Tema 09: Temas avanzados de pipelining

Arquitectura de Computadoras

Ing. Nicolás Majorel Padilla (npadilla@herrera.unt.edu.ar)

http://microprocesadores.unt.edu.ar/arqcom/

Temas que veremos

- Superpipelining.
- Predicción de saltos.
- Paralelismo a nivel de la instrucción (ILP).
  - Procesadores superescalares y VLIW.
  - Desenrollado de lazos.
  - Ejecución fuera de orden.
  - Ejecución especulativa.
  - Limitaciones del ILP.
- Manejo de Interrupciones/Excepciones.

Lectura recomendada

- Computer Organization and Design, RISC-V Edition (2da ed, 2021)
  - Sección 4.9: *Control Hazards*
  - Sección 4.11: *Parallelism via Instructions*
  - Sección 4.10: *Exceptions*
  - Sección 4.15: *Fallacies and Pitfalls*
  - Sección 4.16: *Concluding Remarks*
- Para los más curiosos:
  - Sección 4.12: *The Intel Core i7 6700 and ARM Cortex-A53.*

Recapitulación

- Diseñamos un procesador en pipeline.
  - T pequeño, CPI = 1. Genial.
- Pero tiene riesgos.
  - Los riesgos aumentan el CPI.
  - Los estructurales los solucionamos por diseño.
  - Los de datos los solucionamos por adelantamiento (+Hw) y por reordenamiento (Sw).
  - Los de control los minimizamos, y tratamos de evitarlos mediante predicción.
- Sin embargo, los procesadores reales son **mucho más complejos** que nuestro ejemplo.
  - Veremos algunos de esos problemas, y sus soluciones.
  - Con ejemplos reales y actualizados.

Superpipelining – Idea general

- Nuestro diseño tiene 5 etapas, y produce una aceleración igual a 4 (sin riesgos).
  - *¿Por qué la aceleración no es 5?*
- Sin embargo, recordando el Tema 3, podríamos dividir aún más las etapas.
  - P.ej: hagamos 8 etapas de 100 ps.
  - La teoría dice que aumentaríamos la performance.
  - ¡Y es cierto! Los pipelines de procesadores modernos de alta performance suelen tener más de 5 etapas.

Longitudes típicas de pipelines modernos

- Dependen del tipo de procesador.
  - Procesadores pequeños, optimizados en área, suelen tener entre 1-3 etapas.
  - Procesadores medianos suelen tener entre 5-10 etapas.
  - Procesadores de alta performance suelen tener entre 10-20 etapas.
    - AMD Ryzen 9 7900X3D (2023): 19 etapas.
    - ARM Cortex X2 (2023): 13 etapas.
    - Intel Core i7-13700K (2022): entre 14 y 19 etapas.
    - SiFive P870 (2024): entre 10 y 18 etapas.
- El procesador con más etapas que hubo fue un Pentium 4 (2004), con 31 etapas.

Superpipelining – Inconvenientes

- **Validez del modelo:** se debe poder dividir una etapa.
  - No podemos hacer media operación en la ALU, o media búsqueda en memoria.
  - Los retardos de los buffers intermedios pasan a ser significativos.
- El aumento de frecuencia implica un **aumento del consumo de energía**.
- **Los riesgos se incrementan:**
  - Mayor posibilidad de riesgos estructurales.
  - La unidad de adelantamiento se puede volver muy compleja.
  - El riesgo adicional del LW puede requerir más de una burbuja.
  - La penalidad por no acertar la predicción de saltos puede ser muy grande.
- El último inconveniente es el más difícil de solucionar.

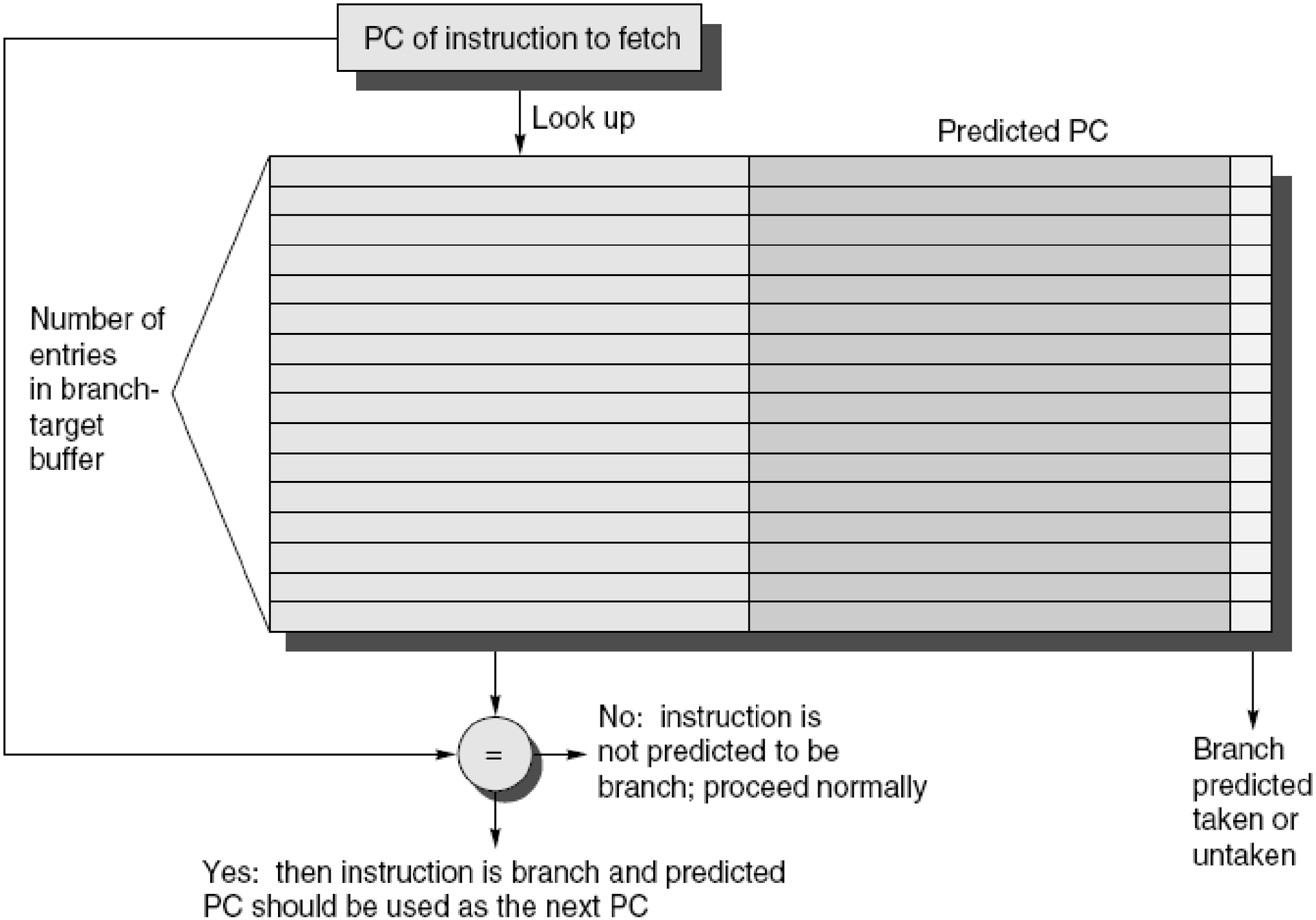
Predicción de Saltos Estática

- Predicción más simple: nunca saltan (o siempre saltan).
  - Si acierto, mejoro. Si no, estamos como antes.
- Mejor predicción: BTFNT (*Backwards Taken, Forwards Not Taken*).
  - Basada en el comportamiento típico de los saltos.
    - Los saltos hacia atrás son casi siempre tomados (lazos).
    - Los saltos hacia adelante son casi siempre no tomados (sentencias if).
- Prácticamente no se usa más ninguna.
  - Porque genera porcentajes de acierto inaceptables actualmente.

Predicción Dinámica de saltos

- Se basa en mantener una historia de cada salto, y repetir lo que pasó al último.
  - Se mantiene una **Tabla de Historia de Saltos (BHT)**, que guarda:
    - La dirección origen del salto (para identificarlo).
    - Bits para saber su historia (para predecirlo).
  - Se mantiene también una especie de **cache de las direcciones destino de los saltos (BTB)**.
    - Para no calcularla nuevamente.
  - Si se falla la predicción, actualizar el historial.

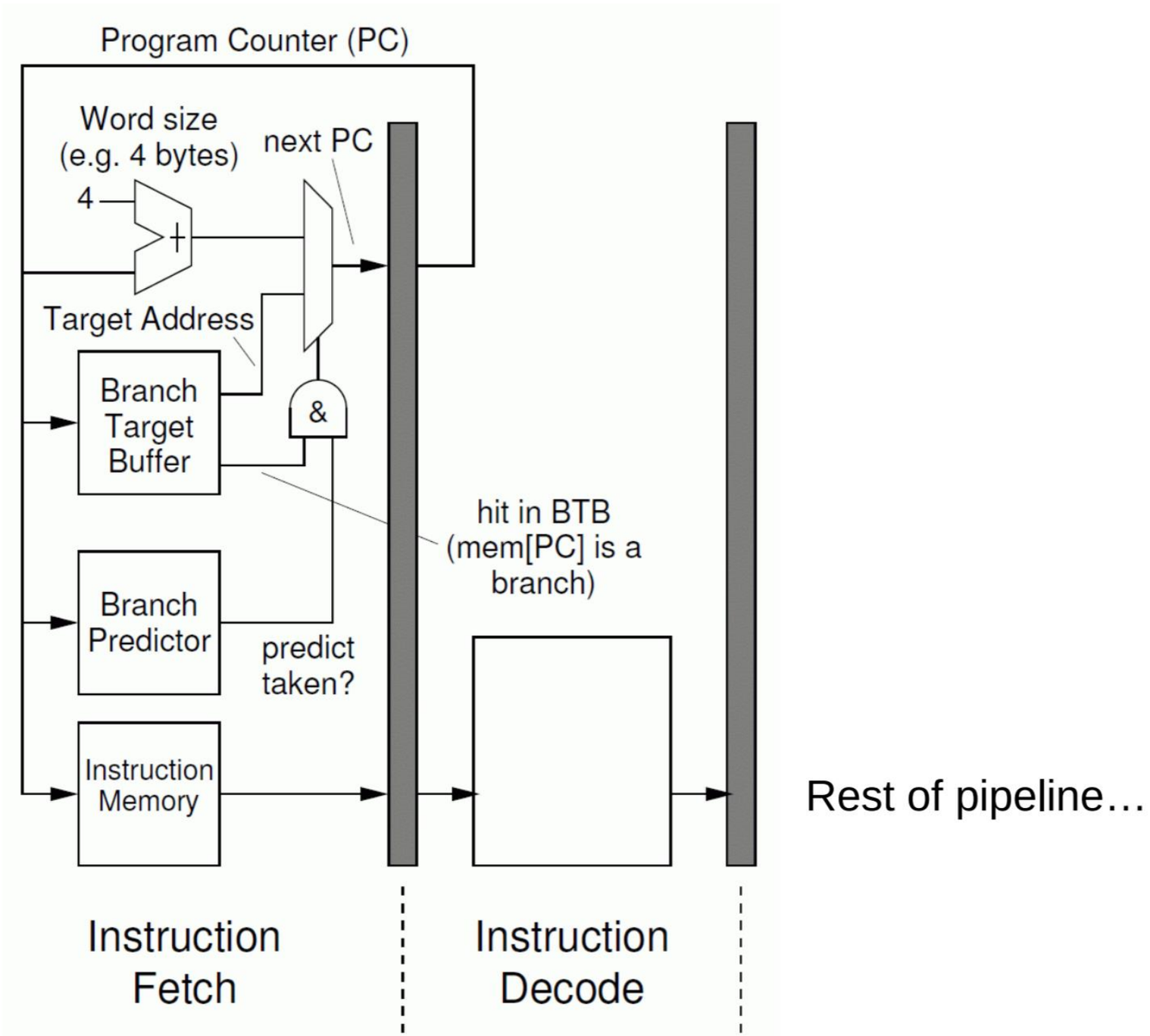
Predicción Dinámica de saltos



Predicción Dinámica de saltos

- Al buscar una instrucción, se revisa la BHT para saber si es un salto.
  - Si lo es, se espera la misma salida de la última vez, y se prepara la dirección destino (del BTB o PC+4).
  - Si no se acierta la predicción, se vacía el pipeline y se actualiza la predicción.
    - Nótese que con BHT y BTB, toda la resolución del salto se puede hacer en la etapa de Fetch.
- *¿Cuántos bits se usan para mantener la historia?*
  - 1 bit: lo más simple, pero tasa de acierto no tan alta.
  - 2 bits: sólo se cambia la predicción al no acertar dos veces consecutivas.
    - Usa una MEF simple de cuatro estados.

Predicción Dinámica de saltos



Predicción Dinámica de saltos

- Aún hoy, es un tópico muy investigado.
  - Se prueban múltiples algoritmos de predicción.
    - ¡Algunos diseños usan redes neuronales!
  - Se prueban múltiple cantidad de bits para la historia.
  - Más info en: https://danluu.com/branch-prediction/
- Procesadores modernos incrementan lo más que pueden la BHT y el BTB.
  - Se busca tasas de predicción cercanas al 99%, para todos los saltos.
  - Se prefiere continuar invirtiendo en Hw, en vez de pagar la penalidad.
    - Impulsados por la Ley de Moore.

Paralelismo a nivel de la Instrucción (ILP)

- Con el diseño en Pipeline, obtuvimos una muy buena performance.
  - CPI = 1, y una duración del ciclo T pequeña.
- Con superpipelining podemos aumentar aún más la performance.
  - Haciendo T más pequeño, manteniendo CPI = 1.
  - En ambos casos, sin considerar los riesgos.
- *¿Podemos mejorar aún más la performance?*
  - *¿Podemos hacer que CPI < 1?*
  - Claro, poniendo **múltiples pipelines en paralelo**, en un diseño **superescalar**.

Procesadores Superescalares

- Un procesador en pipeline con CPI = 1 tiene dos características implícitas:
  - En cada ciclo se emite (*issue*) una nueva instrucción.
  - En cada ciclo se finaliza (*commit*) una instrucción.
  - Y esta es una gran **limitación del pipelining estándar**.
- Un procesador superescalar busca emitir (y finalizar) más de una instrucción en cada ciclo.

```
      1   2   3    4    5    6    7    8    9

i     IF  ID  EX   MEM  WB
i+1   IF  ID  EX   MEM  WB
i+2       IF  ID   EX   MEM  WB
i+3       IF  ID   EX   MEM  WB
i+4           IF   ID   EX   MEM  WB
i+5           IF   ID   EX   MEM  WB
i+6                IF   ID   EX   MEM  WB
i+7                IF   ID   EX   MEM  WB
i+8                     IF   ID   EX   MEM  WB
i+9                     IF   ID   EX   MEM  WB
```

Procesadores Superescalares

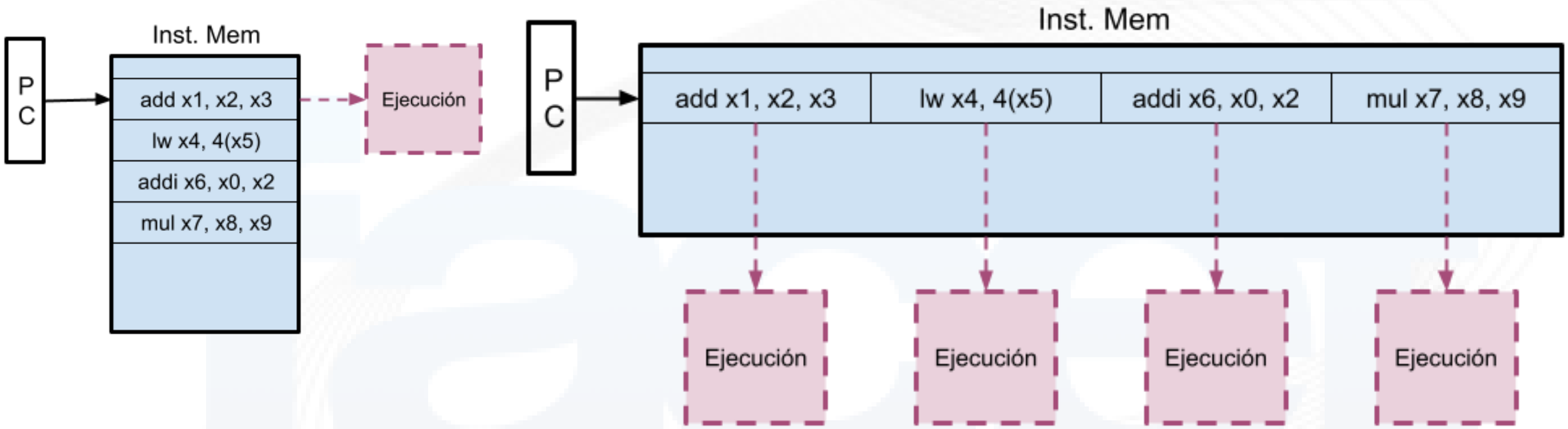
- Cuando CPI < 1 suele invertirse y empezar a llamarse IPC.
- Para ello se basan en tres ideas fundamentales:
  - Emisión dinámica de múltiples instrucciones.
  - Ejecución fuera de orden.
  - Ejecución especulativa.

Emisión múltiple de instrucciones

- Varios pipelines en paralelo no tienen varias memorias de instrucciones.
  - Siempre es una única memoria.
  - Pero debería proveer instrucciones a todos los pipelines en paralelo.
  - Hay dos maneras: **emisión estática y dinámica**.
- **Emisión estática de múltiples instrucciones**.
  - El compilador agrupa las instrucciones que puedan ser emitidas juntas en el mismo ciclo.
  - Se arman “paquetes” según los recursos disponibles.
  - El compilador también se encarga de detectar y evitar los riesgos.
    - Reordenando para evitar dependencias en un paquete.
    - Rellenando con instrucciones *nop* en caso de no poder completar un paquete con instrucciones útiles.

Very Long Instruction Word (VLIW)

- Se puede pensar a esos “paquetes” como instrucciones “largas”.

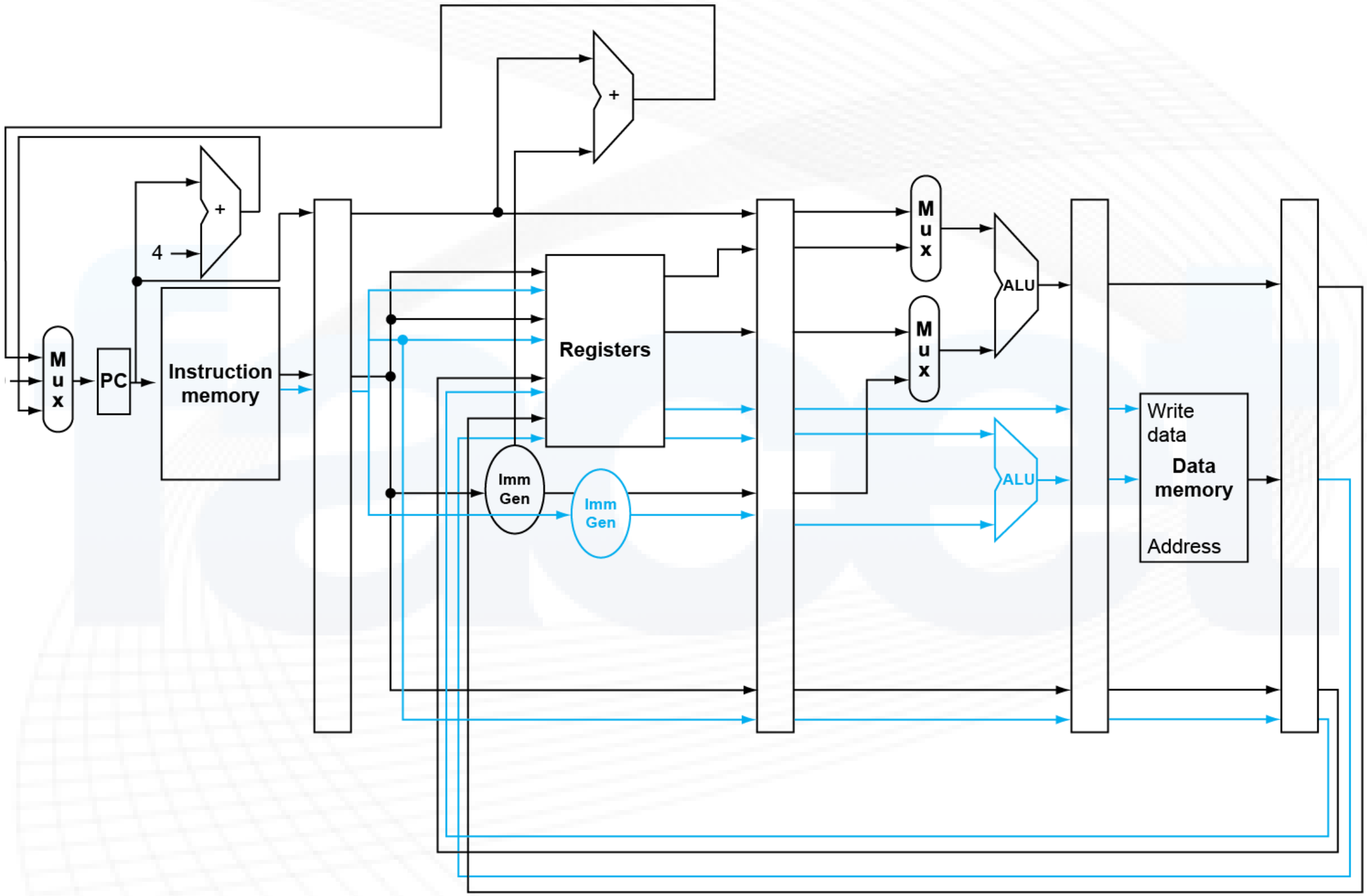


- Si tenemos varios pipelines disponibles, las instrucciones son “muy largas”.
- No es muy usada actualmente, por problemas que veremos en instantes.

Camino de datos con emisión doble

- Veamos cómo sería un ejemplo de nuestro camino de datos, con emisión estática de dos instrucciones por ciclo (*dual issue*).
  - Una instrucción puede ser de Tipo R o Branch, y la otra puede ser Load/Store.
    - El compilador debería acomodarlas así en memoria.
  - De la etapa IF se leen dos instrucciones.
    - Se deben duplicar los puertos de lectura en el banco de registros para la etapa ID.
    - Agregamos una ALU en la etapa EX para el cálculo de la dirección de memoria.
    - La memoria de datos sólo se conecta a esta nueva ALU.
    - Se debe duplicar el puerto de escritura en el banco de registros para la etapa WB.
- El área aumenta aprox. un 25% con respecto al segmentado.

Camino de datos con emisión doble



Ejemplo procesador *dual-issue*

- Supongamos el siguiente bloque de código de RISC-V:

```
Loop: lw   x31,0(x20)      // x31=elemento de un vector
      add  x31,x31,x21     // sumamos el valor en x21
      sw   x31,0(x20)      // guardamos resultados
      addi x20,x20,-4      // decrementamos el puntero
      blt  x22,x20,Loop    // repetimos si x22 < x20
```

- Identificamos las dependencias y los riesgos.
  - Recordemos que un camino para Ld/St y otro para TipoR/B.
  - Que podemos reordenar las instrucciones.
  - Y que rellenamos con nop si no hay otra alternativa.
- *¿En cuántos ciclos se ejecutaría este código en el procesador segmentado original?*

Ejemplo procesador *dual-issue*

- Quedaría de la siguiente manera:

```
Loop: lw   x31,0(x20)     ; nop
      nop                 ; addi x20,x20,-4
      nop                 ; add  x31,x31,x21
      sw   x31,4(x20)     ; blt  x22,x20,Loop
```

- Dejamos un ciclo entre el lw y el add (para evitar la burbuja).
- Movemos el addi antes del add (no hay dependencia).
- El sw se mantiene un ciclo después que add (por la dependencia)
- Ajustamos el offset de sw (porque movimos el addi).
- Movemos el blt junto con el sw (no hay dependencia).
- *¿En cuántos ciclos podemos ejecutar el código?*
  - *¿Cuánto sería el IPC? ¿Cuánto sería el IPC máximo?*
  - *¿Cuánto sería la eficiencia? ¿Cuánto sería la aceleración con respecto al procesador segmentado?*

Ejemplo procesador *dual-issue*

- El ejemplo anterior nos muestra varios inconvenientes:
  - IPC real lejos del máximo teórico. Baja eficiencia.
  - Limitado **fuertemente** por las dependencias.
  - El adelantamiento es mucho más complejo.
    - Un ciclo de demora por culpa de LW, ahora perdemos dos instrucciones.
- Si emitiésemos más instrucciones, estos problemas se agravarían.
- Sin embargo, como la emisión es estática, el compilador puede ayudar aún más.
  - Un ejemplo: **desenrollando el lazo**.

Desenrollado de lazos – Idea

- Es una técnica utilizada por el compilador para mejorar la performance de un programa que incluya lazos.
  - En inglés es *Loop Unrolling*.
- Idea: repetir el cuerpo del lazo para exponer más paralelismo.
  - Reduce el overhead por el control del lazo.
  - Brinda más instrucciones para reordenar.
- Puede ser usada tanto en los procesadores simples como en los superescalares.

Desenrollado de lazos – *Single issue*

- Desenrollando 3 veces, para hacer 4 iteraciones “viejas” en 1 iteración “nueva”, y reordenando:

```
Loop: lw   x28,0(x20)      // cargamos los 4 elementos
      lw   x29,4(x20)      //
      lw   x30,8(x20)      //
      lw   x31,12(x20)     //
      add  x28,x28,x21     // sumamos el valor en x21
      add  x29,x29,x21     // a cada elemento
      add  x30,x30,x21     //
      add  x31,x31,x21     //
      sw   x28,0(x20)      // guardamos resultados
      sw   x29,4(x20)      //
      sw   x30,8(x20)      //
      sw   x31,12(x20)     //
      addi x20,x20,-16     // decrementamos el puntero
      blt  x22,x20,Loop    // repetimos si x22 < x20
```

Desenrollado de lazos – *Single issue*

- *¿En cuántos ciclos se ejecuta el código de la diapositiva anterior?*
  - *¿Aceleración con respecto al procesador segmentado original?*
- Se puede obtener una mejora de performance “casi gratis”.
  - Es un optimización muy utilizada por los compiladores.
- Desventajas:
  - Se necesitan más registros.
    - Para que cada iteración sea independiente de la anterior.
    - Para evitar dependencias de nombre en el mismo registro (antidependencias).
    - ¡Es bueno contar con muchos registros!
  - El código ocupa más memoria.
  - La cantidad de veces a desenrollar depende de la cantidad de iteraciones del lazo.

Desenrollado de lazos – *Dual issue*

- *¿En cuántos ciclos podemos ejecutar el código?*

```
Loop: lw   x28,0(x20)    ; nop
      lw   x29,4(x20)    ; nop
      lw   x30,8(x20)    ; add  x28,x28,x21
      lw   x31,12(x20)   ; add  x29,x29,x21
      sw   x28,0(x20)    ; add  x30,x30,x21
      sw   x29,4(x20)    ; add  x31,x31,x21
      sw   x30,8(x20)    ; addi x20,x20,-16
      sw   x31,-4(x20)   ; blt  x22,x20,Loop
```

- *¿Cuánto sería el IPC? ¿Cuánto sería la eficiencia?*
- *¿Cuánto sería la aceleración con respecto al procesador segmentado?*
- ¡Aumentamos IPC y eficiencia!
  - Pero no llegamos al máximo.
  - Y se necesita un super compilador, que depende del ISA.

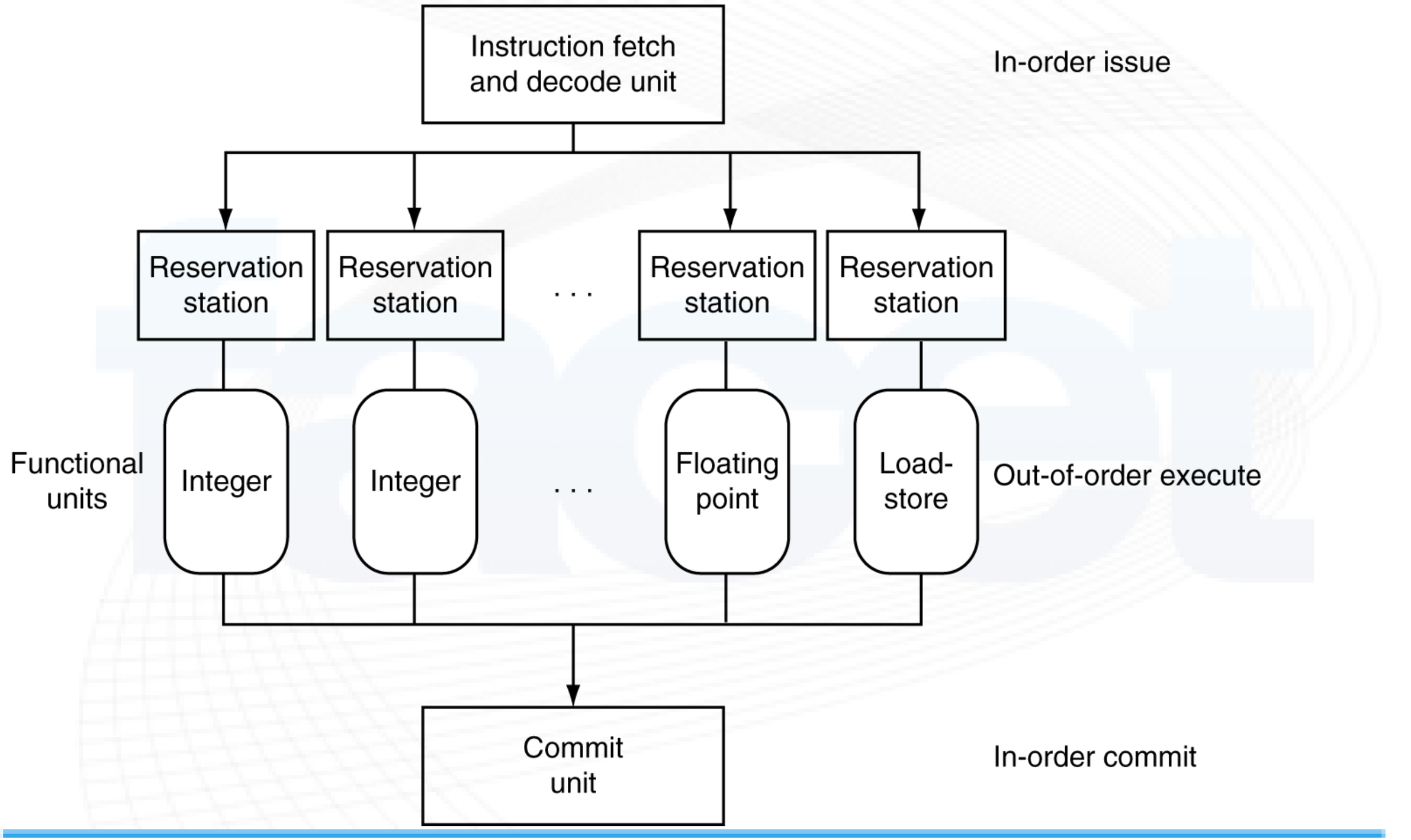
Emisión Dinámica de Múltiples Instrucciones

- En este caso, el mismo procesador analiza la secuencia de instrucciones y decide cuáles instrucciones emitir en cada ciclo.
  - Basado en un **diagrama de precedencias**.
  - En cada ciclo pueden emitirse 0, o 1, o 2, o más instrucciones.
    - Si emite *n* instrucciones por ciclo, se denomina *n-wide*.
    - Es el mismo procesador el encargado de detectar y resolver los riesgos (de datos y estructurales), **en tiempo de ejecución**.
- Se evita la necesidad de un super compilador.
  - Aunque igual puede ayudar también reordenando algunas instrucciones o con otras optimizaciones.
  - Se agrega más Hw para independizarse del Sw (Moore).

Ejecución fuera de orden (OoO)

- La manera de evitar paradas del pipeline en la ejecución dinámica, es permitiendo que **las instrucciones se ejecuten en un orden distinto del que están en el código**.
  - Se ejecuta primero la que pueda ejecutarse, que no tenga dependencias.
- Pero **se siguen terminando en orden**, para mantener el sentido del código.
- **Emisión en orden → Ejecución fuera de orden → Terminación en orden.**
  - Implica que hacen falta buffers antes y después de la ejecución.

Esquema general de un procesador OoO



Esquema de un procesador OoO

- Como la emisión es en orden, se preservan las dependencias.
  - **Preservar el flujo de datos es una de las dos propiedades esenciales para asegurar la exactitud de un programa.**
- A las etapas de IF e ID se las suele denominar “Front-end”.
  - Son encargadas de traer todas las instrucciones que puedan, para mantener siempre las estaciones de reserva llenas.
- Las estaciones de reserva mantienen las instrucciones en espera hasta que puedan ser ejecutadas.
  - O sea, hasta que todos sus operandos estén disponibles.
  - También se las denomina etapas de “*Scheduler*”.

Esquema de un procesador OoO

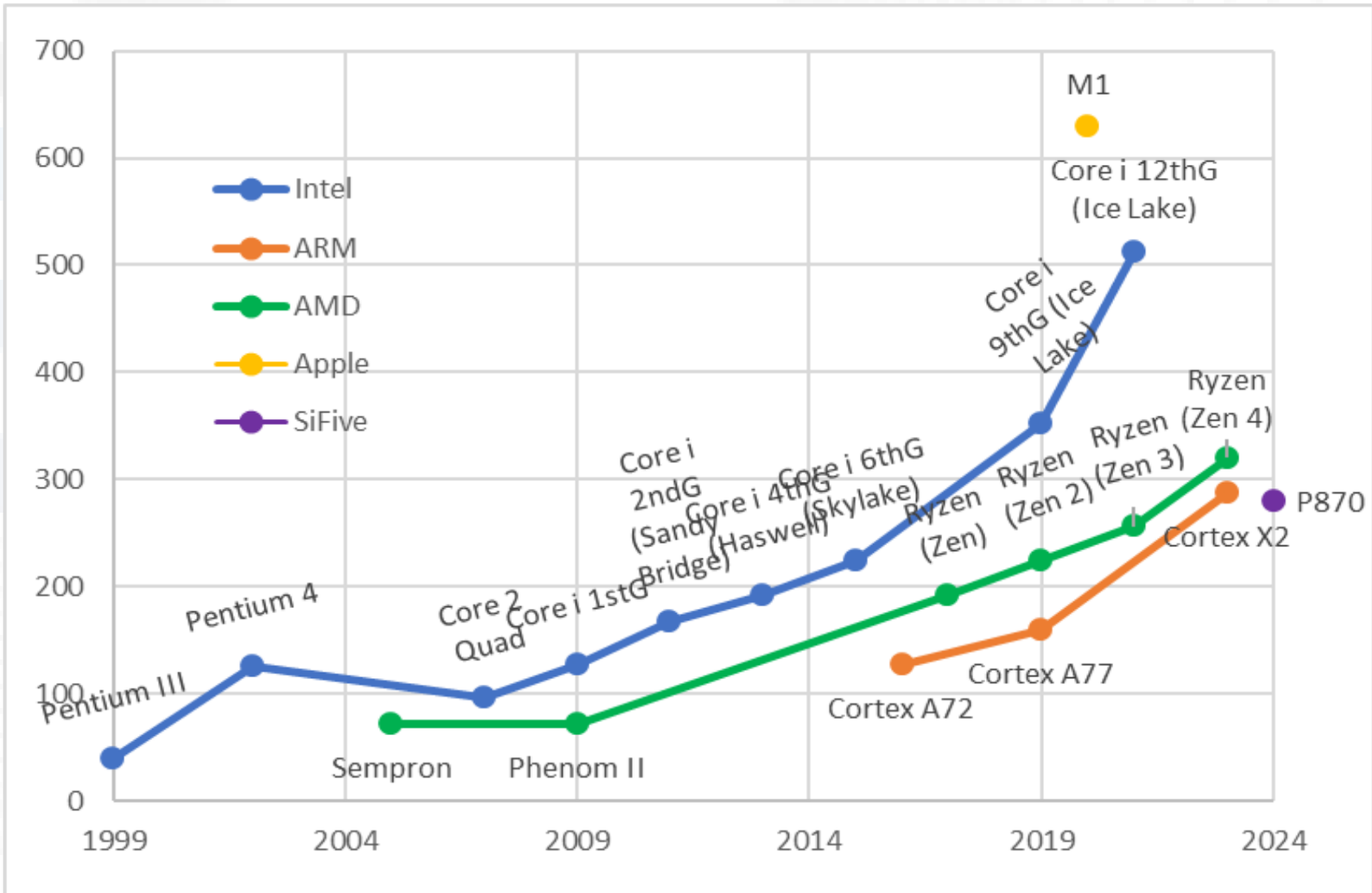
- Hay múltiples unidades funcionales diferentes en paralelo en la etapa de ejecución.
  - Conforman el “Back-end”.
  - Y se requieren caminos de adelantamiento entre todos ellos.
- La unidad de terminación se encarga de reordenar las instrucciones, para asegurar que los registros se escriban en el orden deseado.
  - Se encarga que los resultados calculados se vuelvan permanentes.
  - Tiene un **buffer de reordenamiento (ROB)**.

Evolución del tamaño del ROB

- Cuanto mayor sea el ROB, más instrucciones pueden ser mantenidas en ejecución al mismo tiempo.
  - Desde otro punto de vista, cuantas instrucciones adelante se pueden ir buscando instrucciones independientes para ejecutar.

Los más curiosos pueden probar en:

https://github.com/travisdowns/robsize



Renombramiento de Registros

- En las estaciones de reserva se puede cambiar el “nombre” de los registros, de modo de eliminar dependencias.
  - **Los procesadores modernos tienen muchos más registros físicos que los que se exponen en el ISA.**
    - Por ejemplo, el P870 de SiFive posee 228 registros.
  - Se mantiene una especie de tabla con todos los operandos que están siendo utilizados por instrucciones en ejecución (*RAT, Register Alias Table*).
  - Cuando se emite una instrucción, se revisa si el operando está disponible en el banco de registros o en el ROB.
    - Si está libre, se lo copia a la unidad de reserva y puede ser sobrescrito.
  - Después vuelven a su nombre original en la unidad de terminación.
  - Algoritmo diseñado por Tomasulo en 1967 para IBM.

Emisión Estática vs. Dinámica

- La emisión dinámica con la ejecución fuera de orden es **mucho más compleja** que la estática.
- Sin embargo, es la más usada en procesadores de media y alta performance, porque:
  - Hay muchas paradas que no pueden ser predichas por el compilador, porque ocurren en tiempos de ejecución.
    - Por ejemplo, las que veremos en Tema 11 (caché).
  - Es muy difícil predecir en tiempo de compilación el resultado de los saltos para armar mejor los paquetes.
  - Diferentes implementaciones de un mismo ISA pueden tener distintas latencias y distintos riesgos.

Ejecución Especulativa

- Se puede usar con emisión múltiple estática o dinámica.
- Consiste en “adivinar” qué va a pasar con una instrucción, para comenzar con su ejecución lo antes posible.
  - Dicho de otra manera, ni siquiera esperar sus dependencias, sino intentar adivinarlas.
  - Si acertamos, ganamos tiempo.
  - Si no acertamos, igual hubiésemos perdido tiempo esperando.
- Ejemplo típico: en los saltos.
  - No esperar la resolución del salto, sino **ejecutar ambas alternativas**.
  - Cuando se resuelva el salto, terminamos la alternativa correcta y descartamos la incorrecta.
- ¡También suelen especularse los loads!

Ejecución Especulativa

- Nuevamente, el compilador puede ayudar con la especulación.
  - Reordenando instrucciones.
  - Agregando instrucciones para recomponerse de una especulación errónea.
- Un procesador OoO puede ir anticipando la ejecución de algunas instrucciones “por las dudas”.
  - Solamente porque puede hacerlo (Moore otra vez).
  - Y las mantiene en la unidad de terminación (en el ROB) hasta que determina si las necesita realmente o no.
    - Si no las necesita, las descarta.

Limitaciones del ILP

- La mayoría de procesadores modernos de performance media y alta son superescalares.
  - Funcionan muy bien. Pero no tanto como nos gustaría.
  - En la jerga se los conoce como procesadores "*GreatBigOoO*".
- **ILP es limitado fuertemente por dependencias**.
  - Diagrama de precedencias.
  - La mayoría del código escrito en la mayoría de los lenguajes de programación es secuencial.
  - Es difícil aprovechar todo el paralelismo disponible.
  - Es difícil mantener los pipelines llenos de instrucciones.
    - La ejecución especulativa ayuda bastante.

Limitaciones del ILP

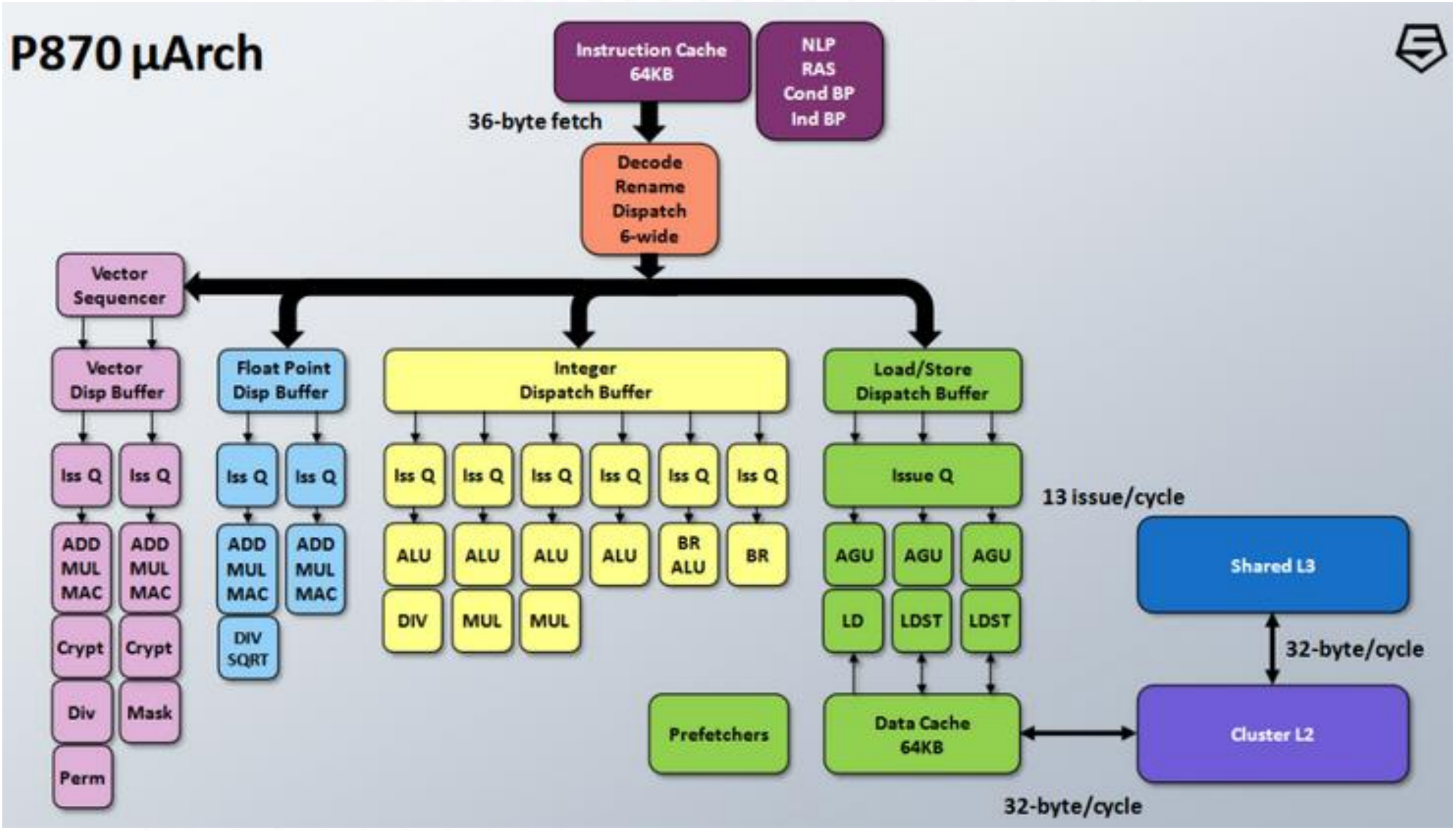
- Tanta complejidad agrega costo (área), y consume energía.
  - El consumo de energía derivó en la *Power Wall* vista en Tema 02.
  - Un procesador *single-issue* con emisión en orden bien diseñado puede llegar a obtener casi la mitad de la performance que uno "*GreatBigOoO*".
    - Pero en 1/10 de área, y consumiendo 1/10 de energía. ¡O menos!
- Además la ejecución especulativa resultó ser un serio problema de seguridad, que veremos en Tema 17.

Ejemplos de procesadores superescalares

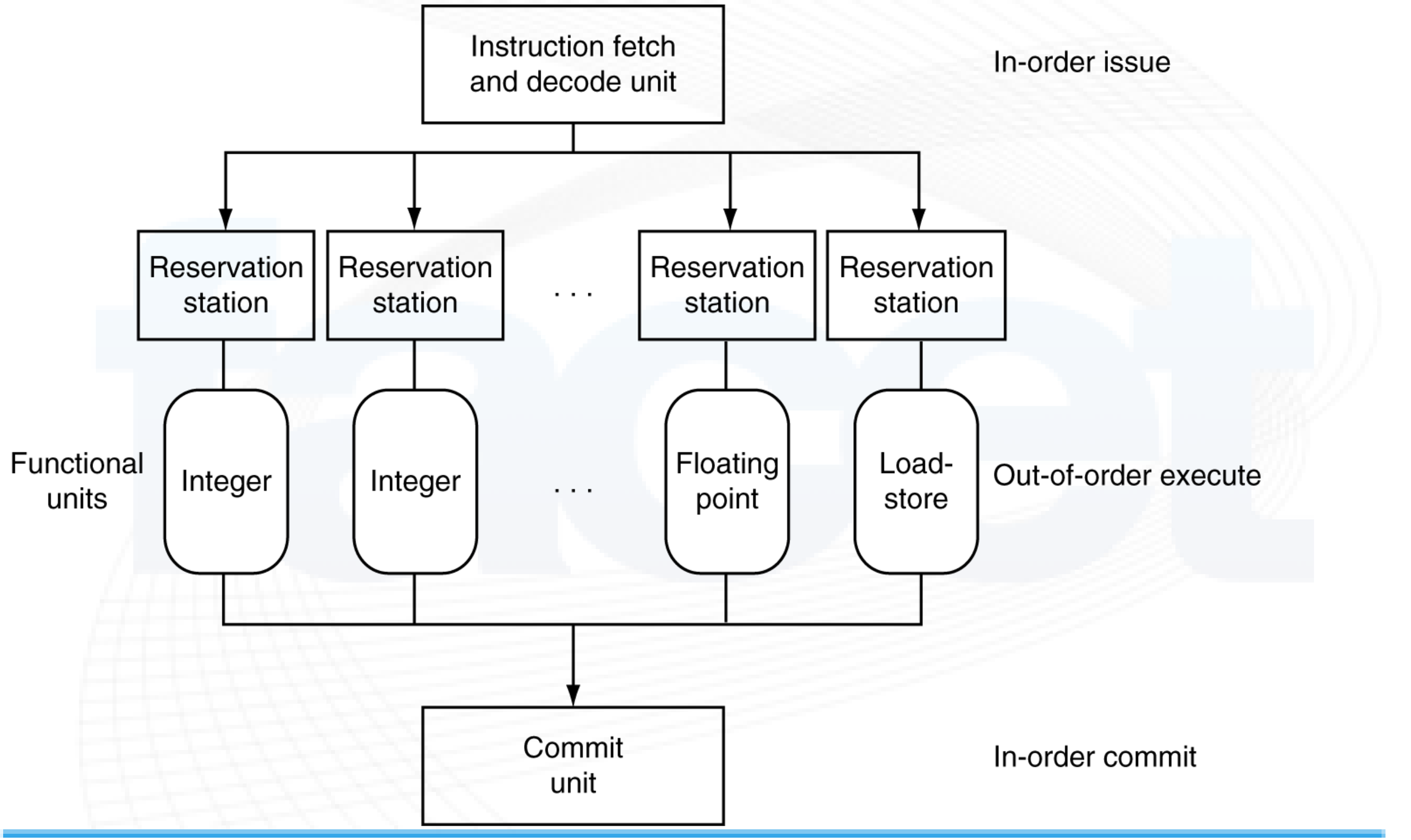
Procesador	Año	ISA	Emisión [Instr/ciclo]	Un. Func. [R/LS/B/FP]	Tamaño ROB
Apple M1 Firestorm	2020	ARMv8.4	**8**	**16** [7/4/1/4]	**630**
ARM Cortex X2	2023	ARMv9	5	13 [4/3/2/4]	288
Intel Core i7 13th gen (Golden Cove)	2022	x86_64	5	10 [4/6/-/-]	352
AMD Ryzen 7 7900X3D (“Zen 4”)	2023	x86_64	6	10 [4/3/-/4]	320
SiFive P870	2024	RV64GCV	6	13 [5/3/1/4]	280

- Los valores resaltados son los máximos en la actualidad.
- Como el pipeline del Ryzen tiene 19 etapas, ¡puede ejecutar en simultáneo **más de 100 instrucciones por ciclo**!

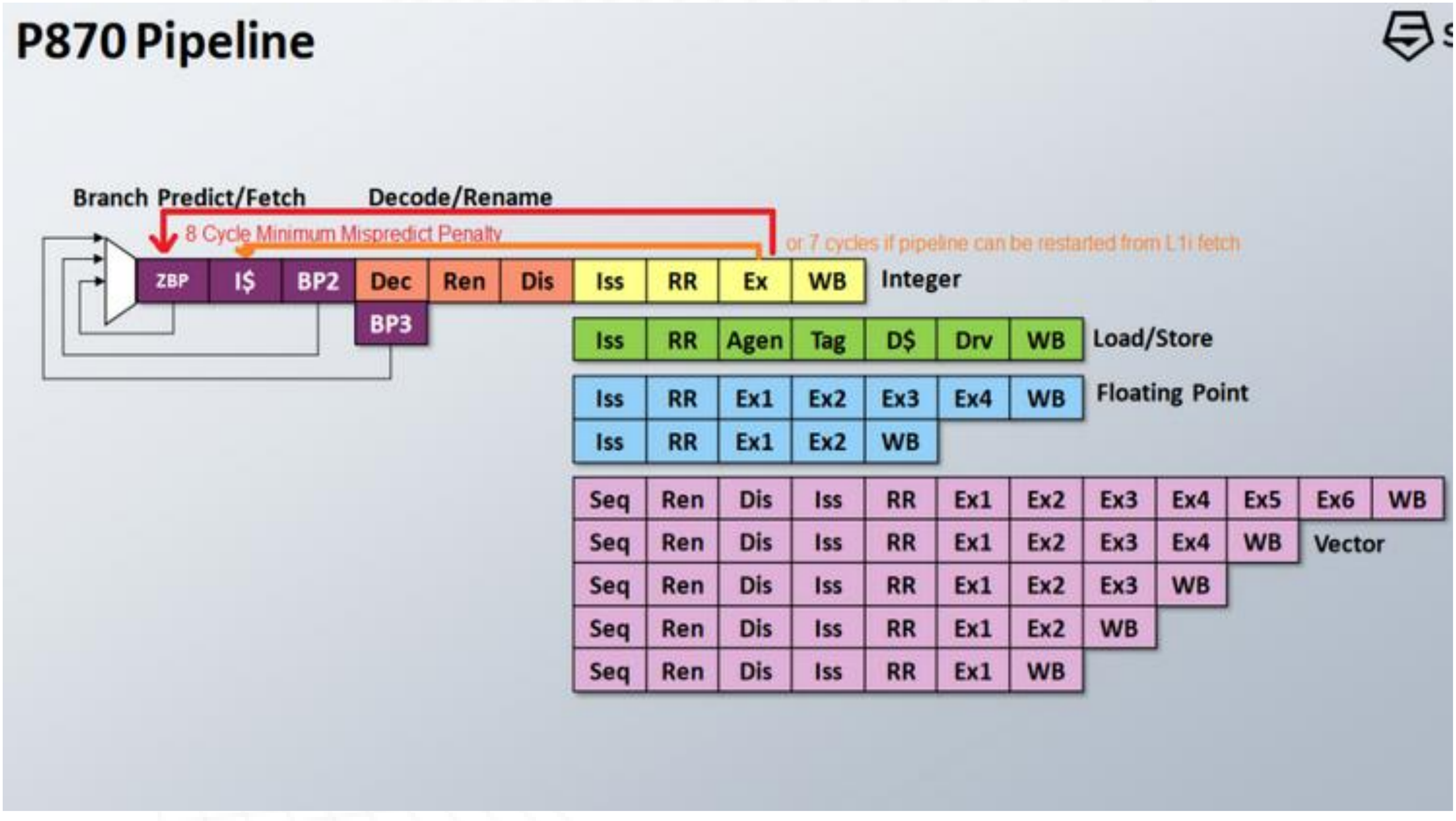
Diagrama de un procesador moderno



Repaso: Esquema de un procesador OoO



Pipeline de un procesador moderno



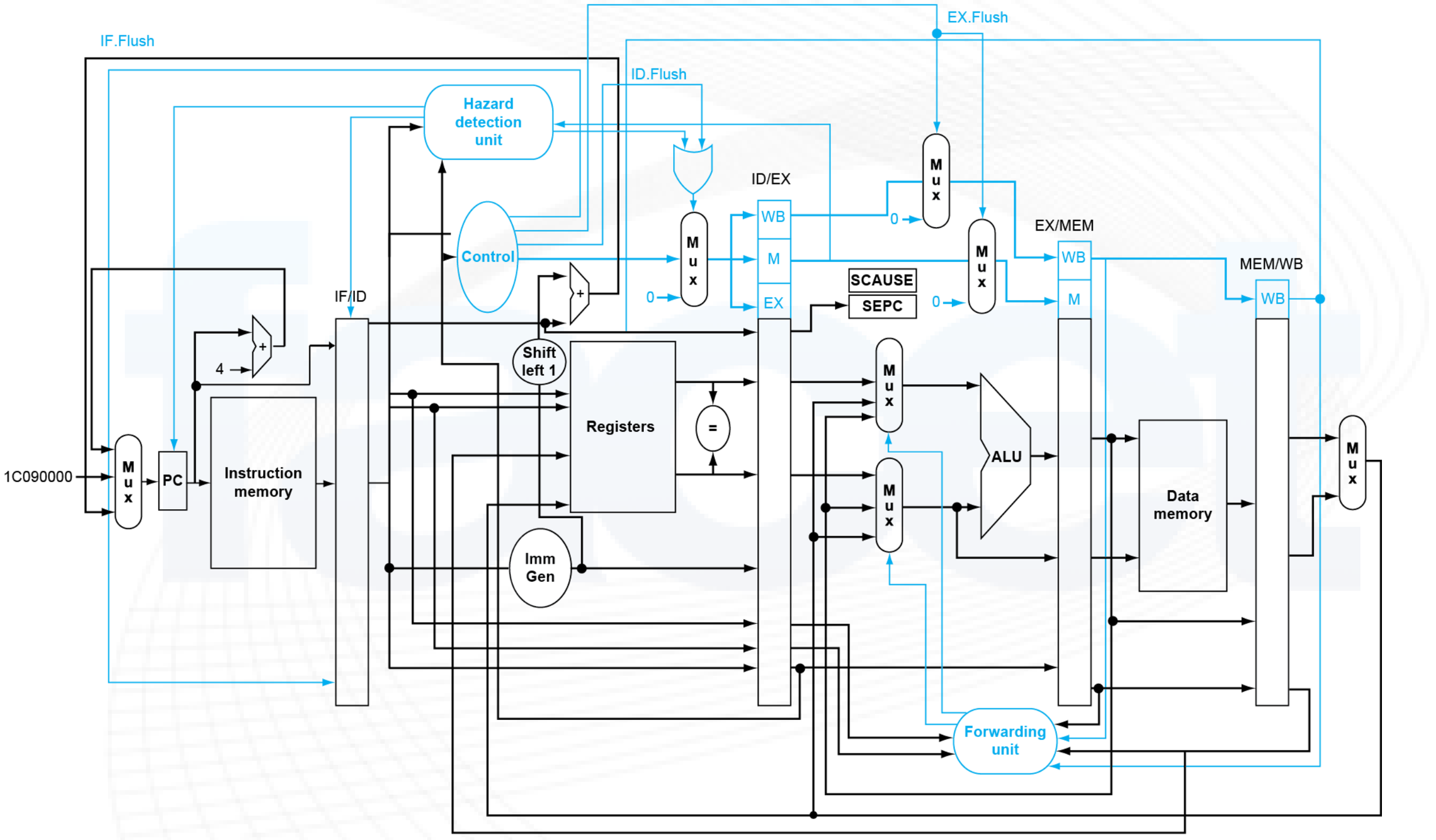
Manejo de Excepciones e Interrupciones

- Por si fuera poco, además los procesadores deben proveer soporte para manejo de excepciones e interrupciones.
  - **Preservar el comportamiento de las excepciones es la segunda propiedad esencial para asegurar la exactitud de un programa.**
- Cuando llega una interrupción (o se produce una excepción):
  - Se debe guardar el PC de la instrucción interrumpida (o que causó la excepción) **en un registro adicional**.
  - Se debe guardar **en otro registro adicional** la causa de la interrupción (o excepción).
  - Se debe escribir el PC con la dirección donde se ubica el manejador de interrupciones/excepciones.
  - Esto agrega al camino de datos un par de registros y **una entrada más al multiplexor de PCSrc**.

Manejo de Excepciones e Interrupciones

- Para el pipeline, las interrupciones son otros riesgos de control.
  - Son tratadas de manera muy similar a un salto mal predicho.
- Supongamos que una instrucción add que está en la etapa EX genera una excepción.
  - Se debe evitar que esa instrucción escriba en el banco de registros.
  - Se deben completar las instrucciones previas, que están en etapa ME y WB.
  - Se quita del pipeline la instrucción que provocó la excepción, y las siguientes.
  - Se guarda el PC de la instrucción que causó la excepción, y la causa de la misma, y se transfiere el control al manejador de excepciones.
  - Cuando se retorna del manejador de excepciones, la instrucción problemática vuelve a ser buscada en IF y ejecutada.

Camino de datos para Interrupciones



Manejo de Excepciones e Interrupciones

- Como en un pipeline se ejecutan varias instrucciones en simultáneo, pueden ocurrir múltiples excepciones.
- El enfoque más simple es manejar primero la excepción de la instrucción más antigua y vaciar las siguientes.
  - Eventualmente, las demás excepciones volverán a ocurrir y serán manejadas en su momento.
  - Esto se denomina **excepciones precisas**.
- En un procesador superescalar, con emisión dinámica de instrucciones múltiples y ejecución OoO, mantener las excepciones precisas **¡es todo un desafío!**
  - Deben manejarse en la unidad de terminación.
  - *¿Y si la excepción ocurre en una instrucción que estaba siendo ejecutada especulativamente?*

Resumen final

- Pipelining no parecía tan difícil.
  - La idea básica es simple.
    - Un buen diseño de ISA ayuda muchísimo.
  - Pero la búsqueda permanente de mayor performance elevó notoriamente la complejidad.
    - Motorizados por la Ley de Moore, se agregaban transistores sin inconvenientes.
  - Las interrupciones precisas también agregan complejidad al pipeline.

Resumen final

- Buscamos mejorar la performance del pipeline mediante:
  - Superpipelining, que disminuye T.
  - Emisión múltiple de instrucciones (superescalar).
    - Ayudados por el compilador, reordenando instrucciones y desenrollando lazos.
  - Ejecución fuera de orden.
    - Con terminación en orden, para preservar el sentido del código.
  - Ejecución especulativa.
  - Predicción de saltos.
- ILP fuertemente limitado por dependencias.
- Tanta complejidad limitada por el consumo de energía.