ARCOM 3 – PARALELISMO A NIVEL DE INSTRUCCIÓN

EMISIÓN MÚLTIPLE

- La SEGMENTACIÓN permite ejecutar varias instrucciones a la vez.
- El objetivo de explotar el ILP (*Instruction Level Paralelism*) es **optimizar el CPI**:
  - $CPI = CPI_{ideal} + \text{Paradas por riesgos estructurales} + \text{Paradas por riesgos de datos} + \text{Paradas por riesgos de control}.$
  - ↳ La predicción de saltos realizada por el compilador, el unrolling, etc. ayudan a exponer más paralelismo.
- Hay dos alternativas para **mejorar el ILP**:
  - Hacer más **profundo el pipeline** (tema 2) ⟶ menos trabajo por etapa ⟶ T_{clock} más corto ⟶ mayor f_{clock}.
  - EMISIÓN MÚLTIPLE (tema 3) ⟶ se replican las etapas del pipeline en **múltiples pipelines** en los que se comenzarán **varias instrucciones en cada ciclo**.
    - ↳ Entonces, $CPI < 1$, por lo que en su lugar se usa el **IPC** (número de instrucciones por ciclo) como medida de rapidez.

DEPENDENCIAS DE NOMBRE

- DEPENDENCIAS DE NOMBRE ⟶ suceden cuando dos instrucciones usan el **mismo nombre de registro**, pero entre ellas **no hay flujo real de información**.
- Aunque no son dependencias reales de datos suponen un problema al reordenar código, ya que **el orden** de las instrucciones en las dependencias de nombre **se debe conservar**.
- Son las dependencias **WAR** y **WAW**.
  - En el MIPS segmentado estudiado estas dependencias no producen riesgos, pero en otras arquitecturas sí.
    - ↳ Se resuelven mediante **renombrado de registros.**

- Ejemplo 1:

```
add x1,x2,x3
beq x4,x0,L
sub x1,x1,x6
L: ...
or x7,x1,x8
```

  - Tipo RAW
    - x1 de add a sub; x1 de sub a or
  - Instrucción or es dependiente de add y sub
  - Tipo WAW
    - X1 de add a sub

- Ejemplo 2:

```
add x1,x2,x3
beq x12,x0,skip
sub x4,x5,x6
add x5,x4,x9
skip:
or x7,x8,x9
```

  - Tipo RAW
    - x4 de sub a add
  - Si x4 no se usa después de "skip", se puede reordenar:
    - Mover sub a antes del salto

EMISIÓN MÚLTIPLE ESTÁTICA VS DINÁMICA

- EMISIÓN MÚLTIPLE **ESTÁTICA** ⟶ el **compilador** agrupa instrucciones en **paquetes de emisión** para emitirlas juntas.
  - El compilador **detecta** y **evita riesgos** con lo cual reduce o elimina **paradas**.
  - ✗ No es eficiente para aplicaciones que no sean científicas.
- EMISIÓN MÚLTIPLE **DINÁMICA** ⟶ la **CPU** examina el flujo de instrucciones y selecciona las que se deben emitir en cada ciclo.
  - La CPU **evita riesgos** utilizando técnicas avanzadas en **tiempo de ejecución**.
  - El **compilador** puede ayudar **reordenando** las instrucciones.
  - ▸ Se usa sobre todo en servidores y procesadores de escritorio.

EMISIÓN MÚLTIPLE ESTÁTICA

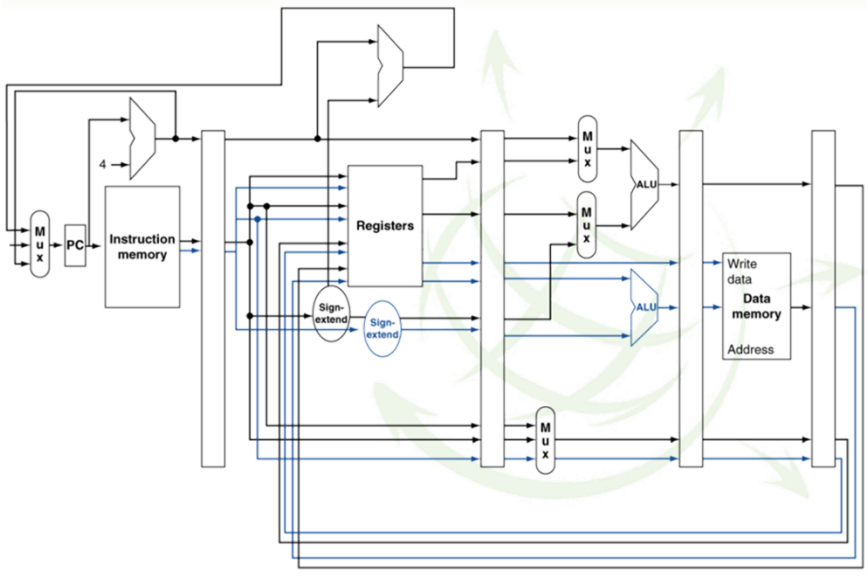
- En la EMISIÓN MÚLTIPLE ESTÁTICA el **compilador** agrupa instrucciones en **paquetes de emisión** para emitirlas juntas.
- Un PAQUETE DE EMISIÓN es un grupo de instrucciones que pueden ser **emitidas** en un solo **ciclo**, y viene determinado por los **recursos del pipeline** que requieran.

PLANIFICACIÓN ESTÁTICA

- En la PLANIFICACIÓN ESTÁTICA el **compilador** debe **eliminar algunos/todos los riesgos**:
  - En un paquete de emisión sólo puede haber instrucciones **sin dependencias entre ellas** (si no, no se podrían ejecutar simultáneamente).
  - Posiblemente haya **dependencias entre paquetes** ⟶ su tratamiento varía entre ISAs. **El compilador debe saberlo**.
  - Se rellenará el resto del paquete con **NOP** si no se encuentran instrucciones adecuadas.

MIPS con emisión dual estática

- Se forman paquetes de emisión con una instrucción **ALU** o un **salto** seguida de una instrucción de **load** o **store**.
- Las instrucciones no utilizadas se rellenan con **NOP**.



Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

- Se ejecutan más instrucciones en paralelo, pero surgen **nuevos casos de riesgos de datos en EX**:
  - El **forwarding** evita paradas que se producirían con la **emisión simple**.
  - Pero ahora **no se puede adelantar** el resultado de una instrucción ALU en una load/store **en el mismo paquete** de dos instrucciones, por lo que hay que dividirlo en **dos paquetes** o se producirá una **parada**.

```
add  $t0, $s0, $s1
load $s2, 0($t0)
```

- Se puede producir **riesgo de carga y uso entre paquetes** de emisión dual → sigue habiendo un ciclo de parada, pero ahora se ejecutan dos instrucciones en cada paquete.
  - Se requiere una **planificación más agresiva** que para la emisión tradicional de una sola instrucción.

➢ Planificar el código para el MIPS de emisión dual considerando salto retardado con una ranura de salto:

```
Loop: lw   $t0, 0($s1)      # $t0=array element
      addu $t0, $t0, $s2    # add scalar in $s2
      sw   $t0, 0($s1)      # store result
      addi $s1, $s1,-4      # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,-4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

- IPC = 5/4 = 1.25 (el máximo sería IPC = 2)

UNROLLING

- El UNROLLING consiste en replicar el cuerpo de un lazo para exponer más paralelismo:
  - ✓ Reduce la sobrecarga por **control del lazo**.
  - ✓ Reúsa datos almacenados en las memoria **caché**.
- Conviene usar **diferentes registros** para la replicación (RENOMBRADO DE REGISTROS).

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1,-16	lw $t0, 0($s1)	1
	nop	lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t3, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

➢ IPC = 14/8 = 1.75
- Más próximo a 2, pero al coste de usar más registros y construir un código más largo

PROCESADORES VLIW

- Las arquitecturas VLIW (*Very Long Instruction Word*) ven los paquetes como una **instrucción muy larga** que especifica **múltiples operaciones concurrentes**.
- Debe haber **suficiente paralelismo** en el código como para **llenar todos los slots** disponibles.
- ✓ **Hardware** de la CPU más **simplificado**.
- ✓ Menor **consumo de potencia**
- × Es necesario **encontrar paralelismo estáticamente** ya que no hay hardware de detección de riesgos.
- × Gran **tamaño** del código.

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
fld f0,0(x1)	fld f6,-8(x1)			
fld f10,-16(x1)	fld f14,-24(x1)			
fld f18,-32(x1)	fld f22,-40(x1)	fadd.d f4,f0,f2	fadd.d f8,f6,f2	
fld f26,-48(x1)		fadd.d f12,f0,f2	fadd.d f16,f14,f2	
		fadd.d f20,f18,f2	fadd.d f24,f22,f2	
fsd f4,0(x1)	fsd f8,-8(x1)	fadd.d f28,f26,f24		
fsd f12,-16(x1)	fsd f16,-24(x1)			addi x1,x1,-56
fsd f20,24(x1)	fsd f24,16(x1)			
fsd f28,8(x1)				bne x1,x2,Loop

➢ Ejemplos de tipos de instrucción en un procesador VLIW:
- Una operación con enteros o una operación de salto
- Dos operaciones independientes en punto flotante
- Dos referencias a memoria independientes

EMISIÓN MÚLTIPLE DINÁMICA

- En la EMISIÓN MÚLTIPLE ESTÁTICA la **CPU** examina el flujo de instrucciones y selecciona las que se deben emitir en cada ciclo.
- Los procesadores que la realizan se denominan PROCESADORES SUPERESCALARES.
- Evita la necesidad de **planificación** del **compilador,** ya que se realiza por hardware:
  - ↳ La CPU asegura la semántica del código.
  - × Da lugar a CPUs más complicadas en cuanto a hardware.
- Permite a la CPU emitir varias instrucciones que:
  - Se **ejecutan fuera de orden** (para evitar paradas).
  - **Terminan fuera de orden** en muchos casos.
  - Pero el WB respeta la semántica del programa.

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```

- Puede comenzar `sub` mientras `addu` está esperando por `lw`

PLANIFICACIÓN DINÁMICA

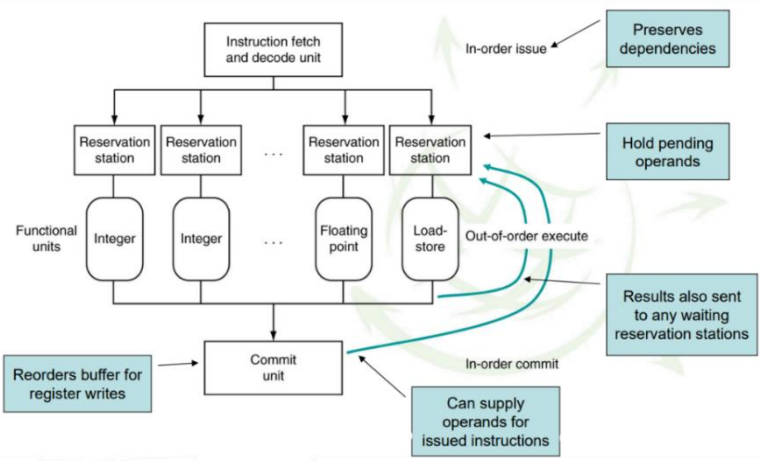
- La PLANIFICACIÓN DINÁMICA consiste en **reordenar** las instrucciones para reducir las paradas a la vez que se mantiene el flujo de datos del programa.
- ✓ El **compilador** no necesita conocer el **hardware**.
- ✓ Puede gestionar casos en que las **dependencias no** se conocen en **tiempo de compilación**.
- × Hay un incremento de **complejidad hardware**.
- × Complica la **gestión de excepciones**.

? ¿Por qué hacer planificación dinámica? ¿Por qué no dejar que el compilador planifique directamente el código?

- No todas las paradas son **predecibles** (por ejemplo, los fallos de caché).
- No siempre se puede planificar estáticamente, por ejemplo, si hay instrucciones de **salto**, ya que el **resultado** del salto se determina **dinámicamente**.
- **Diferentes implementaciones de un ISA** pueden tener diferentes **latencias** y dar lugar a diferentes **riesgos**.

Hay dos técnicas básicas de planificación dinámica:

- MARCADOR ⟶ **retener** las instrucciones emitidas hasta que haya **recursos suficientes** y **no haya riesgos de datos**.
- TOMASULO ⟶ eliminar dependencias de datos WAR y WAW realizando **renombrado de registros**.



➢ Ejemplo:

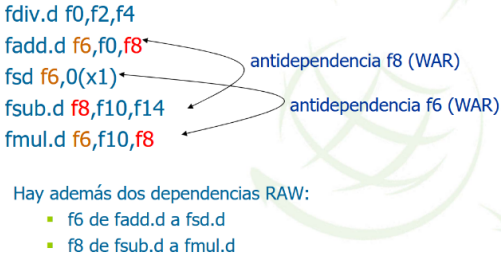
fdiv.d **f0**,f2,f4
fmul.d f6,**f0**,f8
fadd.d **f0**,f10,f14

- fmul.d tiene dependencia RAW con fdiv.d en **f0**. Se producirán paradas.
- fadd.d debido a la dependencia WAR en f0 no puede emitirse tras fmul.d sin que se produzcan paradas, salvo que se aplique renombrado de registros

- INSTRUCTION COMMIT ⟶ proceso que se produce cuando una instrucción actualiza su resultado en el **archivo de registros**.

RENOMBRADO DE REGISTROS

- El RENOMBRADO DE REGISTROS resuelve las dependencias WAR y WAW usando las **estaciones de reserva** y **buffers de reordenamiento**.



Hay además dos dependencias RAW:

- f6 de fadd.d a fsd.d
- f8 de fsub.d a fmul.d

Código con renombrado

fdiv.d f0,f2,f4
fadd.d S,f0,f8
fsd S,0(x1)
fsub.d T,f10,f14
fmul.d f6,f10,T

➢ Ahora solo quedan riesgos RAW, cuyas paradas pueden ser evitadas reordenando

ESTACIONES DE RESERVA

- El renombrado de registros se realiza mediante ESTACIONES DE RESERVA (RS), que tienen una **entrada** por cada **instrucción en ejecución**.
- Las instrucciones **entran** a las RS **en orden**, pero pueden **ejecutarse desordenadas** dependiendo de la disponibilidad de los operandos que necesitan.

Cuando una instrucción se emite a una estación de reserva:

- Si el **operando** está **disponible** en el almacén de registros o en un buffer de reordenamiento ⟶ se copia a la estación de reserva.
  - ↳ Si no se necesita en el registro para más adelante se puede **sobreescribir**.
- Si el **operando no** está **disponible** ⟶ se copia a la estación de reserva cuando esté disponible.
  - ↳ Podría no ser necesario **actualizar** el registro.

Campos de una entrada en la RS:

INST.	B	OP1	V1	OP2	V2	R

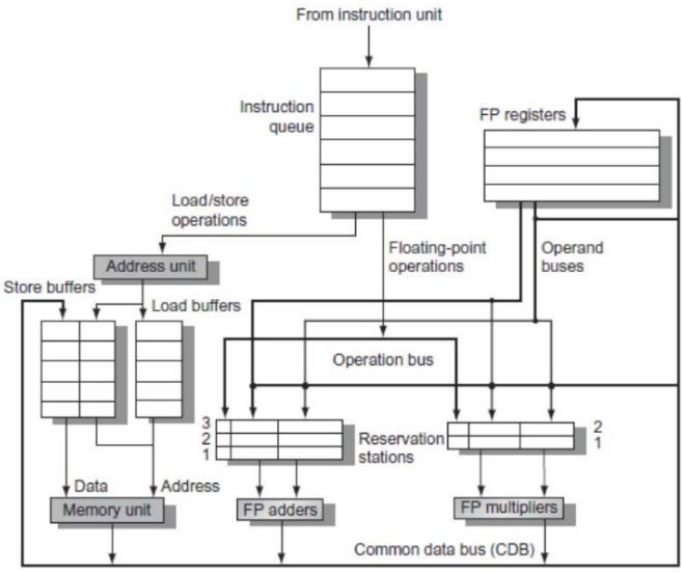
- B ⟶ indica si entrada está ocupada por una instrucción o libre.
- $INST$ ⟶ campo de código de instrucción.
- $OP1$ y $OP2$ ⟶ operandos de entrada de la instrucción: proceden de registros o de caminos de adelantamiento de las unidades de ejecución.
  - Pueden ser un identificador de registro de renombrado, es decir, que su valor está siendo calculado.
- $V1$ y $V2$ ⟶ indica si los operandos $OP1$ y $OP2$ son válidos.
- R ⟶ indica si la instrucción está lista para la emisión.

ALGORITMO DE TOMASULO

- TOMASULO rastrea cuando los operandos están disponibles e introduce renombrado de registros.

Cada instrucción pasa por tres pasos:

1. Emitir:
   1.1. Obtener la siguiente instrucción de la cola FIFO.
   1.2. Si la RS está disponible:
   - Si los valores de los operandos están disponibles ⟶ emite la instrucción a la RS.
   - Si los valores de los operandos no están disponibles ⟶ detiene la instrucción.
2. Ejecutar:
   2.1. Cuando un operando está disponible, lo almacena en cualquier estación de reserva que lo esté esperando.
   2.2. Cuando todos los operandos están listos, emite la instrucción.
   - Las **cargas** y el **almacenamiento** se mantienen en el **orden** del programa a través de la DIRECCIÓN EFECTIVA.
   - No se permite iniciar la ejecución de ninguna instrucción hasta que se completen todas las ramas que la preceden en el orden del programa.
3. Escribir el resultado:
   3.1. Escribir en las estaciones de reserva y en los buffers de almacenamiento el resultado que está en el bus CDB (bus común de datos).
   ↳ Los **almacenamientos** deben **esperar** hasta que la dirección y los valores estén disponibles



ESPECULACIÓN BASADA EN HARDWARE

- La ESPECULACIÓN consiste en hacer conjeturas sobre de lo que hará una instrucción.
  1. Comenzar la operación tan pronto como sea posible.
  2. Chequear si se acertó:
     - Si se acertó ⟶ completar la operación.
     - Si no se acertó ⟶ retroceder y corregir.
- Se realiza tanto en emisión múltiple **estática** como en **dinámica**.

Tipos de especulación:

- Especulación realizada por el **compilador** ⟶ reordena instrucciones (p. e. mover la carga a antes de un salto).
  - Si la especulación es incorrecta, puede incluir **instrucciones de "arreglo"** para recuperarse.
- Especulación realizada por **hardware** ⟶ busca por adelantado instrucciones para ejecutar.
  - Mete los resultados de ejecutar la instrucción en un buffer hasta que se determine si se necesitan realmente, si la especulación es incorrecta se **vacían los buffers**.

ESPECULACIÓN ACERCA DEL SALTO

- Consiste en especular acerca del resultado de un salto y continuar emitiendo instrucciones asumiendo que son las correctas,
- No se actualizan los registros (Instruction Commit) hasta que se determine el resultado del salto.

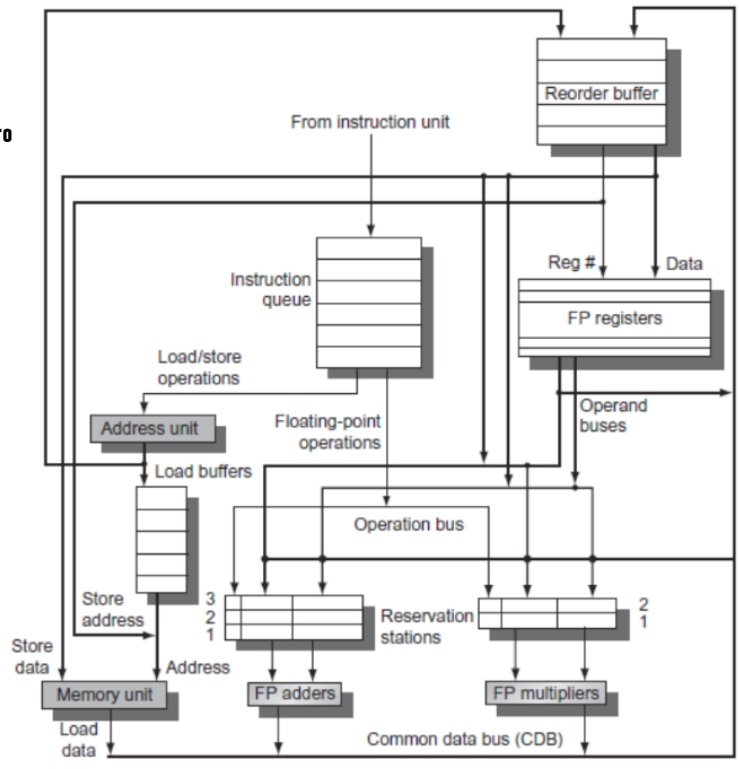
ESPECULACIÓN ACERCA DE LA CARGA

- Trata de evitar el retardo por carga y fallo caché.
  1. Predecir la dirección efectiva.
  2. Predecir el valor cargado.
  3. Pasar los valores almacenados a la unidad de carga.
- No se actualizan los registros hasta que se resuelva la especulación.

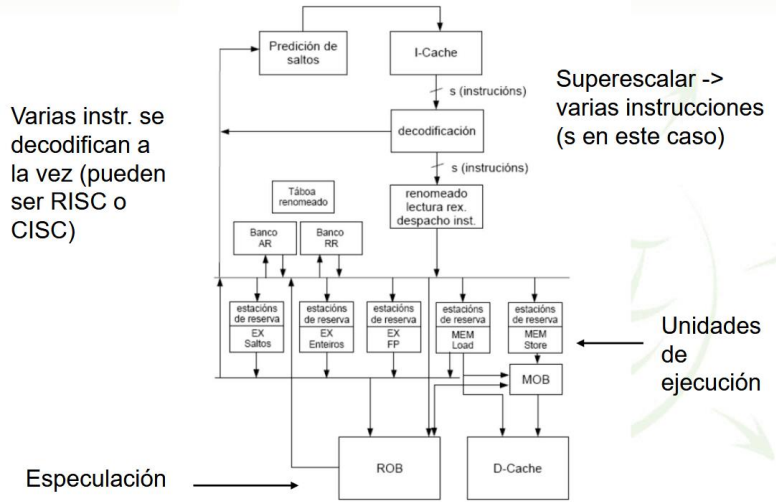
BUFFERS DE REORDENAMIENTO (ROB)

Posibles estados para una instrucción:

- EN EJECUCIÓN ⟶ la instrucción no acabó de ejecutarse.
- FINALIZADA ⟶ la ejecución acabó y se escribe el resultado en un **registro renombrado** o en un **buffer** previo a escribir en memoria (si es un store).
- COMPLETADA ⟶ modifica los registros del núcleo.
- RETIRADA:
  - Si el destino es un registro ⟶ coincide con COMPLETADA.
  - Si el destino es la memoria ⟶ se escribe en la caché de datos.



ARQUITECTURA DE ALTO NIVEL DE UN NÚCLEO



DIFICULTADES DE LA EMISIÓN MÚLTIPLE

- Los programas tienen **dependencias** que limitan el ILP, y algunas son difíciles de eliminar (p. e., las asociadas a los punteros).
- Algún **paralelismo** es **difícil de encontrar** en el código,
  ya que se usa una ventana de emisión (rango de instrucciones analizadas) de tamaño limitado.
- Hay **retardos de memoria** y el **ancho de banda** está **limitado** ⟶ es difícil mantener los pipelines llenos.
- La **especulación** puede ayudar si se hace bien.

TIPOS DE ARQUITECTURAS

- Las microarquitecturas modernas usan **emisión múltiple**, **planificación dinámica** y **especulación.**

Hay dos grandes enfoques:

- Asignar estaciones de reserva y **actualizar** la **tabla de control** del pipeline en **medio ciclo de reloj**.
  - × Sólo permite 2 IPC.
- Añadir **lógica de diseño** para manejar cualquier posible dependencia entre las instrucciones.
  - La lógica de emisión es el **cuello de botella** en los superescalares planificados dinámicamente.

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

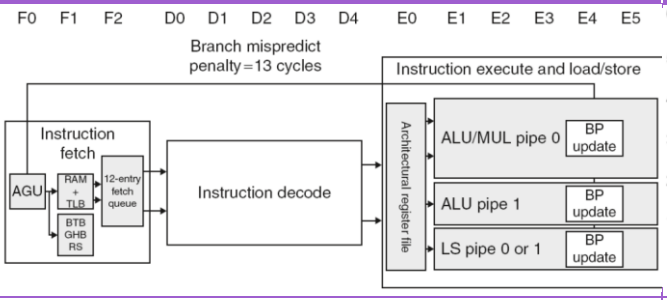
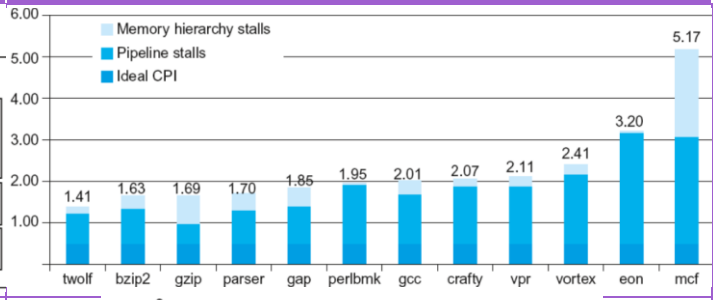
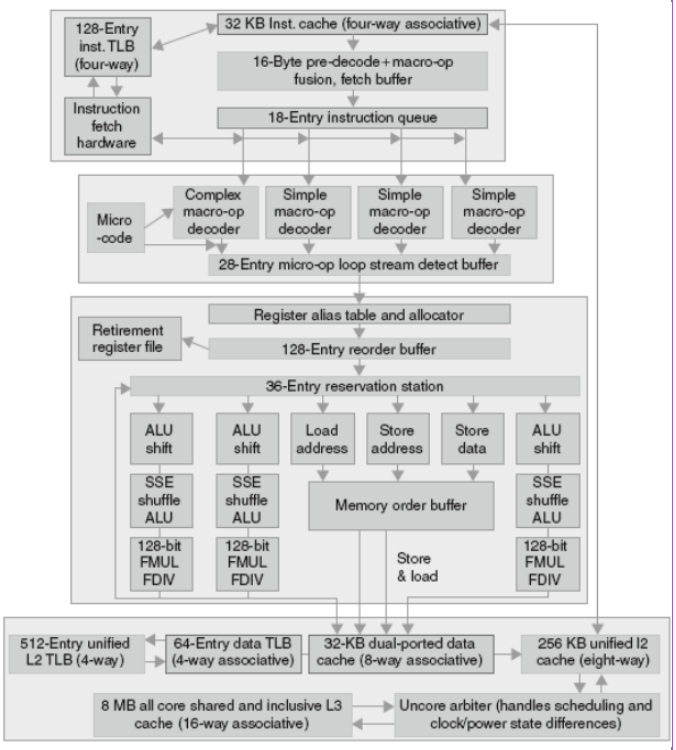
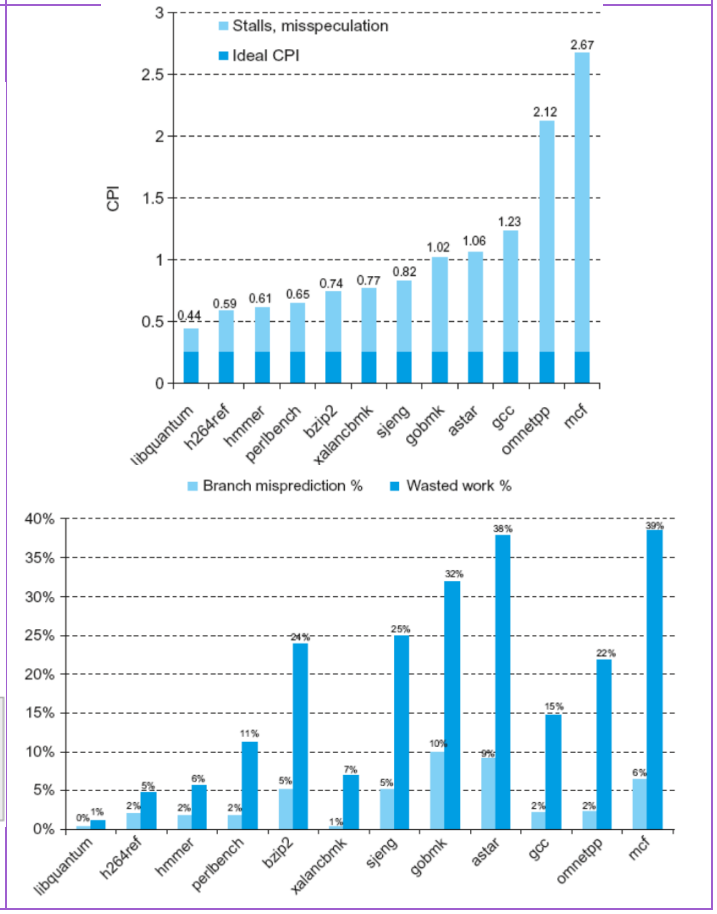
EFICIENCIA ENERGÉTICA

- La complejidad de la **planificación dinámica** y las **especulaciones** requieren **energía**.
- Podría resultar más eficiente tener **muchos núcleos más simples**.

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

EJEMPLO: ARM A8 e INTEL I7

Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple issue? real	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	14	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	2-level	2-level
1st level caches/core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-1024 KiB	256 KiB
3rd level caches (shared)	-	2- 8 MB

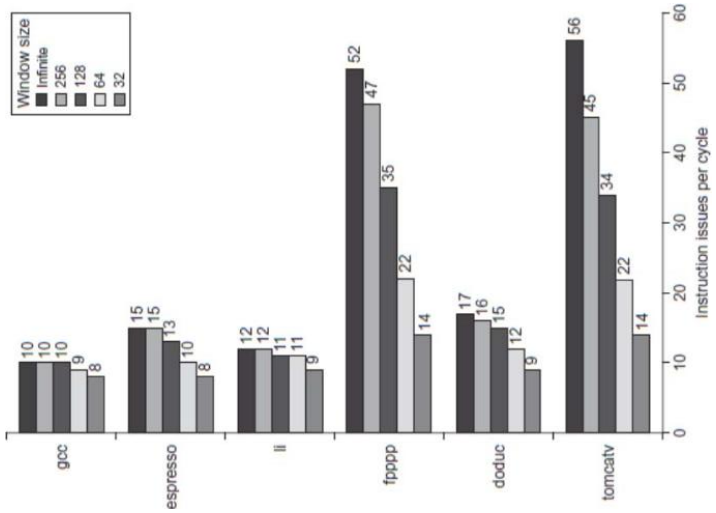
	PIPELINE	RENDIMIENTO EXPRESADO MEDIANTE EL CPI
A8		
I7		

FALACIAS

- Creer que procesadores con **CPIs** más bajos o **frecuencias** de reloj más **altas** son **más rápidos**.

Processor	Implementation technology	Clock rate	Power	SPECCInt2006 base	SPECCFP2006 baseline
Intel Pentium 4 670	90 nm	3.8 GHz	115 W	11.5	12.2
Intel Itanium 2	90 nm	1.66 GHz	104 W approx. 70 W one core	14.5	17.3
Intel i7 920	45 nm	3.3 GHz	130 W total approx. 80 W one core	35.5	38.4

- Creer que hay grandes cantidades de **ILP disponibles** y que se podrían explotar si tuviéramos las **técnicas adecuadas**.



CONCLUSIONES

- El **ISA** influencia el **diseño** del camino de datos y control.
- En las máquinas que realizan emisión múltiple, la **segmentación mejora** la **productividad**, pero la **latencia** por instrucción **no cambia**.
- Los **riesgos** estructurales, de datos y de control pueden producir **paradas**.
- En las arquitecturas de **emisión múltiple** y **planificación dinámica**:
  - Las **dependencias de datos** limitan la cantidad de paralelismo explotable.
  - La **complejidad del hardware** requerido lleva la potencia requerida al límite de lo permisible.