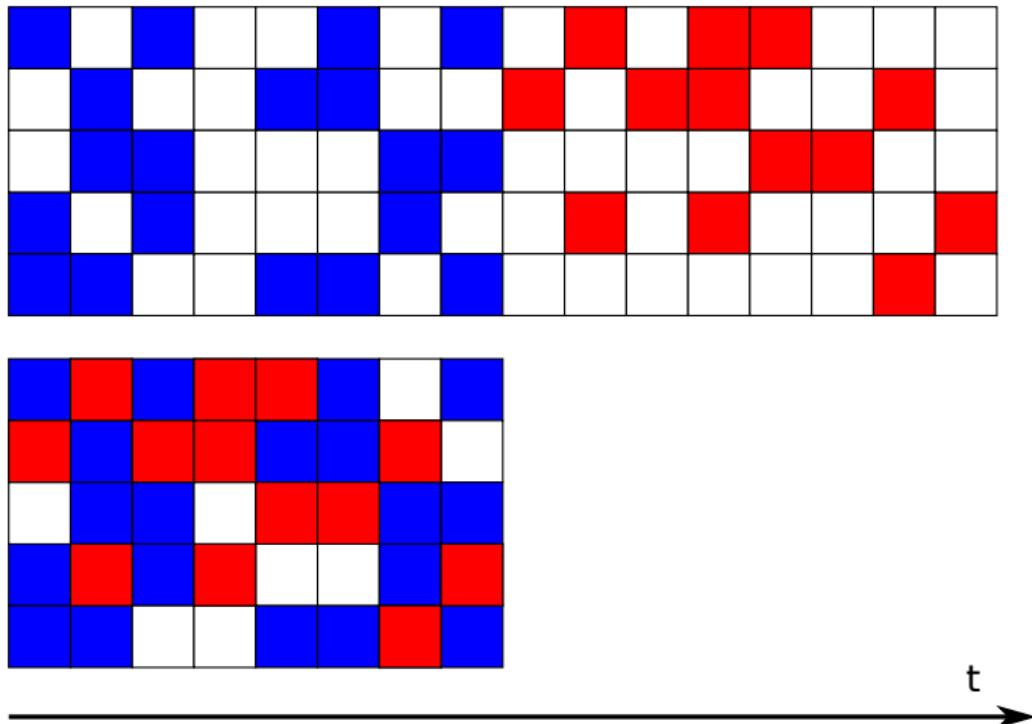
# Ejecución en los Procesadores Multithred



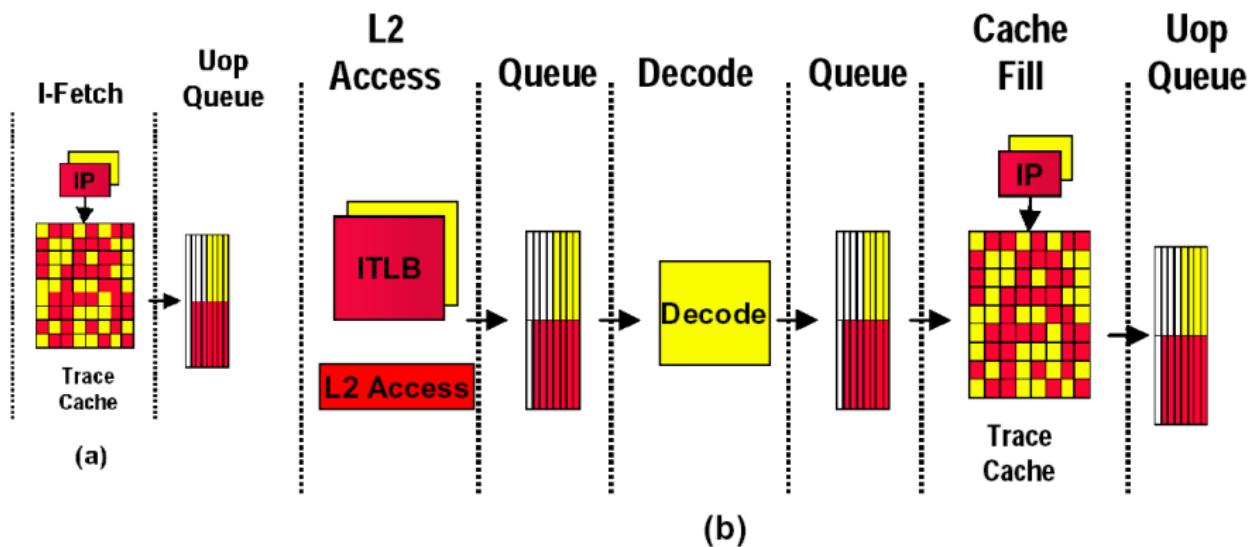
# Ejecución en los Procesadores Multithred



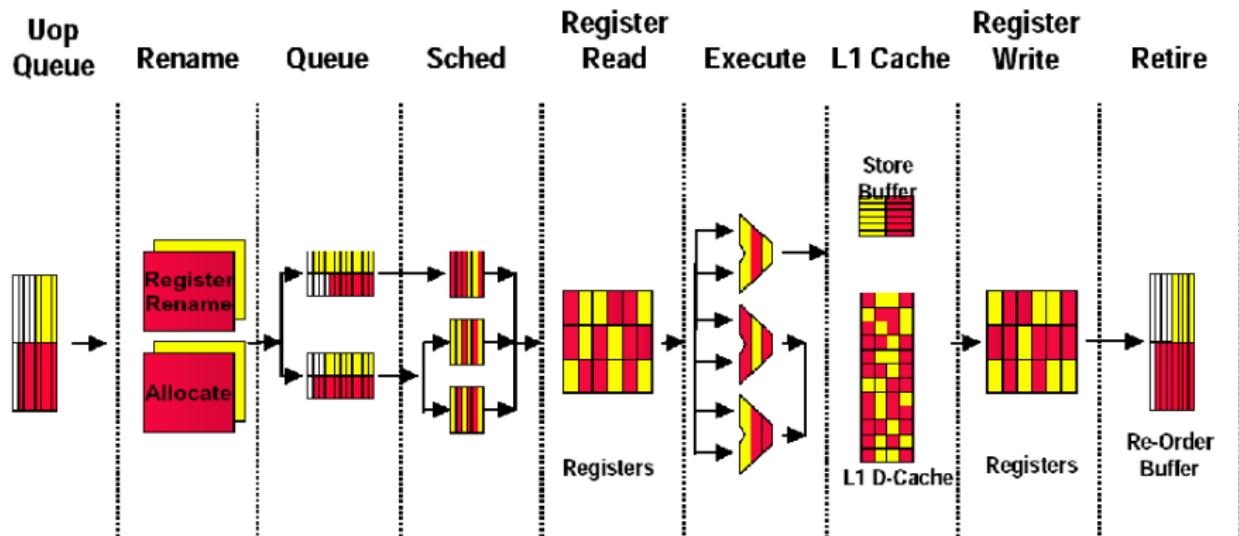
# Ejecución en los Procesadores Multithred



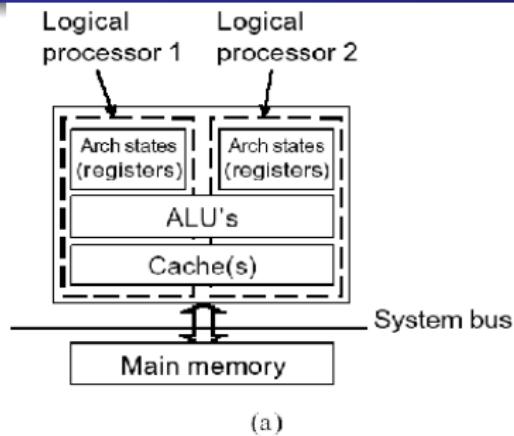
# Intel Hyperthreading



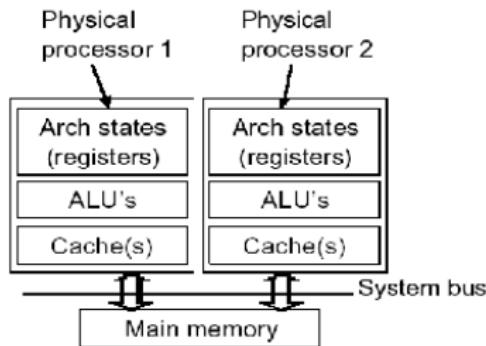
# Intel Hyperthreading



# Hyperthreading versus Multicore



(a)



(b)

# Multicore

**Intel Core Duo Processor**

**Intel Core 2 Duo Processor**

**Intel Pentium dual-core Processor**

Architectural State	Architectural State
Execution Engine	Execution Engine
Local APIC	Local APIC
Second Level Cache	
Bus Interface	



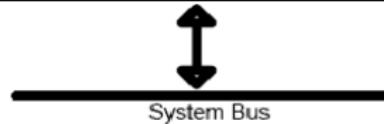
**Pentium D Processor**

Architectural State	Architectural State
Execution Engine	Execution Engine
Local APIC	Local APIC
Bus Interface	



**Pentium Processor Extreme Edition**

Architectural State	Architectural State	Architectural State	Architectural State
Execution Engine		Execution Engine	
Local APIC	Local APIC	Local APIC	Local APIC
Bus Interface		Bus Interface	



OM19809

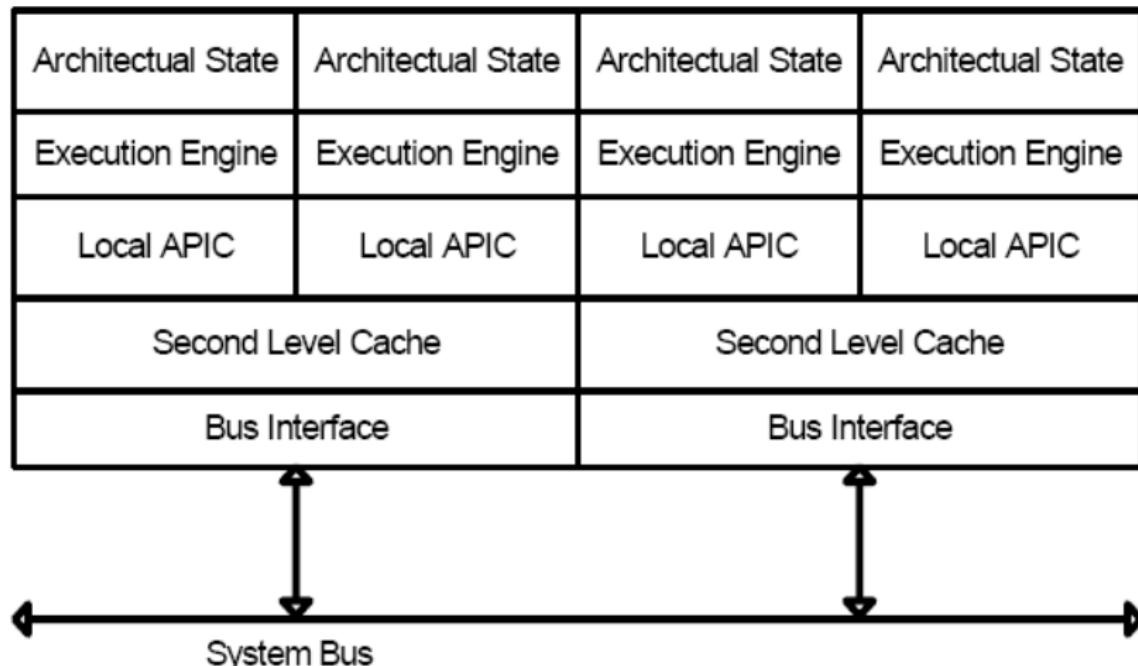
# Multicore

**Intel Core 2 Extreme Quad-core Processor**

**Intel Core 2 Quad Processor**

**Intel Xeon Processor 3200 Series**

**Intel Xeon Processor 5300 Series**



# ¿un core superpoderoso, o muchos cores sencillos?

- Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,, vol.14, no.1, pp.12-31, Jan 1995
- AHi está la respuesta.... en 1995 ya lo delinearon. Y ya se empezaba a avisar que el consumo sería un problema

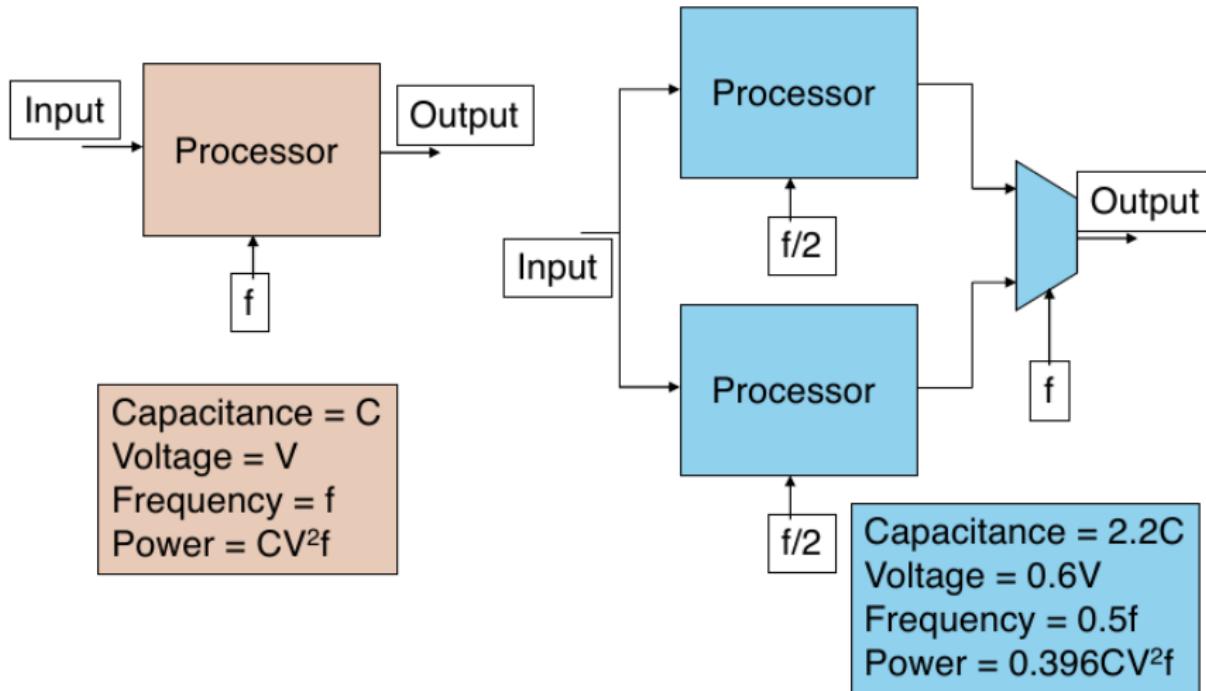
# ¿un core superpoderoso, o muchos cores sencillos?

- Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,, vol.14, no.1, pp.12-31, Jan 1995
- AHi está la respuesta.... en 1995 ya lo delinearon. Y ya se empezaba a avisar que el consumo sería un problema

# ¿un core superpoderoso, o muchos cores sencillos?

- Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,, vol.14, no.1, pp.12-31, Jan 1995
- AHí está la respuesta.... en 1995 ya lo delinearon. Y ya se empezaba a avisar que el consumo sería un problema

# Varios cores: = performance, < consumo



# Resultado en cores y performance/consumo

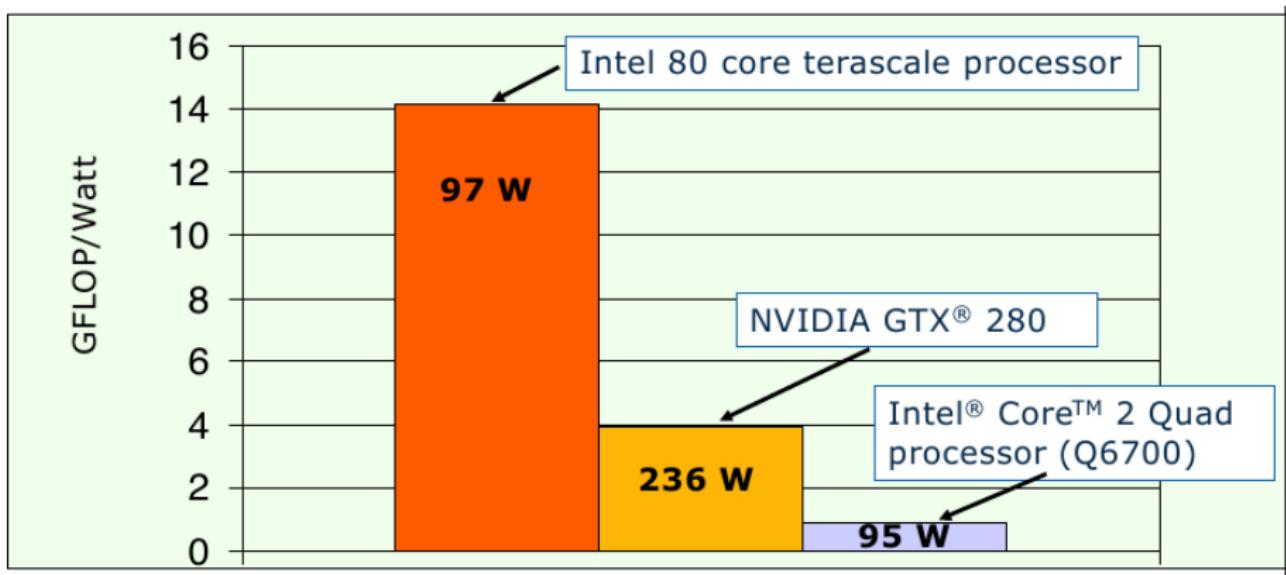
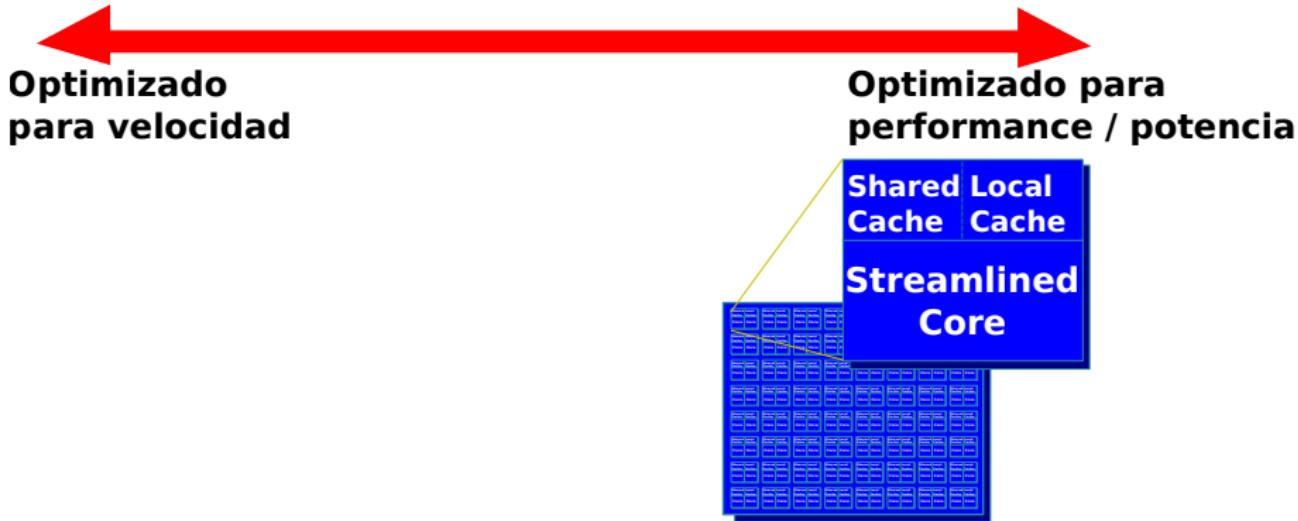


Figura: ©Cortesía Intel

# Derribando vacas sagradas



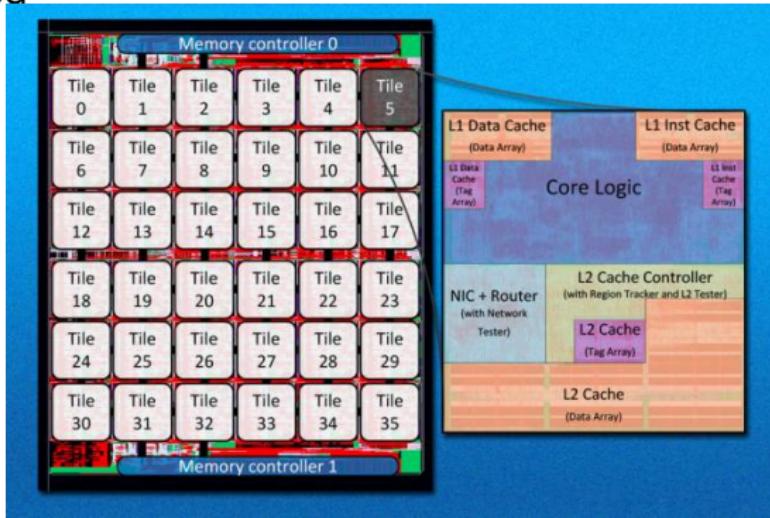
**Era de los Procesadores.**  
Alta frecuencia de clock.  
Superescalares con Ejecución Fuera de Orden.  
Optimizados para ejecutar un thread

**Etapa actual**  
Procesadores de pocos cores optimizados para performance multi threaded y single threaded.

**En 5 a 10 años**  
procesadores con 10 a 100 veces menos de consumo optimizados para multithreading

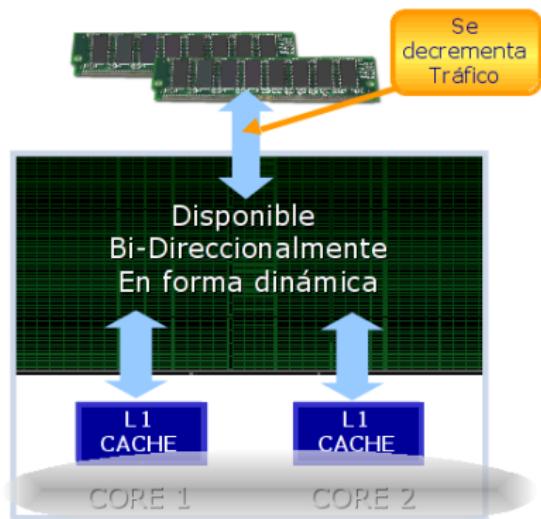
# Issues

- 1 A medida que aumenta la cantidad de cores, el overhead introducido por los protocolos de coherencia es muy alto.
- 2 Una tendencia es configurar redes de multicores.
- 3 Se compone de clusters de  $2^n$  cores y un router.
- 4 Los clusters se derivan mensajes a través de los routers. Como en una red



# Smart Cache

## L2 Compartida Microarquitectura Core



## L2 Independiente

