

1. COMPUTADORES Y FIGURAS DE MÉRITO

ACI: arquitectura del conjunto de instrucciones, equivalente al Lenguaje Máquina (el LM es un lenguaje imperativo)

El conjunto de instrucciones y su codificación, especifica la arquitectura de un computador

- familia: los procesadores que interpretan una misma ACI
- microarquitectura: una implementación concreta de una ACI

Memoria

- latencia: el tiempo entre la solicitud de un dato a memoria y la disponibilidad del dato en la UP
- ancho de banda: el número de bytes que se transmiten por unidad de tiempo
 - multipuerto: en paralelo se pueden efectuar varios accesos a la misma posición de memoria
 - multibanco: entrelaza la información en distintos bancos y permite el acceso concurrente a bancos distintos

$$@ \text{ bloque} = \left\lfloor \frac{@ \text{ mem}}{\text{tamaño del bloque}} \right\rfloor \quad \text{siempre es división exacta}$$

$$\text{banco} = @ \text{ bloque} \bmod \text{número de bancos}$$

Localidad: características observadas de los accesos a memoria de los programas

- temporal: una posición accedida se volverá a acceder
- espacial: una posición cercana a la accedida será accedida

$$OP = \frac{\text{número de operaciones}}{\text{tiempo empleado}} \quad \text{operaciones por unidad de tiempo [op/s]}$$

$$TAM = \frac{1}{OP \cdot \text{accesos de cada operacion}} \quad \text{tiempo de acceso a memoria [s/acceso]}$$

assert: tiempo de proceso de UP negligible

$$AB = \frac{\text{bytes en una transmisión}}{TAM} \quad \text{ancho de banda [bytes/s]}$$

N número de instrucciones del programa [inst]

f_R fallos de cache por cada referencia a memoria [misses/ref]

$f_I = R_I \cdot f_R$ fallos de cache por cada instrucción [misses/inst]

R_I número medio de referencias a memoria por instrucción [ref/inst]

un instrucción como mínimo hace una referencia: ella misma, en BUS

P_f ciclos de penalización en caso de fallo [ciclos/miss]

$CMA = c_{\text{acierto}} + f_R \cdot P_f$ tiempo medio para efectuar una acceso [ciclos/ref]

$CPI = \sum_{i=1}^n \frac{N_i}{N} CPI_i = CPI_{\text{base}} + f_I \cdot P_f$ número medio de **ciclos que tarda una instrucción** del programa, n es

el número de instrucciones distintas del LM [ciclos/inst]

$IPC = \frac{1}{CPI}$ instrucciones por ciclo [inst/ciclo]

f frecuencia del procesador [Hz]

$t_c = \frac{1}{f}$ tiempo de ciclo [s]

$T = N \cdot CPI \cdot t_c$ **tiempo de ejecución** de un programa [s]

$R = \frac{1}{T}$ rendimiento

$MIPS \cdot 10^6 = \frac{N}{T} = \frac{f}{CPI}$ **millones de instrucciones por segundo** [inst · 10⁶ / s]

$G = \frac{T_{\text{original}}}{T_{\text{nuevo}}} = \frac{R_{\text{nuevo}}}{R_{\text{original}}}$ **ganancia** de la nueva versión respecto la original

$$f_m = \frac{t_{\text{donde actúa la mejora}}}{T_{\text{original}}}$$

fracción de tiempo de la versión original donde afectará la mejora

$$g_m = \frac{t_{\text{donde actúa la mejora}}}{t_{\text{donde actúa la mejora tras aplicarla}}}$$

ganancia de tiempo en la parte donde se mejora

$$G = \frac{1}{(1-f_m) + \frac{f_m}{g_m}}$$

ley de **Amdahl**: mejora sólo las partes influyentes

$$CPI_{\text{carga}} = \frac{\sum f_i \cdot T_i}{\sum f_i \cdot N_i} = \sum_{\text{programas}} f_i \frac{N_i}{N} CPI_i$$

CPI medio en una carga de trabajo (f_i frecuencia del i ésimo programa)

$$T_{\text{carga}} = \sum_{\text{programas}} f_i \cdot T_i$$

tiempo de ejecución de varios programas

Procesadores concurrentes para reducir el tiempo de ejecución de un programa de forma transparente al programador:

assert: tiempo distribuido uniformemente entre etapas, no limitación de recursos, instrucciones independientes

- escalares: en cada ciclo se inicia la interpretación de una nueva instrucción y se van solapando

$$G_{\text{segmentación}} = \text{número etapas}$$

- superescalares: la interpretación de una instrucción está segmentada y además se ejecutan varias a la vez

$$G_{\text{segmentación} + \text{paralelismo}} = \text{número etapas} \cdot \text{número de instrucciones por ciclo}$$

- multihilo: aprovecha el tiempo de bloqueo ocioso (e.g. fallo en cache) para cambiar de hilo de ejecución
 - grueso: ejecuta un proceso hasta algún evento
 - fino: en cada ciclo sólo se inician instrucciones de un hilo
 - simultáneo: fino pero con superescalares y con planificación dinámica de instrucciones

Cache no bloqueante: permite servir solicitudes de acceso mientras se está sirviendo un miss

Existe un número máximo permitido de fallos de cache pendientes de servicio

Preferentemente junto con instrucciones de prebúsqueda (como load pero no almacena en registro, sólo en L1)

Load no bloqueante: se siguen interpretando instrucciones tras un miss mientras no se deba usar el dato pendiente

Existe un número máximo permitido de fallos de cache pendientes de servicio

No necesita instrucciones de prebúsqueda, pueden simularse mediante loads

$$C \propto A$$

capacidad efectiva de todo el dado en un ciclo [faradios]

$$V$$

tensión de alimentación [voltios]

$$I$$

intensidad de corriente [amperios]

$$Q$$

carga de una batería [coulombios]

$$E = C \cdot V^2 = Q \cdot V = P \cdot T$$

energía [julios]

$$P = E \cdot f = V \cdot I$$

potencia consumida en conmutación [watts = julios/s]

assert: corrientes de fuga y de cortocircuito negligibles

Teoría del escalado: cada nueva generación escala por 0.7 las dimensiones de los dispositivos y cables. Consecuencias:

$$f' = f / 0.7 \approx 1.5 f$$

assert: el retardo de las puertas: $r' = 0.7 r$

$$C' = 0.7 C$$

assert: el área se escala como indica la siguiente línea

$$A' = 0.7^2 A \approx 0.5 A$$

es decir: el área de las puertas es la mitad

$$P' = 0.7 C \cdot 0.7^2 V^2 \cdot \frac{f}{0.7} \approx 0.5 P$$

assert: escalado del campo eléctrico constante: $V' = 0.7 V$

$$P' = 0.7 C \cdot V^2 \cdot \frac{f}{0.7} = P$$

assert: escalado de tensión constante: $V' = V$

Conclusión:

	contracción (misma arquitectura)	nueva generación (nueva arquitectura)
número transistores	1x	2x
tamaño del dado	0.5x	1x
frecuencia	1.5x	1.5x
consumo de potencia	0.5x	1x
rendimiento	1.5x	3x

MIPS / P	eficiencia cálculos/consumo en portátiles [inst/watt]
MIPS ² / P	eficiencia cálculos/consumo en estaciones de trabajo
MIPS ³ / P	eficiencia cálculos/consumo en servidores
P / MIPS	power-delay product [julios/inst]
P / MIPS ² = E · retardo	energy-delay product
P / MIPS ³	energy-delay ² product
$3 \frac{\Delta f}{f} \approx \frac{\Delta P}{P}$	escalado tensión-frecuencia: para controlar consumo o rendimiento

Productividad = $\frac{\text{número de operaciones}}{\text{tiempo de ejecución}}$ productividad de un dispositivo [tareas/s]

2. SEGMENTACIÓN Y REPLICACIÓN

Retardo del circuito: el número de unidades de tiempo, en el peor caso, que tarda en estabilizarse la última de las señales de salida, a partir del instante en que se han estabilizado todas las señales de entrada

Reloj: dispositivo que produce una señal repetitiva de forma permanente

- periodo o tiempo de ciclo: el intervalo de tiempo de una repetición de la señal
- nivel lógico: nivel 0 es el tiempo durante el cual la señal vale 0, el nivel 1 es cuando vale 1
- flanco: ascendente cuando el nivel lógico de la señal pasa de 0 a 1, descendente cuando pasa de 1 a 0
- forma o aspecto: cuadrada cuando los dos niveles lógicos tienen la misma duración, sino rectangular

Metodología de reloj: define cuándo los elementos de memorización pueden leerse y cuándo escribirse

- usamos actualización por flanco ascendente
- restricciones en el tiempo de ciclo: $t_c \geq t_p + t_{\text{lógica combinacional}}$

Registro: elemento de memorización de un bit, pueden agruparse. Actualización por flanco.

- lectura: puede efectuarse en cualquier momento
- escritura: el valor de la entrada D en el flanco de reloj se observa en la salida Q después de un retardo de propagación t_p

Replicación: se replican los dispositivos para incrementar el número de tareas procesadas por unidad de tiempo.

- las réplicas procesan tareas concurrentemente
- cada réplica procesa tareas de forma serie

Ganancia de la replicación respecto del dispositivo serie cuando se ejecutan n tareas:

assert: todas la réplicas empiezan a procesar tareas en el mismo instante, se inicia una tarea sin demora

$$T_{\text{rep}} | n = \left\lceil \frac{n}{ND} \right\rceil (t_p + t_{\text{lógica}}) \quad \text{donde ND es el número de dispositivos}$$

$$G = \frac{T_{\text{serie}} | n}{T_{\text{rep}} | n} = \frac{n (t_p + t_{\text{lógica}})}{\lceil n/ND \rceil (t_p + t_{\text{lógica}})} \approx ND \quad \text{assert: n múltiplo de ND}$$

Segmentación: para empezar a procesar una tarea antes de haber finalizado la precedente. Construcción:

1. distinguir secuencias de fases
2. identificar los módulos combinacionales que efectúan cada una de las fases o grupos de ellas
3. establecer un secuenciamiento en la utilización de los módulos combinacionales
4. insertar registros de desacoplo entre las conexiones lógicas establecidas entre los módulos. Esto se hace para aislar las distintas etapas durante el periodo de la señal de reloj.

Los registros de desacoplo tendrán varios modos de funcionamiento para permitir el control de riesgos:

- normal: el valor de la entrada se transfiere a la salida en el flanco ascendente
 - burbuja: en la salida del registro hay una NOP después de que se produzca el flanco
 - bloqueo: la salida del registro no se modifica (bloqueo y burbuja simultáneos -> se hace burbuja)
5. el tiempo de ciclo será el necesario en la etapa más lenta: $t_c \geq t_p + \max(t_{\text{lógica combinacional}})$

Ciclos perdidos: los ciclos en los que se reduce el grado de concurrencia en la interpretación de instrucciones

ciclos perdidos = LMI - 1 donde el -1 indica que lo normal es 1 ciclo por instrucción
se contabiliza un ciclo de bloqueo en el ciclo en que finaliza una instrucción NOP

Latencia

- de una operación: tiempo total de procesado de la tarea
- de cálculo: tiempo desde que se dispone de los datos para efectuar el cálculo hasta que finaliza el cálculo
- de inicio: número de ciclos que hay que esperar entre el inicio de dos tareas consecutivas

secuencia de latencias de inicio: lista con el patrón periódico de latencias de inicio, permite calcular la media:

$$LMI = \frac{\sum \text{latencias de inicio}}{\text{instrucciones ejecutadas}} = \frac{\sum \text{lista}[i]}{\text{tamaño lista}} = \frac{1}{\text{Productividad [en inst/ciclo]}}$$

- prohibida: latencia de inicio que da lugar a un riesgo estructural
- efectiva de segmentación: número de ciclos que transcurren desde el ciclo en que se leen los datos hasta el primer ciclo en que se puede utilizar el resultado de un cálculo (ambos inclusive)
- del bucle hardware: número de etapas entre la etapa inicio (destino del bucle) y final (origen del bucle), esta última inclusive
- productor-uso: ciclos que deben transcurrir entre una instrucción productora y una consumidora para que no haya riesgo de datos

Tabla de reserva: diagrama que muestra la ocupación de las etapas de un dispositivo segmentado en función del tiempo.

Permite caracterizar las latencias de inicio permitidas y prohibidas

	ciclo0	ciclo1	ciclo2		ciclo0	ciclo1	ciclo2
etapa0	X		X	<=>	etapa0	etapa1	etapa0/1
etapa1		X	X				

Ganancia de la segmentación respecto del dispositivo serie

assert: distribución uniforme del tiempo serie en los distintos módulos combinacionales de cada etapa

$$t_{\text{etapa}} = t_p + t_{\text{lógica}} / NE \quad \text{donde NE es el número de etapas}$$

$$T_{\text{seg}} = NE \cdot t_{\text{etapa}} = NE \cdot t_p + t_{\text{lógica}}$$

Perdemos rendimiento en una tarea individual, pero cuando se ejecutan n tareas:

assert: $n \gg NE$ y además en cada ciclo se puede iniciar una nueva tarea

$$T_{\text{seg}}|_n = n \cdot t_{\text{etapa}} + (NE - 1) \cdot t_{\text{etapa}} \approx n \cdot t_{\text{etapa}}$$

$$G = \frac{T_{\text{serie}}|_n}{T_{\text{seg}}|_n} = \frac{n(t_p + t_{\text{lógica}})}{n(t_p + t_{\text{lógica}}/NE)} \approx NE \quad \text{assert: } t_p \ll t_{\text{lógica}}$$

y considerando una latencia media de inicio:

$$G \approx \frac{NE}{LMI}$$

Tipos de unidades funcionales segmentadas:

- dispositivo segmentado unifunción: únicamente procesa un tipo de tarea
 - dispositivo segmentado lineal: cada etapa se conecta exclusivamente a etapas sucesoras y ninguna etapa tarda más de un ciclo
 - dispositivo multiciclo o pseudolineal: como el anterior pero una etapa puede tardar más de un ciclo (puede convertirse en lineal aumenando el tiempo de ciclo)
 - dispositivo segmentado no lineal: alguna etapa se utiliza más de una vez y al menos una de las utilizaciones es en ciclos no consecutivos
- dispositivo segmentado multifunción
 - estático: puede procesar varias tareas de varios tipos, pero en un instante todas serán del mismo tipo
 - dinámico: puede procesar de forma solapada varias tareas de distinto tipo

Riesgo estructural: cuando en un dispositivo segmentado tareas concurrentes necesitan un mismo recurso a la vez

- soluciones posibles:
 - replicar el recurso tantas veces como sea necesario
 - añadir caminos de acceso al recurso (e.g. buses de lectura al BR)
 - si el retardo del recurso es inferior al tiempo de ciclo -> usarlo varias veces en un sólo ciclo
 - retardar el uso de un recurso con el objetivo de lograr una segmentación unificada: un mismo patrón de uso de los recursos, es decir que todas las instrucciones usen el recurso (si es necesario) en el mismo ciclo contando desde el inicio de la interpretación
- considerar la frecuencia y el impacto de los riesgos: no siempre es provechoso eliminar los riesgos

Control de dispositivos segmentados: en caso de riesgo estructural dinámico se serializa el procesamiento de tareas

- el control ha de:
 - determinar cuándo una tarea se inicia o prosigue a la siguiente etapa
 - controlar la lógica de cada etapa y el encaminamiento entre las etapas
- tipos:
 - control estacionario en el tiempo: un elemento centralizado que en cada ciclo envía las señales a todos los elementos del dispositivo
 - control estacionario en los datos: la información de control fluye con los datos y el control está integrado en la segmentación. Una etapa puede actualizar las señales de control que pasará a la siguiente etapa.
- etapa de retención: etapa en la que se retiene una instrucción hasta que desaparezcan los riesgos
- NOP: instrucción especial que no modifica el estado del procesador
- ejemplo: segmentado en CP | BUS | D/L | ALU | MEM | ES y en D/L es donde se decodifica -> el conflicto se producirá después -> controlamos la progresión de la interpretación y no el inicio.

Al detectar un riesgo estructural:

- retener etapas CP, BUS y D/L
- inyectar NOP desde D/L hacia ALU

Memoria cache: organizada en contenedores (etiqueta + bloque o línea)

$B = 2^b$ tamaño del bloque [bytes/bloque]

$C = 2^c$ capacidad de la MC [bytes]

$TE = 2^e$ tamaño del elemento [bytes]

$NC = \frac{C}{B} = \frac{2^c}{2^b} = 2^{c-b}$ número de contenedores de MC

$@ \text{bloque} = \frac{@ \text{mem}}{B} = \frac{@ \text{mem}}{2^b}$

$EL = (@ \text{mem} \bmod B) / TE$ número del elemento dentro de su bloque correspondiente

$EN = @ \text{bloque} \bmod NC$ entrada de la MC donde se mapea la línea (mapeo directo)

$ETIQ = \frac{@ \text{mem}}{C} = \frac{@ \text{bloque}}{NC}$ etiqueta que identifica una entrada de la MC

Una @mem tiene el siguiente formato [bits de cada parte, toda la @mem tiene n bits]:

ETIQ [n-c]		EN [c-b]		EL [b-e]		[e]
------------	--	----------	--	----------	--	-----

3. PROCESADOR SEGMENTADO LINEAL

Relación entre instrucciones:

- instrucción vieja: una instrucción p es vieja respecto a otra instrucción q si p se ejecutaría antes que q en un procesador serie
- instrucción joven: una instrucción p es joven respecto a otra instrucción q si p se ejecutaría después que q en un procesador serie

Trabas para lograr un CPI de 1:

- recursos hardware
- semántica del lenguaje:
 - secuenciamiento de instrucciones: no se puede anticipar cuál es la siguiente instrucción
 - dependencias de datos: el orden de lecturas y escrituras puede verse alterado

CPI: suponemos que las instrucciones están dentro de una secuencia mayor de instrucciones. Para calcular el tiempo que tarda contamos los ciclos desde la finalización de la primera instrucción hasta la finalización de la última (ambos ciclos inclusive). Luego se divide por el número de instrucciones que han finalizado (sin contar las NOP)

Bucle hardware o lazo: comunicación entre etapas que permite que en una etapa se utilice información suministrada desde etapas posteriores, es decir desde instrucciones más viejas. Determinan que sea necesario añadir control en el camino de datos segmentado para que se respete la semántica del lenguaje. Son necesarios ya que:

- el resultado de una instrucción lo puede utilizar una instrucción posterior
- una instrucción puede determinar cuál es la siguiente instrucción que debe interpretarse


Riesgo: cuando la semántica del camino de datos difiere de la semántica del lenguaje máquina

- de secuenciamiento: interpretación de una secuencia de instrucciones distinta de la que debería
- de datos: modificación del orden de lecturas y escrituras sobre una misma posición de almacenamiento

Para que haya una dependencia de datos entre dos instrucciones al menos una de ellas ha de hacer una escritura
Nomenclatura:

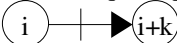
- R: rango de una instrucción: conjunto de posiciones de almacenamiento que escribe una instrucción
- D: dominio de una instrucción: conjunto de posiciones de almacenamiento que lee una instrucción
- orden de programa: es el orden de interpretación de las instrucciones en un procesador serie

Tipos de dependencias de datos:

- $R(i) \cap D(i+k) \neq \emptyset$ 

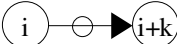
Dependencia Verdadera: lectura tras escritura (RaW)

La segunda instrucción ha de esperar a que la primera produzca el dato

- $D(i) \cap R(i+k) \neq \emptyset$ 

Antidependencia (dependencia de nombre): escritura tras lectura (WaR)

La segunda instrucción no puede escribir hasta que la primera haya leído

- $R(i) \cap R(i+k) \neq \emptyset$ 

Dependencia de Salida (dependencia de nombre): escritura tras escritura (WaW)

La segunda instrucción ha de escribir después de que lo haga la primera

Determinan un grafo de dependencias: los nodos son las instrucciones y los arcos (dirigidos) van de la fuente de dependencia hacia el destino de la dependencia (de la instrucción vieja a la joven). Permite exponer un orden parcial de las instrucciones: explícita cuáles han de ir obligatoriamente antes y cuáles no importa.

Lógica de interbloqueos (de la segmentación): hardware que detecta los riesgos y actúa en consecuencia

- la actuación consistirá en emular un funcionamiento serie, es decir serializar la interpretación de instrucciones
- es necesario considerar las interacciones entre las acciones hechas ante riesgos de datos y de secuenciamiento
 - en el ejemplo se gestiona antes el riesgo de datos que el riesgo de secuenciamiento
- ◆ actuación cuando en decodificación se detecta una instrucción de secuenciamiento
 - ◆ descartar las instrucciones más jóvenes que se hayan iniciado
 - ◆ suspender la interpretación de nuevas instrucciones

cuando la instrucción de secuenciamiento finalice se vuelve a permitir la interpretación de nuevas instrucciones

- en el ejemplo, BUS no modifica el estado y CP se reescribirá cuando finalice la instrucción BR, por lo que no necesitamos eliminar las jóvenes: inyectamos NOP desde BUS mientras haya una instrucción de secuenciamiento en las etapas D/L, ALU, M o ES

- ◆ para el riesgo de datos se han de considerar por separado los distintos elementos de memorización
 - ◆ registros
 - ◆ memoria

y en cada uno de ellos las varias posibilidades

- ◆ $R(i) \cap D(i+k) \neq \emptyset$
- ◆ $D(i) \cap R(i+k) \neq \emptyset$
- ◆ $R(i) \cap R(i+k) \neq \emptyset$

es necesario esperar a decodificación para saber los registros fuente y destino de la instrucción actual

- en el ejemplo, el riesgo $R(i) \cap D(i+k) \neq \emptyset$ se detecta en D/L comparando los registros fuente con los registros destino de las instrucciones que hay en ALU y M; si hay riesgo bloqueamos las etapas CP, BUS y D/L e inyectamos NOP desde D/L.

Para los $D(i) \cap R(i+k) \neq \emptyset$ y $R(i) \cap R(i+k) \neq \emptyset$ nos salvaremos: las lecturas viejas siempre se hacen antes que las escrituras jóvenes, y las escrituras viejas también se hacen antes que las jóvenes.

El acceso a memoria siempre se realiza en el mismo ciclo de interpretación en cualquier caso: no hay riesgo de datos debido a memoria.

El control de riesgos actual no modifica estos comportamientos: las instrucciones inician la ejecución en orden de programa.

Procesador en orden: cuando las instrucciones inician la fase de ejecución en orden de programa (aun habiendo sido segmentado)

4. TÉCNICAS PARA TOLERAR O REDUCIR LA LATENCIA EFECTIVA DE LA SEGMENTACIÓN

Tolerar la latencia: cuando una dependencia de datos entre dos instrucciones no ocasiona un riesgo de datos.

Reducir la latencia: poder utilizar un resultado antes de que se escriba en el elemento de almacenamiento.

Planificación de instrucciones: dada una secuencia ordenada de instrucciones pueden existir otras ordenaciones que efectúen el mismo cálculo -> reordenamos con el objetivo de reducir el tiempo de ejecución

El objetivo es separar una instrucción productora de una consumidora para reducir los ciclos perdidos

- BB: un bloque básico estático es una secuencia de instrucciones donde (1) ninguna instrucción, excepto tal vez la primera, es el destino de una instrucción de secuenciamiento y (2) todas las instrucciones se interpretan si se interpreta la primera instrucción, estático porque se generan en tiempo de compilación.

Identificación de los BB:

1. encontrar líderes, las primeras instrucciones de los BB. Son:
 - la primera instrucción del código
 - la instrucción destino de una instrucción de secuenciamiento
 - la instrucción que sigue a una instrucción de secuenciamiento
2. un líder y todas las instrucciones que le siguen hasta antes del próximo líder conforman un BB

Las instrucciones de un BB siempre se ejecutarán en secuencia -> puede crearse un grafo de dependencias de datos acíclico

- Algoritmo de planificación *assert:* nuestro procesador segmentado lineal que inicia ejecución cada ciclo

```
sub AlgoritmoDePlanificación (instrucciones inst) {
    res = secuencia de instrucciones vacía;
    foreach ( bb : dividir inst en BB ) {
        gd = contruir el grafo de dependencias de bb;
        gd.modificar( λ() {
            foreach ( arco ) {
                etiqueta = 0 if ( es antidependencia o dependencia de salida );
                etiqueta = retardo_productor_uso - 1 if ( es dependencia verdadera );
            }
            if ( bb acaba en una instrucción de secuenciamiento )
                añadir arcos etiquetados con 0 desde el resto de hojas hasta su nodo
            else
                añadir arcos etiquetados con 0 desde todas las hojas hasta un nodo ficticio
        });

        for ( t=0 ; quedan instrucciones por planificar ; t++ ) {
            lista_elegibles = seleccionar de gd:
                · los nodos r que no tengan predecesores
                · los nodos n que todos sus predecesores hayan sido planificados
                  y además el tiempo_más_cercano de n cumpla: n.TMC <= t
            if ( lista_elegibles no es vacía ) {
                p = seleccionar heurísticamente una instrucción de la lista_elegibles:
                    · el nodo que tiene el camino de mayor peso hasta el nodo hoja de bb
                      si hay varios caminos para un mismo elegible, se considera el más pesado
                    · en caso de empate seleccionar el primero en el orden de programa original
                foreach ( s : sucesores de p )
                    s.TMC = max( s.TMC, t+1 + etiqueta del arco de p hasta s );
                gd -= p;
                res += p;
            }
        }
    }
    return res;
}
```

Cortocircuito: es una conexión entre una etapa donde se dispone de un valor producido (y todavía no escrito en el elemento de memorización del cual se leerá) y una etapa donde el valor se utiliza como dato fuente. Es el hardware empleado para permitir que un consumidor disponga de los datos producidos antes de que se escriban

- se crea un bucle hardware nuevo con el objetivo de eliminar riesgos de datos y con ello reducir ciclos perdidos
- añadir hardware al camino de datos (buses, multiplexores y control) puede requerir aumentar el tiempo de ciclo, por lo que no siempre será conveniente (Amdahl)
- tipos extremos, básicos y elementales de cortocircuitos :
 - al inicio de ciclo: un dato disponible al inicio de una etapa se transmite al inicio de una etapa precedente
 - al final de ciclo: un dato disponible al final de una etapa se transmite al final de una etapa precedente

No es necesario que sea de inicio->inicio o final->final, simplemente considerar el retardo que pueden sufrir

Metodología de enumeración de cortocircuitos:

- identificar los distintos tipos de instrucciones (RR, RI, Lo, St, ...) y los registros fuente y destino de cada uno
Notación: tipo.registro
- para determinar si es posible utilizar un cortocircuito hay que tener en cuenta:
 - la latencia de la fase de ejecución
 - la distancia entre las instrucciones
 - la última etapa en que es necesario el dato en la instrucción consumidora
- enumeración en una matriz tridimensional:
 - x - consumidor de dato (cada tipo de instrucción por todos los registros fuente)
 - y - productor de dato (cada tipo de instrucción por todos los registros destino)
 - z - número de ciclos transcurridos entre el inicio de la fase de ejecución de la productora y la necesidad del dato por la consumidora para poder iniciar o proseguir la ejecuciónen cada posición de la matriz marcamos si se puede usar un cortocircuito o no
- a partir de la matriz se obtienen directamente los cortocircuitos lógicos, pero pueden agruparse algunos de lógicos en un mismo cortocircuito físico (también puede darse la situación inversa)

Lógica de interbloqueos

- en un camino de datos segmentado y con cortocircuitos, la lógica de interbloqueos deberá controlar tanto los bloques como los cortocircuitos. Además se ha de considerar que:
 - la lógica que determina los bloqueos por riesgo de datos ha de tener en cuenta la presencia de los cortocircuitos
 - la lógica de activación de los cortocircuitos actúa independientemente de las condiciones de bloqueo (ya sean por riesgos de datos como por riesgos estructurales)
- un dato se ha de obtener de la instrucción más joven que lo esté produciendo: se han de priorizar los cortocircuitos -> tal vez pueda hacerse sólo ordenando los multiplexores que encaminan los datos

Instrucciones de secuenciamiento: la actuación de descartar las instrucciones iniciadas antes de que se decida el salto resulta muy costosa: se ha de intentar resolver un salto lo antes posible

- en el ejemplo las instrucciones de salto condicional pueden resolverse en ALU, las incondicionales en D/L. Es necesario añadir circuitería para que en D/L se efectúe el cálculo de la dirección de salto y poner nuevos bucles

Predicción del sentido

- se actúa según una hipótesis de cómo será el salto en una instrucción de secuenciamiento: tomado o no.
- en caso de acierto reduce los ciclos perdidos, pero no disminuye la latencia efectiva de la segmentación
- en caso de error de predicción se han de deshacer los posibles cambios que se hayan iniciado (recuperación)
- tras realizar una predicción:
 1. las fases que se interpreten de las instrucciones predichas no deben modificar el estado
 2. se ha de verificar la predicción
 3. hay que restaurar el flujo correcto de instrucciones cuando se detecte un error de predicción: descartar las instrucciones predichas y realizar el salto correcto
- el control se complica: instrucciones predichas pueden tener riesgos de datos o ser nuevos saltos
- predicción fija de sentido en saltos condicionales: dada una instrucción de secuenciamiento condicional siempre se efectúa la misma hipótesis sobre su sentido (que puede estar codificada en la instrucción)
 - en el ejemplo: realizamos la predicción en D/L y consiste en seguir en secuencia si el desplazamiento del salto es positivo, y modificar el secuenciamiento si el desplazamiento es negativo (ya que probablemente corresponda a un bucle)
- Seguir en secuencia: si acertamos no se pierden ciclos, sino perdemos 2 ciclos
- Modificar el secuenciamiento: si acertamos perdemos sólo 1 ciclo, sino perdemos 2 ciclos

5. PROCESADOR SEGMENTADO MULTICICLO

Procesador segmentado multiciclo: el camino de datos del procesador está ramificado: CP, BUS y D/L son comunes pero la ejecución y escritura de cada categoría de instrucciones está independizada (paralela, ramificada)

Justificación de su necesidad:

- las operaciones en coma flotante tardan mucho más que el resto -> la unificación conlleva pérdidas
- el acceso a memoria tiene una latencia variable en función de en qué nivel de la jerarquía esté el dato (suponemos memoria de instrucciones con latencia fija)

Ramificación

- cada camino puede adecuarse al tipo de instrucción -> una instrucción usa todas las etapas que atraviesa por lo que el hardware es más eficiente
- la latencia de actualización del BR se adecúa a la latencia de cálculo -> menos cortocircuitos son necesarios
- si los riesgos se gestionan antes de la ejecución, el control de las ramas puede ser completamente independiente del resto ya que no quedarán interacciones posibles

Camino de datos ramificado: cuando se inicia la ejecución de una instrucción por una ramificación, se ha de indicar al resto de ramificaciones que no hagan nada.

- Las ramificaciones que no estén segmentadas son una excepción:
 - cuando se esté iniciando la ejecución de otra ramificación, a las no segmentadas no se les ha de indicar que no hagan nada si en el próximo ciclo todavía van a estar ocupadas en su primera (única?) etapa de ejecución
 - en los bloqueos no se puede inyectar NOP desde D/L a las ramificaciones no segmentadas que vayan a seguir ocupadas el próximo ciclo

Instrucción

- preparada: una instrucción está preparada para iniciar la ejecución cuando sus datos están disponibles (en el BR o a través de un cortocircuito) y está libre de otros riesgos de datos
- lista: una instrucción está lista para iniciar la ejecución cuando no existen riesgos estructurales ni de datos que lo impidan

Ordenación (en nuestro ejemplo)

- inicio de procesado de una instrucción: se realiza en orden -> inicio de ejecución en orden
- finalización de las instrucciones: en desorden de programa -> aparecen riesgos $R(i) \cap R(i+k) \neq \emptyset$

Lógica de interbloqueos (en nuestro ejemplo)

- riesgos de secuenciamiento: como en el lineal pero teniendo en cuenta además los saltos condicionales que evalúan la condición sobre registros en coma flotante
- riesgos estructurales: existe riesgo debido al camino compartido de escritura en el BR CF y en la unidad funcional no segmentada (división)
 - camino compartido: determina una latencia prohibida de 16 en las secuencias: FDIV ... FADD/FMUL
 - UF no segmentadas: usamos un vector de ocupadas (**VO**), cada posición corresponde a una UFNS y almacena un bit que se activa en su primer ciclo de ejecución y se desactiva en el ciclo previo a la actualización del BR CF

Suponemos que el VO se puede escribir y leer (en este orden) en un mismo ciclo.

Si en D/L el bit está activado, entonces bloquear

- riesgos de datos: los de $D(i) \cap R(i+k) \neq \emptyset$ no se producen porque las lecturas viejas siempre se realizan antes que las escrituras jóvenes, y el mecanismo de gestión de riesgos no cambia este comportamiento.

Para los otros dos riesgos de datos, el control se realiza mediante un vector de bits llamado vector de marcas (**VM**) donde cada bit está asociado a un registro concreto de un banco de registros dado. Con el bit indicamos la disponibilidad o no del contenido del registro (tal vez disponible a través de un cortocircuito): un bit a 1 indica que el registro se está calculando, a 0 que ya está disponible. Al iniciar la fase de ejecución se ha de marcar el bit correspondiente al registro destino, en el último ciclo de ejecución se ha de desmarcar (si la latencia productor-uso de la instrucción es 1 no se realiza el marcado del VM). Entonces:

- $R(i) \cap D(i+k) \neq \emptyset$: bloqueamos en D/L si el VM indica que los registros fuente no están disponibles
- $R(i) \cap R(i+k) \neq \emptyset$: este riesgo aparece debido a que las instrucciones finalizan en desorden de programa. Nuestra actuación será conservadora y consistirá en bloquear en D/L hasta que la instrucción i esté en el ciclo previo al de escritura, es decir mientras el VM indique que el registro destino todavía se está calculando por otra instrucción anterior.
- cortocircuitos: como en el lineal, pero ahora hay más productores y los consumidores se han de reconsiderar por cada ramificación. En todas las ramificaciones puede estar acabando una instrucción (excepto en CF, que debido a riesgos estructurales una FDIV y una FADD/FMUL no acabarán en el mismo ciclo) pero la gestión de riesgos $R(i) \cap R(i+k) \neq \emptyset$ garantiza que los registros destino serán distintos en todas las instrucciones que estén en fase de ejecución (es decir, no nos hace falta considerar la edad de las instrucciones)

Terminología de cortocircuitos:

- red de cortocircuitos: el conjunto de multiplexores que permiten disponer de un dato antes de actualizar el banco de registros
- bus de resultados: cada uno de los buses que transportan datos hasta los multiplexores de cortocircuito
- bus de etiqueta: bus que transporta el identificador de registro destino y que se utiliza para controlar los caminos de cortocircuito

Transformaciones de código usadas para incrementar el paralelismo a nivel de instrucción para tolerar la latencia productor-usuario. Después de aplicadas, se usa el algoritmo de planificación para obtener el código máquina final

- Renombre: sirve para eliminar las dependencias de nombre y exponer un mayor paralelismo a nivel de instrucción (hecho para el BR)

Las dependencias de nombre o falsas dependencias son:

- $D(i) \cap R(i+k) \neq \emptyset$
- $R(i) \cap R(i+k) \neq \emptyset$

Son debidas a la actualización de una misma posición de almacenamiento, pero no hay flujo de información de la instrucción vieja a la joven: podríamos usar registros distintos y las dependencias desaparecerían.

Por ello, en la creación de código:

- supondremos que hay un número ilimitado de registros: usaremos registros virtuales
- en un registro sólo escribirá una única instrucción: en cada instrucción de escritura se definirá un nuevo registro virtual
- un registro podrá ser leído las veces que se quiera

Cuando el código haya pasado por el planificador de instrucciones, los registros virtuales tendrán que asignarse a registros reales según la ACI que se esté usando.

- Desenrollar: sirve para aumentar el tamaño de los BB y dar al algoritmo de planificación mayor holgura.

La transformación consiste en:

- los bucles que no contengan instrucciones condicionales
- los ampliamos replicando su cuerpo (número de replicas en función del número de registros disponibles y del tamaño del cuerpo del bucle: no queremos miss en la MC de instrucciones)
- y eliminamos las dependencias de nombre que hayan aparecido

El potencial de esta transformación se explota cuando los load y store acceden a posiciones de memoria distintas, es decir que el planificador puede modificar el orden de programa. Beneficio colateral: se ejecutarán menos instrucciones de secuenciamiento y menos instrucciones que gestionen el bucle (e.g.: $i++$)

Si, por ejemplo, tenemos un bucle que itera N veces y deseamos desenrollarlo C veces, tenemos que poner un bucle no desenrollado que itere $N\%C$ veces y otro desenrollado C veces que itere las $(N-N\%C)/C$ veces restantes (la división se debe a que cada iteración ejecuta C iteraciones del bucle original) y se inicie en $1+N\%C$.

6. PLANIFICACIÓN DINÁMICA DE INSTRUCCIONES

Planificación dinámica: no queremos bloquearnos en D/L, en su lugar se hará esperar a las instrucciones con riesgos hasta que estos desaparezcan y mientras ejecutaremos las siguientes instrucciones -> inicio de ejecución en desorden

Ventana de instrucciones: en D/L no nos bloqueamos por riesgo de datos sino que planificamos la instrucción y esta se esperará en una posición de almacenamiento VL hasta que los datos fuente estén disponibles -> deberemos monitorizar el fin de ejecución de las instrucciones viejas.

Mientras, la ejecución de otras instrucciones se habrá iniciado.

Riesgos que se dan en el procesador ramificado con planificación dinámica:

- $R(i) \cap D(i+k) \neq \emptyset$: porque la latencia de productor-usuario en algunas instrucciones es mayor que 1
- $D(i) \cap R(i+k) \neq \emptyset$: la planificación dinámica añade este tipo de riesgo debido a que no se garantiza que la lectura sea en orden (las lecturas pueden retrasarse en VL)
- $R(i) \cap R(i+k) \neq \emptyset$: porque la finalización es en desorden de programa

Terminología:

- *emitir*: guardar una instrucción en VL
- *iniciar*: al leer del BR y pasar a la unidad funcional pertinente

Algoritmo Marcador: el control está centralizado y con $R(i) \cap D(i+k) \neq \emptyset$ como objetivo principal

- Cada ramificación tendrá su entrada en la ventana de instrucciones \rightarrow hay una asociación biunívoca entre entrada de la ventana y unidades funcionales

Dicha entrada estará ocupada mientras se esté usando la ramificación \rightarrow es como si las unidades funcionales no estuvieran segmentadas

- **Acciones** básicas son:

- Al *emitir* una instrucción desde D/L se la guarda en una entrada de la VL
- Una instrucción se *inicia* desde su entrada de la VL (o desde el registro de desacoplo de entrada a D/L si se ha *emitido* e *iniciado* en el mismo ciclo)

- **Gestión de los riesgos** en el orden en que son tratados:

a) en D/L y antes de la *emisión*:

1. estructurales: se bloquea la *emisión*. Además de los casos típicos, ahora también pueden darse cuando no haya ventanas libres.

2. $R(i) \cap R(i+k) \neq \emptyset$: se bloquea la *emisión*

la simbología para denotarlos será: D/L | D/L | D/L | ...

b) una vez *emitida*:

3. $R(i) \cap D(i+k) \neq \emptyset$: se bloquea el *inicio*: D/L | VL | VL | ALU | ...

4. $D(i) \cap R(i+k) \neq \emptyset$: retardamos la escritura: D/L | ... | R | R | ES

- **Hardware para control de los riesgos**

- estructurales: para la ventana de instrucciones usamos un VO, al *emitir* se marca la ventana que se ocupa y se liberará cuando se haya escrito en el BR (es decir: después de ES)

Para que no se dé este riesgo al menos se tiene que replicar cada ramificación tantas veces como su latencia de lectura-escritura, aunque si se produce algún retraso el riesgo puede reaparecer.

- de datos:

- VM: vector de marcas del BR con un 1 cuando el registró se escribirá
- VLP: vector de lecturas pendientes con el que para cada registro se indica (con n bits) si una ramificación está esperando para leerlo o no (por lo que en realidad es una matriz de bits)

Cada vez que se *emite* una instrucción:

- se pone toda la entrada de su registro destino a 0: las instrucciones que estaban esperando su resultado se referían a alguna instrucción más vieja que la actual
- se marcan sus registros fuente con un 1 en la columna de la ramificación de la instrucción
- Buses de inicio: uno por ramificación, se pone a 1 cuando *inicie* la ejecución, es decir cuando haya leído
- Buses de etiquetas: uno por ramificación, tienen el identificador del registro destino que *ya* se ha actualizado

Entonces cada tipo de riesgo de datos puede detectarse haciendo:

- $R(i) \cap R(i+k) \neq \emptyset$: indexando el VM con el registro destino y viendo si alguien lo está calculando
- $R(i) \cap D(i+k) \neq \emptyset$: al *emitir* guardamos la disponibilidad de nuestros registros fuente; monitorizamos los buses de etiquetas para saber si alguna instrucción más vieja ha finalizado el cálculo de alguno de nuestros registros fuente que no estuviese disponible; mientras no todos estén disponibles será que hay este riesgo.
- $D(i) \cap R(i+k) \neq \emptyset$: al *emitir* guardamos el estado de VLP de nuestro registro destino; actualizamos esta copia cada vez que una instrucción que esperaba para leerlo se haya *iniciado* (y por tanto haya leído, lo sabremos gracias a los buses de inicio); mientras la copia no esté toda a cero será que hay este riesgo.

Unidades funcionales virtuales: una misma entrada de la ventana tiene N entradas (UFV) donde se acumularán instrucciones, un algoritmo de arbitraje escogerá cuál es la que ocupa la ramificación (típico: la más vieja)

- Abrazo mortal: en un $D(i) \cap R(i+k) \neq \emptyset$ la etapa ES puede bloquearse y las instrucciones colisionarán, en el Marcador se soluciona añadiendo N-1 entradas antes de ES para que puedan acumularse en los retrasos.

Algoritmo de Tomasulo: el control está distribuido, realiza renombre dinámico de registros

- Después de leer del BR los operandos fuente, la instrucción se almacena en una estación de reserva
Cada ramificación tiene varias estaciones de reserva -> árbitro para seleccionar la instrucción que se *iniciará*
 - **Acciones básicas** son:
 - Al *emitir* una instrucción desde D/L se leen sus registros fuente (aunque *no* estén disponibles) y se la guarda en una estación de reserva de su ramificación
 - Una instrucción se *inicia* desde su estación de reserva (cuando no esté bloqueada por riesgos y el árbitro la seleccione)
 - En ES se hace una difusión del resultado por los buses de difusión, este dato será capturado por las estaciones de reserva y por el banco de registros (que se actualizará de forma selectiva)
 - **Riesgos que desaparecen** al usar Tomasulo:
 - $D(i) \cap R(i+k) \neq \emptyset$: desaparece debido a que lo primero que siempre se hace es leer
 - $R(i) \cap R(i+k) \neq \emptyset$: la actualización selectiva del BR hará que este riesgo no pueda darse
 - **Gestión de los riesgos** en el orden en que son tratados:
 - a) en D/L y antes de la *emisión*:
 1. estructurales: se bloquea la *emisión* si no hay estaciones de reserva libres
 - b) una vez *emitida*:
 2. $R(i) \cap D(i+k) \neq \emptyset$: se bloquea el *inicio*: D/L | ERL | ERL | ALU | ...
 - **Hardware para control de los riesgos**
 - riesgos estructurales: un VO de estaciones de reserva para cada ramificación, al *emitir* se marca la estación que se ocupa y se liberará sólo cuando haya acabado ES (no es posible hacerlo antes debido al renombre dinámico de registros)
 - de datos: se realiza un renombre dinámico usando:
 - VETiq: vector de etiquetas, cada entrada corresponde a un registro y contiene un identificador que indica la estación de reserva que actualizará ese registro
 - VM: vector de marcas del BR donde un 1 indica que el dato se está calculando y que por tanto debe capturarse del bus de difusión indicado por VETiq, un 0 indica que el BR dispone del dato
 - Buses de difusión: buses que transportan el resultado obtenido en ES hasta el BR y hasta todas las estaciones de reserva para permitir que el dato sea capturado. También transportan el identificador de la estación de reserva de la que procede el dato calculado.
No son cortocircuitos.
- Entonces cada tipo de riesgo de datos se gestiona haciendo:
- $R(i) \cap R(i+k) \neq \emptyset$: cuando se *emite* una instrucción la posición del VETiq de su registro destino se reescribe con el identificador de su estación de reserva; cuando una instrucción llegue a ES y difunda su resultado el BR comparará el identificador de la estación de reserva que esté difundiendo con el guardado en la correspondiente entrada del VETiq y *sólo* actualizará si coinciden -> sólo podrá actualizar la más joven
Nótese que aunque varias escrituras al mismo registro se realizasen en el mismo ciclo, el valor de VETiq sólo podrá coincidir con una de esas estaciones de reserva productoras
 - $R(i) \cap D(i+k) \neq \emptyset$: al *emitir* leemos los registros fuente y guardamos su valor, si según VM no estaban disponibles guardaremos qué estación de reserva los está calculando (indicado por VETiq); monitorizamos los buses de difusión para capturar los datos que no estaban disponibles; mientras no todos los datos fuente estén disponibles será que hay este riesgo
 - $D(i) \cap R(i+k) \neq \emptyset$: desaparece debido a que lo primero que siempre se hace es leer del banco de registros