Tema 11

Pipelining Avanzado

Eduardo Daniel Cohen – dcohen@arnet.com.ar

http://www.herrera.unt.edu.ar/arqcom

¿En qué se pone difícil Pipelining?

- **Excepciones: hay 5 instrucciones ejecutándose en nuestro caso.**
  - **¿Cómo parar el pipeline?**
  - **¿cómo recomenzarlo?**
  - **¿Qué Instrucción es responsable de la excepción?**

  EtapaExcepciones que pueden ocurrir por:

  IF Falla de Página; acceso no alineado a Memoria; violación de protecciones de memoria.

  ID Opcode no definido o ilegal.

  EX Excepción Aritmética (overflow por ej.)

  MEM Falla de Página; acceso no alineado a Memoria; violación de protecciones de memoria

  - **¿Load con falla de pag. de datos, Add con falla de pág de instruc?**
  - **¡Puedo tener más de una y detectar una posterior primero que otra!**
- **Solución 1: vector de instrucción por instruccion , revisar en últ. etapa**
- **Solución 2: terminar todo lo incompleto por Sw y ver las excepciones.**
- **Por "suerte" escribimos sólo al final.**

¿Qué lo hace más fácil?

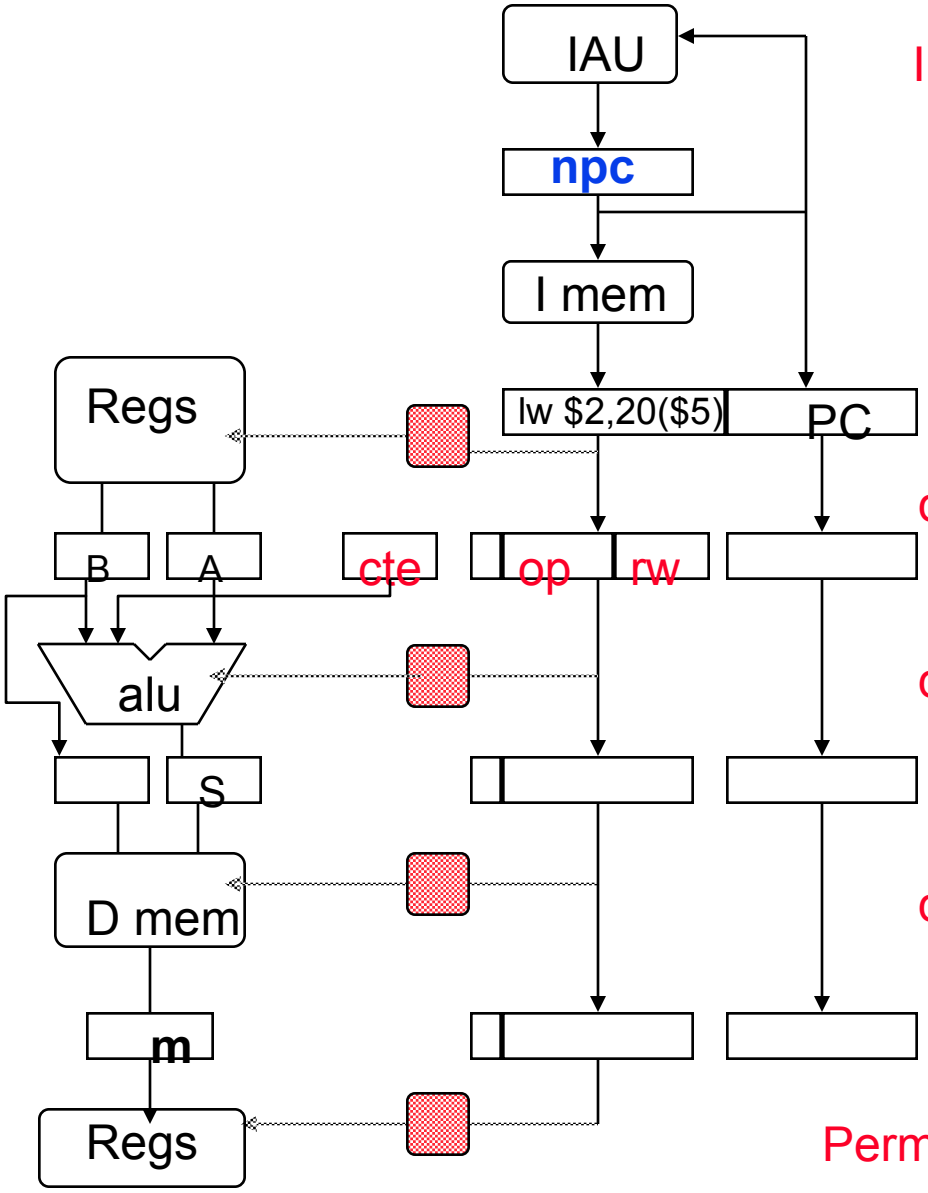
- El diseño del Set de Instrucciones MIPS → óptimo para Segmentación
  - Todas las instrucciones tienen la misma longitud.
  - Pocos formatos diferentes.
  - Posición fija de operandos: RF en paralelo con ID.
  - Operandos a Memoria solo aparecen en Loads y Stores. No aparecen etapas de búsquedas de dátos en M antes de Exec.
- Para el caso del Pentium.
  - Largo variable de instrucción: hasta 16 palabras.
  - Etapa Fetch de largo variable.
  - Pocas instrucciones simples → responsables 90% tiempo de ejec.
  - Se traducen en ID: 1 a 4 microinstrucciones.
  - Las microinstrucciones se ejecutan por Segmentación.
  - Las instrucciones complejas se traducen por Microprograma. Pero son minoría.

¿Qué se hace con Interrupciones y Traps?

- **Interrupciones Externas:**
  - **Esperar que se vacíe el Pipeline,**
  - **Cargar PC con la dirección de la Interrupción.**

- **Fallos (durante una/s instrucción, reparable)**
  - **Problema: podría darse la excepción en una instrucción que siga a un Salto Retardado. Puede ser necesario guardar varios PCs.**
  - **Llevar información del valor de PC con cada instrucción en el Pipe.**
  - **Detectar e incorporar la causa a la información de la instrucción en el Pipe.**
  - **Vaciar todas las instrucciones posteriores (izquierda) hasta que la excepción llegue al final del Pipe.**
  - **Que la instrucción con fallo no provoque cambio de estado en etapas posteriores.**

Manejo de Excepciones



Instruction Address Update

detectar bad instruction address

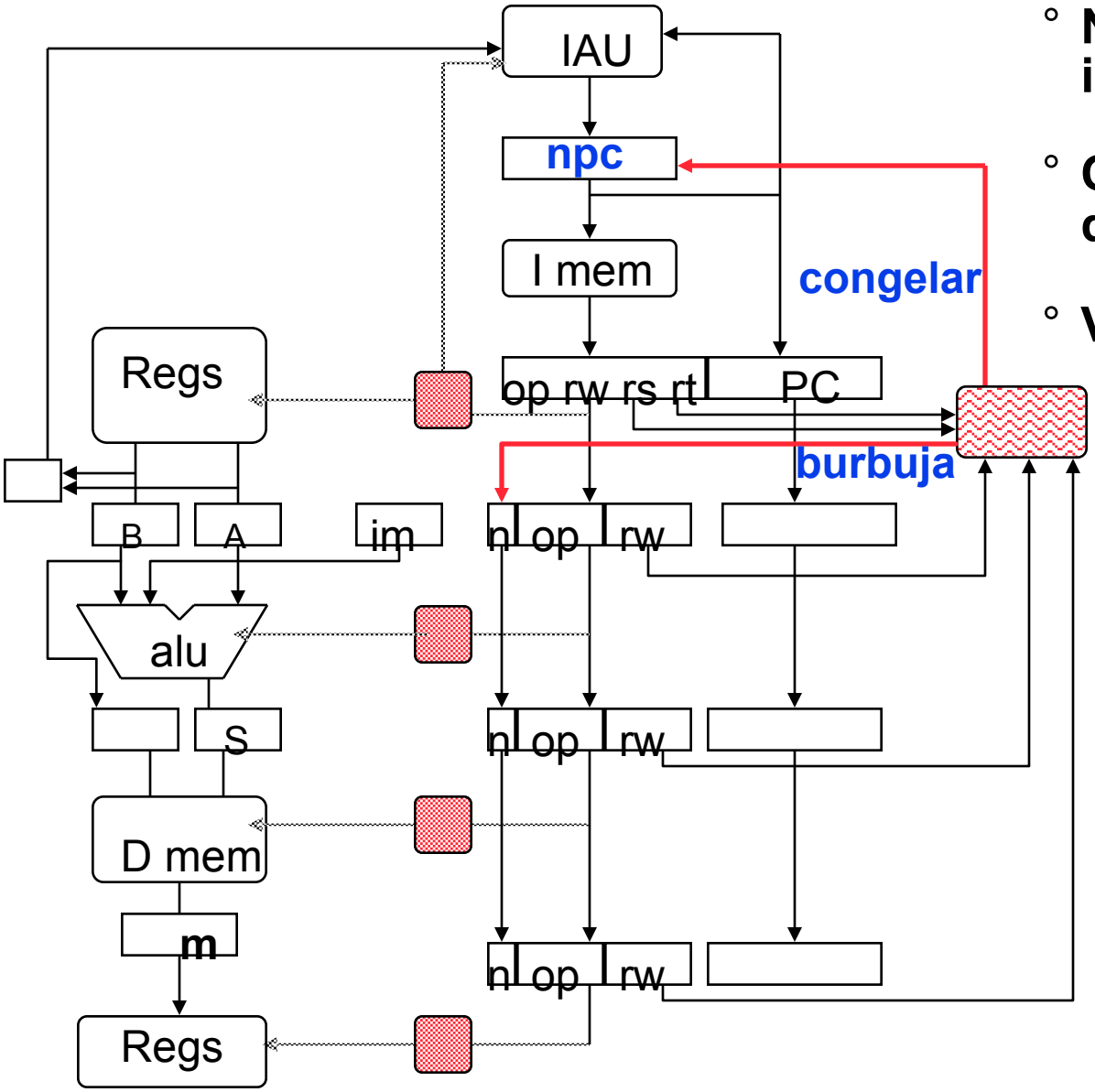
detectar bad instruction

detectar overflow

detectar bad data address

Permitir que la excepción se realice

Solución: Congelar arriba y burbujear abajo



- **No comenzar nuevas instrucciones.**
- **Que la inst nueva ni las de arriba hagan nada.**
- **Vaciar y realizar excep.**

Y..., se pone más difícil cuando hay:

- Instrucciones y Modos de Direcciones Complejos (CISC).
- Modos de direccionamiento: Autoincremento pre provoca cambios en los registros durante la etapa de ejecución.
  - Ya no se escribe solo al final, aparecen riesgos de escritura
    - Write After Read (WAR): una posterior escribe antes que la anterior lea, lee mal.
    - Write After Write (WAW): write ocurre en orden equivocado dejando datos equivocados en los registros.
    - (Los riesgos de datos ya analizados se llaman RAW, Read After Write – una posterior lee antes que la actual escriba)
- Instrucciones de Movimiento de Bloque de Memoria a Memoria.
  - Fallas múltiples de páginas.
  - ¿Está suficientemente difícil?

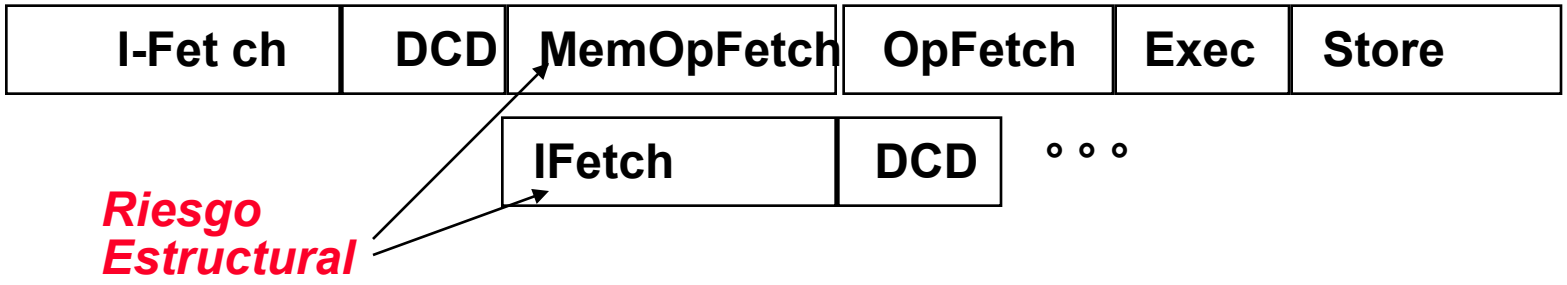
Pero en MIPS también hay más dificultades...

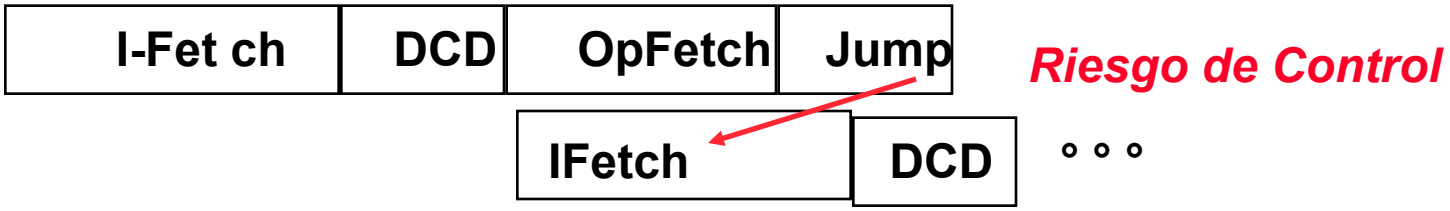
- Punto Flotante: tiempo de ejecución largo
- Además se diseña la unidad de punto flotante para que haya solapamiento y funcione en paralelo.

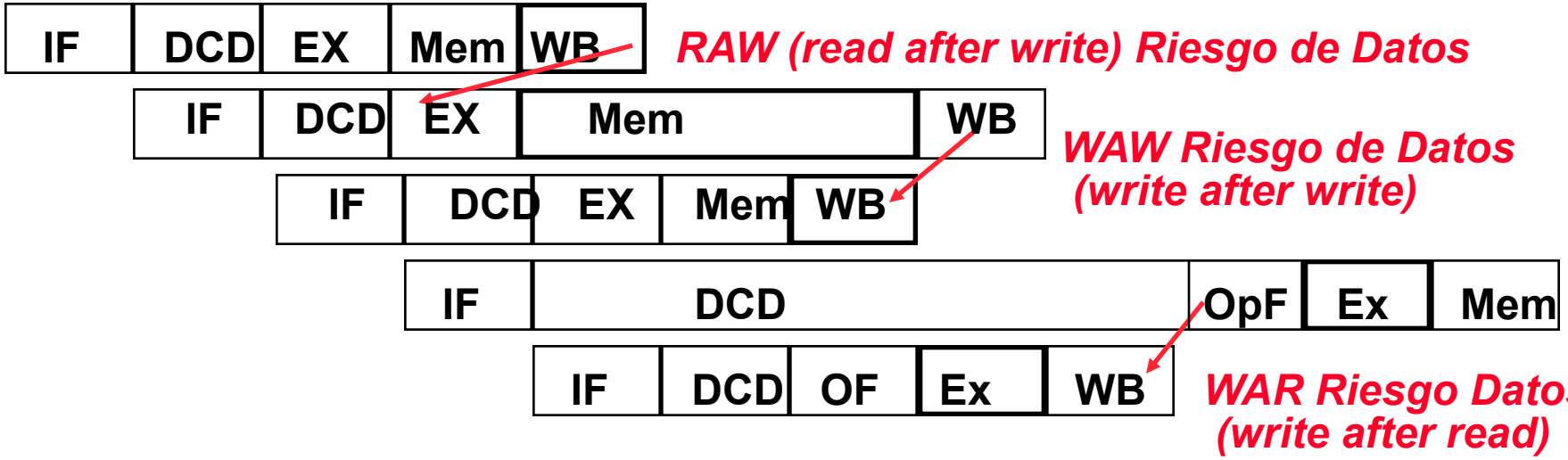
FP Instruction	*Latency*	*Initiation Rate*	*(MIPS R4000)*
Add, Subtract	4	3	
Multiply	8	4	
Divide	36	35	
Square root	112	111	
Negate	2	1	
Absolute value	2	1	
FP compare	3	2	

- División y Raíz Cuadrada duran 10 a 30 veces más que Add
  - Aparecen riesgos WAR y WAW ya que la duración del pipeline no es la misma para todas las instrucciones (lw y sw se ejecutan más rápido y pueden terminar antes siendo posteriores, por ej.)

Riesgos de la Segmentación en General







Algoritmo para Detección de Riesgos

° **Supongamos que la instrucción *i* esté por comenzar y que una instrucción precedente *j* *ya esté en el pipeline (i sucede a j).***

Rregs (*i*) = Registros leídos por la instrucción *i*

Wregs (*i*) = Registros escritos por la instrucción *i*

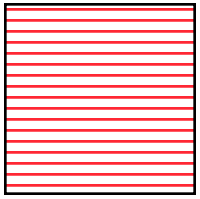
° **Un riesgo RAW ocurrirá en el reg. ρ si ∃ ρ : ρ ∈ Rregs(*i*) ∩ Wregs(*j*)**

– Mantener anotaciones de las escrituras pendientes y comparar con los registros operandos de la instrucción actual. SCOREBOARD.

– Cuando inicie una instrucción, reservar su registro destino.

– Cuando se complete una instrucción, elimine su reserva.

- Demorar toda instrucción hasta que no haya riesgo.



° **Un riesgo WAW ocurrirá en el reg. ρ si ∃ ρ, ρ ∈ Wregs(*i*) ∩ Wregs(*j*)**

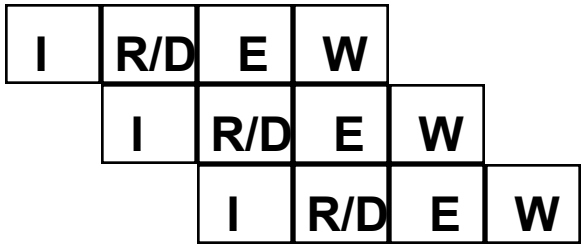
° **Un riesgo WAR ocurrirá en el reg. ρ si ∃ ρ, ρ ∈ Wregs(*i*) ∩ Rregs(*j*)**

Eliminación de Riesgos de Datos mediante Diseño

Supongamos que las instrucciones se inicien en orden secuencial.

° **Eliminación *WAW:* si la escritura en un recurso se realiza en la misma etapa para todas las instrucciones, y todas las etapas duran lo mismo, entonces no pueden haber riesgos WAW..**

Prueba: Las escrituras están en el mismo orden que las instrucciones.



° ***Eliminación WAR*: si en todas las instrucciones las lecturas de un recurso ocurren en una etapa anterior que las escrituras en ese mismo recurso, entonces no pueden haber riesgos WAR.**

Prueba: Una instrucción sucesora debe iniciarse posteriormente, con mayor razón escribirá solo después que se hayan realizado las lecturas de la instrucción actual.

Primera generación de Pipelines RISC

° Todas las instrucciones siguen el mismo orden del pipeline ("ordenamiento estático").

° Las escrituras de registro se realizan en la última etapa:

  – Se evitan riesgos WAW.

° Todos las lecturas se hacen al iniciar.

  – Se evitan riesgos WAR.

° Memoria de Instrucciones y de Datos Independientes.

  – Elimina riesgos de memoria (estructurales).

° Los riesgos de control se resuelven con saltos retardados (1T)

° Los riesgos RAW se resuelven por anticipacion, excepto en resultados de load que son retardados (1T).

Gran uso de pipelining con muy poco costo y bastante simplicidad.

Resumen de Bases de Pipelining I

Factor de Aceleración Pipe en relación a Ciclo Unico.

L Latencia, A Aceleración, T(p) Periodo de Pipe, b paradas promedio

A= t(cu) / t(pipe)

t(cu) = L(cu)

t(pipe) = [T(p) x (1+b)]

T(p) = L(p) / n

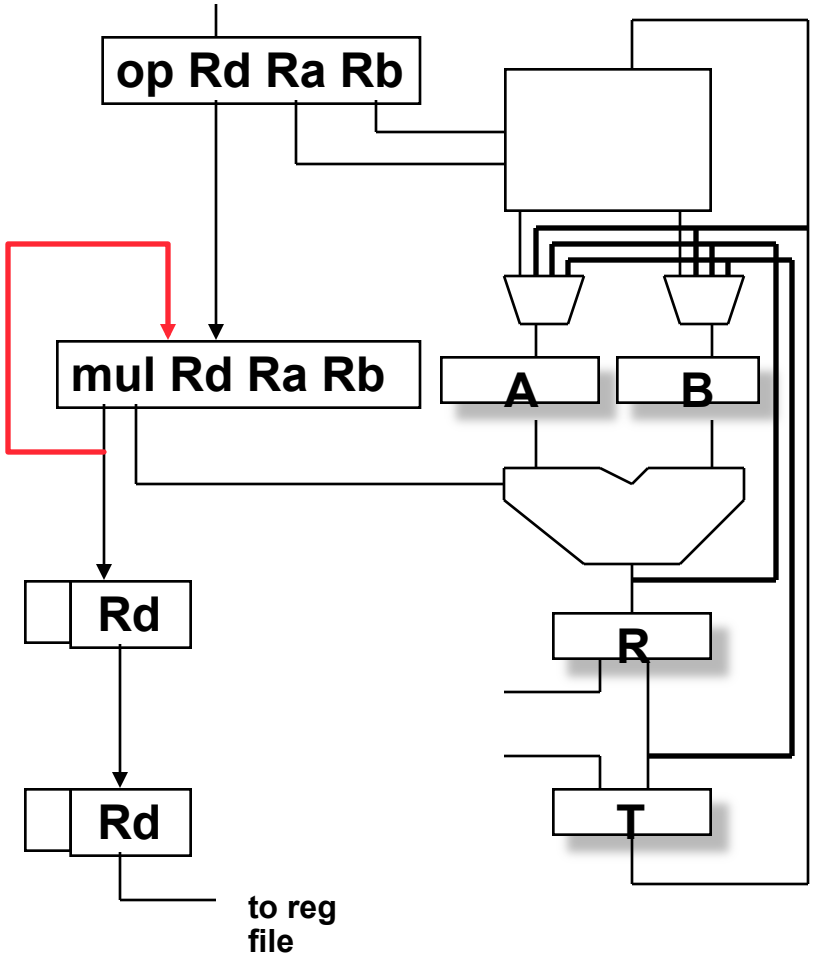
A = [n / (1+b)] * [L(cu) / L(p)]

- **OJO: T(p) no es L(cu) / n.**
- **T(p) es la duraciòn de la etapa cuello de botella más retardo latch intermedia.**

Resumen de Bases de Pipelining II

- Los riesgos limitan la performance de los procesadores:
  - Estructurales: necesitan más recursos de Hw.
  - De Datos: necesitan anticipacion y reordenamiento por Sw
  - De Control: evaluación de condición y actualización de PC anticipada, salto retardado y reordenamiento por Sw.
- Incrementar la longitud del pipe incrementa el impacto de los riesgos.
- Los compiladores son clave en reducir el costo de los riesgos.
  - Huecos de retardo de Loads.
  - Huecos de retardo de Branchs.
- Excepciones, Instrucciones Complejas, FP hacen difícil pipeline

MIPS R3000 Operaciones Multiciclo



Ej: Multiplicación, División, Fallo de Cache

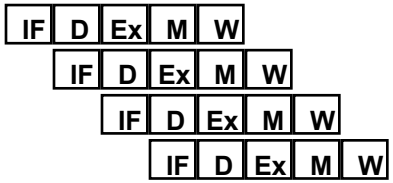
Parar todas las etapas anteriores a la operación multiciclo.

Vaciar (burbujear) las etapas siguientes

Usar la palabra de control de la etapa local Para realizar paso a paso la operación multiciclo.

Aspectos en el diseño avanzado de Pipelines.

° Pipeline común.

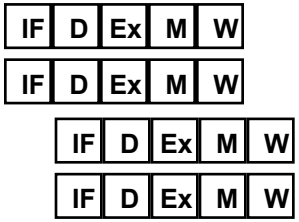


Limitación

Tasa de emisión

° Super escalar

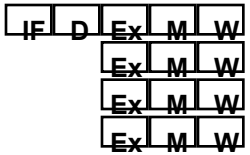
- Emite múltiples instrucciones escalares por ciclo



Resolución de Riesgos y suficiente paralelismo

° VLIW ("EPIC" – Computador de inst. con paralelismo explícito)

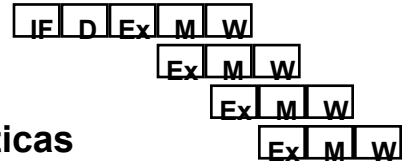
- Cada instrucción especifica múltiples operaciones escalares
-Compilador determina el paralelismo



Suficiente paralelismo

° Procesadores Vectoriales

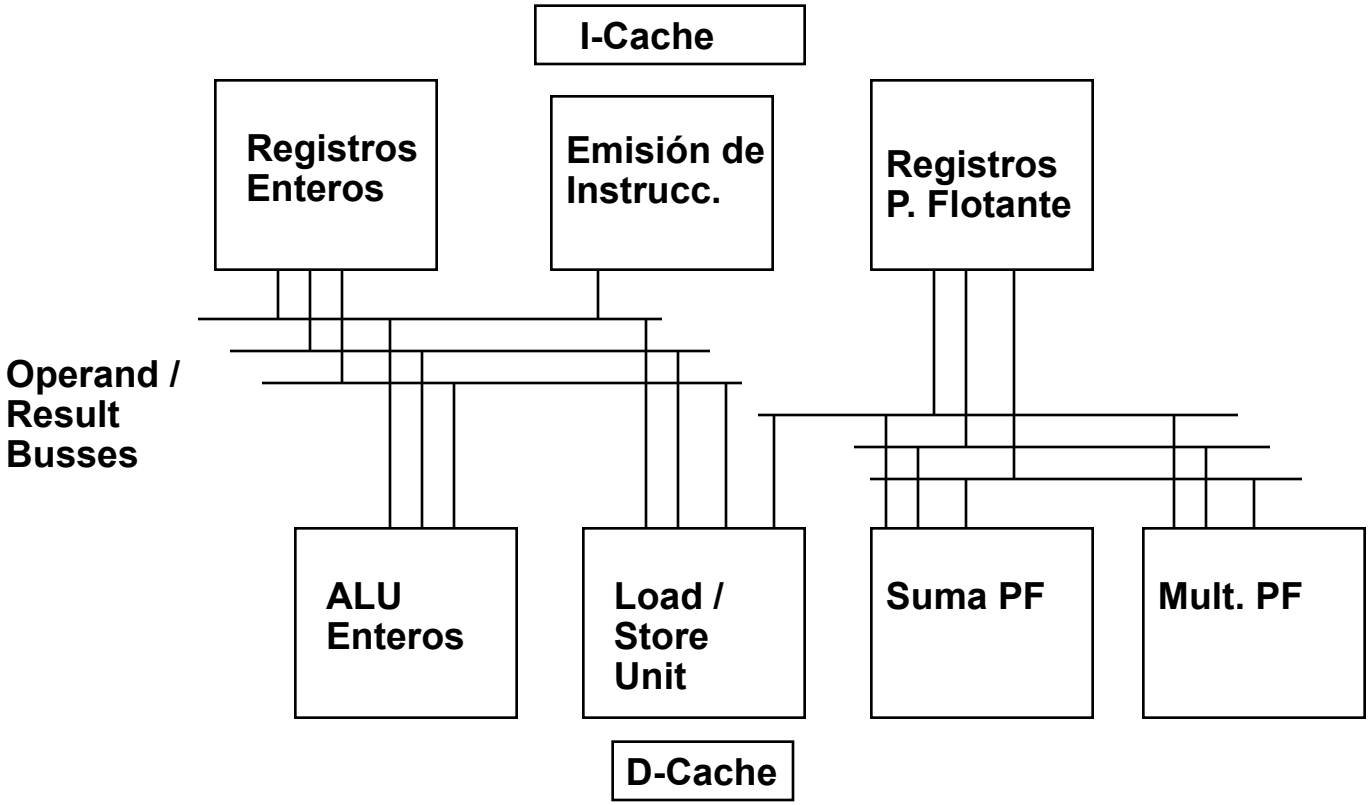
- Cada instrucción especifica una serie de operaciones idénticas



Aplicabilidad

Emisión de Instrucciones Particionada (superescalar simple)

Emisión independiente para operaciones Enteras y Flotantes en pipes separados



Tiempo Total para emisión única = Tiempo Instr. Enteras + Tiempo Instrucciones PF.

Aceleración Máx(sin riesgos): $\dfrac{\text{Tiempo Total}}{\text{MAX(Tiempo Ent, Tiempo FP)}}$

CPI < 1: Emisión de múltiples Instrucciones/Ciclo

- Ej. Superscalar (SS): 2 instrucciones, 1 FP & 1 cualquier otra.
  - Fetch 64 bits/ciclo de reloj; Entera a la izquierda, FP a la derecha.
  - Puede emitirse la segunda sólo si la primera se emite.

Tipo	*Etapas de Segmentación*						
Int. instruction	IF	ID	EX	MEM	WB		
FP instruction	IF	ID	EX	MEM	WB		
Int. instruction		IF	ID	EX	MEM	WB	
FP instruction		IF	ID	EX	MEM	WB	
Int. instruction			IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB

- La demora de 1 ciclo para LOAD se expande a 3 instrucciones en SS.
  - La instrucción en paralelo no la puede usar, ni las dos del próximo T.

Apoyo del Compilador: Desenrollado de Lazos

Ejemplo: Supongamos ADDD produce un retardo de 2 ciclos, LD 1 ciclo.

El siguiente lazo suma un escalar (PF) en F2 a un Vector (PF) apuntado por R1, se comienza por la última componente.

R0 se inicializa para apuntar a la última componente del vector.

```
Loop:   LD     F0, 0(R1)
        ADDD   F4, F0, F2
        SD     0(R1), F4
        SUBI   R1, R1, #8
        BGEZ   R1, Loop
```

¿Cuánto demora?

Apoyo del Compilador: Desenrollado de Lazos

Sin Reordenar, se insertan burbujas para resolver dependencias:

```
Loop:   LD      F0, 0(R1)
        Parada
        ADDD    F4, F0, F2
        Parada
        Parada
        SD      0(R1), F4
        SUBI    R1, R1, #8
        BGEZ    R1, Loop
        Parada
```

Total: 9T por iteración.

¿Y reordenando?

Apoyo del Compilador: Desenrollado de Lazos

Reordenando:

```
Loop:   LD     F0, 0(R1)
        SUBI   R1, R1, #8
        ADDD   F4, F0, F2
        Parada
        BGEZ   R1, Loop
        SD     8(R1), F4
```

Se reduce la duración de 9T a 6T por iteración.

Apoyo del Compilador: Desenrollado de Lazos

Desenrollando 4 veces se reducen instrucciones de control del lazo:

```
Loop:   LD     F0, 0(R1)
        ADDD   F4, F0, F2
        SD     0(R1), F4        ; elimina SUBI y BNZ
        LD     F0, -8(R1)
        ADDD   F4, F0, F2
        SD     -8(R1), F4       ; elimina SUBI y BNZ
        LD     F0, -16(R1)
        ADDD   F4, F0, F2
        SD     -16(R1), F4      ; elimina SUBI y BNZ
        LD     F0, -24(R1)
        ADDD   F4, F0, F2
        SD     -24(R1), F4
        SUBI   R1, R1, #32
        BGEZ   Loop
```

Sin reordenar ya se gana respecto al ejemplo inicial no reordenado. ¿Cuánto?

¡Pero reordenando hay gran mejora!

```
1 Loop: LD     F0,0(R1)
2       LD     F6,-8(R1)
3       LD     F10,-16(R1)
4       LD     F14,-24(R1)
5       ADDD   F4,F0,F2
6       ADDD   F8,F6,F2
7       ADDD   F12,F10,F2
8       ADDD   F16,F14,F2
9       SD     0(R1),F4
10      SD     -8(R1),F8
11      SD     -16(R1),F12
12      SUBI   R1,R1,#32
13      BGEZ   R1,LOOP
14      SD     8(R1),F16    ; 8-32 = -24
```

14 ciclos de reloj, o 3.5 por iteración (14/4)

Y para Superescalar → permite usar paralelismo

	Instrucción Entera	*Instrucción PF*	*Ciclo de Reloj*
Loop:	LD F0,0(R1)		1
	LD F6,-8(R1)		2
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12		8
	SD -24(R1),F16		9
	SUBI R1,R1,#40		10
	BGEZ R1,LOOP		11
	SD 8(R1),F20		12

- Desenrollada 5 veces para evitar paradas (+1 por SS)
- 12 ciclos, es decir 2.4 ciclos por iteración

Antidependencias

- Cada ejecución del lazo emplea las mismas variables.
- Pero se puede desenrollar y si cada ejecución es independiente y se presta para realizarla en paralelo…
- Y si uso la misma variable aparecen riesgos waw (que escriba la variable del segundo desenrollado antes que del primero.
- Entonces la misma variable no significa dependencia ya que se resuelve cambiando el nombre en cada desenrrollado.
- Este caso se denomina “antidependencia”.
- Y por eso los procesadores modernos tienen muchos registros.

Desenrrollado de Lazos en VLIW

Ejemplo: 5 unidades independientes en paralelo

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16			SUBI R1,R1,#56	7
SD 24(R1),F20	SD 16(R1),F24			BGEZ R1,LOOP	8
SD 8(R1),F28					9

Desenrrollado 7 veces se optimiza el uso de Paralelismo.

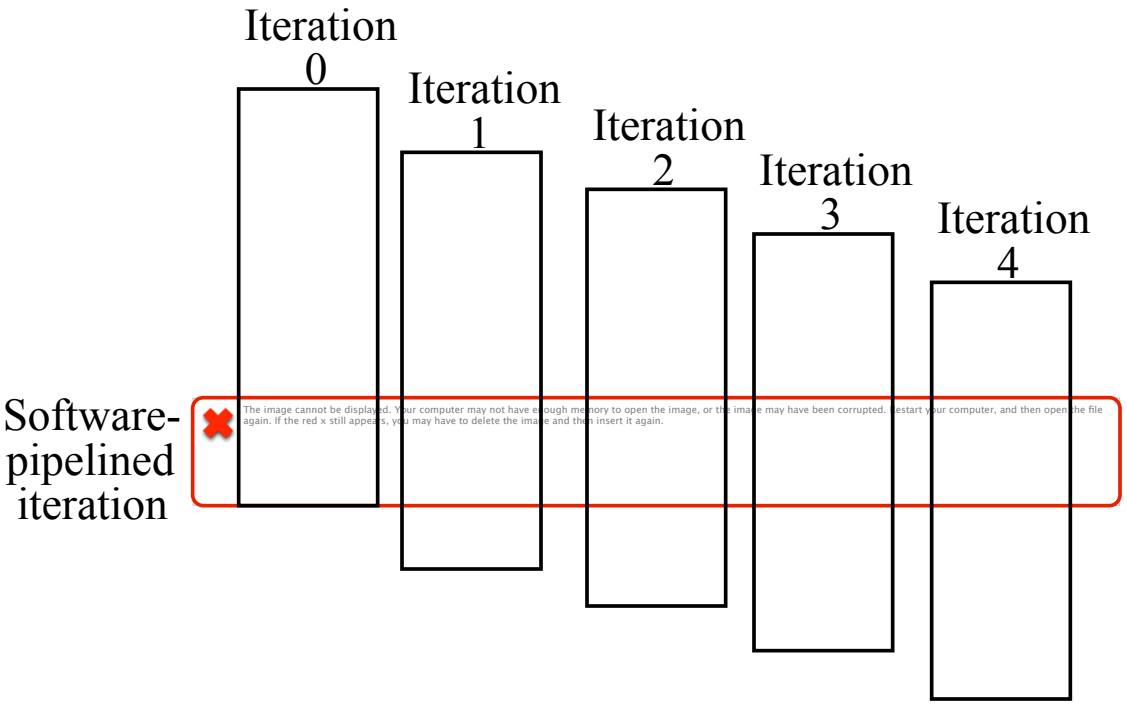
7 resultados en 9 T, o 1.3 T por iteración

Se requieren más registros en VLIW (EPIC => 128int + 128FP)

Software Pipelining

- **Observación: si las iteraciones en el lazo son independientes, se puede tomar distintas instrucciones de distintos lazos y ejecutarlas en paralelo.**

- **Segmentación de Software: se reorganizan los lazos de tal forma que cada iteración esté compuesta por instrucciones de distintas iteraciones del lazo original.**



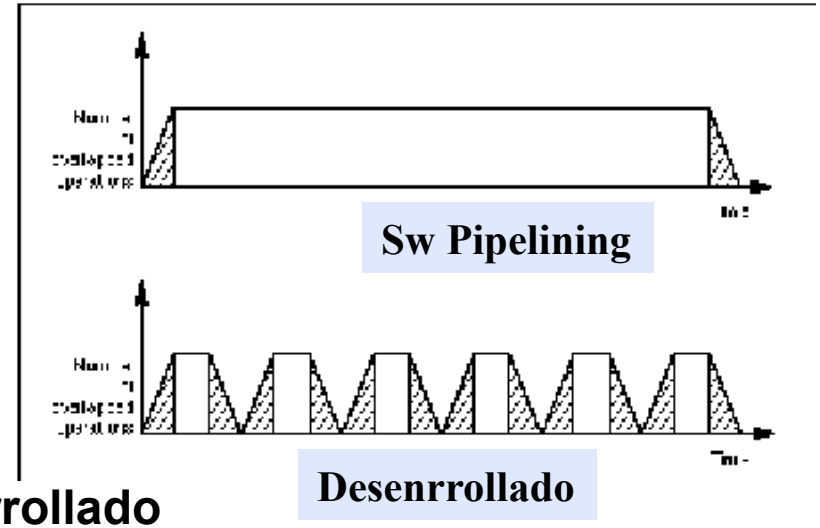
Ejemplo de Segmentación en Sw

Antes: Desenrollando 3 veces.

```
1  LD    F0,0(R1)
2  ADDD  F4,F0,F2
3  SD    0(R1),F4
4  LD    F6,-8(R1)
5  ADDD  F8,F6,F2
6  SD    -8(R1),F8
7  LD    F10,-16(R1)
8  ADDD  F12,F10,F2
9  SD    -16(R1),F12
10 SUBI  R1,R1,#24
11 BGEZ  R1,LOOP
```

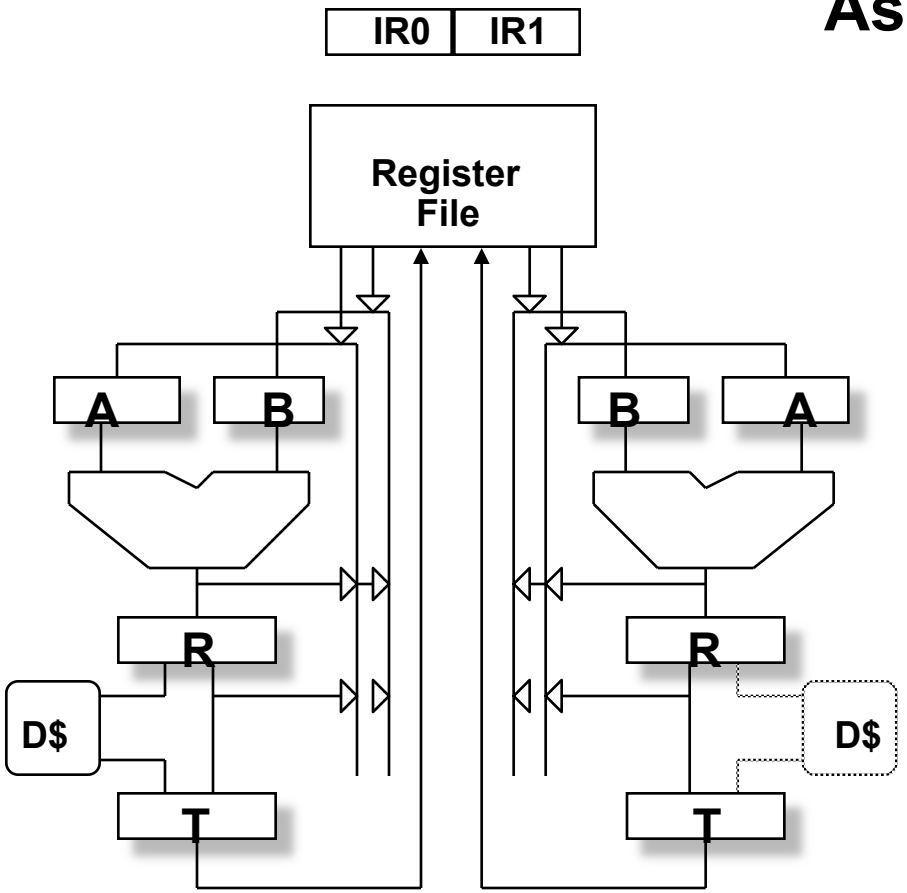
Ahora: con segmentación de Sw

```
1   SD    0(R1),F4 ; Stores M[i]
2   ADDD  F4,F0,F2 ; Adds to M[i-4
3   LD    F0,-16(R1);Loads M[i-2]
4   SUBI  R1,R1,#8
5   BNEZ  R1,LOOP
Cuidado: inicializar y terminar
    adecuadamente los lazos.
```



- **Desenrollo Simbólico del Lazo**
  - **Menor Espacio para el código.**
  - **Llenar y vaciar el Pipe una vez vs. una vez por iteración en el desenrrollado**
  - **Para llenar hueco en BNEZ con SUBI se puede comenzar R1 - 24 y cambiar subind de las 3 inst del lazo, BGEZ resultará correcta.**

Múltiples Pipes/ Superescalar más difícil



Aspectos:

- Puertos del Banco
- de Registros
- Puertos de DM
- Ancho de IM
- Detectar Dependencias
- de Datos
- Anticipaciones
- Riesgos RAW?
- Riesgos WAR?
- Riesgos WAW?
- Múltiples load/store ops?
- Saltos?

Predicción de Saltos y Ejecución Especulativa

Predicción de Saltos.

- En pipes más largos las penalidades son altas.
- La solución de Salto retardado no es la mejor
  - Muy retardado.
  - Código dependiente de la máquina.
- Predecir el salto y continuar, commit solo cuando estamos seguros, sino "unroll".

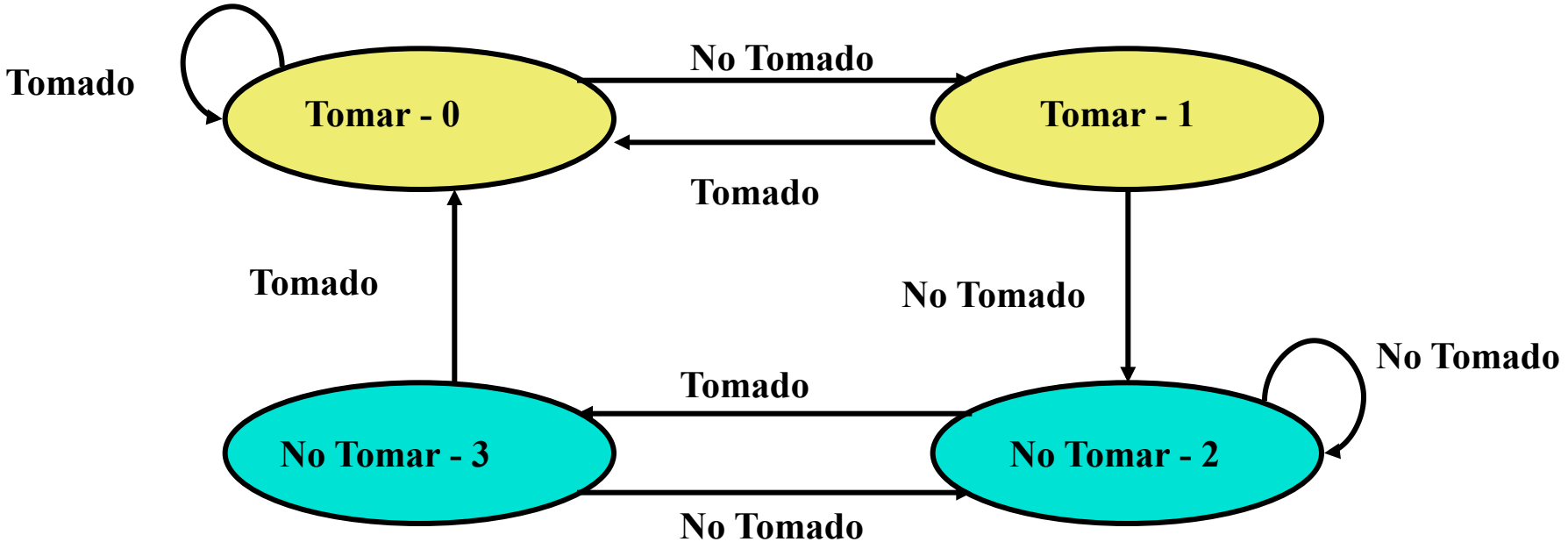
Implementación:

1. Predicción Estática. En un lazo se salta en general (hacia atrás), predecir tomado. Caso contrario... Predecir no tomado. Se tienen códigos distintos para cada instrucción de salto.

2. Predicción Dinámica. No depende de la máquina. Recordar qué pasó la última vez... Repetir esa experiencia.
   - Se emplea una tabla con los últimos bits de las direcciones de salto y allí se guarda qué pasó la última vez.

Predicción Dinámica de Saltos

- **Para un Lazo: Se predice mal al salir del salto y también cuando luego de un tiempo se vuelve a entrar.**
  - **Un salto de 10 iteraciones tiene 80% de predicción correcta.**
- **Mejora: Cambio decisión si equivoco las últimas dos veces:**



Fallas en la Predicción

- **Razones:**
  - **Error al indexar la tabla de predicción: agrandar la tabla.**
  - **Predicción incorrecta en ese salto.**
- **Frecuencia de Fallas**
  - **En tablas de 4096 entradas... 1% a 18% de error en benchmarks.**
  - **No está mal.**
- **Mejora para evitar Paradas.**
  - **En la tabla se escribe el destino del salto.**
  - **Se evita calcular... Y la penalidad por fallo disminuye.**

Ejecución Especulativa

- Instrucciones que se ejecutan si se cumple cierta condición.
- La evaluación de la condición podría detener el pipeline.
- Para no detenerse, se procede con la ejecución de las instrucciones sin conocer el resultado de la condición.
- Todo funciona bien si no se permite que estas instrucciones cambien el estado del CPU. Se proveen “copias de registros”.
- Si la condición es válida, se realiza el commit → los registros se renombran o “copian” a los reales.
- Caso contrario... Se perdió tiempo que igual se hubiera perdido.
- Pero... Sí aparecen costos adicionales: fallos de cache en la ejecución. No permitir a las ejecuciones especulativas provocar fallos, sus resultados en registros quedan marcados con un bit de “veneno”. Sólo cuando se está seguro que no es especulativa se permite el fallo.

Mejoras Adicionales

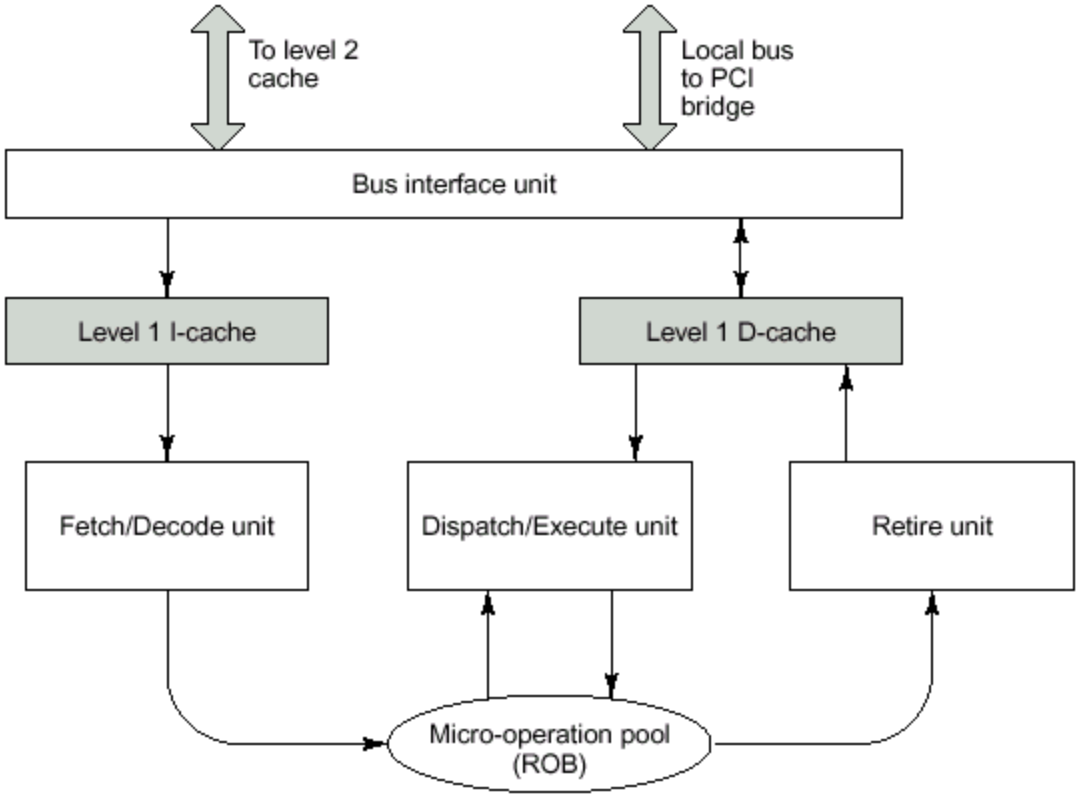
1. **Emisión Fuera de Orden o Dinámica.**
   - Si la emisión de una instrucción está detenida por un riesgo, emitir siguientes que no tengan riesgo (parecida a lo que hacen los compiladores pero se hace "dinámicamente"). Diagr de precedencia.
   - A la instrucción detenida se la deja en una cola de espera. Dim Cola!
   - Ejecución Fuera de Orden. Copias de registros para Antidependencias
   - Las etapas trabajan independientemente con su propia MEF para control.

2. **Completar (Commit) en Orden.**
   - Permite Excepciones Precisas.
   - Se trabaja con un buffer de reordenamiento en la etapa COMMIT.
   - En este buffer están instrucciones en espera de que se completen sus anteriores.
   - Pero hay registros cuyos valores ya se conoce. No son definitivos hasta que no haya el COMMIT de la instrucción que los modifica.
   - Hay otras instrucciones que los necesitan. Se las emite y se provee copias de esos registros (hay muchos registros adicionales para copias). Al finalizar se los renombra si todo está bien.
   - Si algo sale mal, no se tocaron los registros y todo se anula.

Ventajas

- No depende del Compilador.
- Se llama ILP (Instruction Level Paralelism).
- Tomasulo – IBM 360 – 1967.

Arquitectura del Pentium II



ROB: ReOrder Buffer

Pentium II – Fetch / Decode Unit

7 Etapas de Segmentación: IFU0 a ROB.

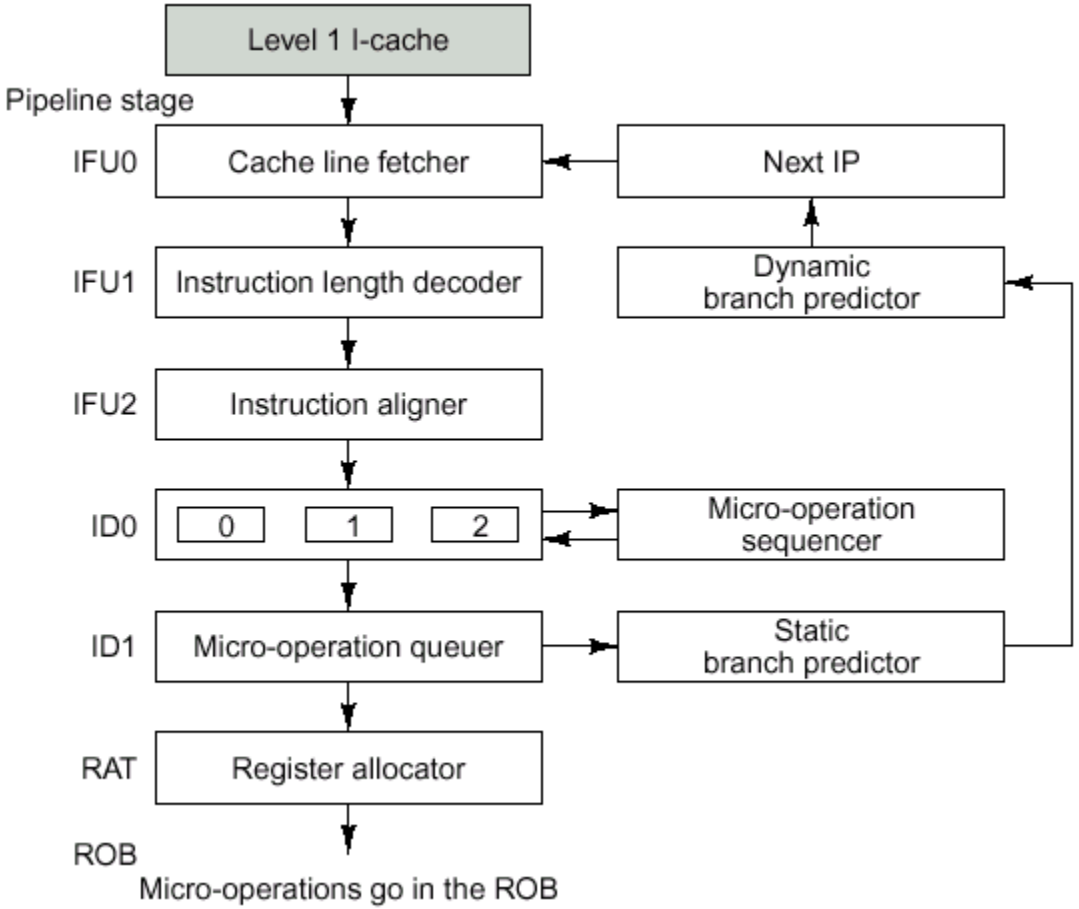
IFU0 – trae 32 bytes del cache a un buffer interno cada vez que se vacía.

IFU1 – Encuentra la longitud de una instrucción (varía).

IFU2 – la alinea al comienzo de ID0

ID0 – decodifica y las convierte en un grupo de MicroInstruc.

ID1 – cola de espera y detección de saltos. La predicción estática es por si no está en la tabla de predicción dinámica de 4 bits.



RAT – se encarga de renombrar registros de un grupo de 40 disponibles.

ROB – Contiene instrucciones pendientes a ejecutar fuera de orden y completar en orden.

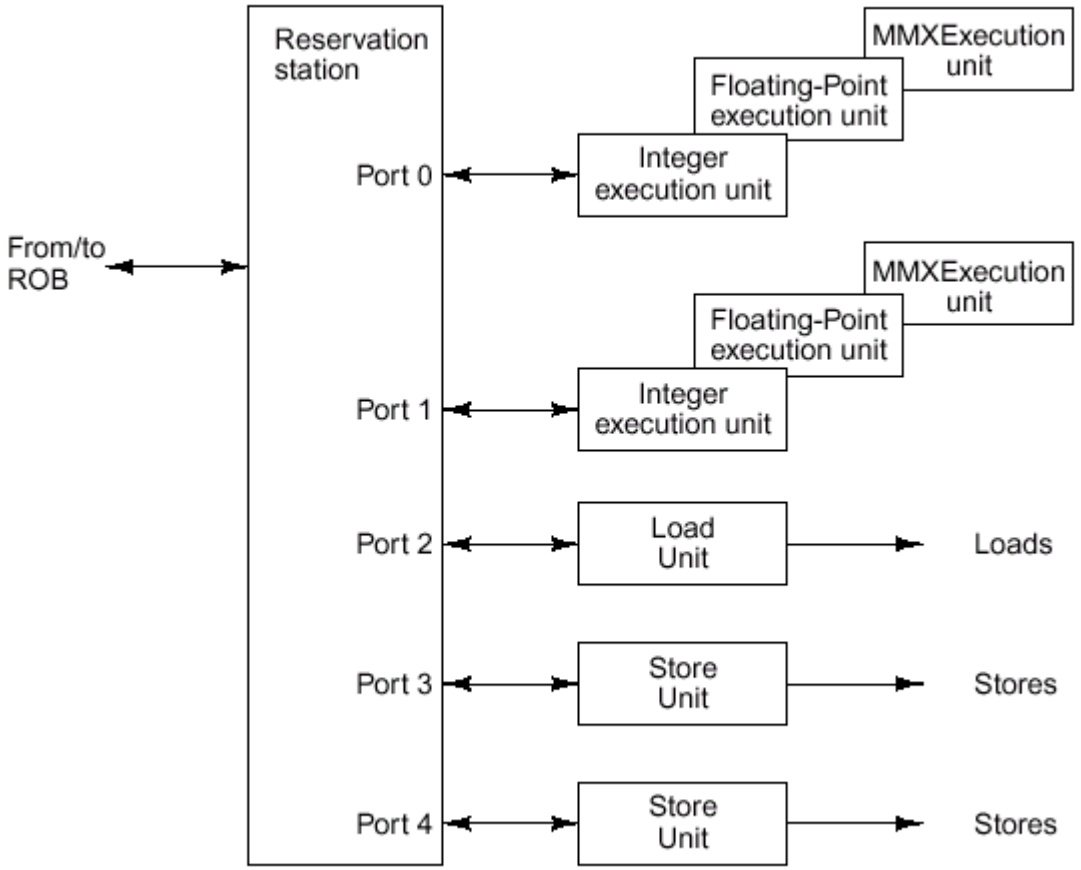
Pentium II – Unidad de Despacho y Ejecución

Se encarga de emisión y ejecución de las instrucciones

Puede emitir hasta 5 inst/ciclo

Maneja un scoreboard para resolver los conflictos antes de emitir.

La estación de reserva guarda hasta 20 operaciones listas para ejecutarse hasta que se liberan sus unidades de ejecución.



La Unidad de Retiro se fija en ROB cuales Inst. están completas, las retira en orden, redenomina los registros y envía sus valores a quienes estén esperando por ellos. Resuelve cuales instrucciones especulativas no se ejecutan (rollback).

Ejecución Dinámica en Pentium IV y Pentium III

Parámetro	PIV	PIII
Max. instructions emitidas / clock	7	5
Profundidad Pipe (duplica frec)	20	16
Conversión a microoperaciones	Trace Cache	PLA
Instrucciones en ROB	126	40
Num de Registros para Renombrar	128	40
Tabla Predicción de Saltos	4K	0,5K
Mejoras en Mem – próximo tema.		

Pentium IV – Desventajas

- Pipeline muy profundo – alta penalidad por riesgos.
- Instrucciones complejas que no se traducen a 1-4 micro operaciones.
- Dependencias largas de datos entre micro operaciones que pueden llevar a riesgos costosos (entre dos inst hay muchas microp.)
- Los riesgos largos son costosos en términos de consumo más que de tiempo, pues sobran recursos.
- Localidad de las Instrucciones baja – veremos Cache.

ITANIUM - Soporte de Hw para más paralelismo

- EPIC – Explicit Paralell Instruction Computer (IA-64, Itanium, Itanium 2)
- EPIC → VLIW – Inst de 128 bits, contienen 3 operaciones y el compilador puede armar paquetes de varias Inst. que pueden emitirse en paralelo.
- Evitar errores en predicción, creando instrucciones condicionales:

  if (x) then A = B op C else NOP
- Si es falso, entonces no se guarda el resultado.
- Se les llama "instrucciones predicadas" o condicionales.
- "Si el salto "if then else" es un problema, ¡elimine el salto!".
- Cada rama del salto se puede ejecutar en paralelo (¡Energía!). Se complementa con desenrollado de lazo para encontrar más paralelismo.
- 64 registros de 1 bit en un campo de 8 bits para condicionar la ejecución.
- Un campo "template" único en la VLIW indica qué unidades funcionales se requieren en cada instrucción y cuándo termina la serie de instrucciones paralelas que preparó el compilador.

ITANIUM

- **LOAD anticipado y Check.**
  - **Un LOAD puede encontrar falla en cache y demorar la ejecución.**
  - **Se anticipa el LOAD unas cuantas instrucciones para atrás.**
  - **Cuando se usa su resultado en una instrucción, se la precede por un CHECK para ver si se completó, sino se espera.**
  - **Si el LOAD está en una rama condicional no se lo anticipa fuera de la rama, y al contrario, si da falla en cache: se espera hasta la evaluación de la condición.**
- **Desventajas de las Ejecuciones Condicionales**
  - **Las condiciones complejas se evalúan tarde y consumen recursos por que se especula durante muchas etapas**
  - **Sin embargo hoy sobran recursos pues no hay en general tanto paralelismo en el programa. Pero… ¡Energía!**

Itanium 2

- Capaz de emitir hasta 6 instrucciones por cada T.
- Capaz de ejecutar (instantánea) hasta 11 ops por cada T.
- Unidades Operativas:
  - 6 Enteras
  - 4 de Memoria.
  - 3 para Saltos.
  - 2 Punto Flotante.
- Frec de reloj 1,5 GHz
- Cantidad Transistores: 221 millones.
- 130 Watts.

En síntesis ILP:

- La segmentación es un tipo de paralelismo transparente al programador (excepto por los riesgos).
- Pipelines más rápidos requieren más etapas. Los riesgos y las etapas largas para algunos casos (FP), hacen inviable un pipe único.
- ILP (Paralelismo a nivel de Instrucción): manejado por el Hw en forma totalmente transparente.
- ¿es totalmente transparente la implementación en pipe del primer MIPS?
- Más rendimiento en ILP.
  - pipelines más profundos y paralelismo.
  - predicción de salto y ejecución especulativa.
  - Ejecución fuera de orden.
  - Renombramiento de registros.
  - Ayuda del compilador mejora pero no define ILP.

En síntesis: VLIW

- **VLIW: arquitectura en que varias operaciones paralelas se colocan en una instrucción muy ancha.**
- **El compilador se encarga de encontrar el paralelismo, no el Hw.**
- **Idea: Hw más simple, Sw más complejo.**
- **Técnicas:**
  - **Sw Pipelining.**
  - **Desenrrollado de lazos.**
  - **Ejecución de más de una alternativa en un salto o case**
- **Ejemplo: Itanium.**
- **Contra: Potencia.**