Dynamic Multiple Issue

Dynamic Scheduling

Static Scheduling (el que veniamos viendo)

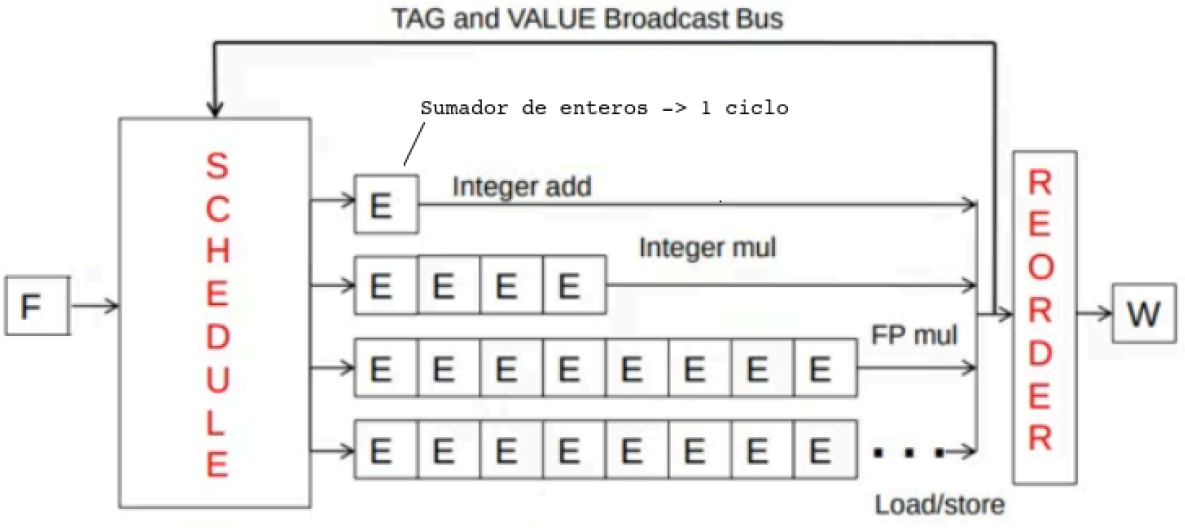
- Se captura una instruccion (o grupo de instrucciones)
- Se las ejecuta, a menos que haya una dependencia que no se pueda resolver
- => El pipeline se detiene (STALL)
- Ejecutabaos la siguiente la siguiente la siguiente
  - Lo unico que puede romper el esquema de ejecutar la siguiente es un salto
  - Pero saltamos y ejecutamos la que sigue la que sigue y la que sigue...
  - O sea seguimos el ritmo al que fue decodificado (Su orden)

Dynamic Scheduling (Solucion a ese problema del micro)

- Es una tecnica donde el hardware reordena la ejecucion de instrucciones
- => Reducimos los stalls al minimo posible
- Manteniendo el flujo de datos y exception behavior
- **Aca se respeta mas el flujo de datos que el orden**

Ejecucion OoO (Out-of-Order)

- Es muy comun que se separen las unidades funcionales segun las distintas latencias



1. Se hace fetch de la instruccion (en orden)

2. [SCHEDULE] -> Decodificar, resolver dependencias y asignarla a las distintas unidades funcionales
3. Se ejecutan las instrucciones (Fuera de orden)
4. Se escriben y se resuelven algunas dependencias <-

Algoritmo de Tomasulo

Aunque existen muchas implementacion
todas se basan en dos principios

- **Determinacion dinamica** - De cuando una instruccion esta lista para ejecutarse
- **Register renaming** - Para evitar WAW

Unidades funcioles (FU)

- Hardware que se encarga de realizar la operacion de la instruccion
- Ej: Memory, ALU, FP, ALU, multiplicador, etc.
- **Cada uno de ellos puede tener distinta latencia** (otra puede demorar menos o mas ciclos de clock)

Reservation Stations (RS)

- Otros registros asociados a las **FU**
- Contiene la operacion y operandos necesarios
- Posee 7 campos
  -

Name	Op	Qj	Vj	Qk	Vk	A	Busy

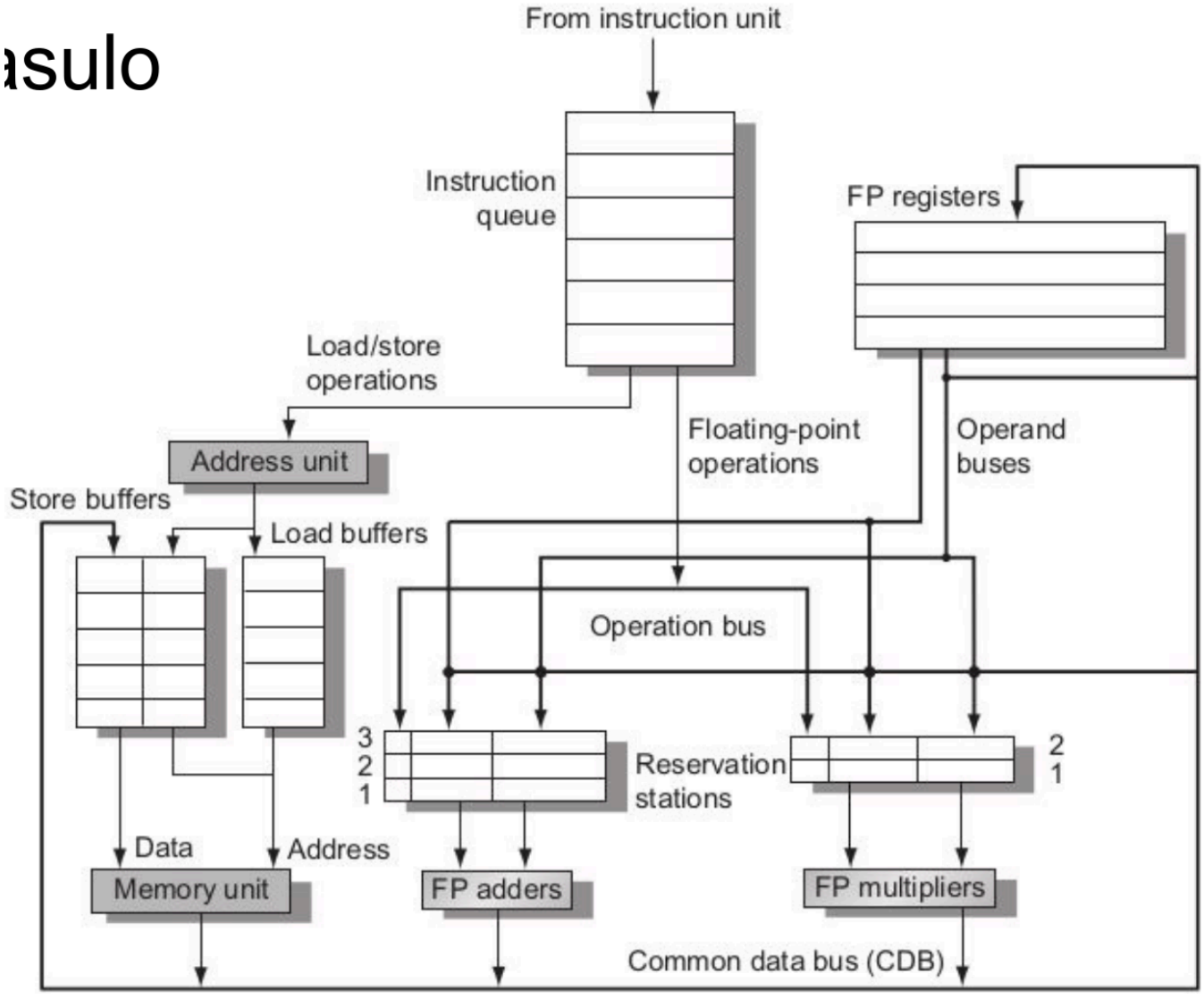
  - Op -> Operacion que tiene que haces
  - Vj - Vk -> el valor de los operandos
  - Qj - Qk -> el tag de la RS asociada a la FU - Si algun operando de ambos no esta disponible se usa
  - A -> Contiene el address a donde se tienen que hacer los LDUR y STUR
  - busy ->Si 0 esta en espera es | Si 1 la FU esta calculando la instruccion

Idea

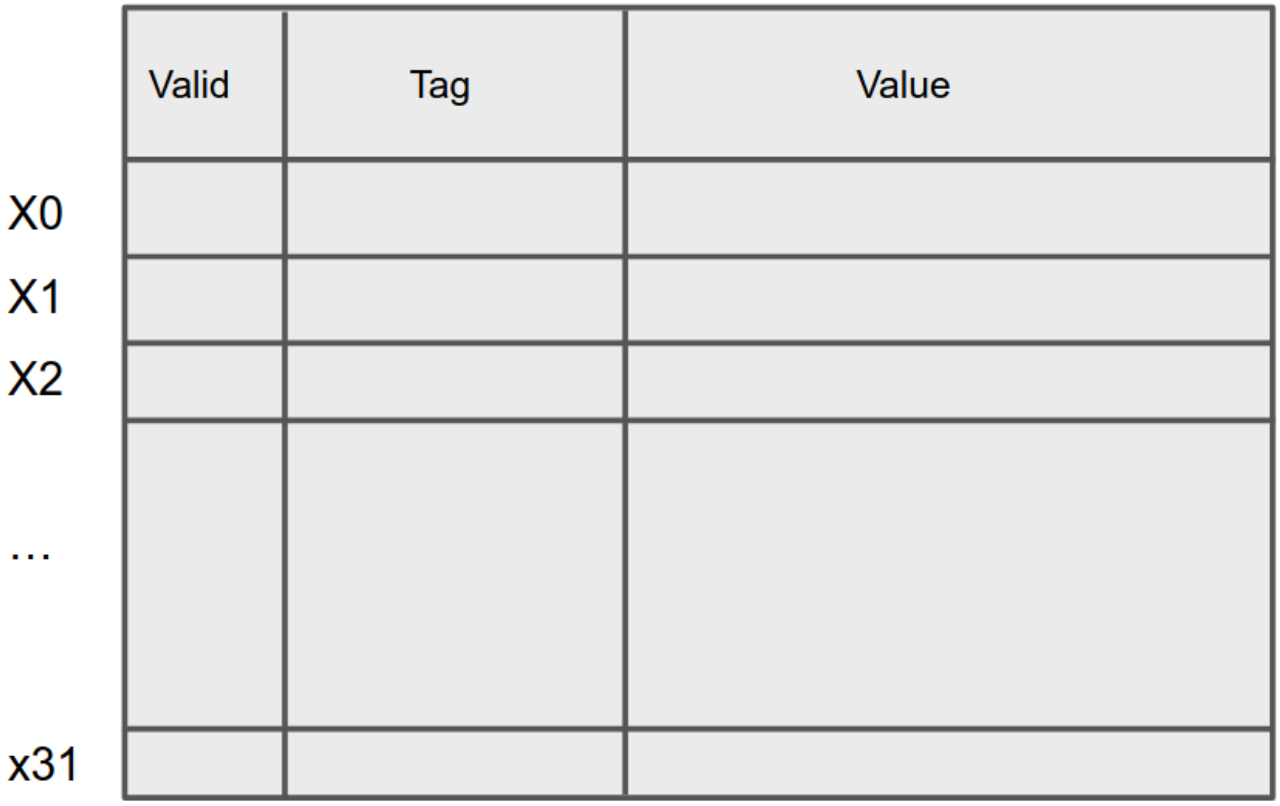
- Capturar ("Bufferearlos en los **RS**") los operando tan pronto como esten disponibles
- Ejecutar la operacion
- O sea no recurrimos al regfile (a los registers)
- **Dependencia**
  - Cuando hay hazard de datos en lugar de poner en el RS el valor

    - Ponemos una ID que representa el RS donde se va a guardar el operando necesario
    - Es como un puntero al hardware que va a producir ese resultado
    - => Ademas de escribirlo en los registros lo escribe en esa **FU** que necesita ese valor
- Si el RS tiene dos operaciones cual ejecuta primero?
    - La **FU** elige cualquiera instruccion para ejecutar
    - Y seguiria funcionando perfectamente
- Entonces una vez que se hace Issue
    - Deja de tener importancia el orden de las instrucciones
    - **Y tiene importancia el flujo real de los datos**
    - => Una instruccion que no necesita esperar datos se ejecuta fuera de orden y listo

Esquema



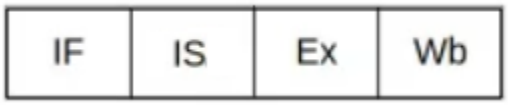
Register file



- Arriba se tiene el Fetch y las va metiendo en la Instr Queue y las va metiendo en orden
- Cuando se esta por pasar la instruccion revisa si las RS tienen algun espacio libre
- Se ve que cada FU tiene asociado un RS
  - **Address unit** -> Calcula la direccion de memoria a la que se tiene que acceder
  - **Memory Unit** -> LDUR/STUR
  - **FP Adders** -> Suma de punto flotante
  - **FP multipliers** -> Multiplicacion de punto flotante
- El de memoria tiene un RS particular que son bufferes pero los vemos iguales a los RS comunes por simplicidad
  - **Por lo tanto los LOAD y los STORE siempre se van a ejecutar en orden** entre ellos
  - **Si hay una dependencia de datos y un STUR no se puede ejecutar se frenan todos los accesos a memoria**
  - Pero se pueden dar hazard de datos a traves de la memoria
  - Es dificil de detectar
  - Por lo menos debemos calcular el address
  - Para saber efectivamente a que direccion estoy accediendo

    - Pero basicamente los Load Store
- Y se tiene los registros (RegisterFIle)

Funcionamiento Tomasulo - 4 etapas



- **Fetch**
    - Se traen las instrucciones de memoria y se colocan en una Queue(FIFO)
    - O sea en clk = 0 se fetchea la primera instruccion
- **Issue**
    - Puede ser issue de 1 o 2 instrucciones lo que diga el hardware
    - Se hacen decode de las instrucciones
    - **Se guarda la info en el RS correspondiente y sobre los registros**
    - **Si RS con espacio libre** => se le pasa la operacion y los operandos (si estan disponbiles - sino añadir el ID de la RS que lo va a producir)
    - **Si RS sin espacio libre** => Stall hasta que se libere (Bloqueando el issue para otros)
- **Execute**
    - **Address** - Solo para **STUR Y LDUR**
        - Para estos en la fase de execute se calcula el Address X + # offset
        - Tarda un ciclo
        - Luego de ese ciclo pueden ser pasadas al FU
    - **Si hay dependencia de datos** => **Stall hasta el siguiente ciclo del WB del dato esperado**
    - **Si la FU esta ocupada** => STALL hasta que se libere
    - **Si estan los datos** y esta el **FU libre=> Busy = 1 en la RS** y esperar los ciclos que corresponden
- **Write back**
    - Una vez terminado los ciclos del FU escribir los datos en los registros y los RS que necesitaban el dato

Ejemplo del funcionamiento

1. Cuando una instruccion esta lista para Issue (para mandarse) ADD X0, X0, X1
2. Procesador se fija si hay un lugar libre en la RS
3. Si lo hay pone en el RS la operacion a realizar y traer del regfile los operandos
    - Operacion y Operandos

4. El valor destino le va a poner un bit indicando que ese registro no tiene un valor actualizado en el register file
    - Valid - 0 Para decir que el valor no esta actualizado
    - Tag - Id del RS donde se asigno esa instruccion
    - Value - Valor del registro
5. Entra otra instruccion que necesita de X0
6. => Guarda en el RS la operacion
7. => Busca los operandos pero X0 sale que no esta actualizado
8. => En lugar de poner el Operando se pone el campo tag de ese registro
9. => Espera a que se ejecute el ADD
10. Se resuelve el ADD
11. El Common data Bus ->
    1. Cambia el valor de x0 y pone el bit de validacion en 1
    2. **y ademas escribe el valor de X0 en el regfile y en todos los RS donde estaba el tag**
12. Entonces todos las operaciones que necesitan X0 lo van a tener disponibles

- **Dependencia de datos en out-of-order pipelines** (RAW - WAW - WAR - CONTROL - COND)
    - **RAW** - 1 instr requiere el dato de otra instr
    - **WAW** - 2 instr que escriben sobre el mismo registro
        - Por ejemplo
        - ADD X0, X0 ,X1 -> ADD X2, X0, X2 -> ADD X0, X10, X1
            1. Se guarda en la RS[1] la primera operacion y sus operandos
            2. Se guarda en la RS[2] la segunda operacion y una tag=1 al RS del primero para X0
            3. Entra la tercera de nuevo escribe X0, se guarda la tercera los operacion y los operandos pero ningun y se sobreescribe el tag de X0 en el regfile
                - O sea que se va a tener el valor mas actualizado de X0 al final
            - Pero el tag de la RS de la segunda operacion no cambio => tendra el valor que necesita de X0 de la primera instruccion
        - Eso me dice que el algoritmo naturalmente ya resulve los WAW ya que la instruccion del medio no se le fue su tag original
    - **Antidependencia WAR** - Leer un registro y la instr que sigue lo escribe
        - Es tipo WAW pero se me actualiza el valor del registro y yo queria el anterior

```
add X0, X1, X2
add X1, X3, X4
```

```
add X0, X1, X2
ldur X1, [X3, #0]
```

-
  - Pues como es out of order puede pasar que se actualize el valor antes
  - Necesitamos que el hardware sea capaz de analizarlo y solucionarlo
  - No es RAW ojo
- **Dependencia estructural**
  - Si no hay FU libre, => bit de busy = 0 se espera
- **Dependencias de control**
  - Todas aquellas instrucciones cuya ejecucion depende de un salto

```
1>        sub x2,x3,x4
2>        cbnz x2,L1
3>        ldur x1, [X2, #0]
4>   L1:
```

  -
  - En este caso out of order si no consideramos que hay instrucciones cuya ejecucion depende de si un salto no se toma o se toma
- **Dependencia de datos condicionales**
  - EL orden del programa es el que establece que predecesor genera el dato necesario para la instruccion

```
1>        add x1,x2,x3
2>        b.eq x4,x0,L
3>        sub x1,x5,x6
4>   L:   ...
5>        or x7,x1,x8
```

  -

Resumen de hazard respecto a las arquitecturas

Tipo de dependencia	In-order (1-issue)	Multiple issue in-order	Multiple issue out-of-order
Estructurales (Hazard)	No*	No*	Puede provocar stalls
Control	Puede provocar stalls	Deben considerarse al armar los issue packet. Puede provocar stalls	Deben considerarse al reordenar el código. Puede provocar stalls
Datos	RAW	RAW, WAW	RAW, WAW, WAR

*Pueden existir por criterios de diseño, debe especificarse el hardware y el problema en particular

RAW: Puede evitarse con stalls o forwarding
WAW: Puede evitarse con register renaming

Hazard Estructurales: En la ejecución out-of-order, puede ocurrir que una instrucción requiere una unidad funcional (ALU, memoria, etc) que está ocupado ejecutando otra instrucción.

Algoritmo Tomasulo - Planila para datos

Para ejercicios que se ejecuten en estos procesadores OoO y poder representar los valores de la ejecucion usamos una planilla del siguiente tipo:

Instruction	Iteration	Instruction status		
		Issue	Execute	Write result
1> LDURD D6, [X2, #32]		0	0	0
2> LDURD D2, [X3, #44]		0	0	0
3> FMULD D0, D2, D4		0	0	0
4> FSUBD D8, D2, D6		0	0	0
5> FDIVD D0, D0, D6		0	0	0
6> FADDD D6, D8, D2		0	0	0
		0	0	0
		0	0	0
		0	0	0
		0	0	0
		0	0	0
		0	0	0
		0	0	0
		0	0	0

Hardware
Issue = 1 instrucción
Load = 6 RS / 1 clk
Store = 6 RS / 1 clk
Suma punto flotante = 3 RS / 2 clk
Multiplicación punto flotante = 2 RS / 6 clk

Name	Reservation stations						
	Busy	Op	Vj	Vk	Qj	Qk	A
load 1	0						
load 2	0						
load 3	0						
store 1	0						
store 2	0						
store 3	0						
FP alu 1	0						
FP alu 2	0						
FP alu 3	0						
FP mult 1	0						
FP mult 2	0						
	0						

Register Status									
	D0	D1	D2	D3	D4	D5	D6	D7	D8
Qi									
	X0	X1	X2	X3	X4	X5	X6	X7	X8
Qi									

Especificacion del hardware

Hardware

- Se especifica el hardware
- Los FU, sus RS y los ciclos de procesamiento

Estado de la instruccion

Instruction	Iteration	Instruction status		
		Issue	Execute	Write result
		0	0	0
		0	0	0
		0	0	0
		0	0	0
		0	0	0
		0	0	0
		0	0	0
		0	0	0
		0	0	0
		0	0	0
		0	0	0
		0	0	0
		0	0	0
		0	0	0

- Se hace para ver en que etapa se encuentran las instrucciones
- Se tiene la columna de iteracion por si hay un Loop se hace como ver en la iteracion en la que se va
- Si alguan de las etapas tiene 1 significa que ya se paso por esa etapa

Reservation Stations RS - representan todas pero con lineas suficientes para el ejercicio

Name	Reservation stations						
	Busy	Op	Vj	Vk	Qj	Qk	A
	0						
	0						
	0						
	0						
	0						
	0						
	0						
	0						
	0						
	0						
	0						
	0						
	0						

- El name es la ID de la RS
- Si tenemos por ejemplo 3 RS para un FU => ponemos en 3 lineas dedicado para ese RS
- Todos con los campos de la RS

Estado de los registros

Register Status									
	D0	D1	D2	D3	D4	D5	D6	D7	D8
Qi									
	X0	X1	X2	X3	X4	X5	X6	X7	X8
Qi									

- Por cada registro se pone el RS ID de donde se va a generar ese resultado

Aclaraciones

La idea es actualizar la planilla por cada ciclo que pasa en el procesador

Ciclos

=> Si el procesador le tarde 6 ciclos terminar de ejecutar

=> Necesitas llevar 6 planillas de algoritmo Tomasulo

Inicio de ejecucion

=> Se asume que por ejemplo en el clock i se fetcheo una instruccion y en el siguiente clock i+1 se hace issue

Campo de address

=> Calcular el address siempre lleva un ciclo

=> Luego de un ciclo se actualizaria el campo A del RS al resultado de [X?,#offset]

=> X? + OFFSET

=> Se considera que es parte de la etapa execute esto pero no que se accedio a una FU

Campo de los registers

=> El campo de cada registro ponemos el ID del RS para decir que el valor no es valido

=> O sea que esta desactualizado

Stall y WB

=> Si una o varias instr estan esperando un operando

=> No pasan a su FU en el mismo ciclo de wb

=> Sino en el siguiente

Branchs

=> Cuando sigue una instruccion de branch se fetchea esa unica instruccion

=> y si bien se fetchean las que siguen

=> No podran acceder a issue

=> Hasta un ciclo despues del wb de la instruccion Branch (QUE HACE ALUCION A TOMAR O NO TOMAR EL SALTO)