Tema 08: Paralelismo a nivel de instrucciones

Arquitectura de Computadoras

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

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

Temas que veremos

- Procesadores superescalares
- Emisión estática de múltiples instrucciones
  - VLIW.
- Desenrollado de lazos.
- Latencias variables.
- Emisión dinámica de múltiples instrucciones
  - Ejecución fuera de orden.
  - Renombramiento de Registros.
  - Ejecución especulativa.
- Limitaciones del ILP.
- Ejemplos de procesadores modernos.

Lectura recomendada

- Computer Organization and Design, RISC-V Edition (2da ed, 2021)
  - Sección 4.11: *Parallelism via Instructions*
  - 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.*

Paralelismo a nivel de la Instrucción (ILP)

- Con el diseño en pipelining, 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 (*WB*) 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.
  - Es una métrica más simple, porque cuanto más alto, mejor.
- Para ello se basan en tres ideas fundamentales:
  - Emisión dinámica de múltiples instrucciones.
  - Ejecución fuera de orden.
  - Ejecución especulativa.

Emisión múltiple de instrucciones

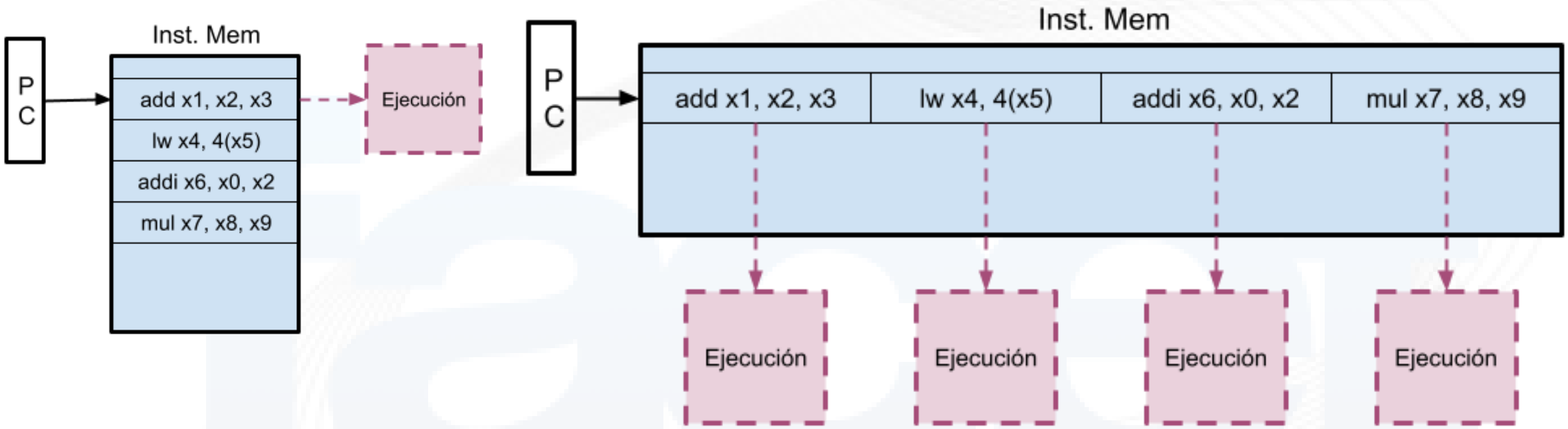
- Varios pipelines en paralelo no tienen varias memorias de instrucciones.
  - Siempre es una única memoria.
  - Pero debería proveer instrucciones a todos los pipelines en paralelo.
    - En vez de buscarse una instrucción en el *Fetch*, se buscan varias.
    - Necesita que la memoria de instrucciones tenga un mayor ancho de banda.
  - Hay dos maneras: **emisión estática y dinámica**.

Emisión estática de múltiples instrucciones

- Se denomina estática porque se realiza antes de la ejecución del programa.
- El compilador agrupa las instrucciones que puedan ser emitidas juntas en un mismo ciclo.
- El compilador arma “paquetes de instrucciones” según los recursos disponibles.
- El compilador se encarga (también) 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 de instrucciones” como si fuesen “instrucciones largas”.

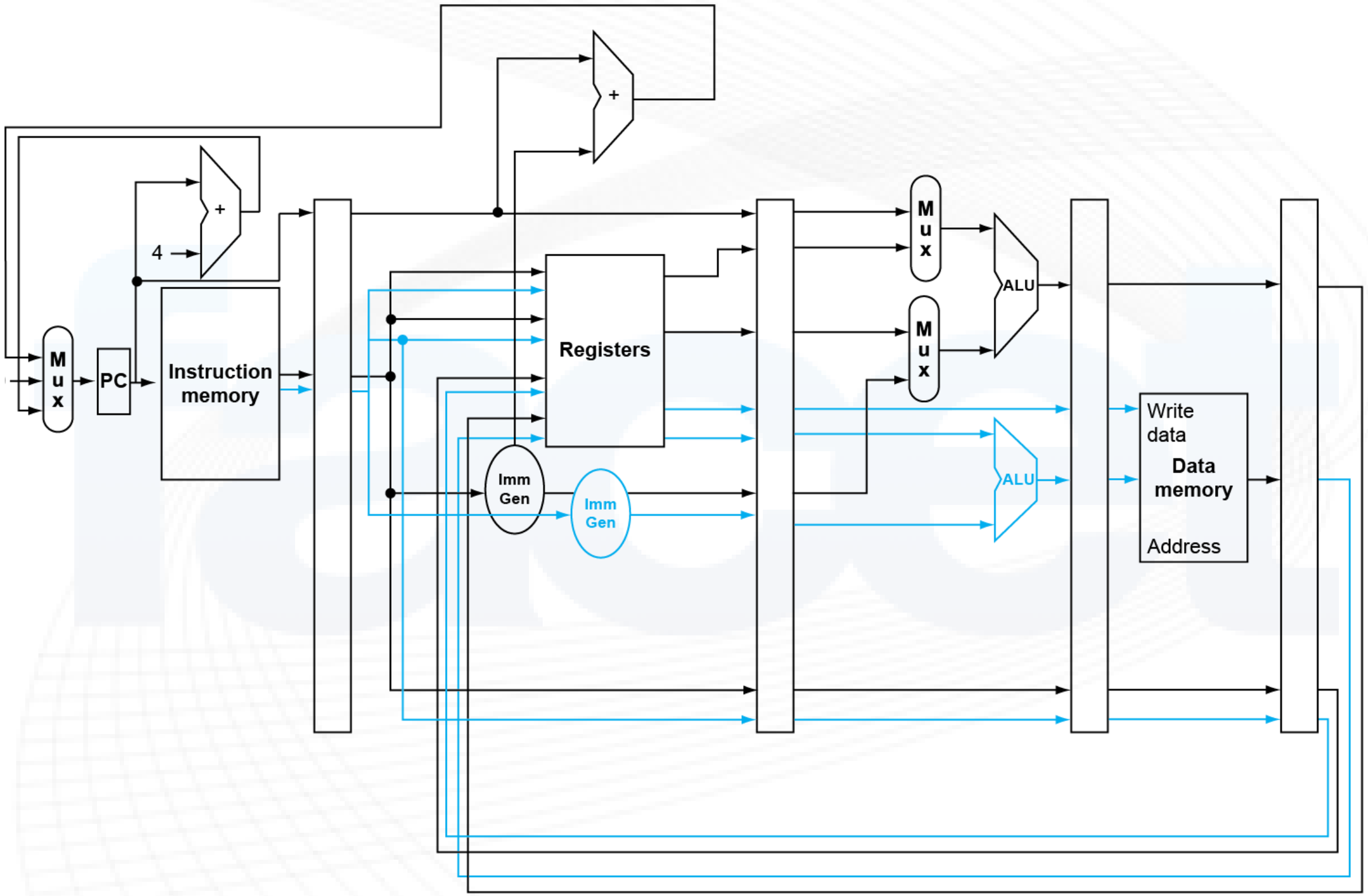


- Si tenemos varios pipelines disponibles, las instrucciones son “muy largas”.
- No es muy usada actualmente, por problemas que veremos en instantes.

Camino de datos con emisión doble

- Modificaremos el camino de datos visto, para que pueda ejecutar dos instrucciones por ciclo (*dual-issue*).
  - Dos pipelines independientes: 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
```

- *¿En cuántos ciclos se ejecutaría cada iteración del código en el procesador segmentado original?*
- Para ejecutar este código en un dual-issue, debemos armar los "paquetes" de instrucciones, como lo haría el compilador.
  - 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.

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).
  - Esto genera una nueva dependencia, que antes no estaba.
- Movemos el blt junto con el sw (no hay dependencia).
- Rellenamos todos los huecos con instrucciones nop.

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
```

- *¿En cuántos ciclos se ejecuta este 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 implica “perder” dos instrucciones.
  - La separación de funciones de los pipelines no resulta muy equilibrada.
- Si intentásemos emitir más instrucciones, haciendo los “paquetes” más grandes, estos problemas se agravarían.
  - Por esto no es tan común VLIW.
- Sin embargo, como la emisión es estática, el compilador puede ayudar aún más.

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.
  - Hacer en una iteración “nueva” varias de las iteraciones “originales”.
  - 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*

- Para “desenrollar” un lazo **una vez**, lo primero que hacemos es duplicarlo:

```
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

      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
```

Desenrollado de lazos – *single-issue*

- Luego, quitamos el control del lazo de la “primera iteración”, ajustando el control de la “segunda iteración” (la última), y los desplazamientos:

```
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

      lw   x31,-4(x20)     // x31=elemento de un vector
      add  x31,x31,x21     // sumamos el valor en x21
      sw   x31,-4(x20)     // guardamos resultados
      addi x20,x20,-8      // decrementamos el puntero
      blt  x22,x20,Loop    // repetimos si x22 < x20
```

- Nótese que al reducir el control del lazo, ya hay una pequeña mejora.
- *¿Cuántos ciclos demora ahora en hacer una iteración “original”?*

Desenrollado de lazos – *single-issue*

- Sin embargo, como ahora tenemos más instrucciones disponibles, es posible reordenarlas, para eliminar riesgos:

```
Loop: lw   x31,0(x20)      // x31=elemento de un vector
      lw   x30,-4(x20)     // x30=elemento de un vector
      add  x31,x31,x21     // sumamos el valor en x21
      add  x30,x30,x21     // sumamos el valor en x21
      sw   x31,0(x20)      // guardamos resultados
      sw   x30,-4(x20)     // guardamos resultados
      addi x20,x20,-8      // decrementamos el puntero
      blt  x22,x20,Loop    // repetimos si x22 < x20
```

- Nótese que ahora son necesarios más registros.
- *¿Cuántos ciclos demora ahora en hacer una iteración “original”?*
- *¿Cuánto es la aceleración con respecto al original?*

Desenrollado de lazos – *single-issue*

- La mejora de performance se obtuvo por dos motivos:
  - Reducir instrucciones de control de lazo.
  - Reordenar instrucciones para evitar paradas.
- El código original era de 5 instrucciones, y demoraba 6 ciclos en ejecutar cada iteración.
  - 3 ciclos (cuerpo del lazo) + 1 ciclo (burbuja) + 2 ciclos (control del lazo)
- El código desenrollado una vez y reordenado tiene 8 instrucciones y demora 8 ciclos en ejecutar cada iteración (4 ciclos por iteración “original”).
  - 3 ciclos (cuerpo) * 2 + 0 ciclo (burbuja) + 2 ciclos (control)
- Si se desenrollara el lazo 2 veces, para hacer 3 iteraciones “originales”, y se lo reordena, *¿cuántos ciclos demoraría cada nueva iteración?*

Desenrollado de lazos – *single-issue*

- Esta mejora de performance es “casi gratis”.
  - Por lo que es un optimización muy utilizada por los compiladores.
  - Misma aceleración que obtuvimos con el *dual-issue*, sin costo adicional.
- 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 – *single-issue*

- Si se desenrolla el lazo 3 veces, para hacer 4 iteraciones “originales”, y se lo reordena, *¿cuántos ciclos demoraría cada nueva iteración?*

```
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 – *dual-issue*

- Todo esto lo analizamos para tratar de superar las limitaciones que tenía la ejecución del lazo en el procesador *dual-issue*.
  - IPC = 1,25; eficiencia = 62.5%; Aceleración = 1.5
- *¿Cómo quedaría el código anterior en el procesador dual-issue?*

```
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
```

Desenrollado de lazos – *dual-issue*

- *¿En cuántos ciclos se ejecuta el código anterior?*
- *¿Cuánto sería el IPC? ¿Cuánto sería la eficiencia?*
- *¿Cuánto sería la aceleración con respecto al procesador segmentado?*
- ¡Aumentamos IPC y eficiencia!
  - Al desenrollar el lazo, se aumentó la cantidad de instrucciones disponibles para armar los “paquetes” de ejecución.
  - Los dos pipelines están bastante más equilibrados.
- Inclusive así, no llegamos al máximo.
- Y se necesita un súper compilador.
  - Para que desenrolle, reordene y arme los paquetes, insertando nops cuando sea necesario.
  - Que a su vez depende del ISA.
- Quizás sea necesaria buscar una solución por Hw…

Latencias variables

- Pero antes, hay otra suposición implícita que venimos haciendo:
  - La etapa de EX demora siempre 1 ciclo.
  - Es válido para el subconjunto de instrucciones con el que venimos trabajando.
- Sin embargo, al hacer más reales los procesadores, es necesario agregar más instrucciones, más complejas.
  - Por ejemplo, en RISC-V, podríamos agregar:
    - Instrucciones de multiplicación (extensión M).
    - Instrucciones para punto flotante (extensiones F y D).
    - Instrucciones vectoriales (extensión V).
  - Estas instrucciones poseen una ejecución que suele demorar más de un ciclo.
    - Por ejemplo, las de punto flotante demoran entre 4 y 6 ciclos.

Latencias variables

- Una primera solución: solución serie del pipelining
  - Dividir la etapa EX en múltiples etapas: EX1, EX2, EX3, etc.
  - La cantidad total estará determinada por el peor caso.
- Ventajas:
  - Mantiene la simplicidad, evita riesgos adicionales.
  - Todas las instrucciones pasan por todas las etapas.
- Desventajas:
  - Aumenta la latencia de todas las instrucciones. No es tan grave.
  - Aumenta la penalidad del load-to-use.
  - Aumenta la penalidad del fallo en la predicción de saltos.
  - No cumplimos con un principio de diseño.
    - Hacer más rápido el caso más común.
- Conclusión: no es buena idea.

Latencias variables

- Una segunda solución: solución paralela del pipelining.
  - La etapa EX se convierte en un conjunto de unidades funcionales independientes en paralelo, cada una con su propia latencia.
- Las instrucciones “cortas” van por el camino rápido.
- Desventajas:
  - Genera riesgos estructurales.
  - Aumenta la penalidad del load-to-use (solamente en algunos casos).
  - Puede generar riesgos tipo WAW.
    - Una instrucción “corta” va después que una “larga”, pero termina antes.
  - Las unidades de Adelantamiento y de Inserción de Burbuja se vuelven extremadamente complejas.
- Conclusión: tampoco es buena idea.

Latencias variables

- El problema de la finalización fuera de orden por las latencias variables es más complejo de lo que parecía.
  - La ejecución de las instrucciones ya no es determinística.
  - Es casi imposible mantener las excepciones precisas.
    - Si una instrucción “corta” genera una excepción mientras una instrucción “larga” anterior aún está ejecutándose, *¿qué estado se guarda?*
- La emisión estática de múltiples instrucciones es incapaz de resolver estos problemas.
  - No hay manera óptima en que el compilador resuelva estas situaciones, para todos los casos.
  - Es necesario que el Hw tome las decisiones en tiempo de ejecución.

Emisión Dinámica de Múltiples Instrucciones

- En este caso, el procesador analiza la secuencia de instrucciones y decide cuáles instrucciones emitir en cada ciclo.
  - El mismo procesador es el encargado de detectar y resolver los riesgos (de datos y estructurales), **en tiempo de ejecución**.
  - Basado en un **diagrama de precedencias**.
  - En cada ciclo pueden emitirse 0 (burbuja), 1, 2, o más instrucciones.
    - Si emite *n* instrucciones por ciclo, se denomina *n-wide*.
- 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).
- Se provee una manera eficiente de solucionar el problema de las latencias variables.

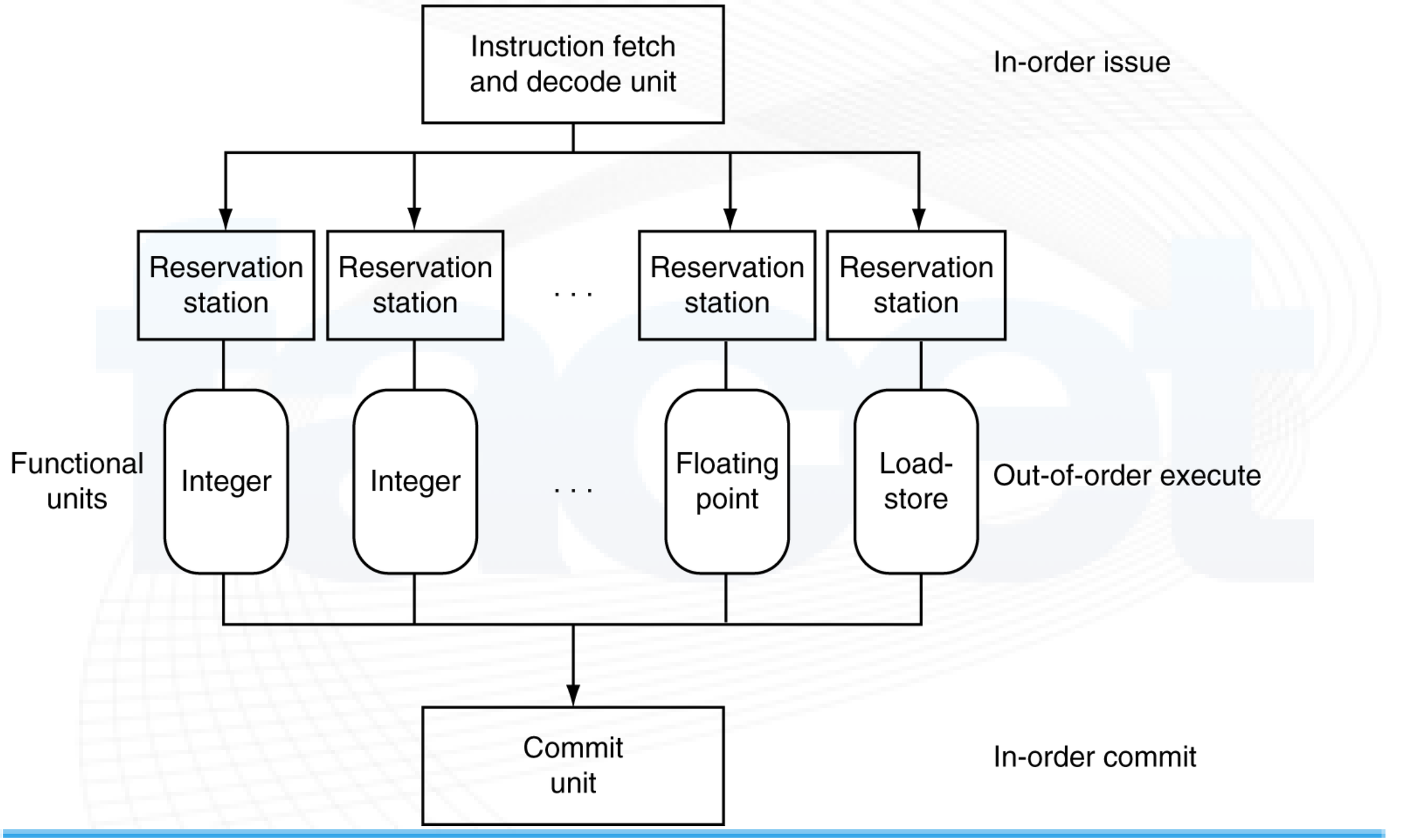
Ejecución fuera de orden (OoO)

- La manera de evitar paradas del pipeline en la emisión 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, siempre y cuando no tenga dependencias.
- Pero **se siguen terminando en orden**, para mantener el sentido del código.
- De esta manera, se divide el pipeline en tres “bloques conceptuales”:
- **Emisión (en orden) → Ejecución (fuera de orden) → Terminación (en orden).**
- Analogía: la cocina de un restaurante.

Ejecución fuera de orden (OoO)

- La separación en esos tres bloques conceptuales implica que hacen falta buffers antes y después de la ejecución.
  - Antes para “aguantar” las instrucciones que fueron buscadas y decodificadas, pero aún no están listas para ejecutarse.
  - Después para “aguantar” las instrucciones que están listas para terminar, pero deben esperar a que terminen las anteriores, para preservar el orden.
- 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.**
- Como la finalización es en orden, se mantienen las excepciones precisas.

Esquema general de un procesador OoO



Esquema de un procesador OoO

- A las etapas IF e ID se las denomina "***Front-end***".
  - Son encargadas de traer todas las instrucciones que puedan, para mantener siempre las estaciones de reserva llenas.
- Estas son las que emiten tantas instrucciones como el "ancho" del procesador (*n-wide*).
- 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***".
  - Definen lo que se conoce como "ventana de instrucciones" (*Instruction window*), que es la cantidad de instrucciones que pueden estar en espera simultáneamente.
  - Evitan que sea necesario parar todo el pipeline.

Esquema de un procesador OoO

- En la etapa de ejecución hay múltiples pipelines con unidades funcionales en paralelo, y se los denomina "***Back-end***".
  - Estos pipelines pueden ser diferentes entre sí, y tener diferentes latencias.
  - Inclusive puede haber varios iguales, para generar un mejor balance de unidades funcionales.
    - Por ejemplo: 3 pipelines para instrucciones TipoR, 2 pipelines para Ld/St y 4 para instrucciones de punto flotante.
- No hay relación directa con el "ancho" del procesador.
  - Se pueden emitir 5 instrucciones por ciclo, y el *Back-end* tener 13 pipelines en paralelo.
- Se requieren caminos de adelantamiento entre todos ellos.

Esquema de un procesador OoO

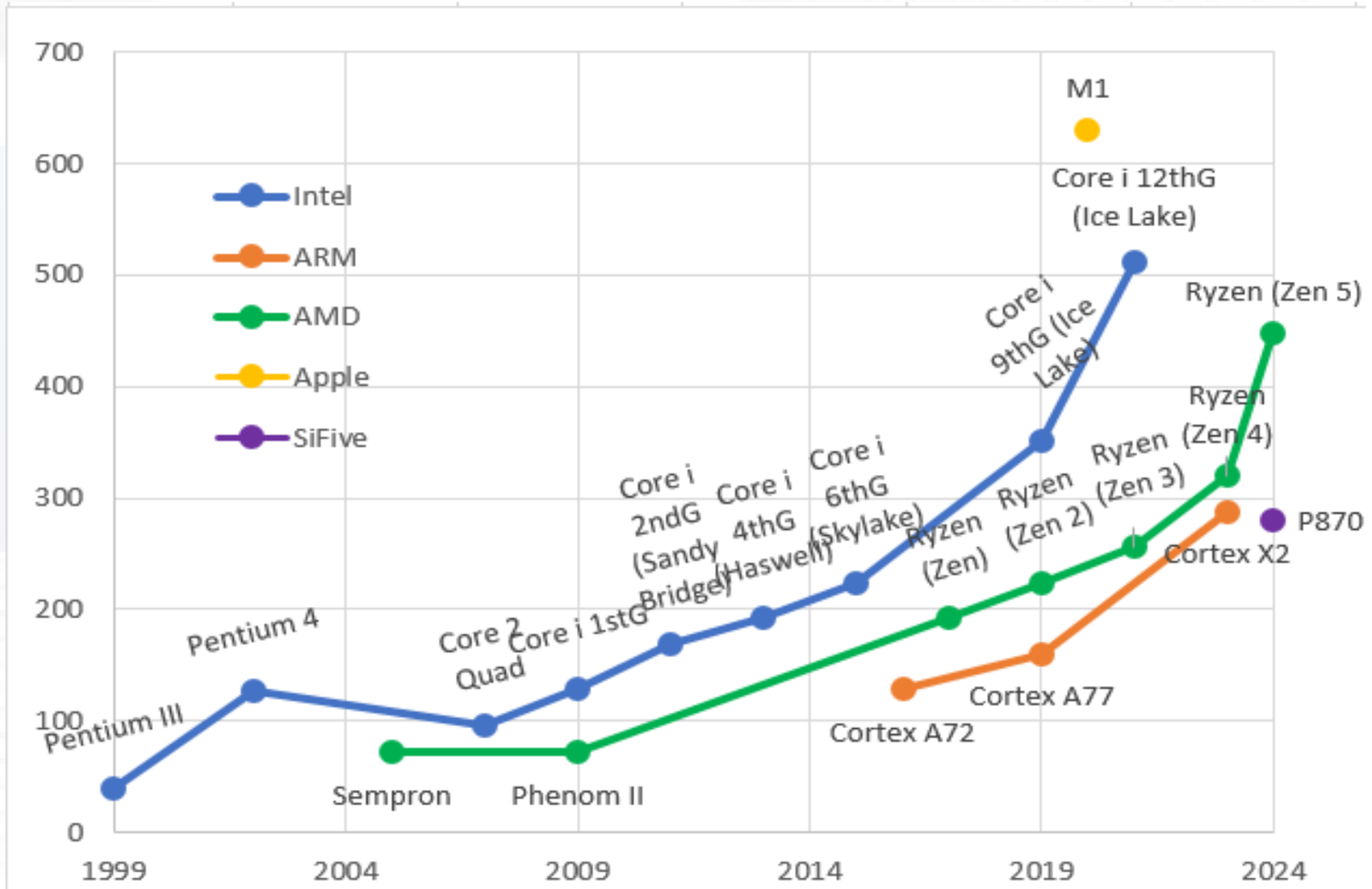
- La Unidad de Terminación (***Commit unit***) se encarga de reordenar las instrucciones, para asegurar que los registros se escriban en el orden deseado.
  - Se encarga que los resultados calculados se vuelvan permanentes.
  - Si una instrucción “posterior” termina antes que una “anterior”, se la retiene hasta que la “anterior” escriba su resultado.
  - Se encarga de manejar las excepciones.
- La estructura usada para retener las instrucciones terminadas se denomina **Buffer de Reordenamiento (*Re-Order Buffer*, 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 de nombre.
  - **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, y los núcleos Zen 5 de AMD tienen 240.
  - Se mantiene una 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 previstas por el compilador, porque ocurren en tiempos de ejecución.
    - Por ejemplo, las que veremos en Tema 10 (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 tanto estática como 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.
  - **Gran Idea en Arquitectura de Computadoras: Predicción.**
    - 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.
- *¿Qué ocurre si una instrucción que está siendo ejecutada especulativamente genera una excepción?*
  - Primero se resuelve la especulación.

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

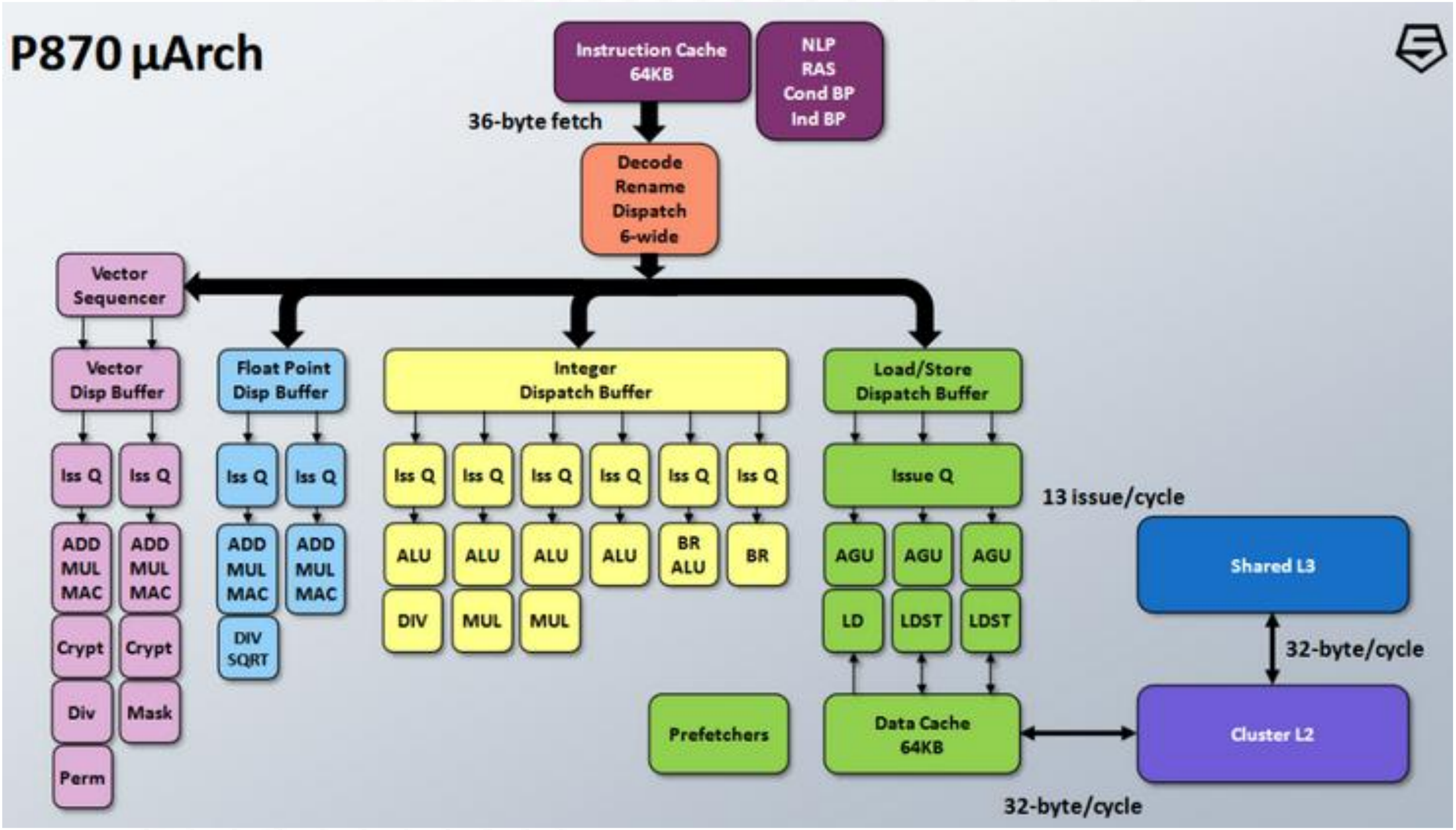
- Tanta complejidad agrega costo (área), y consume energía.
  - El consumo de energía derivó en la *Power Wall* vista en Tema 02.
  - Un procesador *single-issue* con emisión en orden bien diseñado puede llegar a obtener casi la mitad de la performance que uno "*GreatBigOoO*".
    - Pero en 1/10 de área, 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 modernos

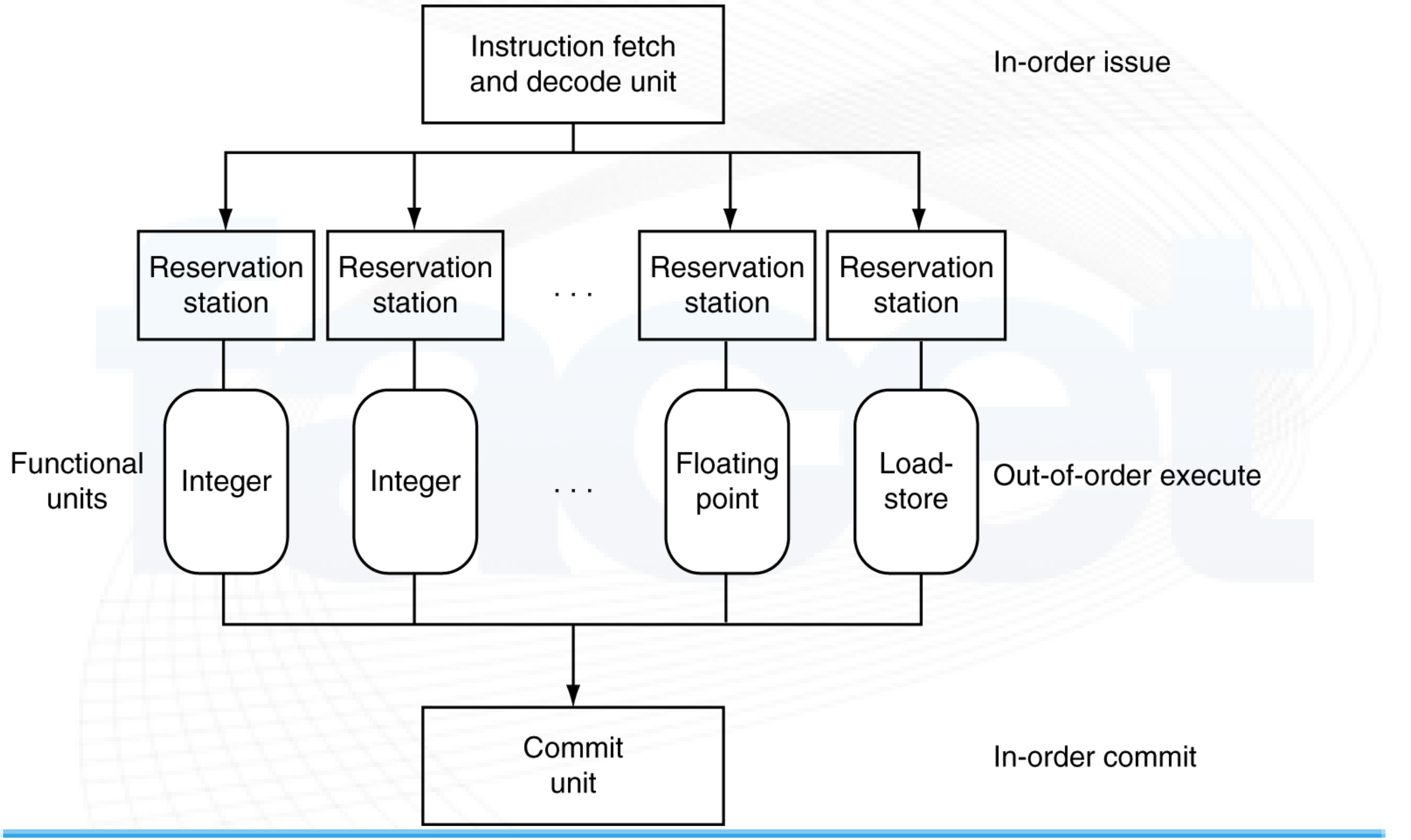
Procesador	Año	ISA	Emisión [Instr/ciclo]	Un. Func. [R/LS/B/FP]	Tamaño ROB
Apple M1 Firestorm	2020	ARMv8.4	**8**	**16** [7/4/1/4]	**630**
ARM Cortex X2	2023	ARMv9	5	13 [4/3/2/4]	288
Intel Core i7 13th gen (Golden Cove)	2022	x86_64	5	10 [4/6/-/-]	352
AMD Ryzen 9 HX 370 (“Zen 5”)	2024	x86_64	**8**	**16** [6/4/-/6]	448
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 llegar a ejecutar en simultáneo **casi 150 instrucciones por ciclo**!

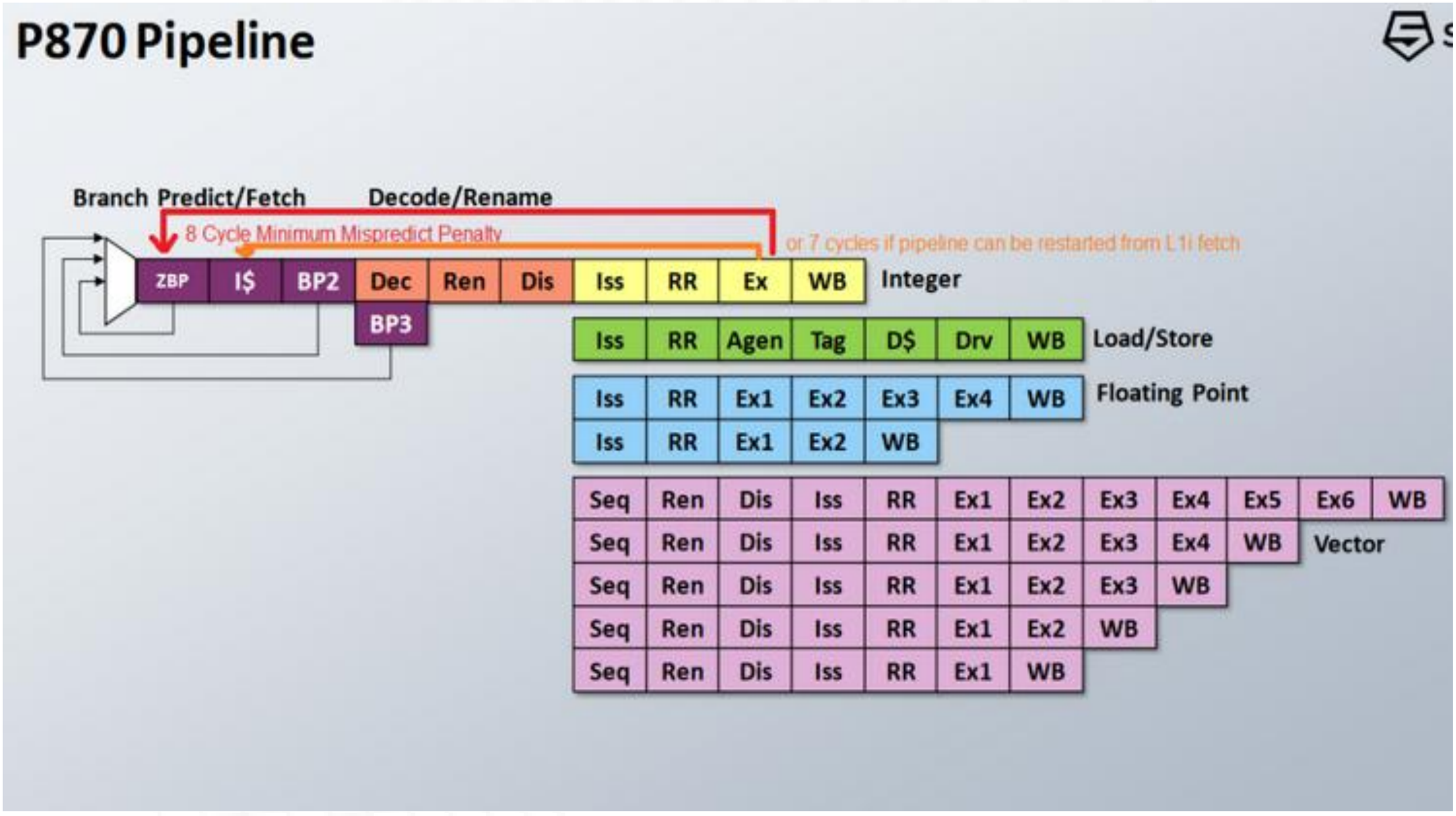
Diagrama de un procesador moderno



Repaso: Esquema de un procesador OoO



Pipeline de un procesador moderno



Resumen final

- Objetivo de ILP: mejorar la performance del pipeline.
  - CPI < 1, se convierte en IPC.
- Técnicas:
  - Emisión múltiple de instrucciones (superescalar)
    - Emisión (*issue*) y terminación (*commit*) de más de una instrucción por ciclo.
    - Dinámica, para independizarse del sw y poder manejar ejecuciones de latencia variable.
    - Ayudados por el compilador, reordenando instrucciones y desenrollando lazos.
  - Ejecución fuera de orden.
    - Mediante múltiples pipelines en paralelo.
    - Con emisión y terminación en orden, para preservar el sentido del código y manejar excepciones precisas.
  - Ejecución especulativa.
- ILP fuertemente limitado por dependencias, consumo de energía y problemas de seguridad.