ARQUITECTURA E INGENIERÍA DE COMPUTADORES

Tema 2.2

Miquel Gómez Corral Ing. Informática ETSINF, UPV

Tema 2.2

Unidades multiciclo y gestión estática de instrucciones

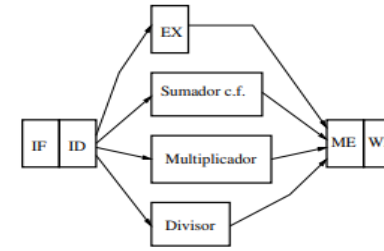
OPERACIONES MULTICILOS

Son de 2 tipos, o **Instrucciones enteras** que realizan operaciones más complicadas (*mul, mulw, div, divw*) o Instrucciones de coma flotante (*fadd.s, fadd.d, fmul.s, fmul.d, fdiv.s, ...*). Necesitan **mayor tiempo en la fase EX**.

Soluciones

- **Aumentar el periodo de reloj** de la unidad de instrucción: *Se ralentiza toda la maquina* ***MALAMENTE***.
- **Variar duración fase EX**: Permitir que la fase EX de estas operaciones se prolongue durante varios ciclos de reloj (*operaciones multiciclo*). El **período de reloj no cambia**.
    - **Ejemplo**: fmul.d con un operador de 40ns Fases: $IF\ ID\ EX\ EX\ EX\ EX\ ME\ WB$

Implementación: **Diferentes posibles** implementaciones, una forma sería hacer cosas extra en la etapa EX, y otra mejor **poner diferentes módulos** (*operadores especializados*) para cada cosa. *Ya se apañará el decodificador de mandar cada instrucción donde toca.*



Tipos de operadores multicilos

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 ha acabado la operación anterior** no puedes darle más (*ya sea xq es muy complicada, xq no renta...*). $IR\ != 1$. *TODOS SON SEGMENTADOS MENOS EL DIV*

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 esta segmentado, IR = 1.*

Ejemplo de operadores añadidos al RISC-V

- Sum/rest coma flotante segmentado. T_{ev}: 4. IR: 1 cada ciclo.
- Mult entero/coma flotante segmentado. T_{ev}: 7. IR: 1 cada ciclo.
- Div entero/coma flotante convencional. T_{ev}: 24. IR: 1 cada 24 ciclos.

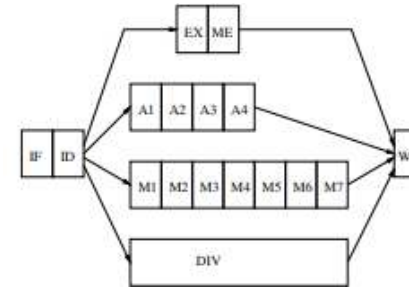
RISC-V Segmentado con nuevos operadores

Antes todas las **instrucciones duraban 5 ciclos**, y aunque a algunas no les hiciera falta pasar por fase de memoria. Se las hacía pasar todas para que fueran 100% de 5 ciclos. Pero ahora hay **instrucciones que son más largas**, por lo que ya no tiene sentido hacerlo con estas nuevas. Se hacen **nuevas rutas para cada una de las operaciones** (*manteniendo lo que teníamos antes para las de toda la vida y ampliando para las nuevas*).

Se necesitan nuevos registros inter-etapas

1. **ID/EX una para cada camino**: ID/EX, ID/A1, ID/M1, ID/DIV
2. **Entre aritméticas de coma flotante**: A1/A2, A2/A3, A3/A4
3. **Para multiplicaciones**: M1/M2, M2/M3, . . . , M6/M7

Hay un único registro a la entrada de WB (*/WB), pero se necesita un multiplexor que determine que instrucción pasa a WB (*Puede ser que varias instrucciones quieran escribir a la vez, jeje que chulo*).



PROBLEMAS / SOLUCIONES

División no segmentada (*Riesgo estructural*): Cuando hay por ejemplo varias instrucciones en MULT pueden entrar xq está segmentado, pero división no, por lo que se tiene que esperar: **Inserción de Stalls**.

i1	IF	ID	DIV	DIV	DIV	DIV	DIV	DIV	WB				
i2		IF	id	id	id	id	id	ID	DIV	DIV	DIV	DIV	...
i3			if	if	if	if	if	IF	ID	EX	M	WB	
i4									IF	ID	EX	M	

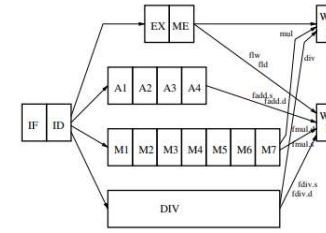
Escritura en WB por varios en el banco de registros (*Riesgo estructural*): Hay 2 posibles problemas, que escriba una de **coma multiciclo** (mult div… **coma flotante**) y **otra entera**, o **2 multicilos**.

i1	IF	ID	M1	M2	M3	M4	WB	
i2		IF	ID	A1	A2	A3	WB	
i3			IF	ID	A1	A2	A3	WB
i4				IF	ID	EX	ME	WB

2
1

1 **Sol. CF y entera**: Se hace que los enteros y los operandos en coma flotante escribían en sitios diferentes. **REGISTROS SEPARADOS PARA CADA TIPO**.

- **Ventajas**: **Reduces riesgos estructurales**, duplica el número total de registros, se duplica el ancho de banda del banco de registros (xq ahora puedo acceder a uno de 32 (entero) a la vez que a uno de 64 (float)).
- **Desventajas**: Para transferir datos de un tipo de registros al otro te hacen falta conversiones explicitas y hemos limitado la cantidad de registros para cada tipo (en vez de 64 que pueden usar los 2 ahora cada uno usan solo 32).



Cancelar instrucciones en MEM: Las instrucciones carga/descarga y NO escriben **se cancelan** tras la etapa MEM.

Como tal no hemos solucionado todo el problema por completo, solo lo hemos mejorado un poco.

2 **Sol. Varias CF y para todo en general**: Se plantea poner **más puertos de escritura** (conexiones en las que se pueden escribir los registros) pero es *más caro y poco eficiente*. Esto ya que en cada ciclo se lanza 1 intrucción, y cada instrucción escribe SOLO 1 vez, por lo que la media sigue siendo 1 → No te rallas, lo retrasas con STALL/Ciclos parada.

Riesgos de 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. *Lo que pasa es que ahora se ponen MUCHOS ciclos de parada xq son MUY largas las operaciones incluso usando cortocircuitos.*

fadd.d **f0**,...	IF	ID	A1	A2	A3	A4	WB			
fmul.d ...,**f0**		IF	id	id	id	ID	M1	M2	M3	...
i3			if	if	if	IF	ID	EX	ME	WB
i4							IF	ID	EX	ME
i5								IF	ID	EX

WAW (*Write After Write*): Se producen riesgos **WAW** cuando dos instrucciones cercanas **escriben en el mismo** registro. *Si A quería escribir antes que B, pero B termina antes que A, se joden el uno al otro xq sobrescribe el valor.*

i1	fmul.d **f0**,f2,f4	IF	ID	M1	M2	M3	M4	M5	M6	M7	WB
i3	fadd.d **f0**,f10,f12		IF	ID	A1	A2	A3	A4	WB		

*Como Solución se hace la Detección en ID e **inserción de ciclos de parada**.*

Puede pensar que, como el valor que de la segunda será la que es ha de quedar, pues elimino la primera y yau, es ineficiente xq no se va a usar, no hace falta usarla. *Pero puede pasar como en el ejemplo este:*

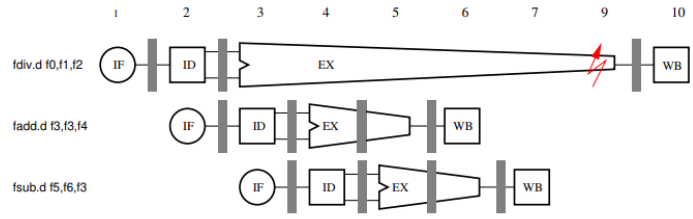
```
if (flag == 0)
   /* camino más frecuente */
   x = x*2;
else
   x = a;
cont: ...
```

$t1 \equiv flag, f0 \equiv x, f2 \equiv 2$

```
      fmul.d f0,f0,f2
      beqz   t1,cont
      fld    f0,a(zero)
cont: ...
```

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: *HACE QUE LAS EXCEPCIONES NO SEAN PRECISAS XQ LAS POSTERIORES A LAS QUE HAN CAUSADO EXCEPCIÓN YA HAN TERMINADO.*



TIPOS DE DEPENDENCIA

Instruction Level Paralelism (*ILP*)

Una Unidad segmentada sin operaciones multiciclo → pocos ciclos de parada → $CPI \approx 1$

Una Unidad segmentada con operaciones multiciclo → muchos ciclos de parada → $CPI \gg 1$

Pero las operaciones multiciclo son necesarias, por lo que hay que hacer algo. La Posibilidad de solapamiento en las secuencias de instrucciones Depende de si las instrucciones son independientes. A esto se le llama *ILP*

$$ILP \uparrow \rightarrow pocos\ conflictos \rightarrow pocos\ ciclos\ de\ parada \rightarrow CPI \downarrow$$

Independencia de instrucciones: **Dos instrucciones son independientes** si pueden ejecutarse simultáneamente sin ningún problema ⇔ Se pueden reordenar. *Tipos de dependías: De datos, de nombre, de control.*

Si las **instrucciones son indepes** el ILP es Alto, si **no lo son** (**hay dependencias**) ILP bajo.

Dependencias

Dependencia de Datos *Sí, otra vez*: Dadas dos instrucciones $i -> j$, **j** ejecutándose **después de i**. Tienen dependencia si **j lee** donde **escribe i** (*se tiene que esperar*).

Dependencia de Nombre *Antidependecia y Dependencia de salida*: Dadas dos instrucciones $i -> j$, **j** ejecutándose **después de i**.

- **Antidependecia**: Si i lee de x y j escribe en x es una ANTIDEPENDENCIA (*si no haces nada está okey, pero si lo reordenas puede cambiar y que j escriba en x antes de que i lo lea, sobrescribiendo el valor*).
  En el que tenemos nunca pasa pero en tomasulu sí.
- **Dependencia de salida**: Sí los 2 escriben en x es una DE SALIDA (*si no haces nada okey pero si cambias el orden y ahora i escribe después de j el valor de x será el de i en vez del que yo quería, j*)

Dependencia de Control: Determina la ordenación de algunas instrucciones respecto a un salto las cuales hay que ejecutar antes la instrucción que ejecuta el salto.

ILP ↓ → Presencia simultánea de instrucciones dependientes en la unidad segmentada → (posibles) riesgos → (posibles) ciclos de parada → CPI ↑

Dependencia:	*Riesgo:*
De datos	RAW
Antidependencia	WAR
De salida	WAW
De control	De control

TÉCNICAS PARA AUMENTAR ILP

El objetivo es el de aumentar el ILP de las instrucciones que están 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.

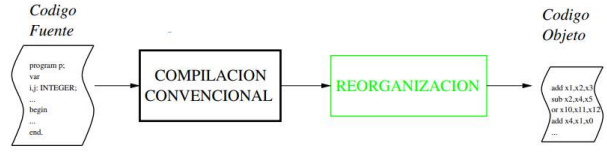
Bloque básico { beqz t1, L / instr / instr / instr / ... / beqz t1, L

De normal hay como 6 o 7 instrucciones en cada bloque, y de normal suelen tener dependencias. Hay que ejecutar en paralelo instrucciones procedentes de distintos bloques básicos para poder explotar al máximo el ILP.

- **Gestión dinámica de instrucciones**: El hardware reordena las instrucciones en tiempo de ejecución.
- **Gestión estática de instrucciones**: El compilador reordena/modifica el código para evitar dependencias.

Gestión estática de instrucciones

El compilador reordena/modifica el código para aumentar ILP. Se pretende reduciendo/eliminando las dependencias. Se **Añade una fase para que el código compilado no tenga tantas dependencias**, usa las estrategias de abajo, una o la otra. Se usan el "**Loop Unrolling**" y el "**Software PipeLining**".



Loop Unrolling

Lo que se hace es **replicar el código base del bucle varias veces**, disminuyendo el numero de iteraciones a realizar e intercalando las instrucciones unas con otras para reducir los riegos. *La idea es tener más operaciones entre cada salto, de manera que pueda tener más instrucciones y que las pueda separar y reordenar como yo quiera*.

Cantidad de iteraciones a "desenrollar": **Máximo número de ciclos parada consecutivos + 1 (***entre 2 instrucciones***).** ***Si entre A y B hay 2, pero entre B y C hay 3, serían 3 + 1 = 4 iteraciones*.**

```
i = 0;
while (i<n) {
  z[i] = a + y[i];
  i = i+1;
}
```

→

```
i = 0;
while (i<n) {
  z[i] = a + y[i];
  z[i+1] = a + y[i+1];
  z[i+2] = a + y[i+2];
  z[i+3] = a + y[i+3];
  i = i+4;
} /* endwhile */
```

Sobrecarga reducida: **Reduce la sobrecarga** por las instrucciones de control de los bucles xq tienes menos iteraciones y por lo tanto menos saltos. *Los ciclos de parada por control son difíciles de evitar → los reduces.*

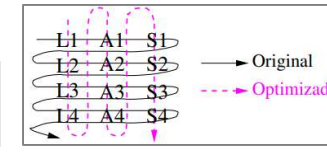
Renombramiento de registros: Hay que **renombrar registros**. *Antes todo usaban por ejemplo f2, Pues ahora uno f2, otro tendrá que usar f4….*

Cambio de incremento índice del bucle: Hay que cambiar el cómo aumenta el índice del bucle. Antes iba de 1 en 1. Ahora será $1 += num\ Unrolling$. *De igual forma ajustar los desplazamiento para que todos vayan bien.*

Podría ser que te quedes sin registros. Como tienes que renombrar los que usabas antes, puedes pasarte. También no puedes con todas las operaciones, por ejemplo como la división no es segmentable, el riesgo estructural te jode.

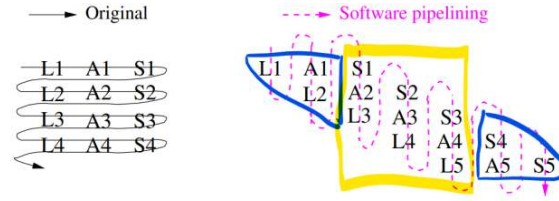
SE EVITAN PONER CICLOS DE PARADA, aumentando el tamaño del cuerpo del bucle

Antes se ejecutaban primero las intrucciones del bulce 1 en orden, luego las del 2nd…
Ahora se ejecuta primero la primera instrucción de todas las iteraciones, luego la segunda…



Software Pipelining *Segmentación Software*

Transformar un bucle con **instrucciones dependientes** e **iteraciones independientes** en otro bucle con **instrucciones independientes** e **iteraciones dependientes**:



Antes hacíamos instrucción uno ano de cada iteración, luego saltas y repites. Ahora lo haces más en escalera: *El nombre le viene por que realiza el procesamiento del bucle original simulando el comportamiento de una unidad segmentada.*

Triángulos de salida y entrada: Tienes un triángulo de inicio y otro de salida que se tienen que estructurar y en medio está una secuencia que se repite en cada iteración que son instrucciones que no depende entre ellas (*L3 no depende ni de A2 ni de S1. L4 ni de A3 ni S2…*).

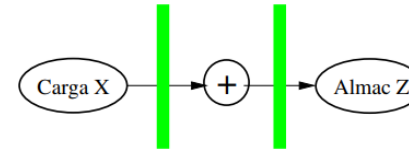
Simula una unidad segmentada de código: Los "**datos**" de la "**unidad segmentada**" **son las iteraciones**. *Para no sobrescribir los resultados intermedios, el procesamiento se realiza desde la última etapa hacia la primera*.

Tiene más sobre carga de bucles y es posible que antes de entrar al bucle y salir del bucle tenga algún ciclo de parada. *Sigue siendo mucho mejor que no hacer nada, pero loop unrolling es mejor.*

```
prebucle o triangulo entrada:
fld f2, 0(t1)       # Lee it. 0
fadd.d f4, f0, f2   # Calcula it. 0
fld f2, 8(t1)       # Lee it. 1
addi t1 ,t1, 16
```

```
loop:
fsd f4, 0(t2)       # Escribe it. i
fadd.d f4, f0, f2   # Calcula it. i+1
fld f2, 0(t1)       # Lee it. i+2
addi t1, t1, 8
addi t2, t2, 8
sub t4, t3, t1
bnez t4, loop
```

```
Resto o triangulo salida:
fsd f4, 0(t2)       # Escribe it. n-2
fadd.d f4, f0, f2 # Calcula it. n-1
fsd f4, 8(t2)       # Escribe it. n-1
```



Carga...	y[0]	y[1]	y[2]	y[3]	y[4]	...
Suma...		y[0]+a	y[1]+a	y[2]+a	y[3]+a	...
Almacena...			z[0]	z[1]	z[2]	...
It. bucle original		It. nuevo bucle	1	2	3	...
1	fld f2,0(t1)	fadd.d f4,f0,f2	fsd f4,0(t2)			
2		fld f2,8(t1)	fadd.d f4,f0,f2	fsd f4,0(t2)		
3			fld f2,0(t1)	fadd.d f4,f0,f2	fsd f4,0(t2)	
4	Preparación			fld f2,0(t1)	fadd.d f4,f0,f2	...
5	nuevo bucle				fld f2,0(t1)	...
...						...
Actualiza t1 (y)		addi t1,t1,16	addi t1,t1,8	addi t1,t1,8	addi t1,t1,8	...
Actualiza t2 (z)			addi t2,t2,8	addi t2,t2,8	addi t2,t2,8	...