Tarea 5

IIC2343 – Arquitecutura de Computadores
Rafael Fernández Sánchez

Pregunta 1:

a)
En primer lugar, ya que el pipeline tiene más etapas, se necesitan más registros intermedios que guarden el estado de cada una de estas, por lo que el hardware se hace más caro y complejo, se aumenta el delay total (tiempo fijo de acceso a los registros), se necesita más espacio físico y el consumo energético aumenta.
Por otra parte, al haber muchas etapas ocurriendo simultáneamente, se vuelve más compleja la detección y manejo de hazards, lo que resulta en hardware más complicado para las forwarding units y por lo tanto también, más caro.
Respecto al stalling, se puede decir que el costo de hacerlo es más caro, ya que por la cantitdad de etapas es probable que se tengan que esperar varios ciclos para resolver la dependencia.
También, el costo de fallar en una predicción de salto es mucho más alto, ya que se tiene que hacer *flush* de más etapas, causando que se desperdicie más tiempo computacional.
Por último, llega un momento donde el agregar más etapas solo aumenta el tiempo de ejecución. Este momento es cuando la etapa de acceso a memoria es la etapa más lenta, ya que por limitaciones tecnológicas esta no se puede acortar más.

b)
El objetivo de la predicción de salto consiste en aprovechar de mejor manera el pipeline, agregando instrucciones a éste incluso si es posible que no se vayan a ejecutar. De esta forma, en caso de que la predicción de salto sea correcta, no se desperdiciarían ciclos de la CPU.
Sin embargo, existen en problemas en el caso de que la predicción no sea correcta. Estos consisten en que las instrucciones que ya estan siendo procesadas en el pipeline, no deben ser ejecutadadas, por lo que es necesario que se eliminen y se empiecen a cargar en el pipeline las instriucciones correctas del salto. Para realizar esto, es necesario agregar una unidad de hazard de control que detecte el problema y realice un *flush* de las instrucciones inútiles. Esto complejiza el hardware y lo hace más caro.

c)
En el caso de que esta instrucción se ejecute múltiples veces, sería posible buscar cuales son independientes de otras instrucciones, y mediante un compilador agruparlas todas juntas de modo que se ejecuten simultáneamente en el mismo pipeline. Una vez hecho esto, es necesario que una pieza de hardware sea capaz de detectar cuándo se va a ejecutar la instrucción en una etapa en la que no hace nada (Generalmente MEM) y que permita traspasar la información de esa instrucción (Señales de control, valores de registros) a una etapa subsiguiente. De esta forma, no hay ciclos donde hayan etapas inactivas. Cabe notar que, como se habían agrupado previamente las instrucciones con la misma etapa inactiva, al momento de traspasar la información a la etapa subsiguiente no se produce un conflicto entre dos instrucciones utilizando la misma etapa, ya que la instrucción previa también había sido adelantada, permitiendo que la etapa esté libre. Esto es cierto para todas las instrucciones que tienen la misma etapa inactiva, excepto por la primera de ellas, ya que la etapa previa en el pipeline no se puede adelantar, produciendo un "choque". Para solucionar esto, se puede hacer *stalling por software,* agregando un (o más dependiendo de la etapa) NOP antes de la primera instrucción del grupo, de modo que al momento de adelantar las instrucciones en el pipeline no existan conflictos.

d)
Para un computador genérico con las etapas vistas en clases, IF, ID, EX, MEM, WB, es posible que ocurra que en el mismo instante haya una instrucción en la etapa ID leyendo algún registro (MOV A, B) , por ejemplo, que lee el registro B, y que también haya una instrucción en la etapa WB escribiendo este registro (MOV B, 0). En este caso una instrucción trata de leer el registro mientras que otra lo escribe, generando problemas. Para solucionar esto, se podría implementar un mecanismo que permita realizar la lectura de los registros antes que la escritura, permitiendo así que aunque haya una instrucción leyendo y otra escribiendo en el mismo registro. Una de las muchas formas de implementarlo podrías ser hacer uso de los flancos de subida y bajada del clock, dando el valor en el flanco de subida y cargandolo en el de bajada, permitiendo así que se lea antes de que se escriba.

Pregunta 2:

En primer lugar, los procesadores modernos hacen uso de pipelining, por lo que es necesario que de alguna forma se pueda hacer stalling para resolver los hazards que no se pueden arreglar mediante el uso de forwarding. A continuación se explica por qué se puede preferir la implementación por hardware de stalling frente a la de software:

Algunas de las desventajas que tiene el stalling por software respecto al de hardware son:
-Mayor procesamiento a la hora de compilar código
-El compilador tiene que saber exactamente cómo es la arquitectura del computador, ya que de lo contrario podría generar stalls erróneos o no generarlos cuandos se necesitan.
-Ya que el compilador no sabe de antemano los distintos caminos que puede tomar el programa en tiempo de ejecución, tiene que agregar NOPs en todos los posibles casos, ocupando más espacio en memoria (física y caché).

Es debido a estas desventajas (con énfasis en la última) que tiene sentido que se agregue de todas formas una stalling unit a la aquitectura de los computadores, logrando así un diseño más eficiente.
El computador básico no incluye una stalling unit ya que no posee el mecanismo de pipelining. Esto implica que no van a haber dependencia de datos entre una instrucción y otras, por lo que no se producirán hazards y no se necesita hacer stalling para resolverlos.

Aunque para un computador sin pipeline la stalling unit no tiene sentido, una forma de implementarla en caso de que se quisiera, podría ser agregando una compuerta lógica AND, tal que una entrada sea la señal de clock del PC, otra sea una señal de control STALLING con una compuerta NOT (Para que semánticamente tenga sentido) y la salida sea la entrada de señal de clock del PC. La Control unit se tiene que adaptar para poder envíar la señal de control STALLING.
De esta menera, cuando se envía una señal de control de STALLING, la compuerta AND no deja cambiar el clock del PC y por lo tanto este no aumenta, generando un stall.

Pregunta 3:

a)
RISC: Reduced Instruction Set Computer. Es un tipo de arquitectura de set de instrucciones donde el hardware tiene funcionalidades básicas y se deja que las funcionalidades más complejas se implementen por software. De esta forma se tiene un hardware simple, con pocas instrucciones y poco específico, pero códigos más largos.

CISC: Complex Instruction Set Computer. Se ecuentra en el otro extremo de RISC. Acá el hardware tiene muchas funcionalidades y es más específico, generando que la circuitería sea compleja. Al ser tan específico, se tienen muchas instrucciones distintas, pero se pueden escribir programas en menos líneas de código.

Diferencias entre CISC y RISC

CISC	**Criterio**	**RISC**
Hardware	Énfasis	Software
Múltiples	Ciclos por instrucción	Uno
Extendida	Cantidad de instrucciones	Reducida
Traducido a microcódigo	Relación con Assembly	Directa
Compuestos	Modos de direccionamiento	Simples
Corta	Longitud de código Assembly	Extensa

Ventajas RISC:
- Diseño de hardware simple y por lo tanto más barato.
- Una instrucción por ciclo hace que sea más fácil de implementar pipeline.
- Instrucciones son simples y fáciles de entender.

Desventajas RISC:
- El programador del compilador de un debe hacer un gran trabajo para implementar funciones complejas a partir de las funcionalidades básicas que provee el hardware.
- El código generado por un compilador es mucho más largo, lo que consume más memoria.

Ventajas CISC:
- El programador del compilador no debe preocuparse de realizar muchas operaciones ya que la mayoría vienen implementadas a nivel de hardware.
- Programación en assembly más fácil.
- Menos uso de memoria gracias a que para muchas instrucciones se necesitan menos instrucciones (implementación hardware).

Desventajas CISC:
- Hardware más complejo y más caro.
- Implementación de pipel ine es difícil ya que las instrucciones se ejecutan en múltples ciclos.

b) Dado que el pipeline es deseable en los computadores debido a las mejores en rendimiento que provee, no resulta conveniente implementar un procesador CISC con unidades específicas que resuelven completamente las operaciones. Esto se debe a que, para poder implementar un pipeline, es necesario que cada instrucción pase por las mismas etapas y que la ejecución de una instrucción se vaya haciendo por partes. Si se tienen unidades que resuelven la operación completa, se hace difícil mantener cada etapa de un pipeline ocupada con una instrucción, por lo que no es aconsejable. Además, en un computador CISC cada instrucción se puede ejecutar en múltiples ciclos de la CPU, lo que hace también que la implementación del pipeline sea mucho más compleja, ya que no se puede separar uniformemente la ejecución de una instrucción en una cantidad de etapas fijas.
Es por esta razón que conviene más construir un computador RISC que emule la arquitectura CISC mediante la ejecución de multiples instrucciones para cada operación de CISC. De esta forma se puede aprovechar la simplicidad de RISC y el hecho de que cada una de sus instrucciones se ejecuta en un ciclo para tener un pipeline y así tener mejor rendimiento, pero a costa de un codigo assembly más complejo que emule las operaciones.

Pregunta 4:

a)

- SISD: Single Instruction, Single Data. Maneja instrucciones donde cada una está asociada a un dato. Ocurre en un sólo procesador y permite agregar múltiples unidades de ejecución con sus registros respectivos, de modo que se puedan ejecutar muchas operaciones simultáneas en un pipeline, estando varias instrucciones en la misma etapa pero en distintas unidades de ejecución. Un ejemplo de computador SISD es el computador básico con el que hemos trabajado durante el curso.

- SIMD: Single Instruction, Multiple Data. Ejecuta la misma instrucción sobre múltiples datos. Su implementación es común cuando se requieren hacer operaciones vectoriales, por ejemplo, ya que en estos casos se hace la misma operación (Ej: suma) sobre mucho datos (varios vectores). Esto hace que se puedan agregar unidades de ejecución para varios datos y no solo dos, paralelizando la operación. Un ejemplo de SIMD son las unidades de procesamiento gráfico o GPU (véase NVIDIA o AMD), que básica son procesadores especializados en realizar operaciones vectoriales.

- MIMD: Multiple Instruction, Multiple Data. Corresponde a un sistema multiprocesador, donde se tienen varios procesadores ejecutando distintas instrucciones con sus datos respectivos. Estos computadores pueden estar comunicados mediante un mecanismo de paso de mensajes o bien tener una memoria compartida donde transferir la información.
Un ejemplo de esto son la mayoría de los procesadores modernos, que se caracterizan por ser "multicore". Un ejemplo más concreto es el popular Intel Core i5, que en sus últimas versiones tiene 6 procesadores comunicados por una memoria compartida.

- MISD: Multiple Instruction, Single Data. Corresponde a un computador donde se ejecutan múltiples operaciones sobre el mismo dato. Este tipo de paralelismo no es común en la práctica, ya que los casos de usos normales se relacionan de mejor manera con los otros tipos de paralelismo. Si extendemos un poco la definición y nos refiremos a "Dato" como lo que procesa el computador al inicio de una serie de operaciones (independiente del estado durante estas operaciones), podríamos decir que un computador RISC con pipeline, como el visto en los apuntes de paralelismo a nivel de instrucción, página 17, es un computador MISD. Esto se debe a que en cada etapa del pipeline (operaciones distintas) se estaría trabajando sobre el mismo dato. (El que entró al pipeline).

Cuadro comparativo arquitecturas paralelas

	SISD	SIMD	MIMD	MISD
Instrucciones	Única	Única	Múltiple	Múltiple
Datos por Instruc.	Único	Múltiple	Mútiple	Único
Ventaja	Mayor rendimiento en único procesador	Gran rendimiento en operaciones matriciales	Rendimiento mejorado con múltiples procesadores	Eficiencia en ciertas operaciones mátematicas (arreglos sistólicos)
Desventaja	Hardware más complejo	Bajo rendimiento en operaciones generales	Se necesitan complejos mecanismos de sincronización	Pocos casos de uso en la práctica

b)
Para determinar cuáles son las operaciones que sacan más beneficio de SIMD, se analizarán distintos casos de uso en el cuadro a continuación

Caso de uso	Detalles de Operación	Arq. Recomendada
Ejecutar una simulación	Muchas operaciones distintas, saltos condicionales	SISD
Ejecutar Servidor Web	Manejo de usuarios concurrente, resistencia ante falla de una máquina	MIMD
Multitarea	Manejo de múltiples programas independientes	MIMD
Jugar Fortnite	Operaciones mátemáticas intensivas sobre los pixeles de la pantalla	SIMD

Se puede ver que las operaciones que no son uniformes, como los muchos casos diferentes que se pueden dar en una simulación, o bien tareas que requieren ejecución de múltiples operaciones distintas de manera simultánea, no son el punto fuerte de SIMD, sin embargo, Jugar Fornite, según el cuadro, lo es.

Esto se debe a que una de las operaciones que tienen muy buen desempeño en las arquitecturas paralelas SIMD son las vectoriales. Esto, ya que los vectores corresponden a grandes arreglos de datos, y cuando se realiza una operación matemática sobre ellos, se realiza la misma acción sobre cada uno de los datos del arreglo, aprovechando así las bondades de SIMD. Es por esta razón que para el procesamiento de gráficos de computador, que involucra operaciones con matrices, se utilizan GPUs, que son múltiples procesadores con arquitectura paralela SIMD.

c)

Sí es posible tener de manera simultánea paralelismo del SISD y SIMD. Para que esto sea posible, se requiere las distintas unidades de ejecución a lo largo del pipeline (ALU, FPU, etc) sean capaces de manejar múltiples datos al mismo tiempo (muchos entradas y salidas por unidad). Un caso donde esta situación se puede dar es el consumo de multimedia por internet. En este caso, la información llega al computador comprimida (Ej: formato .mp4), por lo que hay que decodificarla mediante una serie de instrucciones distintas (SISD) y luego realizar operaciones sobre los pixeles de la pantalla para mostrar la información (SIMD). Es para este caso de uso principalmente que se agregó la "Streaming SIMD Extension" a la arquitectura x86.

Pregunta 5)

1)

Bits Dir. Virt	Bits Dir. Fís	Tam. Pág	Bis Pág.	Bits Marco	Bits entrada
32	32	16	18	18	22
32	26	8	19	13	17
36	32	32	32	17	21
40	36	32	25	21	25
64	40	64	48	24	28

Nota: Me quedé sin tiempo para explicar los cálculos, tengo que entregar :(

2)

No tiene sentido tener una memoria caché de mayor tamaño que la memoria principal, ya que la primera guarda un acceso rápido a la información de la segunda, de modo que es imposible tener más bloques de información escritos en la caché que en la memoria principal, siendo inútil entonces el espacio extra. Además, en la práctica, los costos de tener una caché con el tamaño de una memoria principal serían extremadamente caros, haciendo que esta opción sea inviable económicamente.

El análisis no cambia para un tamaño igual al de la memoria direccionable, ya que este es, generalmente, mucho mayor al de la memoria principal, produciendose lo explicado anteriormente.

Cabe mencionar que en un caso muy poco común donde la memoria direccionable sea de igual o menor tamaño que la memoria física, tiene sentido reemplazar la memoria principal por una caché, que escencialmente se comportaría como una memoría física pero más rápida.