Arquitectura e Ingeniería de Computadores

Parcial 1

etsinf

Curso 2024-2025

Resumen completo de todo el contenido presente para el primer parcial de la asignatura.

AIC

ÍNDICE

TEMA 1.1 – INTRODUCCIÓN A LA ARQUITECTURA DE COMPUTADORES

CONCEPTO DE ARQUITECTURA

La arquitectura define el hardware del computador, distinguiendo entre la ISA, la organización del procesador y la tecnología base.

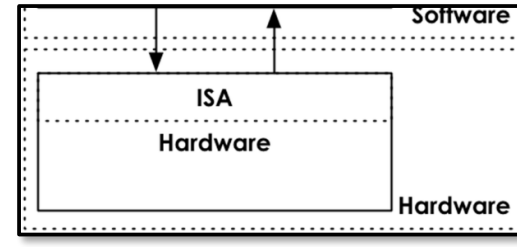
- **ISA:** define la funcionalidad del hardware, lo que es capaz de hacer. Es el juego de instrucciones.

Esta indica todo lo que puede hacer, si deseas realizar algo que no esté en dicho juego, entonces se debe utilizar software específico para ello, como por ejemplo una raíz cuadrada.

- **Organización del procesador:** descripción de la arquitectura del computador, usando los bloques que necesitemos. Describe los elementos lógicos que permiten ejecutar las instrucciones: *registros, decodificadores, operadores aritmeticológicos, etc.*

La ISA está hecha de circuitos eléctricos, pero al estar compuesto por elementos muy minúsculos se abstrae y se indica que el conjunto es una *puerta lógica,* o *memoria.*

- **Tecnología base:** es el circuito electrónico que establece la realización del dispositivo en forma de transistores, conexiones, etc. (*Implementación*)



La arquitectura, en general, comprende: *ISA, Organización y Circuito Electrónico.*

TAREA DEL INGENIERO DE COMPUTADORES

Dados unos requisitos u objetivos, se debe tener en cuenta la tecnología que tienes y los costes para conseguir lo mejor.

- **Ver requisitos:** para qué lo quieres, qué te hace falta, etc.
- **Restricciones:** definir qué debe hacer y qué no debe hacer.

Una vez establecidos, se debe seleccionar el mejor diseño. Para ello, se seleccionan las mejores alternativas, se comparan, y se selecciona al mejor.

Tenemos dos tipos de compatibilidad:

- **Código fuente:** si es compatible a nivel de ISA, se recompila para que use las instrucciones del ISA.
- **Binario:** dado un ISA definido, no se necesita software nuevo.

En el SO, te debes guiar por el espacio de direccionamiento que limita el tamaño de las aplicaciones, y el cómo se gestiona la memoria y los procesos.

TECNOLOGÍA, CONSUMO Y COSTE

El trabajo del diseñador debe tener en cuenta: la tecnología disponible, el consumo y disipación de calor y el coste.

- **Tecnología disponible:** Según la Ley de Moore, las capacidades de cada procesador (número de transistores), cada 18 meses se duplica. Esto se debe a que la tecnología evoluciona cada día, por lo que la ley de Moore se cumple.
    - Al tamaño del transistor se le puede llamar "*Feature size".*

- **Consumo y disipación del calor:** los transistores consumen, y tienen potencia estática y dinámica:
    - **Dinámica:** cuando están en saturación o cuando cambian de estado (frecuencia con la que cambian).
    - **Estática:** debida a la corriente de fuga que se escapa por estar en corte.

  Para hacer que no consumiera tanto, se ha tenido que bajar la potencia (V), sin embargo, tiene un límite en 0.7V. Esto es importante, dado que se tiene que evacuar el calor para que no se queme.

- **Coste:** principalmente, el coste se centra en el silicio, y el cómo se utiliza este para el desarrollo de los microchips. El coste crece muy rápido con el tamaño del silicio empleado, el cual depende de su finalidad.

A lo largo del tiempo, han ido apareciendo utilidades que han reducido bastante el consumo, como el uso de ISAs con menos instrucciones (RISC: *Reduced Instruction Set Compiler*), segmentación, uso de cachés, etc.

CLASES DE COMPUTADORES

Tenemos los siguientes tipos de computadores:

- **Personal Mobile Devices (PMD):** son los *smartphones, PDAs, etc.* El consumo es limitado, depende de la batería y no hay ventilación forzada.
- **Sistemas Empotrados:** incluyen electrónica diversa. El PMD es un caso particular de sistema empotrado, pero el sistema empotrado tiene mejor diseño y prestaciones.
- **Computadores personales:** tienen una potencia equilibrada de cálculo y gráficos.
- **Servidores:** computador que ofrece servicios dentro de una red. La disponibilidad es muy importante, son escalables, y deben hacer varias cosas a la vez.
- **Clústers:** es una colección de computadores, cada uno con sus sistemas y memorias, que están conectados a una red. Son muy escalables, y distribuyen bien la carga de trabajo.
    - **Clústers a gran escala:** se unen en grandes servicios de internet.
- **Supercomputador:** máquinas diseñadas para obtener unas prestaciones muy elevadas, sin importar el coste. Poca interacción con el usuario, gran productividad en aritmética de coma flotante.

TEMA 1.2 – ANÁLISIS DE PRESTACIONES

DEFINICIÓN DE PRESTACIONES

Las prestaciones tienen diferentes puntos de vista, con solo un usuario o con varios:

- **Usuario:** están relacionadas con tener el mínimo tiempo de ejecución para un programa dado. Lo que se le puede llamar tiempo de respuesta o ejecución.

$$Productividad = \frac{1}{Tiempo\ de\ ejecución}$$

- **Administrador (varios):** cierta cantidad de usuarios ejecutando programas. Se usa la productividad, que sería la cantidad de trabajos que se hacen en el tiempo.

$$Productividad = \frac{n}{Tiempo\ de\ ejecución}$$

COMPARACIONES

Cuando se comparan dos computadores, "X" e "Y", se escoge el más lento como referencia y se obtiene la aceleración "S" dividiendo el tiempo de uno entre el del otro:

$$S = \frac{T_X}{T_Y}$$

Si hay varios, se escoge uno diferente que no está en el grupo para que actúe de referencia y se dividen todos los tiempos entre el de ese computador.

Se calcula de la siguiente forma:

$$S = \frac{T_Y}{T_X} = \frac{P_X}{P_Y} = 1 + \frac{n}{100}$$

- **n:** cantidad de mejora en %, siendo: $n = (S - 1) \cdot 100$

PRINCIPIOS CUANTITATIVOS DEL DISEÑO DE COMPUTADORES

Podemos calcular las prestaciones del procesador siguiendo esta fórmula:

$$T_{ej} = \frac{Segundos}{Programa} = \frac{nº\ instr.}{Programa} \cdot \frac{Ciclos}{nº\ instr.} \cdot \frac{Seg}{Ciclos} = I \cdot CPI \cdot T$$

- **I:** Son el número de instrucciones. Depende del ISA y del compilador.
- **CPI:** Cantidad de ciclos por instrucción. Depende del ISA y de la organización.
- **T:** Cuánto tiempo dura cada ciclo. Es la inversa de la frecuencia. Depende de la tecnología y de la organización.

Son dependientes unos de otros, no es posible reducir uno sin afectar a otro.

Para reducir el tiempo de ejecución, habría que reducir alguno de los 3 factores. Sin embargo, dependen de casi las mismas cosas, por lo que es muy difícil.

LEY DE AMDAHL

Describe cómo afecta el cambio de una parte de un proceso en el total. No porque una parte sea el doble de rápida el resultado final también lo será.

- **Nuevo tiempo:** F es la fracción de tiempo que cambia, y la S la aceleración local que se le aplica, por lo que ahora el tiempo pasa de ser 1 a ser: $t' = \frac{F}{S} + 1 - F$

Para $t \neq 1$ (general), tenemos: $t' = t \cdot \frac{F}{S} + t \cdot (1 - F) = t \cdot (\frac{F}{S} + 1 - F)$

- **Aceleración global:** para sacar S', tenemos: $S' = \frac{t}{t'} = \frac{1}{F/S + 1 - F}$

Por otra parte, dado el porcentaje en el que se gasta el componente que quieres mejorar, se puede sacar el límite para saber hasta cuánto podrías mejorar el computador solo mejorando eso:

$$S'_\infty = \lim_{S \to \infty} S' = \frac{1}{1 - F}$$

La Ley de Amdahl se puede generalizar para múltiples fracciones o componentes. Los que mejores serán divididos por su factor de aceleración, mientras que los que no se mejoren permanecerán igual.

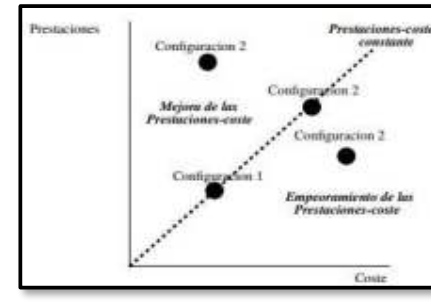
$$S' = \frac{1}{F_1/S_1 + F_2/S_2 + \cdots + F_n/S_n}$$

Luego, tenemos la aceleración local compuesta, donde una aceleración local "S_i" puede ser el resultado de la composición de varias aceleraciones locales independientes: $S_i = S_{i_1} \times S_{i_2} \times \ldots \times S_{i_m}$.

RELACIÓN PRESTACIONES COSTE

Mide la relación entre las prestaciones alcanzadas y el coste (precio, watts, etc.) de una configuración dada. Permite comparar varias alternativas:

- **Óptimo:** coste 0, y tiene x prestaciones justo en el eje Y.
- **Medio:** si vale x, y tiene y prestaciones, se mejora el doble el coste (2x), las prestaciones serán 2yy, en la línea del medio.
- **Empeora:** por debajo de la línea del medio, sube algo el coste, pero no se consigue la misma cantidad de prestaciones en proporción.
- **Mejora:** por encima de la línea del medio. Si sube algo el coste, sube mucho las prestaciones.



La Medida de Prestaciones

Para medir las prestaciones, hace falta escoger programas de prueba para poder tener una referencia, programas de los que se disponga su código fuente para poder compilarlo para cada computador. Los mejores programas para probar son los reales, en concreto los que vayas a usar con dicho ordenador. En caso de que no se elijan, entonces se debe usar unos que representen a la gran mayoría de usuarios.

Benchmarks Suites (Conjunto de Aplicaciones)

Son programas reales sin interactividad y kernels especializados para medir las prestaciones dentro de un perfil de uso. Testean la E/S, la CPU, los gráficos, servidores, etc. Estos se actualizan constantemente para que los programas incluidos en el paquete representen el trabajo que hace un usuario típico en el momento.

Las medidas han de ser reproducibles, se debe indicar y especificar todos los detalles de:

- **Hardware:** procesador, caché, memoria, disco, etc.
- **Software:** SO, programas y versiones, datos de entrada, opciones de ejecución, etc.

Comparación de Computadores

Para obtener una medida resumen de la ejecución de varios programas, podemos apreciar que una característica de buena medición de tiempos es: *el valor medio debe ser directamente proporcional al tiempo total de ejecución.*

Tiempo Total	Media aritmética	Tiempo de Ejecución
$T_t = \sum_{i=1}^{n} Tiempo_i$	$T_a = \frac{1}{n} \cdot \sum_{i=1}^{n} Tiempo_i$	$T_W = \sum_{i=1}^{n} w_i \cdot Tiempo_i$

Donde w_i representa la frecuencia del programa i en la carga de trabajo. Luego, tenemos la media geométrica (SPEC) de los tiempos de ejecución normalizados a una máquina de referencia: R veces más rápido que la referencia. Se calcula de la siguiente forma:

$$R = \sqrt[n]{\prod_{i=1}^{n} \frac{T_{ref_i}}{T_i}}$$

Otras Medidas

Como otras medidas para tener en cuenta, tenemos:

- **MIPS:** millones de instrucciones por segundo. Depende del programa que se use, así como del juego de instrucciones de cada máquina. No tiene en cuenta el tipo de instrucción.
- **MFLOPS:** millones de operaciones en coma flotante por segundo. Pueden variar entre ordenadores. Solo es útil si el programa es el mismo y hace falta que usen coma flotante.

TEMA 1.3 – DISEÑO DE LOS JUEGOS DE INSTRUCCIÓN

GENERALIDADES

Las ISA son la interfaz entre los programas y la ruta de datos. Se debe optimizar el caso frecuente, centrarse en él para mejorarlo. Sin embargo, el caso raro tampoco se debe descuidar, pero basta con garantizar un buen resultado.

Siempre que tenga sentido, las operaciones, modos de direccionamiento y tipos de datos deben ser independientes, lo que simplifica la generación de código, sobre todo si la decisión se toma en dos fases de la compilación distintas.

- **Regularidad:** se dice cuando el juego de instrucciones no sorprende al usuario.
- **Ortogonalidad:** si se tienen varios modos de direccionamiento o de tipo de datos, si combino uno con otro, funciona sin que se tenga que mirar si cuadra o no.

Se debe ofrecer siempre primitivas y no soluciones, es decir, evitar incluir soluciones concretas que den soporte directo a construcciones de alto nivel, pero solo funcionarán con un lenguaje, al ser muy específicas. Lo que hay que hacer es dar operaciones sencillas y rápidas.

- **Principio "uno o todo":** o hay una sola forma de hacer una determinada cosa, o todas las formas son posibles. El objetivo es simplificar el coste del cálculo de cada alternativa. Según la arquitectura es una u otra.

Por último, si existen elementos que se sabe con certeza que no van a variar, no es necesario almacenarlo en memoria, por lo que se puede ahorrar dicha acción.

JUEGO DE INSTRUCCIONES RISC-V: rv64imfd

Oculta los detalles de implementación, y debe verse como una interfaz software para una gran variedad de implementaciones. Se usa en una gran cantidad de sitios. Debemos conocer su conjunto básico y extensiones:

- **Conjunto básico:** conjunto mínimo, presente en cualquier procesador RISC-V, y soporte aritmético (suma/resta) de enteros y operaciones lógica.
- **Extensiones:** algunas añaden multiplicación y división de enteros (M), operaciones atómicas (A), operaciones en coma flotante (simple y doble, F y D), etc.

El RISC-V es modular, eso quiere decir que, tengo un set básico y puedo "añadir" más según dónde lo vayas a usar. Luego, le indica al compilador qué instrucciones tiene disponibles y este se adapta para ser lo más óptimo posible.

REGISTROS Y TIPOS DE OPERANDO

Tenemos dos:

- **Registros IA-32:** cada uno es para una cosa diferente. Se pueden tratar de diferentes formas: *4 registros de 16 bits u 8 registros de 8 bits.*
- **Registros RISC-V (rm64imfd):** 32 registros de 64 bit: x0...x31.

Ahora, entraremos más en detalle acerca de cada tipo de registro:

- **RISC-V:**
    - Los registros son numerosos, y tienen todos igual tamaño.
    - Cada instrucción de cálculo opera con el contenido completo de los registros: conversión entre tipos si las *load* cargan datos de tamaño inferior al tamaño del registro.
    - Algunas instrucciones operan con medio registro.
    - Se pueden hacer accesos solo a bytes, half o word enteras.

- **IA-32:**
    - Pocos registros y adaptados a los tipos de datos disponibles, donde cada instrucción aritmética tiene un código de operación para cada tipo, por ejemplo: *en x86-64 hay cuatro versiones de "add" para operandos de 8, 16, 32 y 64 bits.*
    - En esta se añaden tantas operaciones aritméticas como tipos, mientras que en el RISC hay una o dos genéricas para todos, aquí se deben de especificar y eso complica mucho las cosas.

Es más fácil cambiar la longitud de palabra en RISC-V, dado que en este solo tienes instrucciones genéricas, mientras que si se quiere cambiar en IA se debe modificar y añadir múltiples elementos.

CODIFICACIÓN DE INSTRUCCIONES

Las instrucciones se almacenan en la memoria según un formato, que indica la operación a realizar (código de operación) y los operandos. Formato fijo vs variable:

- **Fijo:** todas las instrucciones se codifican utilizando el mismo número de bits.
    - Facilita la búsqueda de instrucciones y su decodificación.
    - A veces, derrocha bits en el formato, ya que no todas las instrucciones requieren el mismo espacio.
- **Variable:** el número de bits requerido para codificar la instrucción varía según el tipio de instrucción.
    - Complica la búsqueda de instrucciones y su decodificación.
    - Optimiza espacio ocupado por las instrucciones y, por lo tanto, por los programas. Se adapta a cada instrucción sin usar memoria innecesaria.

Para instrucciones similares, es mejor el fijo, mientras que para una instrucción de gran tamaño es mejor el variable.

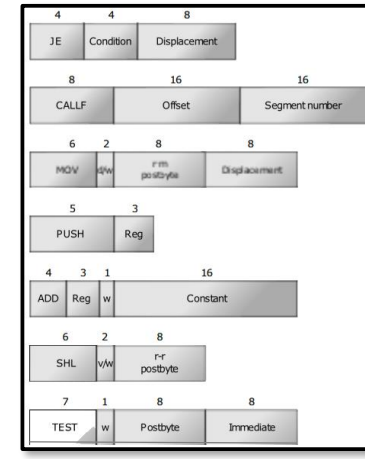
El número de bits destinado al formato impone un límite al espacio destinado a cada uno de los campos, el cual limita el número de variantes de este.

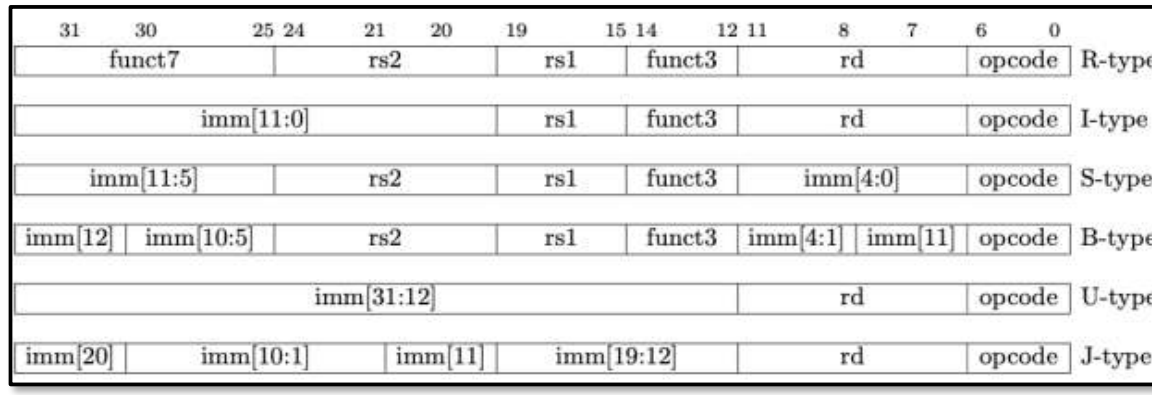
- **Formato único:** la correspondencia entre los bits del formato y los campos es siempre la misma.
    - Facilita la decodificación de la instrucción.
    - Derrocha bits en el formato, ya que no todas las instrucciones requieren todos los campos previstos.
- **Múltiples formatos:** cada formato puede tener campos distintos, y se relacionan estos y los bits del formato.
    - Permite ajustar mejor los bits ocupados por la instrucción y los campos requeridos.

ESTRATEGIAS DE CODIFICACIÓN

Distinguimos, de nuevo, entre IA-32 y RISC-V:

- **Codificación IA-32:**
  - **Formato variable:** una instrucción puede ocupar entre 1 y 17 bytes.
  - **La descodificación es secuencial:** hay que conocer el valor de los bits de un campo para descodificar el siguiente.
- **Codificación RISC-V:**
  - Formato fijo de 32 bits.
  - Pueden descodificarse varios campos en paralelo.
  - 6 tipos de formatos: R, I, S, B, U y J.
  - rd, rs1 y rs2 en la misma posición.
  - Bit de signo de inmediato en la misma posición.





DIRECCIONAMIENTO DE MEMORIA

Referido a cómo se especifican las direcciones. Podemos tener modos de direccionamiento sofisticados o más sencillos.

- **Reducción del número de instrucciones de los programas:** Baja el número de instrucciones (I).
- **Hardware más complejo:** Sube el CPI y/o el tiempo de ciclo (T).

En IA se incluye hasta el modo escalado. Combina libremente los más complejos, y una instrucción puede tener mucho trabajo. Por otra parte, en el RISC-V solo se tiene modo desplazamiento, y a partir de ahí se simula el resto.

Modo	Ejemplo	Significado
Directo a registro	add x1, x2, x3	x1←x2+x3
Inmediato	add x1, x2, 1	x1←x2+1
Directo o Absoluto	lw x1,(1000)	x1←Mem[1000]
Registro indirecto	lw x1,(x2)	x1←Mem[x2]
Desplazamiento	lw x1,100(x2)	x1←Mem[100+x2]
Indexado	lw x1,(x2+x3)	x1←Mem[x2+x3]
Indirecto a mem.	lw x1,@(x2)	x1←Mem[Mem[x2]]
Autoincremento	lw x1,(x2)+	x1←Mem[x2] x2←x2+*d*
Autodecremento	lw x1,–(x2)	x2←x2–*d* x1←Mem[x2]
Escalado	lw x1,100(x2)(x3)	x1←Mem[100+x2+x3**d*]

- **Rango de valores inmediatos:** solución entre disponer de constantes grandes y el número de bits ocupados en el formato. 12 bits típico en RISC-V, pero hay instrucciones para facilitar el trabajo con valores inmediatos más grandes. No se está limitando a 12 bits. Puede operar y conseguir en un registro inmediatos más grandes:

```
lui x1, Uvalor          :Regs[x1] ← Uvalor << 12 (Parte alta)
ori x1, x1, Lvalor      :Regs[x1] ← Regs[x1] OR Lvalor (Parte baja)
```

- **Acceso a memoria con inmediatos:** igual que antes, se está limitando a 12 bits, si se quieren más, entonces se tendrá que operar y pasar por un registro usando otras instrucciones, después ya se puede usar esta dirección o inmediato.

CONTROL DE FLUJO

Tenemos tres tipos de saltos: *saltos condicionales (Branch), saltos incondicionales (jump) y llamadas/retorno a/de procedimiento (jal, jalr).*

Tenemos también diversos modos de indicarle al programa el direccionamiento en el salto:

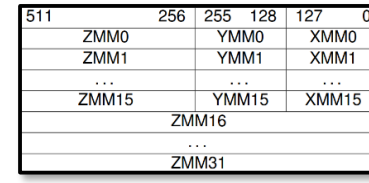
- **Relativo al PC:** el destino suele estar cerca de la instrucción actual. Estas direcciones relativas consumen pocos bits. Los bits para el salto son 13 para condicionales, y 21 en incondicionales.
- **Indirecto a registro:** útil si el destino del salto es desconocido durante la compilación. Sentencias que seleccionan una de entre varias alternativas. Métodos virtuales en lenguajes orientados a objetos, paso de funciones como parámetros a otra función, o librerías enlazadas dinámicamente.

INSTRUCCIONES SIMD

Las instrucciones SIMD (Single Instruction – Multiple Data) operan sobre registros que contienen varios datos empaquetados.

- **Tipos de datos utilizables:** enteros de 8, 16, 32, etc. bits, y CF en simple y doble precisión.
- El número de datos contenido en un registro es: $n = \frac{longitud\ registro}{longitud\ tipo\ de\ datos}$
- Los registros tienen longitud fija, pero el juego de instrucciones permite empaquetar datos de diferente longitud.

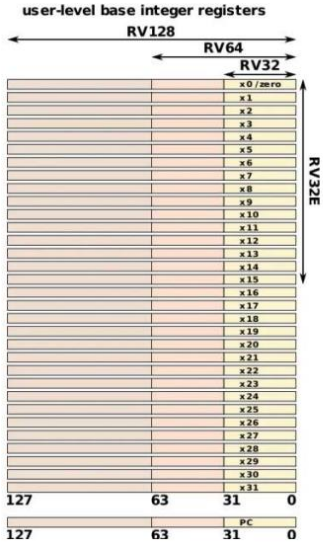
El esquema de los registros AVX-512 (ZMM) como extensión de los registros AVX (YMM) y los SSE (XMM) sería el de la derecha:



La compilación se basa en los tipos de datos y en las expresiones para generar código. Si el programador expresa el código de forma escalar, un compilador clásico no utilizará instrucciones SIMD. Tenemos tres maneras de insertar instrucciones SIMD en el código:

- **Vectorización automática:** el programador no explicita el paralelismo. Un compilador avanzado extrae el paralelismo del código fuente escalar.
- **Usar tipos de datos SIMD:** el programador define las variables de interés con un tipo específico, y las utiliza en expresiones comunes del lenguaje.
- **Intrinsic functions:** el programador utiliza una biblioteca de funciones SIMD.

REGISTROS



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

MIPS	RISC-V
Registros “r0 – r31”	Registros “x0 – x31”
Float “f0 – f31”	Float “f0 – f31”
r0 = 0	x0 = 0
r31 = *return address*	x1 = *return address*

INSTRUCCIONES

Tipo	pseudo-instrucción	Equivalencia
salto	j offset	jalr x0, rs1, offset
retorno de subrutina	ret	jr ra
no operación	nop	addi x0, x0, 0
carga de inmediato	li rd, imm	muchas secuencias
copiar registro	mv rd, rs	addi rd, rs, 0
llamada subrutina	call offset	auipc x1, offset[31:12] jalr x1, x1, offset[11:0]

Tipo	Instrucción	Ejemplo	Descripción
mult.	mul	mul rd, rs1, rs2	$rd = (rs1 \times rs2)_{63..0}$
	mulh	mulh rd, rs1, rs2	$rd = (rs1 \times rs2)_{127..64}$
	mulhsu	mulhsu rd, rs1, rs2	$rd = (rs1 \times arg\_sin\_signo(rs2))_{127..64}$
	mulhu	mulhu rd, rs1, rs2	$rd = (arg\_sin\_signo(rs1) \times arg\_sin\_signo(rs2))_{127..64}$
	mulw	mulw rd, rs1, rs2	$rd = ext\_signo((rs1_{31..0} \times rs2_{31..0})_{31..0})$
división	div	div rd, rs1, rs2	$rd = rs1 \div rs2$
	divu	divu rd, rs1, rs2	$rd = arg\_sin\_signo(rs1) \div arg\_sin\_signo(rs2)$
	divw	divw rd, rs1, rs2	$rd = ext\_signo((rs1_{31..0} \div rs2_{31..0})_{31..0})$
	divuw	divuw rd, rs1, rs2	$rd = ext\_signo((arg\_sin\_signo(rs1_{31..0}) \div arg\_sin\_signo(rs2_{31..0}))_{31..0})$
resto	rem	rem rd, rs1, rs2	$rd = rs1 \,\%\, rs2$
	remu	remu rd, rs1, rs2	$rd = arg\_sin\_signo(rs1) \,\%\, arg\_sin\_signo(rs2)$
	remw	remw rd, rs1, rs2	$rd = ext\_signo((rs1_{31..0} \,\%\, rs2_{31..0})_{31..0})$
	remuw	remuw rd, rs1, rs2	$rd = ext\_signo((arg\_sin\_signo(rs1_{31..0}) \,\%\, arg\_sin\_signo(rs2_{31..0}))_{31..0})$

Tipo	Instrucción	Ejemplo	Descripción
suma	add	add rd, rs1, rs2	$rd = rs1 + rs2$
	addw	addw rd, rs1, rs2	$rd = ext\_signo(rs1_{31..0} + rs2_{31..0})$
	addi	addi rd, rd1, imm	$rd = rs1 + ext\_signo(imm_{11..0})$
	addiw	addiw rd, rs1, imm	$rd = ext\_signo(rs1_{31..0} + ext\_signo(imm_{11..0}))$
resta	sub	sub rd, rs1, rs2	$rd = rs1 - rs2$
	subw	subw rd, rs1, rs2	$rd = ext\_signo(rs1_{31..0} - rs2_{31..0})$
lógicas (bit-wise)	xor	xor rd, rs1, rs2	$rd = rs1\ xor\ rs2$
	xori	xori rd, rs1, imm	$rd = rs1\ xor\ ext\_signo(imm_{11..0})$
	or	or rd, rs1, rs2	$rd = rs1\ or\ rs2$
	ori	ori rd, rs1, imm	$rd = rs1\ or\ ext\_signo(imm_{11..0})$
	and	and rd, rs1, rs2	$rd = rs1\ and\ rs2$
	andi	andi rd, rs1, imm	$rd = rs1\ and\ ext\_signo(imm_{11..0})$
salto	jal	jal rd, imm	$rd = PC + 4;\ PC = PC + ext\_signo(imm_{20..1} << 1)$
	jalr	jalr rd, rs1, imm	$rd = PC + 4;\ PC = rs1 + ext\_signo(imm_{11..0})$
salto cond.	beq	beq rs1, rs2, imm	$si\ (rs1 == rs2)\ PC = PC + ext\_signo(imm_{12..1} << 1)$
	bne	bne rs1, rs2, imm	$si\ (rs1 <> rs2)\ PC = PC + ext\_signo(imm_{12..1} << 1)$
	blt	blt rs1, rs2, imm	$si\ (rs1 < rs2)\ PC = PC + ext\_signo(imm_{12..1} << 1)$
	bge	bge rs1, rs2, imm	$si\ (rs1 >= rs2)\ PC = PC + ext\_signo(imm_{12..1} << 1)$
	bltu	bltu rs1, rs2, imm	$si\ (rs1 < rs2)\ PC = PC + ext\_signo(imm_{12..1} << 1)$ (*)
	bgeu	bgeu rs1, rs2, imm	$si\ (rs1 >= rs2)\ PC = PC + ext\_signo(imm_{12..1} << 1)$ (*)
comp.	slt	slt rd, rs1, rs2	$si\ (rs1 < rs2)\ rd = 1\ caso\ contrario\ rd = 0$
	slti	slti rd, rs1, imm	$si\ (rs1 < ext\_signo(imm_{11..0}))\ rd = 1\ caso\ contrario\ rd = 0$
	sltu	sltu rd, rs1, rs2	$si\ (rs1 < rs2)\ rd = 1\ caso\ contrario\ rd = 0$ (*)
	sltiu	stliu rd, rs1, imm	$si\ (rs1 < ext\_signo(imm_{11..0}))\ rd = 1\ caso\ contrario\ rd = 0$ (*)

Tipo	Instrucción	Ejemplo	Descripción
carga/alm. byte	lb	lb rd, imm(rs1)	$rd = ext\_signo(Memoria[rs1 + ext\_signo(imm_{11..0})][7:0])$
	lbu	lbu rd, imm(rs1)	$rd = ext\_ceros(Mem[rs1 + ext\_signo(imm_{11..0})][7:0])$
	sb	sb rs2, imm(rs1)	$Mem[rs1 + ext\_signo(imm_{11..0})][7:0] = rs2_{7..0}$
carga/alm. media palabra	lh	lh rd, imm(rs1)	$rd = ext\_signo(Mem[rs1 + ext\_signo(imm_{11..0})][15:0])$
	lhu	lhu rd, imm(rs1)	$rd = ext\_ceros(Mem[rs1 + ext\_signo(imm_{11..0})][15:0])$
	sh	sh rs2, imm(rs1)	$Mem[rs1 + ext\_signo(imm_{11..0})][15:0] = rs2_{15..0}$
carga/alm. palabra	lw	lw rd, imm(rs1)	$rd = ext\_signo(Mem[rs1 + ext\_signo(imm_{11..0})][31:0])$
	lwu	lwu rd, imm(rs1)	$rd = ext\_ceros(Mem[rs1 + ext\_signo(imm_{11..0})][31:0])$
	sw	sw rs2. imm(rs1)	$Mem[rs1 + ext\_signo(imm_{11..0})][31:0] = rs2_{31..0}$
carga/alm. doble palabra	ld	ld rd, imm(rs1)	$rd = Mem[rs1 + ext\_signo(imm_{11..0})][63:0]$
	sd	sd rs2, imm(rs1)	$Mem[rs1 + ext\_signo(imm_{11..0})][63:0] = rs2$
carga registros	lui	lui rd, imm	$rd = imm_{19..0} << 12$
	auipc	auipc rd, imm	$rd = PC + (imm_{19..0} << 12)$
desplazamiento lógico izquierda	sll	sll rd, rs1, rs2	$rd = rs1 << rs2_{5..0}$
	slli	slli rd, rs1, imm	$rd = rs1 << imm_{5..0}$
	sllw	sllw rd, rs1, rs2	$rd_{31..0} = rs1_{31..0} << rs2_{4..0}$
	slliw	slliw rd, rs1, imm	$rd_{31..0} = rs1_{31..0} << imm_{4..0}$
desplazamiento lógico derecha	srl	srl rd, rs1, rs2	$rd = rs1 >> rs2_{5..0}$ (se insertan 0s)
	srli	srli rd, rs1, imm	$rd = rs1 >> imm_{5..0}$ (se insertan 0s)
	srlw	srlw rd, rs1, rs2	$rd_{31..0} = rs1_{31..0} >> rs2_{4..0}$ (se insertan 0s)
	srliw	srliw rd, rs1, imm	$rd_{31..0} = rs1_{31..0} >> imm_{4..0}$ (se insertan 0s)
desplazamiento aritmético derecha	sra	sra rd, rs1, rs2	$rd = rs1 >> rs2_{5..0}$ (se extiende el bit de signo)
	srai	srai rd, rs1, imm	$rd = rs1 >> imm_{5..0}$ (se extiende el bit de signo)
	sraw	sraw rd, rs1, rs2	$rd_{31..0} = rs1_{31..0} >> rs2_{4..0}$ (se extiende el bit de signo)
	sraiw	sraiw rd, rs1, imm	$rd_{31..0} = rs1_{31..0} >> imm_{4..0}$ (se extiende el bit de signo)
otras	fence	fence	sincroniza accesos a memoria de varios núcleos
	ecall	ecall	petición de servicio al entorno de ejecución
	ebreak	ebreak	devuelve el control a un entorno de depuración

Tipo	rv64f	rv64d	Ejemplo (rv64f)	Descripción
carga/alm.	flw	fld	flw fd, imm(rs1)	$fd = Mem[rs1 + ext\_signo(imm_{11..0})]$
	fsw	fsd	fsw fd, imm(rs1)	$Mem[rs1 + ext\_signo(imm_{11..0})] = fd$
copia registros	fmv.x.w	fmv.x.d	fmv.x.w rd, fs1	rv64f: $rd_{31..0} = fs1$; rv64d: $rd = fs1$
	fmv.w.x	fmv.d.x	fmv.w.x fd, rs1	rv64f: $fd = rs1_{31..0}$; rv64d: $fd = rs1$
aritméticas	fadd.s	fadd.d	fadd.s fd, fs1, fs2	$fd = fs1 + fs2$
	fsub.s	fsub.d	fsub.s fd, fs1, fs2	$fd = fs1 - fs2$
	fmul.s	fmul.d	fmul.s fd, fs1, fs2	$fd = fs1 \times fs2$
	fdiv.s	fdiv.d	fdiv.s fd, fs1, fs2	$fd = fs1 \div fs2$
	fsqrt.s	fsqrt.d	fsqrt.s fd, fs1	$fd = \sqrt{fs1}$
	fmadd.s	fmadd.d	fmadd.s fd, fs1, fs2, fs3	$fd = (fs1 \times fs2) + fs3$
	fmsub.s	fmsub.d	fmsub.s fd, fs1, fs2, fs3	$fd = (fs1 \times fs2) - fs3$
	fnmadd.s	fnmadd.d	fnmadd.s fd, fs1, fs2, fs3	$fd = -(fs1 \times fs2) - fs3$
	fnmsub.s	fnmsub.d	fnmsub.s fd, fs1, fs2, fs3	$fd = -(fs1 \times fs2) + fs3$
signo	fsgnj.s	fsgnj.d	fsnj.s fd, fs1, fs2	$fd = fs1, fd_{signo} = fs2_{signo}$
	fsgnjn.s	fsgnjn.d	fsnjn.s fd, fs1, fs2	$fd = fs1, fd_{signo} = fs2_{signo}\ xor\ 1$
	fsgnjx.s	fsgnjx.d	fsgnjx.s fd, fs1, fs2	$fd = fs1, fd_{signo} = fs2_{signo}\ xor\ fs1_{signo}$
comparación	feq.s	feq.d	feq.s fd, fs1, fs2	$si(fs1 == fs2)\ fd = 1 caso\ contrario\ fd = 0$
	flt.s	flt.d	flt.s fd, fs1, fs2	$si(fs1 < fs2)\ fd = 1 caso\ contrario\ fd = 0$
	fle.s	fle.d	fle.s fd, fs1, fs2	$si(fs1 <= fs2)\ fd = 1 caso\ contrario\ fd = 0$
	fmin.s	fmin.d	fmin.s fd, fs1, fs2	$fd = min(fs1, fs2)$
	fmax.s	fmax.d	fmax.s fd, fs1, fs2	$fd = max(fs1, fs2)$
conversión float a entero	fcvt.w.s	fcvt.w.d	fcvt.w.s rd, fs1	$rd_{31..0} = flotante\_a\_entero(fs1)$
	fcvt.l.s	fcvt.l.d	fcvt.l.s rd, fs1	$rd = flotante\_a\_entero(rs1)$
	fcvt.wu.s	fcvt.wu.d	fcvtwu.s rd, fs1	$rd_{31..0} = arg\_sin\_signo(flotante\_a\_entero(fs1)$
	fcvt.lu.s	fcvt.lu.d	fcvtlu.s rd, fs1	$rd = arg\_sin\_signo(flotante\_a\_entero(fs1))$
conversión entero a float	fcvt.s.w	fcvt.d.w	fcvt.s.w fd, rs1	$fd = entero\_a\_flotante(rs1_{31..0})$
	fcvt.s.wu	fcvt.d.wu	fcvt.s.wu fd, rs1	$fd = arg\_sin\_signo(entero\_a\_flotante(rs1_{31..0}))$
	fcvt.s.l	fcvt.d.l	fcvt.s.l fd, rs1	$fd = entero\_a\_flotante(rs1_{63..0})$
	fcvt.s.lu	fcvt.d.lu	fcvt.s.lu fd, rs1	$fd = arg\_sin\_signo(entero\_a\_flotante(rs1_{63..0}))$
conversión float - double		fcvt.s.d	fcvt.s.d fd, fs1	$fd = double\_a\_simple\_precision(fs1)$
		fcvt.d.s	fcvt.d.s fd, fs1	$fd = simple\_a\_doble\_precision(fs1)$
clase	fclass.s	fclass.d	fclass.s rd, fs1	$rd = clase(fs1)$: $-\infty$, -0, $+0$, $-\infty$, . . .

FORMULARIO COMPLETO – TEMA 1

A continuación, se detallarán las fórmulas utilizadas a lo largo del tema 1.

PRODUCTIVIDAD

$$P = \frac{Número\ de\ trabajos}{Tiempo\ de\ Ejecución}$$

COMPARACIÓN S ENTRE DOS COMPUTADORES X E Y (Y MÁS LENTO)

$$S = \frac{T_Y}{T_X} = \frac{P_X}{P_Y} = 1 + \frac{n}{100}$$

- X es S veces más rápido que Y.
- X es n% veces más rápido que Y.

TIEMPO DE EJECUCIÓN DEL PROCESADOR

$$T_{ejecución} = \frac{seg}{programa} = \frac{nº\ instr.}{programa} \times \frac{ciclos}{nº\ instr.} \times \frac{seg}{ciclo} = I \cdot CPI \cdot T$$

- I es el número de instrucciones
- CPI es el número de ciclos por instrucción.
- T es el periodo en segundos (inversa de la frecuencia)

LEY DE AMDAHL

TIEMPO DE PROCESO RESULTANTE PARA VALORES $t \neq 1$:

$$t' = t \cdot (1 - F) + t \cdot \frac{F}{S}$$

ACELERACIÓN GLOBAL

$$S' = \frac{t}{t'} = \frac{1}{(1 - F) + \frac{F}{S}}$$

ACELERACIÓN MÁXIMA QUE SE PUEDE ALCANZAR

$$S'_\infty = \lim_{S \to \infty} S' = \frac{1}{1 - F}$$

GENERALIZACIÓN PARA N FRACCIONES

$$S' = \frac{1}{\frac{F_1}{S_1} + \frac{F_2}{S_2} + \cdots + \frac{F_n}{S_n}}$$

ACELERACIÓN LOCAL

$$S_i = S_{i,1} \times S_{i,2} \times \ldots \times S_{i,m_i}$$

Comparación de Computadores

Tiempo Total de Ejecución

$$T_T = \sum_{i=1}^{n} Tiempo_i$$

Media Aritmética

$$T_A = \frac{1}{n} \cdot \sum_{i=1}^{n} Tiempo_i$$

Tiempo de Ejecución Ponderado

$$T_W = \sum_{i=1}^{n} w_i \cdot Tiempo_i$$

SPEC (R veces más Rápido que la Referencia)

$$R = \sqrt[n]{\prod_{i=1}^{n} \frac{Tiempo\ Referencia_i}{Tiempo_i}}$$

Otras Medidas de Prestaciones

$$MIPS = \frac{nº\ instrucción}{T_{ej} \cdot 10^6} = \frac{I}{I \cdot CPI \cdot T \cdot 10^6} = \frac{1}{CPI \cdot T \cdot 10^6} = \frac{f}{CPI \cdot 10^6}$$

$$MFLOPS = \frac{nº\ operaciones\ en\ coma\ flotante\ del\ programa}{T_{ej} \cdot 10^6}$$

Anotaciones

TEMA 2.1 – CONCEPTO DE SEGMENTACIÓN

SEGMENTACIÓN

Se basa en descomponer una tarea en varios subprocesos independientes entre ellos, de tal manera que puedes hacer varios de forma simultánea. Esto funciona si la tarea a realizar tiene varias fases, y se realiza de forma que separas cada fase y se divide la carga de trabajo entre las diferentes fases. Este proceso reduce el CPI, idealmente, a 1.

Con esto, debemos conocer el concepto de *periodo de reloj*, el cual nos indica el tiempo mínimo de reloj que hace falta para que se complete la tarea más larga (caso real).

- **Caso ideal:** mismo retardo en todos los módulos, siendo D el retardo del circuito original, y k el número de etapas.

$$\tau = \frac{D}{k}$$

- **Caso real:** módulos con retardos distintos, registros inter-etapa y desfase de reloj, siendo τ_i el retardo del módulo i, T_r el retardo del registro inter-etapa, y T_s el desfase del reloj entre etapas.

$$\tau = \max_{i=i\to k}(\tau_i) + T_r + T_s \geq \frac{D}{k}$$

A su vez, debemos conocer también el concepto de *aceleración*, que nos indica la aceleración percibida por segmentar en etapas. Se calcula de la siguiente manera:

$$S = \frac{T_{ns}}{T_s} \approx \frac{nD}{n\tau} = \frac{D}{\tau}$$

Siendo T_{ns} el tiempo para procesar n datos en unidad original, y T_s el tiempo en la unidad segmentada. Tenemos los siguientes casos:

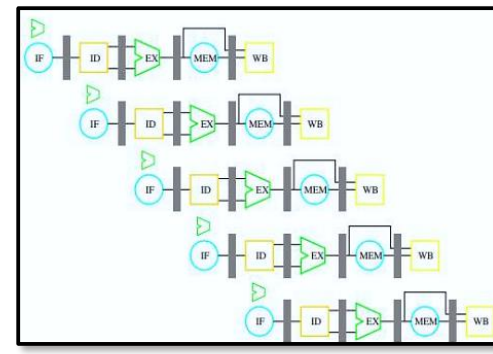
- **Caso ideal:** $\tau = D \cdot k \quad \rightarrow \quad S = \frac{D}{\tau} = k$
- **Caso real:** $\tau \geq D \cdot k \quad \rightarrow \quad S = \frac{D}{\tau} \leq k$

Por último, resta por conocer el concepto de productividad, el cual indica la cantidad de operaciones por unidad de tiempo. Tenemos:

- **Unidad no segmentada:** un resultado cada D segundos.
- **Unidad segmentada:** un resultado cada τ segundos = un resultado/ciclo de reloj. CPI = 1 (idealmente).

ºREQUISITOS HARDWARE PARA LA SEGMENTACIÓN

Para entenderlo mejor, nos apoyaremos en el siguiente esquema, en el que están expuestos con el mismo color aquellos segmentos que acceden al mismo recurso:



- **Azul:** IF y MEM. Podrían dar conflicto si estuvieran unificadas en la caché de datos y memoria.
- **Verde:** operaciones aritméticas. Se utilizan dos ALU diferentes.
- **Naranja y Amarillo:** ID lee, mientras que WB escribe en los registros. En la primera mitad del ciclo se escribe, mientras que en la primera mitad se lee.

RIESGOS DE LA SEGMENTACIÓN

Podemos obtener resultados diferentes de un mismo proceso al ejecutarlo de forma segmentada y no segmentada. Es por ello por lo que tenemos los siguientes tipos de riesgos:

- **Datos:** el resultado de una instrucción se utiliza como dato en la(s) instrucción(es) siguiente(s).
- **Control:** en el flujo de instrucciones aparece una instrucción de salto.
- **Estructurales:** dos o más instrucciones pretenden utilizar el mismo recurso.

Como posibles soluciones generales, tenemos lo siguiente:

SOLUCIÓN 1 – CICLOS DE PARADA

Podemos insertar ciclos de parada, para impedir el avance de la instrucción que origina el conflicto y de todas las que le siguen. Durante estos ciclos, no se buscan nuevas instrucciones, por lo que hay una pérdida de prestaciones:

$$CPIs = CPIs_{ideal} + Ciclos\ de\ \frac{Parada}{Instrucción} = 1 + \frac{stalls}{instructions} > 1$$

SOLUCIÓN 2 – MODIFICACIÓN DE LA RUTA DE DATOS

A su vez, también podemos modificar la ruta de datos para detectarlos y resolverlos dinámicamente. La modificación puede reducir el número de ciclos de parada necesarios para resolverlo, o resolver el riesgo completamente, con el problema de que a veces no es posible.

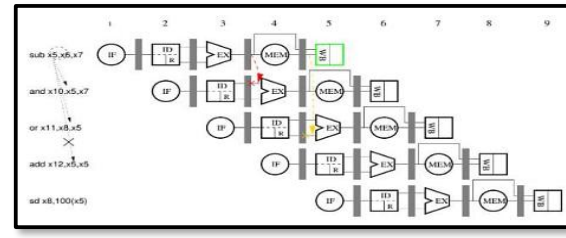
SOLUCIÓN 3 – TRABAJO POR EL COMPILADOR MEDIANTE INSTRUCCIONES "NOP"

Por último, se puede modificar la arquitectura del juego de instrucciones, con la finalidad de impedir que aparezca el problema, prohibiendo la generación de ciertas secuencias de instrucciones.

Para evitarlas, el compilador puede insertar instrucciones NOP, o reordenar instrucciones independientes. El problema con esto es que se pierde la compatibilidad a nivel binario.

RIESGOS DE DATOS

Profundizando más en la prevención de los riesgos ocasionados por los datos, podemos observar en primer lugar un ejemplo que muestre como puede llegar a ocasionarse una inconsistencia derivada de estos.



La primera instrucción escribe en un registro, pero lo hace en el ciclo 5. Mientras tanto, las instrucciones 2 y 3 leen este mismo registro en los ciclos 3 y 4, antes de que la primera instrucción pueda llegar a escribir en él; por lo que se provoca una *inconsistencia.*

Para prevenirlo, tenemos las siguientes soluciones:

SOLUCIÓN 1 – INSERCIÓN DE CICLOS DE PARADA *(STALLS)*

Se inserta 1 o 2 ciclos de parada para que avance la instrucción hasta que la instrucción escriba en el registro, y quede listo para su posterior lectura.

- **Control:** se deben de poner las señales de control antes de la etapa EX, para que retengan el flujo como si de una instrucción NOP se tratara. Lo que se logra con ello es conservar las instrucciones en IF e ID.
    - **Señales:** mediante las señales ID.stall e ID.nop se detiene el paso de las instrucciones, y se introducen las NOP. Posteriormente, el IF.stall evita que el PC siga avanzando.
- **CPI:** el CPI aumenta, dado que hay más ciclos para el mismo número de instrucciones.

SOLUCIÓN 2 – CORTOCIRCUITOS

En la práctica, el resultado de las operaciones es conocido al finalizar el procesado de la ALU (fin ciclo 3), y la instrucción que lea dicho resultado, lo usará antes de aprovechar la ALU (inicio ciclo 4).

Es por ello por lo que se puede realizar un cortocircuito para pasar dicho dato de MEM a EX. Sin embargo, este proceso solo es recomendado en caso de instrucciones consecutivas, dado que en caso de haber otra intercalada entre ambas, el dato que se desea escribir ya estará en WB, por lo que se deberá realizar dicho cortocircuito entre MEM y WB.

En este último caso, en vez de recibir el resultado en el ciclo 3, se recibirá en el cuarto, por lo que se debe insertar un ciclo extra para esperar a conocer el dato. Para ello se realiza una combinación de un ciclo de parada y el cortocircuito explicado anteriormente, de forma que se espera a que el dato esté almacenado, para luego pasarlo mediante un cortocircuito.

SOLUCIÓN 3 – REORGANIZACIÓN DEL CÓDIGO

El compilador tiene una fase más después de compilar y hacer todo el trabajo, la cual es la reorganización del código. Con ella, se pretende evitar que haya ciclos de parada.

Código convencional	Código reorganizado
ld x5,b(gp)	ld x5,b(gp)
ld x6,c(gp)	ld x6,c(gp)
add x7,x5,x6	ld x8,e(gp)
ld x8,e(gp)	add x7,x5,x6
ld x9,f(gp)	ld x9,f(gp)
sub x10,x8,x9	sd x7,a(gp)
sd x7,a(gp)	sub x10,x8,x9
sd x10,d(gp)	sd x10,d(gp)
8 ins + 2 stalls = 10 ciclos	8 ins = 8 ciclos

RIESGOS DE CONTROL

Ahora, toca profundizar en los riesgos de control, en los cuales aparecen instrucciones que modifican el contador de programa (PC), pero dicha modificación se realiza en etapas posteriores, inutilizando así las instrucciones que ya habían empezado a ejecutarse.

- **Latencia de salto:** número de ciclos transcurridos entre la etapa de búsqueda (IF) y la que modifica el PC. Si se hace en MEM es 3, en EX 2 y en ID 1.

Se puede solucionar evitando que entren más instrucciones cuando hay un salto, en concreto tres. Sin embargo, esto ralentizará bastante el proceso.

Es por ello por lo que podemos utilizar la estrategia de **predict-not taken**, en la cual suponemos el salto no efectivo de tal forma que las instrucciones buscadas a continuación de la instrucción de salto son válidas.

Si finalmente el salto es efectivo, se abortan las otras instrucciones en curso. Estas instrucciones no deben de haber modificado el estado.

SOLUCIÓN – REDUCCIÓN DE LATENCIA DE SALTO

En el contexto de la reducción de latencia en saltos, se plantea la posibilidad de optimizar el tiempo necesario para actualizar el contador de programa (PC) y evaluar la condición de salto. Originalmente, esta actualización y cálculo se realizan en la fase MEM, lo que toma 3 ciclos. Sin embargo, se sugiere reducir este tiempo a 2 ciclos realizando dichas operaciones en la fase EX, es decir, adelantando el cálculo.

Una solución aún más eficiente consiste en trasladar estas operaciones a la fase ID, donde tanto el comparador como el sumador se ejecutan en paralelo durante esta fase. Al hacerlo, la latencia se reduce a un ciclo, ya que las operaciones involucradas no requieren un tiempo significativo dentro de un ciclo.

Este adelanto implica que los cortocircuitos ya no necesitarán los datos en la fase EX, sino en la fase ID, lo que requiere un ajuste en la estructura del pipeline para asegurar que los datos estén disponibles en el momento adecuado.

Para medir el tiempo de ejecución de un bucle, se toma como referencia el número de ciclo en el que se ejecuta el primer IF de la primera instrucción del bucle. Luego, se mide el número de ciclo en el que se ejecuta el primer IF de la primera instrucción en la siguiente iteración del mismo bucle. La fórmula para calcular el tiempo de ejecución del bucle es:

$$T_{ej} = nº\ ciclos\ primer\ IF\ en\ segunda\ iteración - nº\ ciclos\ primer\ IF\ en\ primera\ iteración - 1$$

Riesgos Estructurales

Por último, resta profundizar en los riesgos estructurales, en los que el hardware no permite todas las combinaciones posibles de las instrucciones presentes en la unidad si cierto recurso no ha sido replicado suficientemente.

Para solventarlo, se presentan las siguientes soluciones

Solución 1 – Modificación de la Ruta de Datos

Consiste en replicar el recurso para que sea posible esa combinación. En caso de que sea en la caché, se deberá utilizar una arquitectura Harvard, en la que la caché se encuentra separada en instrucciones y datos. Sin embargo, esto aumenta el coste y no siempre es posible o tiene sentido.

Solución 2 – Inserción de Ciclos de Parada

Retrasar una de las operaciones que causa el conflicto con stalls. Esto causa una pérdida de prestaciones.

- **Lógica de control:** cuando una instrucción de carga/almacenamiento (Load/Store) está en la etapa MEM del pipeline, no se puede acceder simultáneamente a la memoria de instrucciones. Para evitar conflictos, se conserva la instrucción en la etapa IF y, al mismo tiempo, se entrega una instrucción NOP a la etapa ID.

Este mecanismo introduce un retraso de un ciclo de reloj en el pipeline, bloqueando temporalmente el acceso a la memoria de instrucciones mientras se procesa la carga o almacenamiento. Como consecuencia, la etapa ID recibe un NOP, lo que detiene temporalmente el flujo de instrucciones sin afectar el resto del pipeline, permitiendo que la operación de memoria se complete sin interferencias.

Excepciones

Tienen diferentes nombres: *interrupción, excepción, falta, etc.* En definitiva, es un evento que se tiene que atender. Hay diferentes tipos:

- **Síncrona y asíncrona:** es síncrona si el evento ocurre en el mismo lugar cada vez que el programa se ejecuta, y asíncrona en caso contrario.
- **Solicitada por el usuario o lanzada hacia el usuario:** cuando es el usuario el que busca la excepción, entonces es solicitada; cuando no lo hace, entonces es lanzada.
- **Enmascarables por el usuario y no enmascarables:** enmascarables son, por ejemplo, la E/S, y no enmascarables son aquellas que se deben atender obligatoriamente.
- **Durante una instrucción o entre instrucciones:** cuando la excepción se encuentra en el proceso de una instrucción o después del final de una y antes del principio de otra.
- **Continuar o terminar el programa:** cuando es más fácil terminar el programa que seguir con su ejecución.

Cuando la unidad está segmentada, en caso de producirse una excepción, habrá seguramente varias instrucciones en ejecución posteriores a la que origina la excepción. Entonces, se deja ejecutar la excepción hasta WB y comienza una nueva secuencia.

Excepciones Precisas

Se dice que un computador soporta un comportamiento preciso frente a las excepciones si:

- Las instrucciones anteriores a la que origina la excepción terminan correctamente.
- La instrucción que origina la excepción y todas las posteriores son abortadas.
- Tras completar la rutina de servicio se puede relanzar el programa, comenzando por la instrucción fallida.

Para asegurar que las excepciones se manejan en el orden natural de ejecución, se permite que una instrucción que ha causado una excepción avance hasta la etapa WB, sin que realice modificaciones en memoria ni otros efectos secundarios. Esto evita la complejidad de decidir qué instrucciones previas deben ser canceladas.

1. En la etapa WB, se verifica si la instrucción ha causado una excepción en alguna de las etapas anteriores.
    a. Si es así, las instrucciones posteriores se convierten en NOP y se actualiza el PC con la dirección de la rutina de servicio de la excepción.
2. Se guarda la dirección de la instrucción que originó la excepción para su manejo posterior.
3. La rutina de servicio de la excepción toma el control del flujo de ejecución.
4. Una vez que la rutina de servicio finaliza, el PC se restaura con la dirección guardada, y la ejecución continúa desde ese punto, garantizando la coherencia en el flujo del programa.

TEMA 2.2 – UNIDADES MULTICICLO Y GESTIÓN ESTÁTICA DE INSTRUCCIONES

OPERACIONES MULTICICLO

Existen dos tipos principales de instrucciones que requieren más tiempo en la fase EX, las instrucciones enteras que realizan operaciones complejas (como *mul, mulw, div, divw*), y las instrucciones de coma flotante (como *fadd.s, fmul.d, fmul.s, etc.*). Estas operaciones demandan más tiempo en la fase EX debido a su complejidad.

Existen dos soluciones posibles para abordar este problema:

- **Abordar el período de reloj de toda la máquina:** esta opción ralentiza todo el sistema de manera global, lo cual es ineficiente y afecta negativamente al rendimiento general, por lo que no es recomendable.
- **Variar la duración de la fase EX:** se permite que la fase EX de estas instrucciones complejas se extienda durante varios ciclos de reloj, manteniendo el período de reloj sin cambios. Esto se conoce como operaciones multiciclo.

Para su implementación, seguimos tenemos las siguientes opciones:

- Una opción es realizar las operaciones adicionales en la misma etapa EX de manera secuencial.
- Una implementación más eficiente sería utilizar módulos especializados para cada tipo de operación (como multiplicadores o divisores dedicados), de modo que el decodificador se encargue de enviar cada instrucción a su unidad correspondiente, optimizando así el uso del pipeline sin modificar el período de reloj del sistema.

Los operadores multiciclo pueden ser convencionales o segmentados:

- **Segmentado:** a cada ciclo le puedes mandar una nueva operación. (IR = 1)
- **Convencional:** al no estar segmentado, hasta que no se ha finalizado la operación previa no se le puede asignar otra.

Cuentan con las siguientes características:

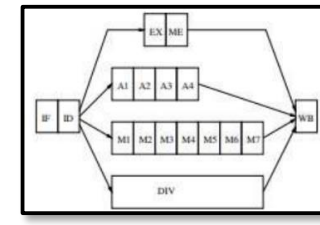
- **Latencia:** latencia o tiempo de evaluación es el tiempo en obtener el primer resultado.
- **Tasa de iniciación:** "IR" inversa del tiempo entre resultados. Si está segmentado, IR = 1.

RISC-V SEGMENTADO CON NUEVOS OPERADORES

Anteriormente, todas las instrucciones tomaban 5 ciclos para completarse, incluso si algunas no necesitaban pasar por la fase de MEM. Se aplicaba este diseño uniforme para que todas las instrucciones duraran exactamente 5 ciclos. Sin embargo, con la introducción de instrucciones más largas (como las de coma flotante y multiplicación), ya no tiene sentido aplicar esta uniformidad.

Por lo tanto, se crean nuevas rutas en el pipeline para cada tipo de operación, manteniendo el esquema anterior para las instrucciones tradicionales de 5 ciclos, pero extendiendo el diseño para las nuevas.

Nuevas rutas en el pipeline:



- **Registros inter-etapas para cada camino:**
  - *Instrucciones tradicionales:* ID/EX
  - *Instrucciones específicas:*
    - *Aritméticas (coma flotante):* ID/A1
    - *Multiplicación:* ID/M1
    - *División:* ID/DIV
- **Registros entre etapas adicionales:**
  - *Para instrucciones aritméticas de coma flotante:* A1/A2, A2/A3, A3/A4, ...
  - *Para multiplicaciones complejas:* M1/M2, M2/M3, ..., M6/M7

Se mantiene un único registro de entrada a la etapa WB (*/WB), pero debido a la posibilidad de que múltiples instrucciones finalicen simultáneamente y quieran escribir en el mismo ciclo, es necesario un multiplexor. Este multiplexor decide qué instrucción se envía a WB. Este escenario plantea un reto interesante, ya que podría haber varias instrucciones que intenten escribir a la vez, lo que requiere un control preciso para evitar conflictos y asegurar que las escrituras se realicen correctamente.

PROBLEMAS Y SOLUCIONES

El problema surge con la división no segmentada, que provoca stalls cuando hay más de una instrucción de división, ya que no puede procesar varias a la vez, a diferencia de la multiplicación.

En la escritura en WB, pueden ocurrir conflictos cuando una instrucción de coma flotante multiciclo y una entera, o dos instrucciones multiciclo, intentan escribir al mismo tiempo.

Como soluciones, tenemos:

1. **Registros separados para enteros y coma flotante:**
   a. *Ventajas*: Reduce los conflictos y aumenta el ancho de banda del banco de registros.
   b. *Desventajas*: Se necesitan conversiones explícitas entre tipos y se reduce la cantidad de registros para cada uno (32 en vez de 64).
2. **Cancelación en MEM:** Las instrucciones de carga/descarga que no escriben se cancelan tras MEM, pero no resuelve todo el problema.
3. **Stalls:** En lugar de añadir más puertos de escritura (costoso), se introduce stalls para gestionar los conflictos de escritura.

RIESGOS DE DATOS

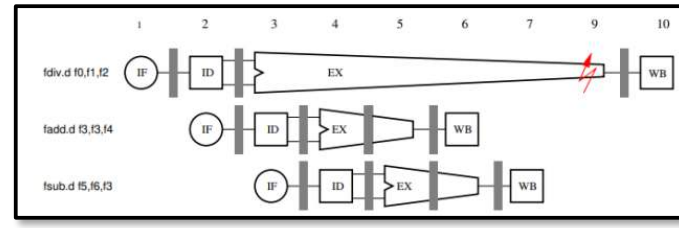
Tenemos los siguientes tipos de riesgos por datos:

- **RAW (Read After Write):** se producen riesgos RAW cuando una instrucción produce un resultado que es consumido por otra instrucción posterior lo suficientemente cercana.
- **WAW (Write After Write):** se producen riesgos WAW cuando dos instrucciones cercanas escriben en el mismo registro.

Para tratar de solucionarlo, se realiza la detección en ID e inserción de ciclos de parada.

Tratamiento de Excepciones

Con operaciones multiciclo se altera el orden de finalización de la ejecución de las instrucciones. En el momento en que se produce la excepción, algunas instrucciones posteriores han terminado ya.



Tipos de Dependencia – Instruction Level Paralelism (ISP)

Sabemos que:

- En una unidad segmentada sin operaciones multiciclo, tenemos pocos ciclos de parada, por lo que el CPI ≈ 1.
- En una unidad segmentada con operaciones multiciclo, tenemos muchos ciclos de parada, por lo que el CPI >> 1.

Las operaciones multiciclos son necesarias, por lo que se debe hacer algo. La posibilidad de solapamiento en las secuencias de instrucciones depende de si las instrucciones son independientes. A esto se le conoce como ILP.

Un ILP elevado es sinónimo de pocos conflictos, lo que provoca pocos ciclos de parada y con ello decrementa el CPI.

Dos instrucciones son independientes si pueden ejecutarse simultáneamente sin ningún problema, o lo que es lo mismo, se pueden reordenar. Tenemos los siguientes tipos de dependencias: *datos, nombre y control.*

- **Dependencias de datos:** dadas dos instrucciones, estas tienen dependencia si una lee donde la otra escribe.
- **Dependencias de nombre:** tenemos:
  - **Antidependencia:** si una lee de x, y la otra escribe en x es una antidependencia.
  - **Dependencia de salida:** si los dos escriben en x, es una dependencia de salida.
- **Dependencia de control:** determina la ordenación de algunas instrucciones respecto a un salto, las cuales hay que ejecutar antes de la instrucción que ejecuta el salto.

Técnicas para Aumentar el ILP

El objetivo es el de aumentar el ILP de las instrucciones que están en ejecución a la vez.

- **Bloque básico:** se consideran secuencias de instrucciones comprendidas entre instrucciones de salto y se estudia el grado de paralelismo extraíble de la ejecución de estas.

Para maximizar el ILP, se pueden utilizar dos enfoques:

- **Gestión dinámica de instrucciones:** El hardware reordena las instrucciones durante la ejecución para aprovechar el paralelismo.
- **Gestión estática de instrucciones:** El compilador reordena y modifica el código antes de la ejecución para minimizar dependencias y facilitar la paralelización.

Gestión Estática de Instrucciones

El compilador reordena o modifica el código para aumentar el ILP. Con esto, se pretende reducir o eliminar las dependencias. Para ello, se utilizan las siguientes estrategias:

Loop Unrolling

El Loop Unrolling implica replicar el código de un bucle varias veces, disminuyendo el número de iteraciones y reorganizando las instrucciones para reducir dependencias. Cuenta con las siguientes características:

- **Iteraciones:** se desenrolla según el máximo de ciclos de parada consecutivos entre instrucciones, más uno.
- **Reducción de sobrecarga:** menos iteraciones significan menos instrucciones de control y saltos.
- **Renombramiento de registros**: Se deben renombrar los registros para evitar conflictos.
- **Cambio en el índice**: El índice del bucle se incrementa en múltiplos del número de iteraciones desenrolladas.
- **Riesgo de registros**: Puede haber un agotamiento de registros y conflictos en operaciones no segmentables.

Ventaja:

- **Evitación de ciclos de parada**: Aumenta el tamaño del bucle y minimiza ciclos de parada.

Software Pipelining

El Software Pipelining transforma bucles con dependencias en bucles con instrucciones independientes, ejecutando instrucciones de manera escalonada.

Características:

- **Ejecución escalonada**: Procesa múltiples instrucciones en orden escalonado, simulando una unidad segmentada.
- **Triángulos de entrada y salida**: Se estructuran triángulos para el inicio y el final, con instrucciones independientes en medio.
- **Simulación de unidad segmentada**: Procesamiento desde la última etapa hacia la primera para evitar sobrescribir resultados intermedios.

Aunque el software pipelining tiene más sobrecarga que el Loop Unrolling, en definitiva, ambos métodos mejoran el rendimiento al optimizar el procesamiento de bucles.

TEMA 2.3 – PREDICCIÓN DINÁMICA DE SALTOS

Hasta el momento, la mejor predicción que hemos podido realizar es el "predict not taken", lo cual es aceptable ya que, con una latencia de uno, solo perdemos un ciclo por cada salto. Sin embargo, en algunos casos esta estrategia puede resultar insuficiente.

Se busca desarrollar un mecanismo de predicción de saltos más eficiente, el cual debe ser de naturaleza dinámica y estar implementado en hardware. Es necesario determinar si el objetivo es simplemente verificar si se cumple la condición del salto, o si también se desea conocer su cumplimiento y la dirección de este.

Además, es importante considerar cómo se llevará a cabo la predicción (basándose en el Program Counter), cuántos bits se utilizarán, la forma en que se almacenará (en tablas de caché) y el momento en que se realizará esta predicción (si se llevará a cabo tras la decodificación, en la etapa de ID, o antes de decodificar, en la etapa de IF).

BRANCH PREDICTION BUFFERS (BPB)

En la actualidad, se contempla únicamente la condición de salto, utilizando algunos bits y almacenando la información mediante correspondencia directa, lo cual resulta irrelevante en qué etapa se determina dicha condición.

Se implementa una tabla (buffer/cache) que almacena una parte del Program Counter (PC), específicamente algunos bits de menor peso, con el fin de identificar a qué instrucción corresponde cada entrada.

Además, se incluyen uno o varios bits de predicción que indican la acción esperada del salto. La información almacenada representa el estado del predictor. A partir de este estado, se realiza la predicción sobre si se producirá un salto o no, siendo este estado el correspondiente a un autómata.

PREDICTOR 1 BIT

El predictor de 1 bit representa el caso más sencillo, donde el estado del autómata se basa en la condición obtenida en la última ejecución del salto. La predicción se realiza en función del estado del autómata. Si el salto se ha producido, el predictor asume que en el siguiente salto también se realizará; si, en cambio, no se lleva a cabo, el estado se modifica, y la próxima predicción se orientará hacia la no realización del salto. Si posteriormente vuelve a saltar, el estado se actualizará nuevamente, y así se repetirá el proceso.

Un inconveniente de este enfoque es que, al utilizar solo unos pocos bits del Program Counter (PC), es probable que distintas instrucciones compartan los bits de menor peso, lo que podría dar lugar a múltiples instrucciones que lean y modifiquen la predicción del salto de manera simultánea.

Solución: Aumentar el tamaño de la tabla o utilizar un esquema de correspondencia asociativa o asociativa por conjuntos en lugar de correspondencia directa.

PREDICTOR DE 2 BITS: HISTÉRESIS Y SATURACIÓN

Las instrucciones de salto que implementan un bucle siguen un patrón predecible. En un bucle de n iteraciones, el salto se ejecuta con éxito n - 1 veces, mientras que en la última iteración no se realiza. Este comportamiento se puede aprovechar mediante la implementación de una solución con cuatro estados. En este contexto, la predicción debe fallar dos veces antes de que se modifique el estado del predictor.

Existen dos versiones de este enfoque: en la primera, es necesario que la condición falle de manera consecutiva dos veces para cambiar el estado; en la segunda, se puede permanecer en un estado intermedio, permitiendo un cambio de estado tras solo una falla adicional. En la primera versión, si se producen dos fallos, el predictor transitará al estado fuertemente opuesto; en la segunda, se moverá al estado débil.

- **Strongly Not Taken (Estado 00):** Hay una clara tendencia a no realizar el salto. Se asegura que no se realizará.
- **Weakly Not Taken (Estado 01):** Existe cierta tendencia a no realizar el salto, pero puede ocurrir. No, pero es posible.
- **Weakly Taken (Estado 10):** Hay una cierta tendencia a realizar el salto, aunque no es seguro. Sí, pero puede no serlo.
- **Strongly Taken (Estado 11):** Hay una clara tendencia a realizar el salto. Se asegura que sí se realizará.

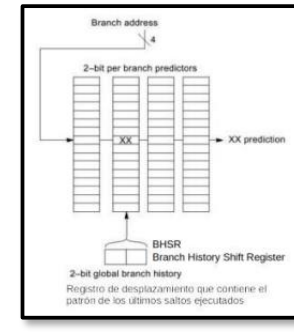
PREDICTORES DE DOS NIVELES

El predictor bimodal realiza la predicción utilizando una única tabla indexada por el *Program Counter (PC)* de la instrucción de salto en cuestión. En esta tabla se almacena el comportamiento de dicha instrucción, similar a lo que se implementa en los predictores de 1 y 2 bits.

Por otro lado, el *predictor de dos niveles* agrega a la predicción el patrón de ejecución (salto/no salto) de los saltos ejecutados previamente. Esto permite tener en cuenta no solo el comportamiento del salto actual, sino también el de otros saltos. Esta consideración es especialmente relevante en situaciones donde hay múltiples instrucciones condicionales encadenadas, ya que el resultado de un salto puede depender de los resultados de otros saltos anteriores.

En lugar de almacenar únicamente la información del salto ejecutado más recientemente, se registra lo que han realizado los 2 o 3 saltos anteriores. De esta manera, al repetirse una combinación de saltos, se puede predecir con mayor certeza si el salto en cuestión se llevará a cabo o no.

La memoria caché en este esquema es más amplia, ya que debe almacenar todas las posibles combinaciones de resultados de saltos anteriores. Se utilizan los bits del *Branch History Shift Register (BHSR)* para llevar un registro de estas combinaciones. En el diagrama, los símbolos "XX" representan el estado actual del predictor (los autómatas).



Para determinar qué columnas se deben consultar, es necesario observar los resultados de los saltos anteriores. Por ejemplo, si ambos saltos anteriores resultaron en un "sí" (0,0 o 1,1), se consulta la primera columna; si uno fue "sí" y el otro "no" (0,1 o 1,0), se consulta la segunda columna.

PREDICTOR HÍBRIDO – CONTADOR

Cada tipo de predictor es adecuado para distintos patrones de comportamiento, por lo que se propone combinar varios predictores y seleccionar el más apropiado para cada caso específico. Un ejemplo de este enfoque es el *Tournament Predictor.*

El mecanismo de selección funciona de la siguiente manera: se elige el predictor que ha mostrado un mejor rendimiento hasta el momento. Si el predictor P2 acierta, se incrementa un contador (cont++), y si acierta el predictor P1, se decrementa el mismo contador (cont--). Cuando ambos predictores aciertan o fallan, el contador permanece sin cambios; sin embargo, si uno acierta y el otro no, el contador se ajusta a favor del predictor que tuvo éxito.

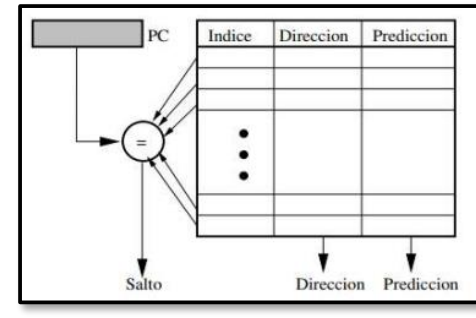
Si el contador es negativo, se selecciona el predictor P2; si es positivo, se opta por el predictor P1. Como el contador es binario, se puede utilizar el primer bit para determinar si su valor es positivo o negativo.

BRANCH TARGET BUFFERS (BTB)

El *Branch Target Buffer (BTB)* combina tanto la condición como la dirección de los saltos, utilizando el Program Counter (PC) y almacenando esta información de manera totalmente asociativa. Esta operación se realiza en la etapa de búsqueda de instrucciones (IF). El PC se utiliza para determinar si se producirá un salto y hacia dónde se realizará (es importante identificar si se trata de un salto en la etapa de IF).

La caché en este contexto es completamente asociativa, ya que se utiliza el PC completo como índice. En esta caché se registra la dirección de destino del salto y el resultado de la predicción.

Anteriormente, la predicción se realizaba utilizando el método "Predict Not Taken" con una latencia de uno en la etapa de decodificación (ID). Sin embargo, es necesario modificar el procesador para implementar un esquema segmentado que permita llevar a cabo esta operación en la etapa IF, eliminando así la penalización y logrando una latencia de cero. Esto permite conocer el resultado de la predicción antes de la decodificación.



Cuando una instrucción de salto llega al procesador, se consulta la caché:

- **Si no está presente:** Si el salto no se encuentra en la caché, se predice que no ocurrirá, y se procede a ejecutar la instrucción normalmente.
- **Si está presente:** Si el salto está en la caché, se tienen dos opciones:
  - **Salta:** Si se predice que el salto se ejecutará, se inicia la ejecución del salto y se comienza a buscar nuevas instrucciones en la dirección de destino.
  - **No salta:** Si se predice que no se ejecutará, se continúa con la ejecución de las instrucciones de manera normal.

En caso de una **predicción errónea**, será necesario cancelar todas las instrucciones que se hayan comenzado a ejecutar, restaurar el PC a la dirección correcta y actualizar la tabla de la caché con la nueva predicción.