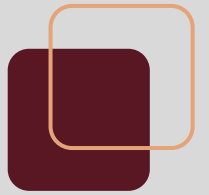
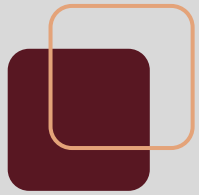


Aspectos avanzados de arquitectura de computadoras Pipeline II

Facultad de Ingeniería - Universidad de la República

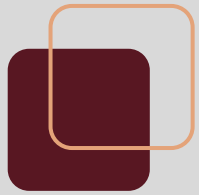
Curso 2017

Excepciones (1/5)



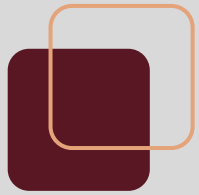
- Tipos de excepciones:
  - Externas, provocadas por hardware externo (interrupciones)
  - Internas:
    - Invocadas 'adrede' por el CPU (System Calls)
    - Provocadas por fallas (dividir por 0, ejecutar instrucción inválida, fallo de página)

Excepciones (2/5)



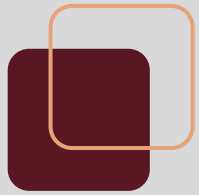
- En un procesador de ejecución secuencial, el manejo de interrupciones es simple. Luego de finalizada una instrucción y antes de iniciar la siguiente, se atiende la interrupción.
- En procesadores con pipeline, esto es más complejo, pues generalmente hay múltiples instrucciones en ejecución!

Excepciones (3/5)



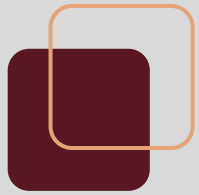
- ¿Cómo manejarlas?
  - En el caso de una interrupción externa, una solución 'simple' es guardar el PC de la última instrucción sin finalizar y reiniciar la ejecución desde ahí.
  - Es 'simple' si las instrucciones solo pueden finalizar en la última etapa del pipeline!
    - ¿Qué ocurre en el pipeline MIPS?

Excepciones (4/5)



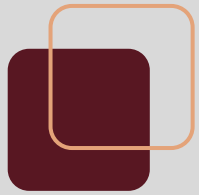
- En el caso de una interrupción interna (excepción), es más complejo.
  - La excepción puede ser provocada en cualquier etapa del pipeline (etapa EX para divisiones por cero, etapa IF o MEM para *page faults*, etc).
  - La ejecución no puede ser finalizada inmediatamente pues pueden haber instrucciones anteriores sin finalizar!
  - Una solución posible es propagar la señal de excepción, hasta que la instrucción finalice.

Excepciones (5/5)



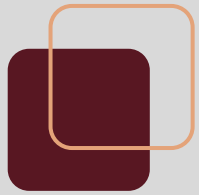
- En todos los casos anteriores, la complejidad mayor surge de no estar definido completamente cuándo *finaliza* una instrucción.
- Más adelante se verá que estos problemas se solucionan con la adición de una etapa de *commit*.

Precise Exceptions



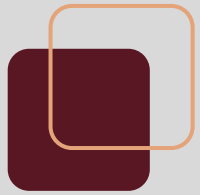
- Si al ocurrir una excepción, es posible detener el pipeline de modo que todas las instrucciones anteriores a aquella que provoque la excepción hayan finalizado, se dice que el procesador tiene *excepciones precisas*.

Cambio de contexto

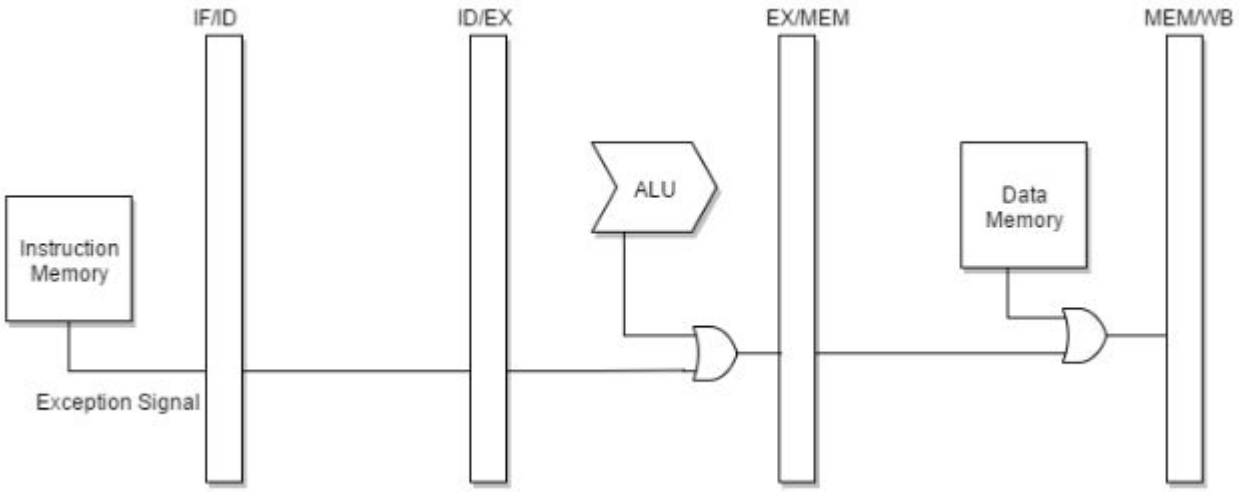


- Generalmente, el cambio de contexto en pipelines NO guarda absolutamente todos los valores de los registros de pipeline (costo), por lo tanto, al retomar la ejecución luego de una ISR, deben ser reiniciadas las ejecuciones de instrucciones que no habían finalizado su ejecución.
  - Esto provoca que las excepciones tengan un costo en performance más alto (aún) que en procesadores secuenciales!

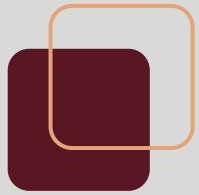
Excepciones en el Pipeline MIPS (1/2)



- El pipeline MIPS de 5 etapas puede implementar excepciones precisas con los siguientes agregados:
  - Se asocian señales de control a cada instrucción, que siguen el pipeline junto a la instrucción.

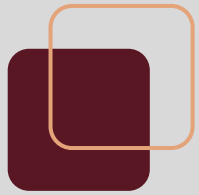


Excepciones en el Pipeline MIPS (2/2)



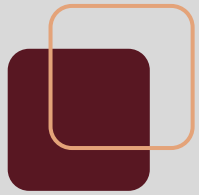
- Una vez que la bandera de excepción está activada, se detienen todas las posibles *escrituras* de la instrucción, de modo de evitar que impacten sus resultados. También se evitan las escrituras de instrucciones anteriores.
- Al llegar a la etapa WB, si la bandera está encendida, se procesa la excepción, realizando el cambio de contexto correspondiente.

Operaciones Multiciclo (1/11)



- Una simplificación introducida en el pipeline MIPS de 5 etapas es que todas las operaciones tienen la misma duración en etapa de ejecución.
- En la práctica esto no ocurre, ya que como mínimo se diferencian:
  - Operaciones de ALU enteras
  - Multiplicaciones/Divisiones enteras
  - Operaciones de ALU PF

Operaciones Multiciclo (2/11)



- Al agregar operaciones multiciclo, hay dos alternativas:
  - Alargar el ciclo de reloj para mantener el pipeline.
  - Separar las unidades funcionales como diferentes *paths* de ejecución, cada una con diferente cantidad de etapas de duración.

Operaciones Multiciclo (3/11)

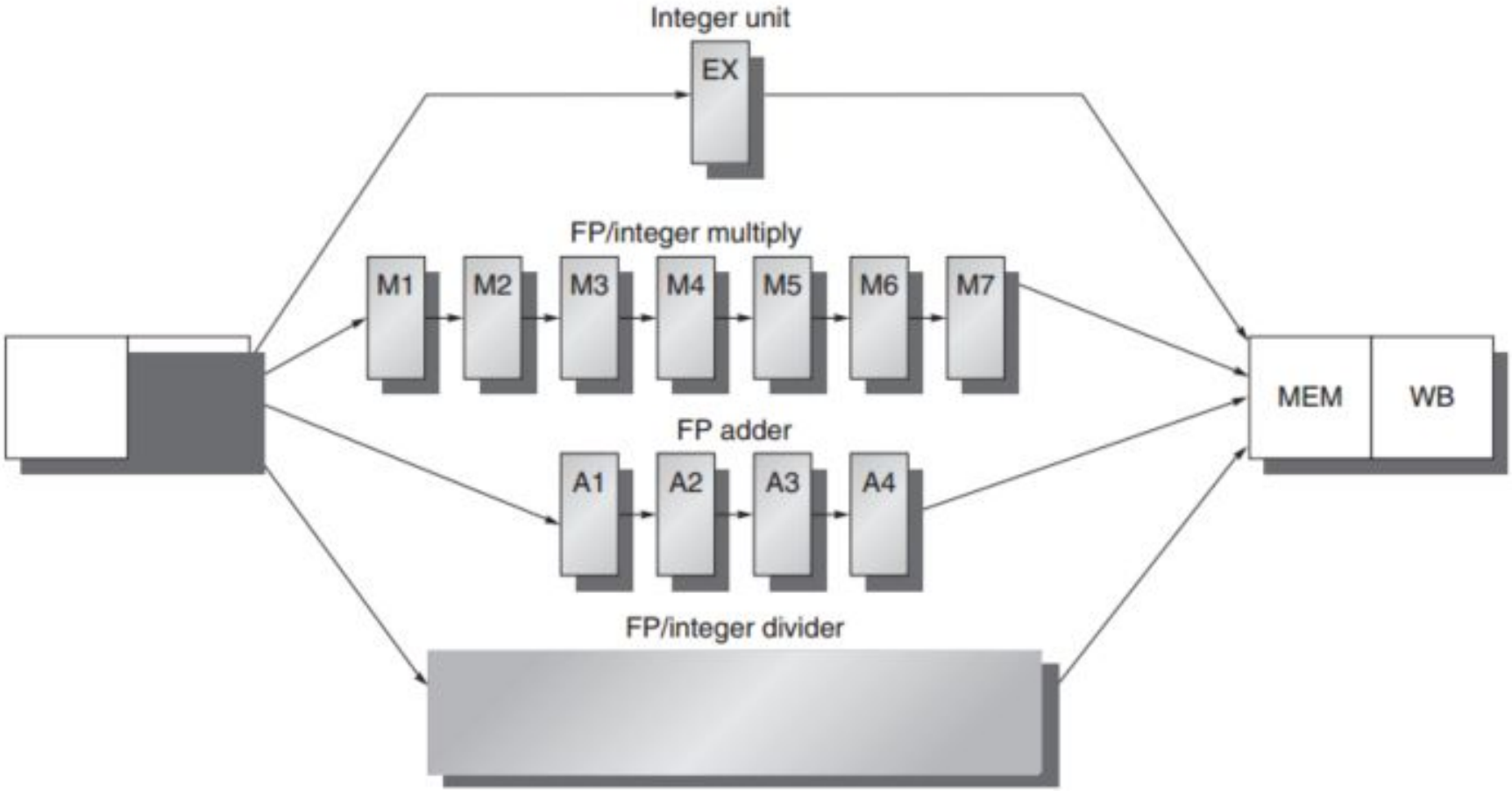
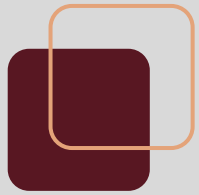


Imagen tomada del libro Computer Architecture, a Quantitative Approach. John L. Hennessy y David Patterson (2011)

Operaciones Multiciclo (4/11)



- El pipeline presentado tiene las siguientes unidades funcionales:
  - ALU entera (1 ciclo)
  - Multiplicador FP/Entero (7 ciclos, en pipeline)
  - Sumador FP (4 ciclos, en pipeline)
  - Divisor (24 ciclos, sin pipeline)

Operaciones Multiciclo (5/11)

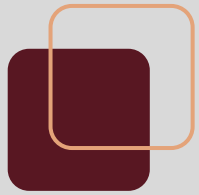
- ¿Cómo se ve la ejecución ahora?

MUL.D	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
ADD.D		IF	ID	*A1*	A2	A3	**A4**	**MEM**	WB		
L.D			IF	ID	*EX*	**MEM**	WB				
S.D				IF	ID	*EX*	*MEM*	WB			

- Como las unidades funcionales tienen diferentes latencias, las instrucciones pueden finalizar *fuera de orden*, esto es llamado *out of order completion!*

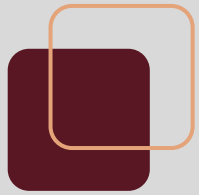
Imagen tomada del libro Computer Architecture, a Quantitative Approach. John L. Hennessy y David Patterson (2011)

Operaciones Multiciclo (6/11)



- Como los largos de las instrucciones son diferentes, es posible que múltiples instrucciones lleguen a la etapa MEM a la misma vez.
- Es posible tener hazards WAW dado que las instrucciones no terminan su ejecución en orden.
- Dado que el divisor no está en pipeline, pueden ocurrir hazards estructurales (múltiples divisiones deben esperar)

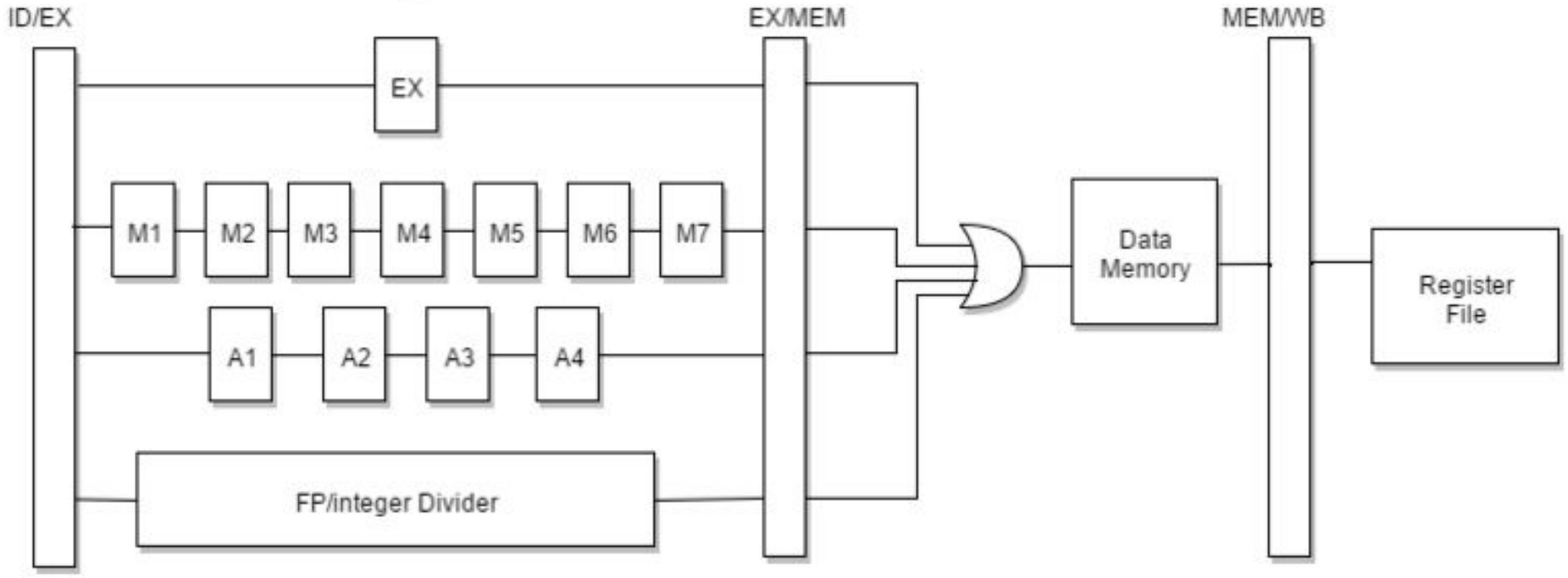
Operaciones Multiciclo (7/11)



- Dado que múltiples instrucciones pueden llegar a la etapa MEM a la misma vez, se debe decidir si a partir de ahí se multiplicarán las conexiones (permitiendo la finalización de múltiples instrucciones a la misma vez), o si se obligará a finalizar una instrucción por vez, obteniendo un *hazard estructural* en el caso de que varias instrucciones completen la ejecución a la vez.

Operaciones Multiciclo (8/11)

- Sin múltiples caminos:



- Se debe verificar en etapa ID que no ocurran hazards estructurales!

Operaciones Multiciclo (9/11)

- Ejemplo:

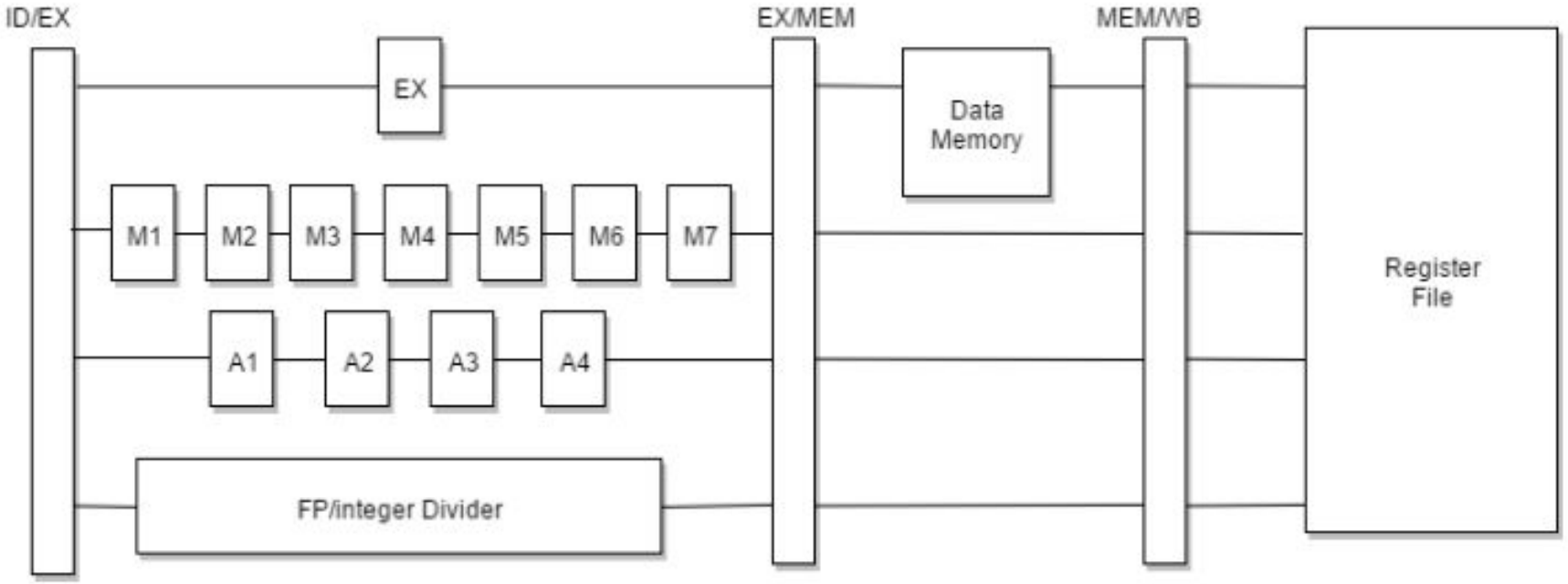
```
ADD.D F2,F0,F8
XOR    R1, R2, R3
OR     R4, R5, R6
ADD    R7, R8, R9
```

Instrucción\Ciclo	1	2	3	4	5	6	7	8	9
ADD.D F2, F0, F8	IF	ID	A1	A2	A3	A4	MEM	WB	
XOR R1, R2, R3		IF	ID	EX	MEM	WB			
OR R4, R5, R6			IF	ID	EX	MEM	WB		
ADD R7, R8, R9				IF	ID	ID	EX	MEM	WB

Hazard Estructural!

Operaciones Multiciclo (10/11)

- Con múltiples caminos:



- Requiere múltiples puertos de escritura en banco de registros pero NO presenta hazards estructurales.

Operaciones Multiciclo (11/11)

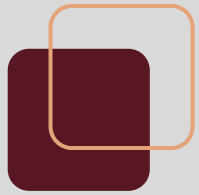
- Ejemplo:

```
ADD.D   F2,F0,F8
XOR     R1, R2, R3
OR      R4, R5, R6
ADD     R7, R8, R9
```

Instrucción\Ciclo	1	2	3	4	5	6	7	8
ADD.D F2, F0, F8	IF	ID	A1	A2	A3	A4	MEM	WB
XOR R1, R2, R3		IF	ID	EX	MEM	WB		
OR R4, R5, R6			IF	ID	EX	MEM	WB	
ADD R7, R8, R9				IF	ID	EX	MEM	WB

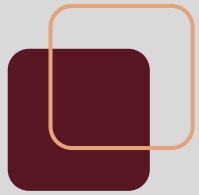
- Notar que hay múltiples instrucciones en etapas MEM/WB en el mismo ciclo!

Instrucciones Complejas (1/2)



- El caso extremo de la situación anterior se da con instrucciones *realmente* complejas de la arquitectura, como copia de strings o movimientos memoria-memoria.
- Adaptar el modelo de *pipeline* a estas instrucciones exige incluir unidades funcionales de decenas (o cientos!) de ciclos.

Instrucciones Complejas (2/2)



- Unidades funcionales largas impactan por varios lados:
  - Mayor cantidad de conexiones en *forwarding*.
  - Mayor densidad de instrucciones finalizadas fuera de orden
- Para evitarlo, la estrategia desde la arquitectura IA-32 en adelante ha sido poner en el pipeline las *microinstrucciones* generadas por la UC en lugar de las propias instrucciones.

Hazards WAW (1/3)

- La finalización fuera de orden introduce la posibilidad de que ocurran hazards WAW.
- Ejemplo:

```
MUL.D   F2,F0,F8
ADD.D   F2,F4,F6
```

Instrucción\Ciclo	1	2	3	4	5	6	7	8	9	10	11
mul.d F2, F0, F8	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
add.d F2, F4, F6		IF	ID	A1	A2	A3	A4	MEM	WB		

- Hazard WAW!

Hazards WAW (2/3)

- Para evitar el hazard, la dependencia puede detectarse en decodificación y detener el pipeline.

Instrucción\Ciclo	1	2	3	4	5	6	7	8	9	10	11	12
mul.d F2, F0, F8	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB	
add.d F2, F4, F6		IF	ID	ID	ID	ID	A1	A2	A3	A4	MEM	WB

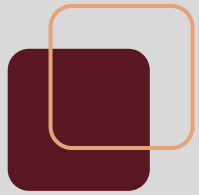
Hazards WAW (3/3)

- Una segunda opción es detectar el hazard WAW y propagar una señal que evite que la primer instrucción escriba sus resultados.

Instrucción\Ciclo	1	2	3	4	5	6	7	8	9	10	11
mul.d F2, F0, F8	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
add.d F2, F4, F6		IF	ID	A1	A2	A3	A4	MEM	WB		

- La instrucción pasa por WB, pero no actualiza sus resultados

Excepciones (1/3)



- El manejo de las excepciones se vuelve más complejo en los pipelines con finalización fuera de orden.
- Ejemplo:

```
DIV.D    F0, F2, F4
ADD.D    F10,F10,F8
SUB.D    F12,F12,F14
```

¿Qué sucede si la división obtiene una excepción de división por 0?

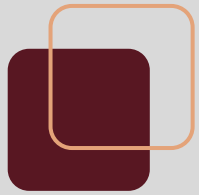
Excepciones (2/3)

- El manejo de las excepciones se vuelve más complejo en los pipelines con finalización fuera de orden.

Instrucción\Ciclo	1	2	3	4	5	6	7	8	9	10	11	12	13
div.d F0,F2,F4	IF	ID	DIV	DIV	DIV	DIV	DIV	DIV	DIV	DIV	DIV	DIV	...
add.d F10,F10,F8		IF	ID	A1	A2	A3	A4	MEM	WB				
sub.d F12, F12, F14			IF	ID	A1	A2	A3	A4	MEM	WB			

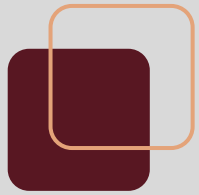
- Si se toma una excepción en este punto, las siguientes instrucciones no pueden ser revertidas!

Excepciones (3/3)



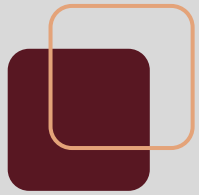
- ¿Soluciones?
  - No hacer nada: admitir excepciones imprecisas (obliga al programa a terminar).
  - Mantener un *buffer* con las operaciones finalizadas, impactando sus resultados una vez que todas las instrucciones que se despacharon antes finalizaron, obligando a las instrucciones a finalizar en orden.

Commit Stage (1/3)



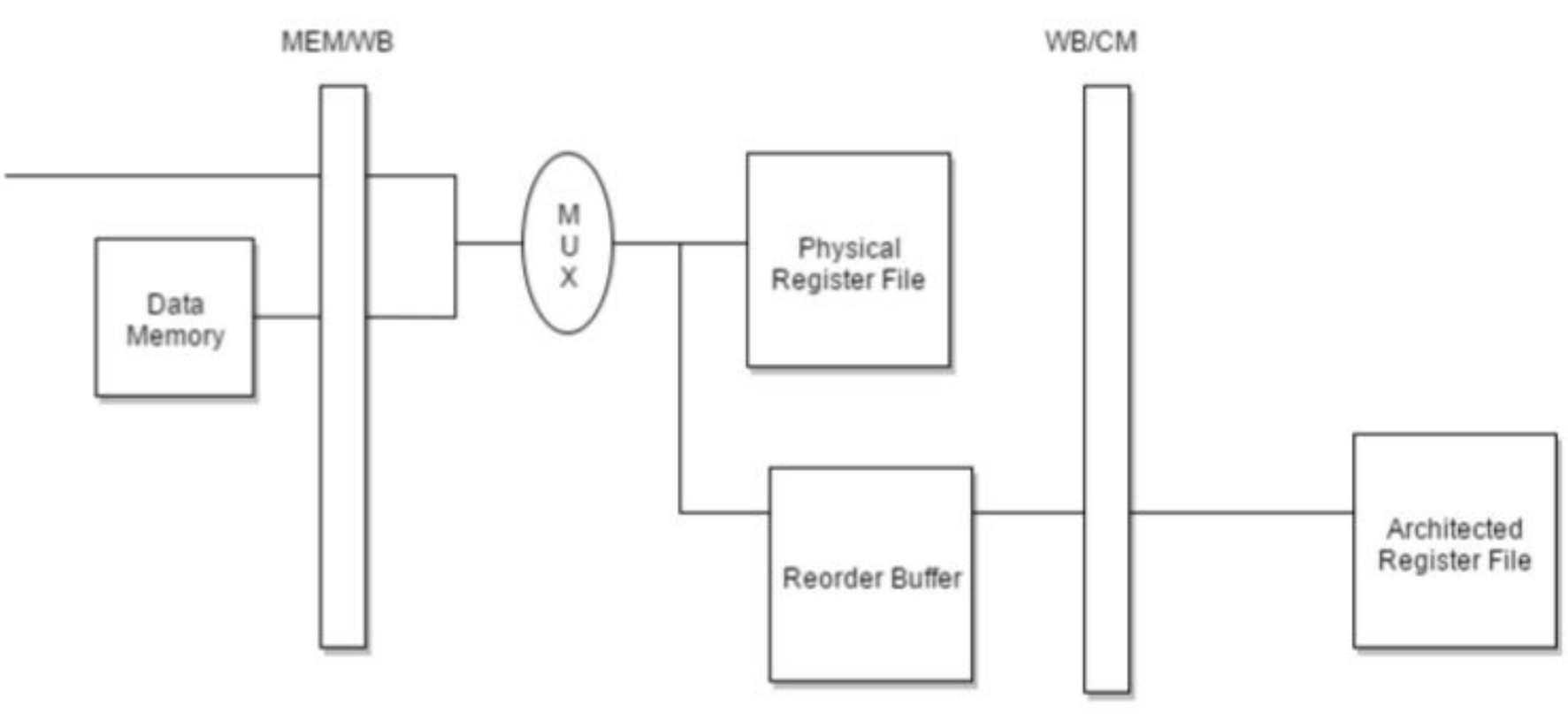
- La definición de *precise exceptions* es incompatible con la finalización fuera de orden.
- Para obtener excepciones precisas, se debe agregar una etapa adicional, denominada *commit*. Al llegar a dicha etapa una instrucción se considerará *finalizada*.

Commit Stage (2/3)



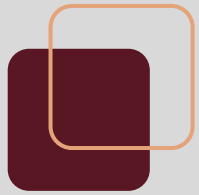
- Se agrega un nuevo banco de registros, el cual será actualizado en la etapa commit, y que tendrá únicamente el propósito de servir para restaurar el estado en caso de una excepción.
- De este modo, se diferencian el *banco de registros físico* (el usado hasta el momento), del *banco de registros de la arquitectura* (el de la etapa commit).

Commit Stage (3/3)



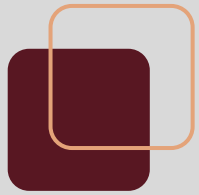
- El pipeline presentado tiene *ejecución fuera de orden* y *commit en orden.*

Reorder Buffer (1/3)



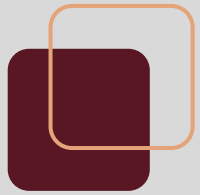
- Para asegurar el acceso en orden a la etapa commit, se puede utilizar una estructura de datos llamada *Reorder Buffer (ROB)*.
- El ROB es el encargado de detener instrucciones que finalicen fuera de orden, es decir, que finalicen cuando haya aún instrucciones anteriores según el orden del programa que no hayan finalizado.

Reorder Buffer (2/3)



- Un ROB es esencialmente una cola circular:
  - Al hacer issue una instrucción, se agrega al ROB con estado pendiente (*queue*)
  - Al llegar a la etapa WB se marca como finalizada.
  - Al final del ciclo, si la última instrucción del ROB está finalizada, se realiza el commit y se quita de la estructura (*dequeue*).

Reorder Buffer (3/3)



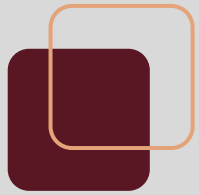
- Estructura:

Head →

Tail →

Instrucción	Estado
-	
-	
add R1, R3, R5	P
subi R3, R6, 500	P
xor R3, R5, R1	F
mul R5, R1, R5	P
-	
-	

Fin



¿Preguntas?