DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
ESCUELA DE INGENIERÍA
PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

IIC2343 - Arquitectura de Computadores

Paralelismo Avanzado

1. Motivación

El enfoque que tradicionalmente se utilizó para lograr que un computador realice más instrucciones por unidad tiempo, y que por tanto el computador mejore su rendimiento, fue ir logrando que el computador pudiese correr a clocks más rápidos. Ya sea mediante la disminución de tamaño de transistores u otras técnicas como pipeline, el aumentó velocidad del clock de la CPU logró por muchos años permitir las mejoras continuas de rendimiento en los computadores. Sin embargo, el aumento de velocidad trajo consigo un aumento de consumo de energía y disipación de temperatura, llegándose a niveles que hicieron imposible seguir aumentando más la velocidad.

Para lograr que los computadores sigan mejorando su rendimiento se comenzaron a implementar distintas técnicas de paralelismo en las arquitecturas de los computadores, las cuales permiten que un computador ejecute más instrucciones por unidad de tiempo, pero sin necesariamente aumentar la velocidad del clock.

2. Arquitecturas paralelas

Existen distintas formas en que se puede paralelizar el procesamiento en un computador, las que incluyen paralelismo dentro y fuera del procesador. Una forma de categorizar las arquitecturas paralelas es mediante la **Taxonomía de Flynn** (tabla 1) la cual define cuatro categorías de paralelismo según si se utiliza uno o múltiples streams de instrucciones (i.e. programas) y uno o múltiples streams de datos.

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

Tabla 1: Taxonomía de Flynn.

De estas cuatro categorías, las tres más relevantes y que tienen aplicación práctica son: SISD, paralelismo dentro de un procesador sobre un programa, MIMD, paralelismo de múltiples procesadores sobre múltiples programas, y SIMD, paralelismo que puede ser en uno o múltiples procesadores, y que realiza un mismo procesamiento sobre un conjunto de datos. A continuación se analizaran distintos ejemplos de implementación de estos sistemas

2.1. Paralelismo SISD

El paralelismo SISD nace de una idea simple: que pasa si agregamos una unidad de ejecución secundaria al procesador, y así permitir que este ejecute más de una instrucción al mismo tiempo.

En la figura 1 se observa un diagrama simplificado del computador básico con pipeline, en la cual los registros han sido agrupados en un **register file** o conjunto de registros, y se renombran los registros A y B como R1 y R2. Supongamos ahora que se le agrega una FPU a este computador. Para poder realizar operaciones con esta unidad, es necesario agregar registros de punto flotante al computador, como se observa en la figura 2.

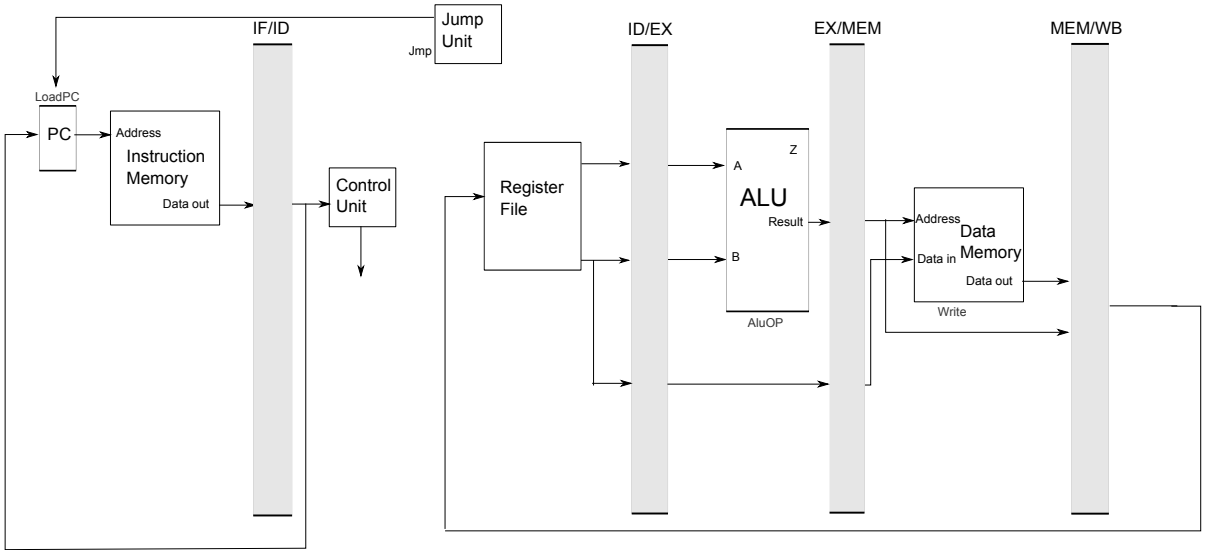

Figura 1: Diagrama simplificado del computador básico con pipeline, agrupando los registros en un register file.

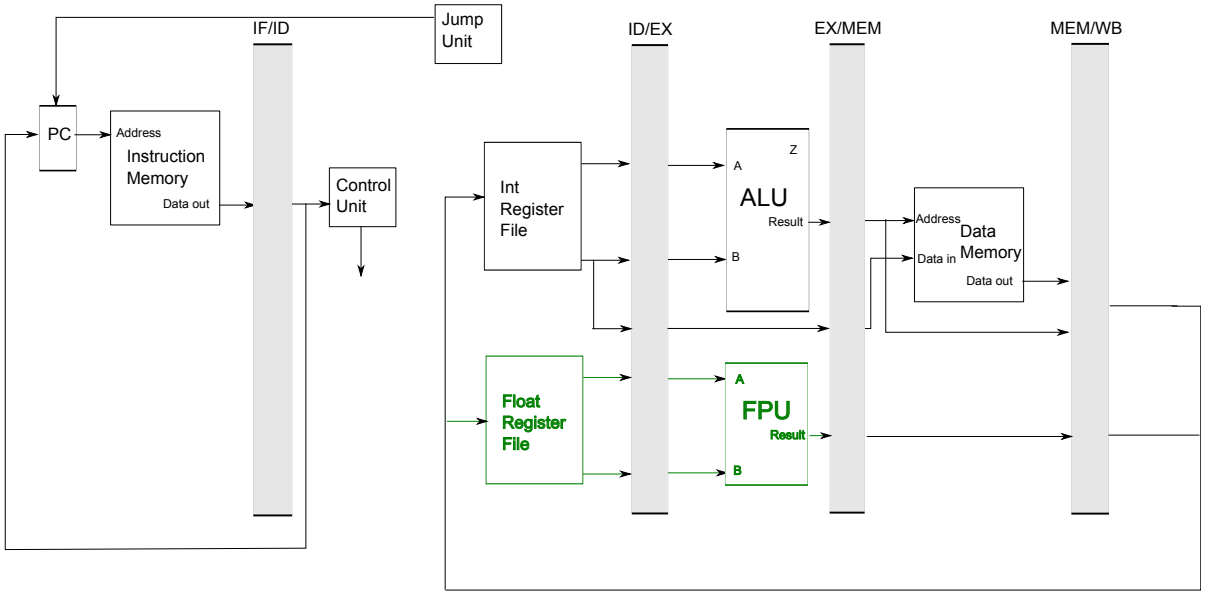

Figura 2: Al agregar una FPU y registros de punto flotantes agregados al computador básico, el flujo de la instrucción puede tomar dos caminos en la etapa de ejecución.

Se puede observar que ahora una instrucción puede tomar dos caminos, dependiendo de si se ejecuta una operación en la ALU o en la FPU. Para lograr ejecutar operaciones en la FPU es necesario agregar instrucciones especiales de punto flotante y permitir el acceso a los registros de punto flotante. Supongamos en este caso que se agrega la instrucción `FADD` para realizar la suma y los registros de punto flotante `F1` y `F2`. Como en general las operaciones aritméticas de punto flotante son más complejas que las con números enteros, la ejecución de una operación de la FPU tomará más tiempo que la ejecución de una operación de

la ALU. Supongamos que la etapa de ejecución de la FPU se puede dividir en tres subetapas: `FEX1`, `FEX2` y `FEX3`. El diagrama del pipeline para la ejecución de las instrucciones `ADD R1, R2` seguida de `FADD, F1, F2` se observa en la figura 3.

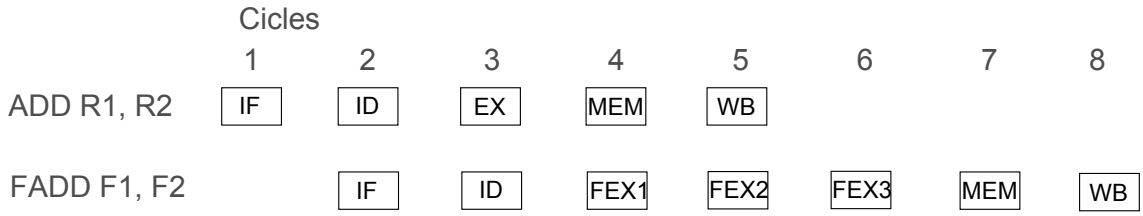

Figura 3: Las instrucciones `ADD R1, R2` y `FADD F1, F2` utilizan registros y unidades de ejecución distintas.

Lo interesante de este par de instrucciones es que están ocupando unidades de ejecución distintas (una la ALU y otra la FPU) y registros de operación y resultado distintas (R1 y R2 la primera, F1 y F2 la segunda). De esta forma las etapas de ejecución y write back de ambas instrucciones son completamente independientes. A partir de esto se intuye que sería posible paralelizar estas instrucciones si las etapas de fetch y decode fueran capaces de trabajar con más de una instrucción al mismo tiempo, como se observa en el diagrama del computador de la figura 4 y el diagrama de pipeline de la figura 5.

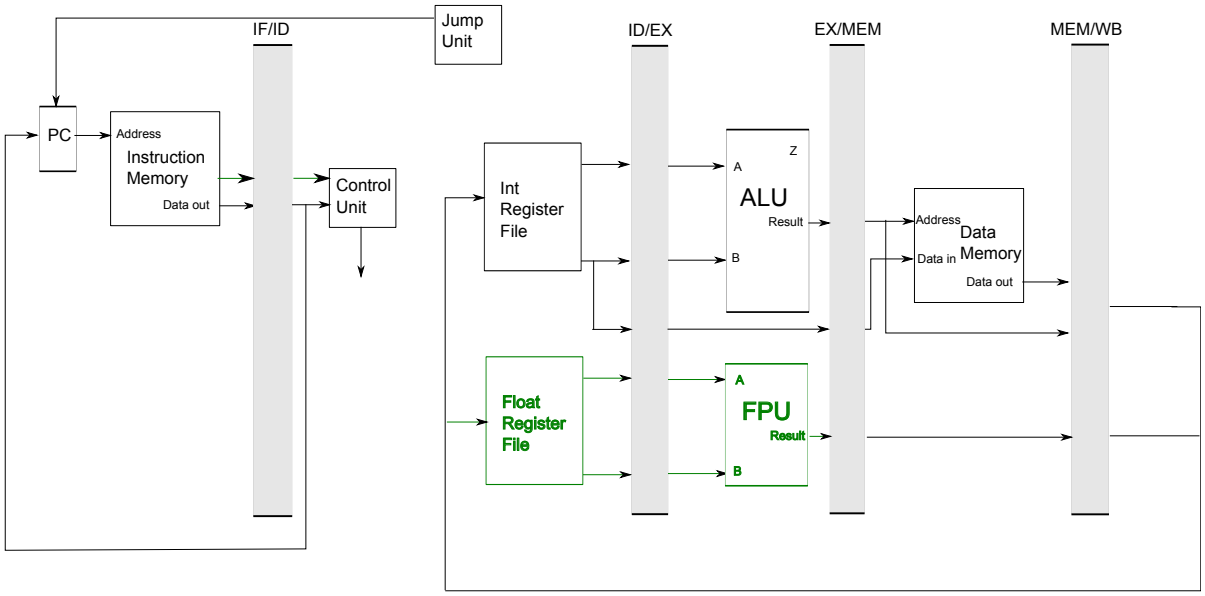

Figura 4: Computador capaz de ejecutar dos instrucciones al mismo tiempo, una ocupando la ALU, la otra la FPU.

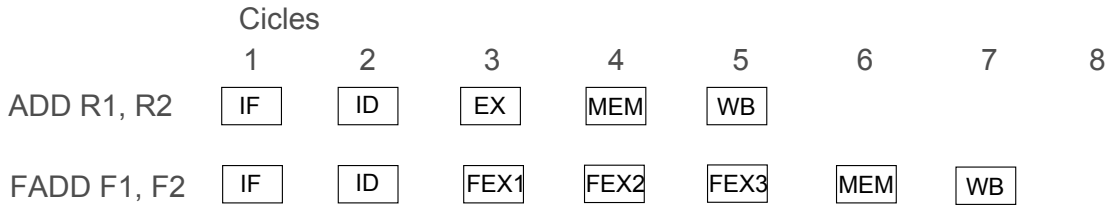

Figura 5: Diagrama de pipeline de dos instrucciones paralelizables.

Podemos ir más allá y extender esta idea, agregando ahora múltiples ALUs, y más registros, lo que permite ejecutar más instrucciones al mismo tiempo, como se observa en el ejemplo de la figura 6. Un procesador que permite obtener, decodificar y ejecutar múltiples instrucciones al mismo tiempo se conoce

como **multiple-issue**. Si es capaz de procesar dos instrucciones al mismo tiempo, será un procesador 2-issue, si procesa 4, un 4-issue, y así. En la figura 7 se observa el esquemea del flujo de las instrucciones en un computador 4-issue.

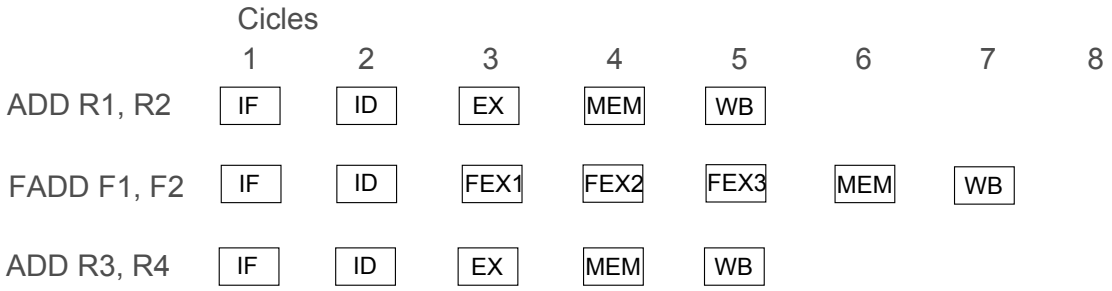

Figura 6: Diagrama de pipeline de un computador con múltiples ALUs y FPU.

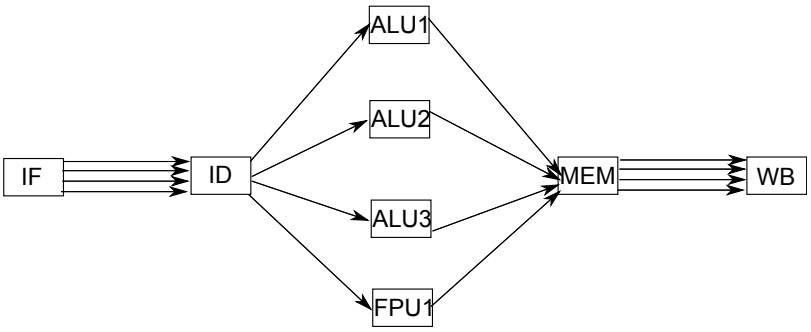

Figura 7: Esquema del flujo de instrucciones en un computador 4-issue.

En un procesador multiple-issue es necesario agregar elementos adicionales que permitan determinar que instrucciones pueden ejecutarse efectivamente en paralelo y cuales son. Existen dos tipos de técnicas para realizar esto: **estáticas** y **dinámicas**. Las técnicas estáticas se basan en que el compilador sea capaz de agrupar los conjuntos de instrucciones paralelizables y enviárselas en conjunto al procesador. Las técnicas dinámicas se basan en que el procesador sea capaz de determinar en tiempo de ejecución que instrucciones paralelizar, pudiendo despachar instrucciones a distintas unidades de ejecución dinámicamente.

Dentro de las técnicas dinámicas, la más utilizada se conoce como **Arquitectura Superescalar**. En este tipo de arquitecturas, antes de cada unidad de ejecución se agrega una estación de reserva o **reservation station** la cual será encargada de ir almacenando los operandos necesarios para una determinada operación, reservándolos hasta que estén todos y solo en ese momento se enviarán a la unidad de ejecución. Luego de que se ejecute la operación, el resultado se almacena temporalmente en una **commit unit** (también llamada reorder buffer) la cual se encargará de enviar los datos a memoria o a los registros sólo cuando haya seguridad de que no hábrán problemas de dependencia de datos (figura 8).

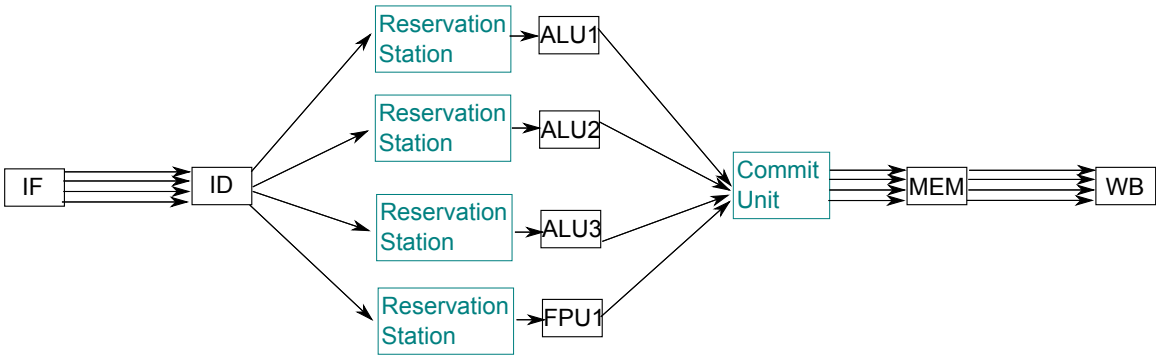

Figura 8: Esquema del flujo de instrucciones en un computador 4-issue superescalar.

Dentro de las técnicas estáticas, la más utilizada se conoce como **Very Long Instruction Word (VLIW)**, la cual consiste en que el compilador agrupe un paquete o **bundle** de instrucciones juntas, las cuales pueden ejecutarse en paralelo, y envíe al procesador este bundle como una «instrucción muy larga» que el procesador directamente envié a distintas unidades de ejecución en paralelo. Para lograr esto, el compilador puede implementar técnicas de **code motion**, es decir, reordenar el código de manera de permitir paralelizar instrucciones que no tienen dependencia de datos, pero sin perder la lógica original del algoritmo, como se observa en las tablas 1 y 2.

Dirección	Instrucción
0x00	Instrucción 1
0x01	Instrucción 2
0x02	Instrucción 3
0x03	Instrucción 4
0x04	Instrucción 5
0x05	Instrucción 6
0x06	Instrucción 7
0x07	Instrucción 8
0x08	Instrucción 9

Tabla 2: Secuencia de instrucciones sin VLIW.

Dirección	Bundle			
0x00	Instrucción 1	Instrucción 6	Instrucción 7	NOP
0x01	NOP	NOP	Instrucción 3	Instrucción 4
0x02	NOP	Instrucción 2	NOP	NOP
0x03	NOP	Instrucción 5	Instrucción 9	NOP
0x04	NOP	NOP	NOP	Instrucción 8

Tabla 3: Secuencia de bundles con VLIW.

Para lograr paralelismo en en arquitecturas multiple-issue es necesario explotar el paralelismo a nivel de instrucción que pueda tener disponible un determinado programa. Existen distintas métricas que de pueden utilizar sobre un programa para determinar su grado de paralelismo. Una de esas métricas se conoce como **Instruction Level Parallelism (ILP)**, la cual mide a nivel de las instrucciones del programa que tan paralelizable es un cierto código. La idea de esta métrica es determinar que tanto más rápido se puede ejecutar un código si se pudiera paralelizar lo más posible.

Para calcular el ILP se asume que cada instrucción toma un ciclo. El ILP se obtiene como la razón entre el número de ciclos que toma el programa sin nada de paralelismo, dividido por el número de ciclos que toma el programa aprovechando al máximo el paralelimso. Por ejemplo el siguiente código toma 4 ciclos en completarse sin paralelismo. Para determinar el número de ciclos con paralelismo hay que revisar la dependencia entre las instrucciones. En este caso, la instrucción 3 depende de la 1, por lo cual no pueden ser ejecutadas en paralelo, entonces las instrucciones 1, 2 y 4 pueden ejecutarse en paralelo (un ciclo), pero la instrucción 3 debe ejecutarse después de la 1, por lo tanto la mínima cantidad de ciclos en que se puede ejecutar el programa considerando paralelismo es 2 ciclos. El ILP en este caso entonces sería $\frac{4}{2} = 2$ lo que se interpreta como que el código puede correr hasta 2 veces más rápido si se tiene todo el paralelimso posible.

```
ADD [var1], 2    /*Instrucción 1*/
SUB [var2], 3    /*Instruccin 2*/
ADD [var1], 5   /*Instrucción 3*/
INC [var3]      /*Instrucción 4*/
```

```
RET

Data:

var1  db  2
var2  db  4
var3  db  5
```

Para determinar el ILP de programas más extensos, una herramienta útil es el **grafo de dependencia** del programa, el cual mapea visualmente todas las dependencias entre las instrucciones de un programa. Una vez obtenido el grafo de dependencias, es fácil determinar la cantidad de ciclos del código al ser ejecutado en paralelo, ya que corresponderá a la cantidad de instrucciones de la rama más profunda del grafo.

Para el siguiente código que originalmente toma 9 ciclos, se muestra su grafo de dependencia en la figura 14. Se puede observar que la rama mas profunda tiene 4 instrucciones, y por tanto al paralelizar el programa se puede reducir el tiempo a 4 ciclos. Con esto, el ILP de esté codigo sería $\frac{9}{4}$.

```
ADD [var1], 2
SUB [var2], 3
ADD BL, [var1]
ADD [var3], BL
MOV [var5], 0
ADD AL, [var5]
AND [var4], AL
XOR [var6], 7
INC [var3]

RET

Data:

var1  db  2
var2  db  4
var3  db  5
var4  db  6
var5  db  7
var6  db  1
```

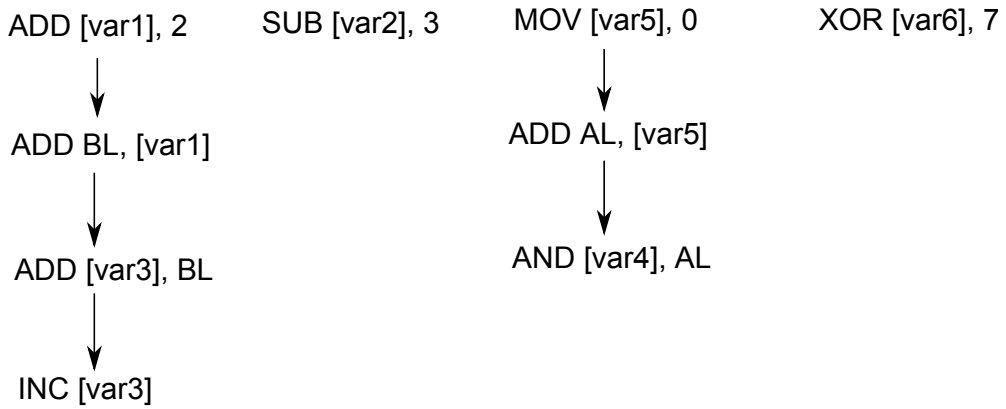

Figura 9: Ejemplo de un grafo de dependencias de instrucciones.

2.2. Paralelismo MIMD

Aunque el paralelismo dentro de un procesador logra mejoras de desempeño, tiene el costo asociado de aumentar la complejidad del procesador, lo que también repercute en el consumo de energía de éste. Una

alternativa para aumentar el rendimiento del computador es en vez de complejizar más el procesador, trabajar con múltiples procesadores simples, repartiendo el procesamiento de múltiples programas independientes entre éstos, lo que se conoce como un sistema **multiprocesador**.

2.2.1. Tipos de multiprocesador

Existen dos tipos principales de multiprocesadores: multiprocesador por **paso de mensaje** y multiprocesador de **memoria compartida**. En un multiprocesador por paso de mensaje, cada procesador tiene una memoria y espacio de direccionamiento propio, y la comunicación con los demás procesadores se realiza a través de algún sistema de red que los interconecte (figura 10).

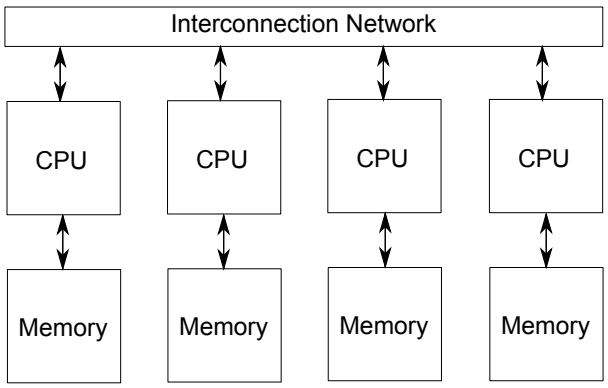

Figura 10: Diagrama multiprocesador de paso de mensajes

Dependiendo de la cercanía que tengan los distintos procesadores, existirán distintos tipos de multiprocesadores de paso de mensaje. Un **cluster** corresponde a un multiprocesador de paso de mensaje formado por distintos computadores comunicados por red local ubicados en una misma locación física. Un **sistema distribuido** corresponde a un multiprocesador de paso de mensajes en el cual los distintos computadores se encuentra ubicados físicamente distantes, pudiendo estar en países distintos, y realizando la comunicación mediante Internet.

El segundo tipo de multiprocesador, los multiprocesadores de memoria compartida se caracterizan por tener un nivel de memoria compartida entre los distintos procesadores (figura 11). Debido a esto es más simple compartir información entre los distintos procesadores, ya que no se requiere estar enviándola a través de la red, pero se generan problemas de sincronización, ya que se debe evitar que ambos procesadores modifiquen un mismo dato de memoria al mismo tiempo.

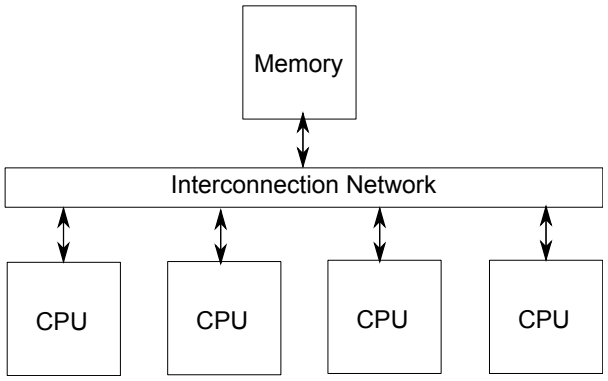

Figura 11: Diagrama multiprocesador de memoria compartida

Un problema particular en la sincronización de dos procesadores ocurre por el hecho de que a pesar de tener un nivel de memoria compartida, es probable que cada procesador tenga un caché no compartido

(figura 12), lo que puede generar problemas de **coherencia de caché**, ya que si un procesador escribe un valor en su caché y esto no se le notifica al otro procesador, la información sobre una misma variable puede quedar almacenada de manera inconsistente en el sistema.

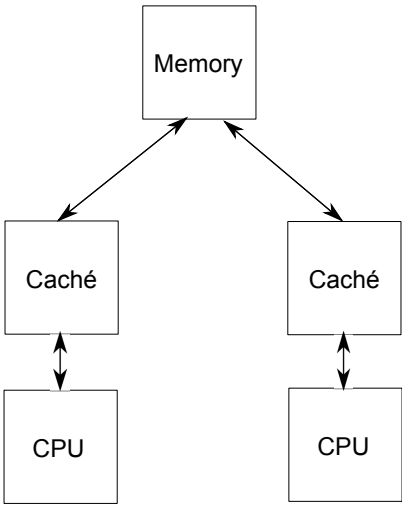

Figura 12: Diagrama de multiprocesador de memoria compartida, con cachés individuales

Un mecanismo para solucionar los problemas de coherencia de caché es el protocolo **MSI**. En este sistema, cada bloque puede estar en uno de cuatro estados:

- Modified: bloque distinto de memoria principal y no existe en otras caches.
- Shared: bloque coincide con memoria principal y puede estar presente en otras caches.
- Invalid: datos no son válidos.

La idea del protocolo es la siguiente (se asume que las cachés tienen política de escritura write-back):

- En un comienzo todos los bloques comienzan en estado Invalid.
- Si un procesador realiza una lectura de un valor de memoria, y lo almacena en su caché, ese bloque pasa a estar en estado Shared en el caché de ese procesador.
- Luego de esto pueden ocurrir distintas alternativas:
  - Si el mismo procesador escribe en ese bloque de caché, el estado del bloque pasa a Modified.
  - Si el segundo procesador lee esa dirección de memoria y lo almacena en su caché, ese bloque pasa a estar en estado Shared en la caché del segundo procesador.
- En caso de que el estado de bloque quedó en Modified, si ahora el segundo procesador intenta leer esa dirección de memoria, ocurrirá un problema de coherencia. Para evitar esto, el controlador de la caché con el bloque Modified, debe encargarse de hacer **bus snooping**, es decir «espiar» el bus de direcciones, para determinar si hubo un acceso a la dirección de memoria asociada al bloque Modified. En caso de detectar un acceso, el controlador le envía una señal a la segunda caché, indicando que el segundo procesador debe esperar (stall), por que si lee ahora habrá problema de inconsistencia. En ese momento el controlador de la primera caché pasa a escribir el bloque en la memoria principal compartida, para que cuando el segundo procesador reintente el acceso, ahora el bloque ya no esté modificado, sino que se haya cambiado a estado Exclusive y no haya problemas de coherencia.

  Supongamos los siguientes ejemplos de accesos a memoria de parte de dos procesadores: Ejemplo 1:

1. El primer procesador ejecuta la instrucción `MOV A, (120)`, copiando el dato de memoria en la dirección 120 a un bloque de la caché. El bloque de caché en este procesador pasa de estado Invalid a Shared.
2. El segundo procesador ejecuta la instrucción `MOV B, (120)`, copiando el dato de memoria en la dirección 120 a un bloque de la caché. El bloque de caché en ambos procesadores pasa a Shared.
3. El primer procesador ejecuta la instrucción `MOV (120), B`, escribiendo en el bloque de caché, el cual pasa a estado Modified.

Ejemplo 2:

1. El primer procesador ejecuta la instrucción `MOV A, (120)`, copiando el dato de memoria en la dirección 120 a un bloque de la caché. El bloque de caché en este procesador pasa de estado Invalid a Shared.
2. El primer procesador ejecuta la instrucción `MOV (120), B`, escribiendo en el bloque de caché, el cual pasa a estado Modified.
3. El segundo procesador ejecuta la instrucción `MOV B, (120)`, intentando copiar el dato de memoria en la dirección 120 a un bloque de la caché. El controlador de caché del primer procesador está haciendo bus snooping y por tanto detecta que se quiere leer un bloque modificado y le envía una señal para que espere. Este controlador se preocupará de escribir inmediatamente en memoria, y actualizar el estado a Shared. Cuando el segundo procesado reintente el acceso, lo podrá hacer, y ahora ambos bloques quedarán en estado Shared.

2.3. Arquitecturas Multicore

Una arquitectura particular de multiprocesadores son las arquitecturas multicore, las cuales no corresponde a un tipo de multiprocesador, sino a una tecnología en la cual los distintos procesadores del sistema están incorporados en el mismo chip. El gran atractivo de estas arquitecturas es que al estar en el mismo chip la comunicación entre los procesadores será mucho más rápida.

Los sistemas multicore pueden ser tanto de paso de mensajes como memoria compartida, aunque habitualmente son de este último tipo, por ejemplo los procesadores multicore usados en los computadores personales. En éstos, en general el nivel de memoria que comparten los procesadores corresponde al nivel L2 de caché, teniendo cada uno una caché L1 independiente (figura 13).

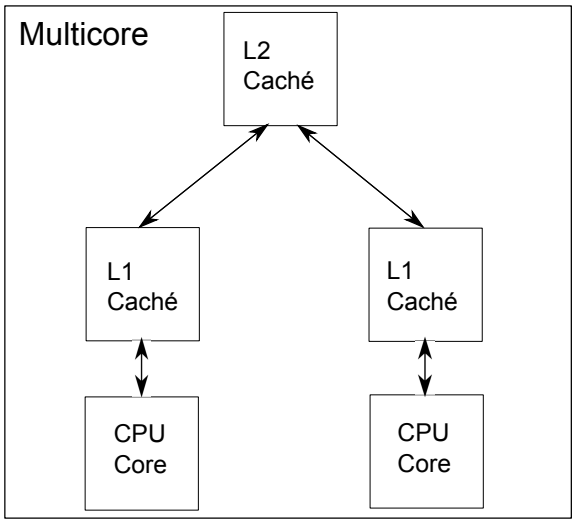

Figura 13: Diagrama de multiprocesador multicore

2.4. Paralelismo SIMD

Una categoría importante dentro de la taxonomía de Flynn son los sistemas SIMD, Single Instruction Multiple Data. La idea de estos sistemas es que en muchas circunstancias se requiere ejecutar la misma operación sobre distintos datos, por lo que es posible implementar naturalmente cierto grado de paralelismo en estos casos. Este paralelismo es particularmente importante para el procesamiento de operaciones vectoriales, que por su naturaleza suelen requerir operaciones de tipo SIMD: sumar un vector con otro, por ejemplo, corresponde a sumar (una instrucción) los distintos pares de componentes de ambos vectores (múltiples datos).

2.4.1. Extensiones Multimedia

Un ejemplo de un sistema SIMD implementado para el procesamiento de operaciones vectoriales corresponde a la extensión para procesamiento de datos multimedia que se incorporó en los computadores x86, denominada **Streaming SIMD Extension (SSE)**. Está extensión se agregó como una unidad de ejecución adicional al procesador, capaz de realizar operaciones sobre 8 registros vectoriales especialmente agregados (`XMM0` a `XMM7`). Estos registros son de 128 bits cada uno, pero para la unidad SSE pueden ser vistos tanto como un único registro, pero también como 4 registros de 32 bits, sobre los cuales se puede ejecutar una misma instrucción. En la figura 9 se observa como ejemplo la instrucción `MULPS XMM1, XMM0`, la cual multiplica cada uno de los 4 subregistros de 32 bits del registros `XMM0` con los del registro `XMM1` y los almacena en `XMM1`.

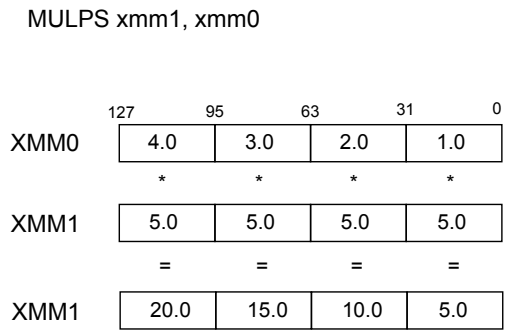

Figura 14: Ejemplo instrucción SSE.

2.4.2. Graphics Processing Unit

Una segunda arquitectura SIMD son las unidades de procesamientos gráficos o GPU. Las GPU son sistemas de multiprocesadores en los cuales cada procesador tiene funcionalidades específicas. En su conjunto, los procesadores de la GPU realizan el proceso de transformar representaciones de objetos 3D (vértices), primero a polígonos, y finalmente a una imagen 2D (figura 15).

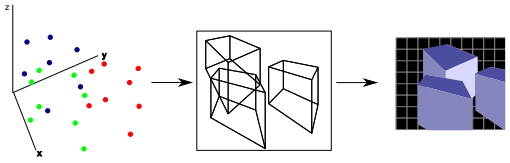

Figura 15: Etapas del procesamiento gráfico.

El procesamiento gráfico tiene la ventaja de ser extremadamente paralelizable, dado que en cada etapa se puede procesar de manera independiente las distinta unidades de información (vértices, polígonos, pixeles), lo que lo hace ideal para una arquitectura paralela de tipo SIMD, ya que se está realizando el mismo procesamiento sobre distintos datos al mismo tiempo. Además, el procesamiento es de tipo streaming, es

decir, es en una dirección, y una vez que un dato pasa por una etapa, ya no vuelve a esta, por lo que no hay dependencia de datos, y se puede «pipelinizar» el proceso (figura 16).

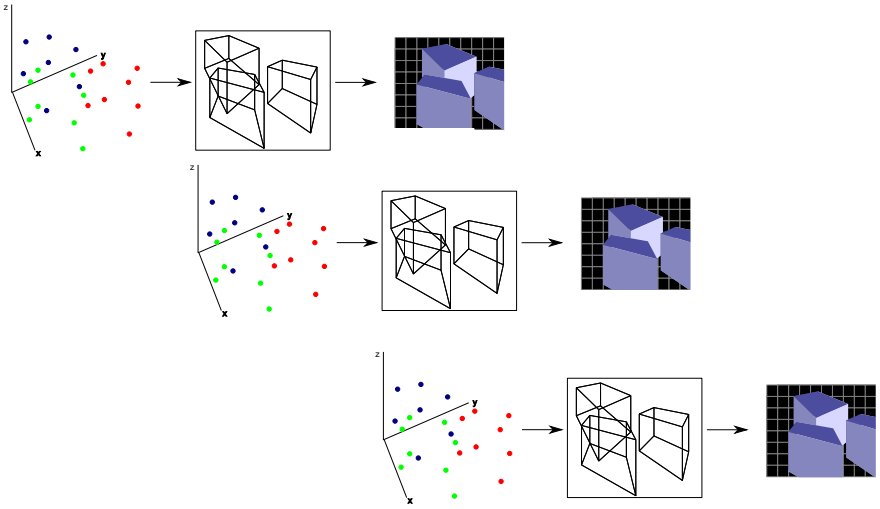

Figura 16: Graphics pipeline.

Las GPUs modernas llegan a tener cientos de procesadores, lo que permite que ejecuten muchas más operaciones por segundo que una CPU, lo que ha impulsado que en los últimos años se hayan comenzado a utilizar para realizar procesamiento general, no solamente gráfico. A pesar de esto, la CPU sigue siendo necesaria, ya que la GPU está optimizada para resolver un tipo particular de problema, pero no es óptima para problemas generales, en los que hayan dependencias de datos, saltos, y procesamiento lógico mas avanzado.

3. Referencias e información adicional

- Hennessy, J.; Patterson, D.: Computer Organization and Design: The Hardware/Software Interface, 4 Ed., Morgan-Kaufmann, 2008. Chapter 7: Multicores, multiprocessors and clusters.