EMA | Dynamic Scheduling (Unidad 7)

Material importante

Clases

- Carpeta de clases: https://drive.google.com/drive/u/1/folders/10weMj2M6tQZ2Lr0OFWIIZoBTb4GMhhU0
  - Unidad 7: Dynamic Scheduling
    1. https://drive.google.com/file/d/1JhJXtHCnxXwkJg7laC8VPA1XzEMzPDC_/view?usp=sharing
    2. https://drive.google.com/file/d/1RtSs4-cvS7udi9qvT8Z0tuDmwkNnTUqn/view?usp=sharing

Filminas

- Unidad 7: Dynamic Scheduling
  - Filmina de profes: https://famaf.aulavirtual.unc.edu.ar/pluginfile.php/28612/mod_resource/content/0/DynamicScheduling.pdf
  - Práctico: https://famaf.aulavirtual.unc.edu.ar/pluginfile.php/28733/mod_resource/content/0/TP7_

Algoritmo de Tomasulo.pdf

- Template de Tomasulo (qué nos van a dar para resolver los ejs.): https://famaf.aulavirtual.unc.edu.ar/mod/resource/view.php?id=8824

Notas

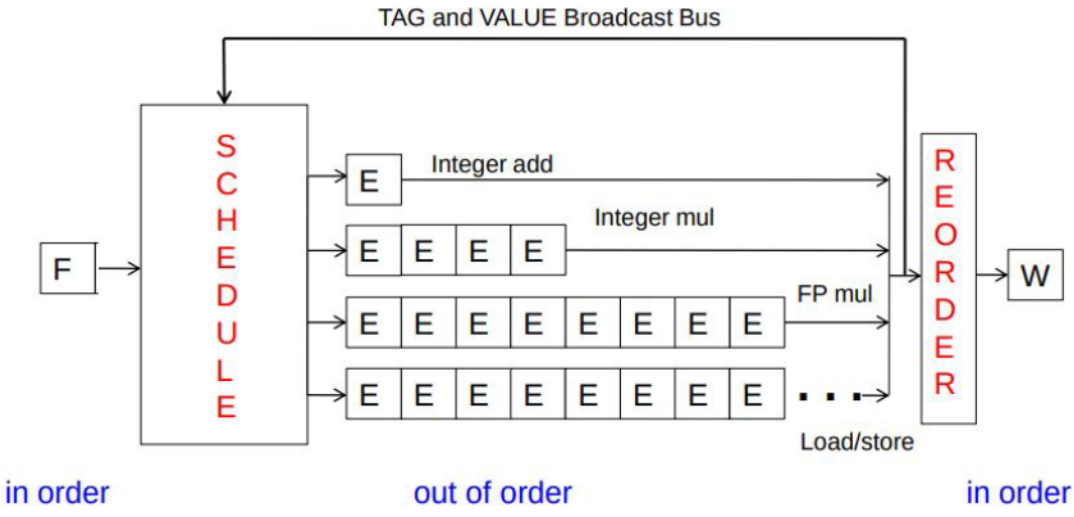
> 💡 Es un procesador **nuevo**, dado que todos los que estuvimos viendo hasta el momento se llaman / consideran de tipo *in order execution*, mientras que este va a ser *out order execution.*

Respecto a lo mencionado anteriormente, consideramos los siguientes dos tipos de *Scheduling:*

- Static Scheduling: se captura una instrucción (o un grupo de instrucciones) y se la ejecuta, a menos que haya una dependencia que no se pueda resolver, en ese caso, el pipeline se detiene (*stall*)
- Dynamic Scheduling: es una técnica donde el *hardware* **reordena** la ejecución de instrucciones para reducit los *stalls* manteniendo el flujo de datos y *exception behavior* (i.e., comportamiento de excepciones).

Máquina OoO (Out-of-order)

- Tenemos el siguiente esquema de nuestro *nuevo hardware* sobre el *scheduler*:



- Lo que hace es ir tirando las instrucciones a las distintas unidades funcionales y “retener” los resultados momentaneamente cuando cada una termina, y luego va reordenándolas al final para escribir.
- Esquema de funcionamiento:

- El FETCH se hace en orden
- Las instrucciones se despachan en orden
- Se ejecutan las instrucciones *fuera de orden*
- Se reordena
- Y al final se hace un *Write*

Algoritmo de Tomasulo

- Algoritmo que se utiliza para la realización del *Scheduler*
- Hay muchas formas de implementarlo, pero en gral se basa en dos principios fundamentales
  - **Determinación dinámica** de cuando una instrucción está lista para ejecutarse
  - **Register renaming** para evitar dependencias de nombre (de dato WAW)

Hardware necesario que se va a utilizar

- Para su implementación, vamos a considerar dos tipos de hardware *principales*:
  - **Unidades Funcionales (FU)** → Hardware que se encarga de realizar la operación propiamente dicha
    - Ej.: memoria, ALU, FP ALU, Multiplicadores, etc.
    - Cada uno de estos puede tener distinta latencia
  - **Reservation Stations (RS)** → Es básicamente *memoria*
    - Tenemos un grupo de RS asociadas a un grupo de FU del mismo tipo, la cual va almacenando los operandos a medida que estén disponibles para que, apenas se pueda hacer la operación, se realice efectivamente
      - !!! Nota → Contiene el *valor real* de los operandos involucrados. Es decir, si tengo que hacer 4 + 2 sumando dos registros, los valores que vamos a considerar que tenemos son 4 y 2, no punteros a los registros o a memoria
    - Cuando una FU termina de resolver un dato, lo va a escribir en los registros, pero también lo va a hacer en todas las otras RS que están esperando ese resultado (como una especie de *forwarding* que se hace directamente)
      - La idea del algoritmo es capturar y *bufferear* los operandos tan pronto como estén disponibles con el objeto de no tener que recurrir a los registros
      - En el caso en que existan instrucciones con dependencia de datos, se designa la RS que producirá el operando necesario
    - ¿Qué tienen adentro las *reservation stations*? Vamos a considerar que tienen estos 7 campos (donde nuestros operandos son j, k):

Name	Op	Qj	Vj	Qk	Vk	A	Busy

        - Op → operación a realizarse entre los operandos
        - Qj, Qk → Tag de la RS asociada a la UF que va a producir el operando necesario
            - Un valor de cero indica que el operando ya se encuentra disponible en Vj o Vk
        - Vj, Vk → Valores de los operandos
            - En las instrucciones *load*, Vk se utiliza para el *offset* !!!
        - A → contiene el resultado del cálculo de dirección para los *load* o *store*
        - Busy → *un sólo bit* que indica que se está generando este resultado en esta FU
- A su vez, para los *registros* de nuestro **micro**, vamos a agregar nuevos campos los cuales son:

    - Tag → Indica qué RS va a calcular el valor que se va a colocar en este registro
        - Es 0 si el resultado de ninguna instrucción activa tiene como destino este registro
    - Valid → *un sólo bit* que indica si el valor que está en el registro es válido o se está generando y debe esperar a que se calcule en la FU

Nuevas etapas de instrucción

- **FETCH**
    - Se levantan las instrucciones de memoria y se colocan en una *FIFO*
- **ISSUE y DECODE**
    - Se decodifican las instrucciones y si está disponible la RS correspondiente, se le pasa la instrucción y los operandos (si están en registros).
    - Si no hay una RS disponible, la instrucción debe esperar (*stall*) hasta que se libere → Implica hazard estructural
    - Si los operandos no están en los registros, se linkea con la RS que lo producirá y la instrucción queda en espera hasta que pueda ejecutarse
- **EXECUTE**

    - Cuando todos los operandos están listos, la operación puede ejecutarse en la FU correspondiente
    - Los *loads* y *stores* requieren un **orden**, por lo que sus RS sí van a implicar una *FIFO*. Además, requieren que primero se calcule la dirección efectiva para luego acceder a la memoria → (esta se almacena en el *buffer* correspondiente en orden de ejecución del programa, previniendo *hazard* en la memoria)
        - Es una *FIFO* porque no queremos que haya dependencia de datos en la memoria y arreglarlo es bastante *complejo*
        - El hecho de *calcular el address* primero implica que se va a usar SIEMPRE un clk para esto antes de realizar el *load* o *store* correspondiente.
- **WRITE RESULT (o Write Back)**
    - Cuando el resultado está disponible, se utiliza el CDB (c*ommon data bus*) para escribirlo en los regitros y en las RS que lo necesitan
    - Básicamente, lo que hace es que cuando se terminó de ejecutar una instrucción y se tiene un resultado nuevo, se escribe en los registros y en todas las RS que están esperando ese resultado → se escribe el valor tal cual (⇒ las RS no contienen punteros a memoria o a registros, sino que directamente contienen los valores)
    - La **magia** es que la escritura en los registros y en las RS que se necesitan se debe hacer en *un ciclo de clock*. !!!

Dependencias de datos en OoO (out-of-order) pipelines

- **Real dependencia de datos ⇒ RAW**
    - La instrucción i1 es dependiente en datos con i2 cuando i1 produce un resultado que debe ser utilizado por i2
- **Dependencias de nombre**
    - Dos instrucciones usan el mismo registro o posición de memoria, pero *no hay flujo real de datos*
    - Tipos:
        - **Dependencia de salida** → Cuando i1, i2 escriben la misma posición de memoria o registro ⇒ **WAW**
            - El orden original se debe preservar para asegurar que el valor final se corresponda con el valor de i2
        - **Antidependencia** → Cuando i2 escribe un registro o posición de memoria que i1 necesita leer ⇒ **WAR**
            - El orden original se debe preservar para asegurar que i1 lea el dato correcto
    - Solución → *register renaming*
        - Se implementa a partir de los Tags de los RS

        - Una vez que se despachó una instrucción, no me importa qué registro se utilizó, porque para el operando voy a poner, o bien su valor real, o bien el Tag a la RS que lo va a generar ⇒ Estoy haciendo *register renaming* porque “me olvido” de los nomrbes que estoy poniendo
            - En el caso de los WAW, como las RS de los *load* y *store* son FIFO, entonces los valores de las Tags de “pisan” en orden, por lo que siempre queda la Tag a la RS que va a generar el resultado final a guardar ⇒ De aquí la importancia de que las RS mantengan orden
            - Es decir, si varias escrituras se superponen en ejecución, solo la última se utiliza para actualizar el registro
- **Dependencia estructural**
    - Si todos los operandos están listos pero no hay una FU libre, se debe *esperar*.
- **Dependencia de datos condicional (o *dependencias de control*)**
    - Ahora, en base a que luego de que se generan las *issue* ya perdemos el *orden* de ejecución de nuestro programa, la dependencia de control que vamos a considerar respecto al salto va a ser la dependencia de qué instrucciones se tienen que ejecutar respecto al salto

      💡 Para **resolver los ejercicios**, vamos a hacer lo siguiente (para *simplificar*):
      - Una vez que tengo un branch, stalleo el *issue* de instrucciones
      - Es decir, no voy a hacer más *issue* de instrucciones hasta que ese salto sea resuelto

      !!! Otra cosa **importantísima** → Vamos a considerar que nuestro predictor es tan bueno que no tenemos considerar casos de *flush* de las instrucciones incorrectamente predichas.
      Además, vamos a considerar que el *FETCH* siempre tiene listas las instrucciones *correctas* para hacer las *issue* (i.e., no tenemos que esperar un clk).

        - Esto nos permite que cuando hagamos *issue*, se haga directamente sobre las instrucciones correctas
    - Respecto a esto, **ver el *Ejercicio 2*** para saber cómo se comporta el algoritmo de *Tomasulo* con branchs.

      💡 En este caso, se coloca en *Vk* el inmediato que indica cuántas instrucciones se saltan desde mi posición (pej., -4 para ir 4 instrucciones para arriba de donde está el *branch*).

En definitiva, tenemos el siguiente **RESUMEN DE HAZARD RESPECTO A LAS ARQUITECTURAS** que vimos:

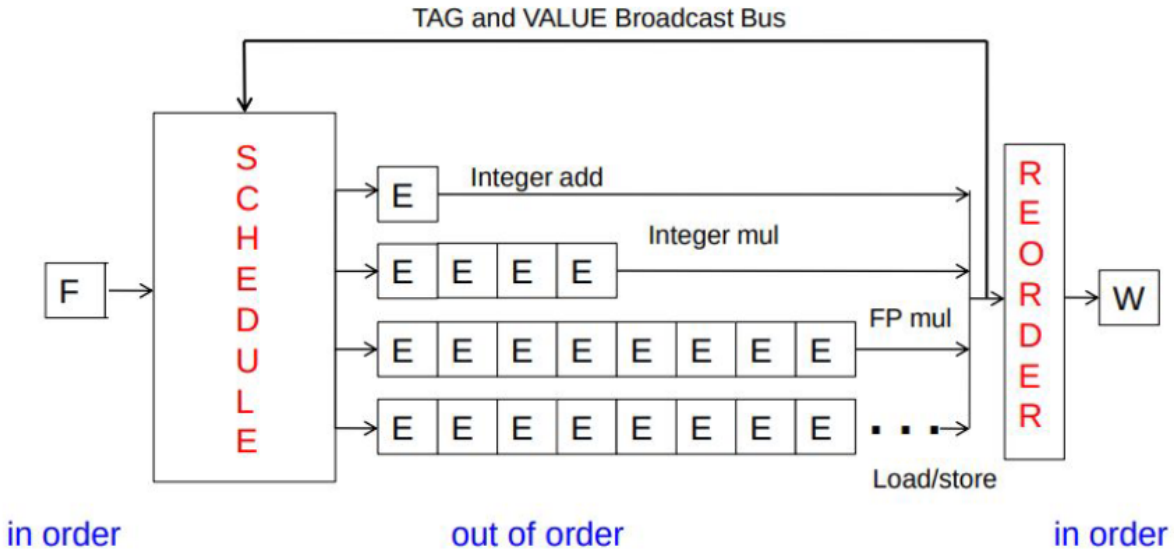
Tipo de dependencia	In-order (1-issue)	Multiple issue in-order	Multiple issue out-of-order
Estructurales (Hazard)	No*	No*	Puede provocar stalls
Datos de datos condicional	Puede provocar stalls	Deben considerarse al armar los issue packet. Puede provocar stalls	Deben considerarse al reordenar el código.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

ReOrder Buffer (ROB) → *Manejo de excepciones*



- En caso de que en una *FU* salte una *excepción*, es necesario saber el contexto "real" de ejecución en orden para poder procesarla correctamente
- Motivo de ello, a la **Máquina OoO** se le agrega la etapa **ReOrder** !!! → Como está el esquema gral. desde el principio
  - Para esto, necesitamos tener un *reorder buffer*, la cual va a tener los *tags* de las instrucciones a medida que se le hacen *issue*, para saber el orden real en el código
- Como la etapa de *reorder* se realiza **antes** del WB, es en esta etapa donde "saltan" las excepciones (ya que se procesa cada resultado de las FU en orden)

- Cada línea del ROB contiene
  - Tipo de *instrucción*
  - Campo de *destino*
  - Campo de *valor*
  - Campo de *ready*

Speculation / Especulación

- La idea es que, dado que *predije* el salto, no me voy a quedar solo con eso, sino que voy a seguir ejecutando instrucciones como si la predicción fuera correcta
- Cuando el salto llegue a ReOrder y vea que es cierto, listo, sigo ejecutando como si no pasó nada
- Caso contrario, si llega y le erró, entonces todo lo que se está ejecutando que sea posterior, se *levanta*
- Para esto, vemos que se agrega lo siguiente:
  - *Buffer de reordenamiento (ROB)* que mantiene los resultados hasta qiue el salto termine de ejecutar
  - Fase de instrucción *commit* en la que se actualiza la memoria y los registros cuando una instrucción no es más especulativa

Ejercicios resueltos (Práctico)

> 💡 **Para el parcial podemos llevar muchas hojas impresas de la *template de Tomasulo* para usar para guiarnos y hacer paso por paso.**

Ejercicio 1

Recomiendo ver la explicación de los profes en la *clase 1* → https://drive.google.com/file/d/1JhJXtHCnxXwkJg7laC8VPA1XzEMzPDC_/view?usp=sharing
(es MUY importante verlo para entender cómo va a ser la onda de los ejercicios de *Tomasulo* que nos van a dar)

Ver desde el minuto **54:47** al **01:38:21**.

Pd.: tiene un *error* que arrastra desde el CLK 4, el cual corresponde a no poner la Tag de D8 para la 4ta instrucción.

Pd2.: el profe mencionó al final que con 2 CLKs más ya termina, pero eso es falso dado que las FUs de ALU es de 2 CLKs, mientras que la de Mult es de 6 CLKs ⇒ Tenemos que tener *muy* en cuenta cuánto tarda cada FU !!!

Pd3.: para los *load* y *store*, tenemos que considerar que el *offset* SIEMPRE se debe poner en el campo de address, dado que en los *store* se tienen tres operandos (registro que se lee + registro de address + offset) ⇒ Se explica mejor en el video para el *Ejercicio 2* dado que en la 1er clase lo dijo mal !!!

Ejercicio 1:
Considerando un microprocesador out-of-order execution implementado mediante el algoritmo de Tomasulo, el código en assembler el hardware especificado que se muestra a continuación, muestre el contenido de las tablas de *Status*, las *Reservation stations* y el flujo de ejecución para el 6to clock de ejecución (inclusive).

Código:

```
1> LDURD D6, [X2, #32]
2> LDURD D2, [X3, #44]
3> FMULD D0, D2, D4
4> FSUBD D8, D2, D6
5> FDIVD D0, D0, D6
6> FADDD D6, D8, D2
```

Especificaciones de hardware:

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

Mi solución al ejercicio *teniendo en cuenta TODOS los CLKs de ejecución* (i.e., no solo el 6to como pide el enunciado) es la siguiente:

https://prod-files-secure.s3.us-west-2.amazonaws.com/206ee924-3842-491f-9926-b87d27a727f3/e8867aaf-97e8-4ab0-a812-3a900cf69598/Arqui.pdf

Ejercicio 2

💡 Recomiendo ver la explicación de los profes en la *clase 2* → https://drive.google.com/file/d/1RtSs4-cvS7udi9qvT8Z0tuDmwkNnTUqn/view?usp=sharing
(es MUY importante verlo para entender cómo va a ser la onda de los ejercicios de *Tomasulo* que nos van a dar que incluyen *Branchs* y/o se hace *issue* de más de una instrucción).

Ver desde el minuto **10:59** al **01:05:25**. (*saltear muchas partes porque se van por las ramas a veces*) ⇒ No es *tan* clara la clase pero algunas cosas sí sirven. Si se ve, recomiendo ir salteando varias secciones y verla con x1.25 o x1.5 !!!

Pd.: Para los *branchs*, se coloca en *Vk* el inmediato que indica cuántas instrucciones se saltan desde mi posición (pej., -4 para ir 4 instrucciones para arriba de donde está el *cbnz*).

Pd2.: por más que las instrucciones *branch* no tengan que escribir nada, consideramos que deben pasar por la etapa de *WriteBack*. !!!

Ejercicio 2:
Considerando un microprocesador out-of-order execution implementado mediante el algoritmo de Tomasulo, el código en assembler el hardware especificado que se muestra a continuación, muestre el contenido de las tablas de *Status*, las *Reservation stations* y el flujo de ejecución para el 8vo clock de ejecución (inclusive).

Código:

L:	1> LDURD D0, [X1, #0]
	2> FMULD D4, D0, D2
	3> STURD D4, [X1, #0]
	4> SUBI X1, X1, #8
	5> CBNZ X1, L

Especificaciones de hardware:

Hardware
El salto se predice correctamente como taken.
Issue = 2 instrucciones
Load = 6 RS / 1 clk
Store = 6 RS / 1 clk
Alu entera = 3 RS / 1 clk
Alu punto flotante = 3 RS / 2 clk
Multiplicación punto flotante = 2 RS / 3 clk
Branch = 1 RS, 1 FU / 1clk

https://prod-files-secure.s3.us-west-2.amazonaws.com/206ee924-3842-491f-9926-b87d27a727f3/57c4bb4f-7f41-4cf4-8092-13bdef351f95/Arqui-1.pdf

Ejercicio 3 (no hecho aún)

Ejercicio 3:
Considerando un microprocesador out-of-order execution implementado mediante el algoritmo de Tomasulo, el código en assembler el hardware especificado que se muestra a continuación, muestre el contenido de las tablas de *Status*, las *Reservation stations* y el flujo de ejecución para el 8vo clock de ejecución (inclusive).

Código:

```
1> ADDI X2, X0, #50
2> ADDI X3, X0, #70
3> ADDI X4, X0, #40
4> LDURD D0, [X4, #0]
5> LDURD D1, [X2, #0]
6> ADDI X5, X2, #16
7> FADDD D2, D1, D0
8> LDURD D1, [X2, #8]
9> STURD D2, [X3, #0]
10> ADDI X2, X2, #2
11> FADDD D2, D1, D0
12> LDURD D1, [X2, #0]
13> ADDI X2, X2, #1
14> ADDI X3, X3, #1
```

Especificaciones de hardware:

Hardware
Issue = 4 instrucciones
Load = 4 RS / 2 clock cycle
Store = 4 RS / 2 clock cycle
Suma entera = 2 FU - 4 RS / 1 clock cycles
Multiplicación entera = 2 FU - 4 RS / 1 clock cycles
Suma flotante = 2 FU - 4 RS / 2 clock cycles
Multiplicación flotante = 2 FU - 4 RS / 4 clock cycles

Ejercicio 4 (no hecho aún)

Ejercicio 4:
Considerando un microprocesador out-of-order implementado mediante el algoritmo de Tomasulo (sin especulación), muestre el contenido de las tablas de Status, Registros, las Reservation stations y el flujo de ejecución para la siguiente secuencia de código para el 7mo ciclo de clk (incluido):

Código:

```
    1>> addi  x3,xzr,#80
L:  2>> ldurd D2, [x1]
    3>> ldurd D6, [x1, #100]
    4>> fmuld D4, D2, D0
    5>> faddd D6,D4,D6
    6>> sturd D6, [x1, #100]
    7>> subi  x1,x1,#8
    8>> cbnz  x1, L
    9>> add   x1, xzr, #160
```

Asumir que X1 inicialmente contiene el valor 80. Asumir que el salto fue predicho correctamente como taken. Considerar el siguiente Hardware:

Hardware
Issue = 4 instrucciones
Load = 4 RS / 1 clock cycle
Store = 4 RS / 1 clock cycle
Suma entera = 4 RS, 2 FU / 1 clk
Suma punto flotante = 4 RS, 2 FU / 2 clk
Multiplicación punto flotante = 4 RS, 2 FU / 3 clk
Branch = 1 RS, 1 FU / 1clk

Ejercicio 5 (no hecho aún)

Ejercicio 5:
Considerando un microprocesador out-of-order implementado mediante el algoritmo de Tomasulo (sin especulación), muestre el contenido de las tablas de Status, Registros, las Reservation stations y el flujo de ejecución para la siguiente secuencia de código para el 5to ciclo de clk (incluido):

Código:

```
     1>> addi x1,xzr,#80
     2>> ldurd D0, [x1]
     3>> ldurd D1, [x1, #100]
     4>> fmuld D4, D2, D0
     5>> faddd D6, D4, D1
     6>> faddd D3, D5, D7
     7>> subi x1,x1,#8
     8>> sturd D6, [x1, #100]
     9>> andi x4, x2, #0xFF
     10>> cbnz x1, L
     11>> addi x1, x1, #16
L:   12>> mult x2, x3, x5
```

Especificaciones de hardware:

Hardware
Los saltos son predichos como taken
Issue = 4 instrucciones
Load = 4 RS / 2 clock cycle
Store = 4 RS / 2 clock cycle
Suma entera = 4 RS, 2 FU / 1 clk
Suma punto flotante = 4 RS, 2 FU / 2 clk
Multiplicación punto flotante = 4 RS, 2 FU / 3 clk
Branch = 1 RS, 1 FU / 1clk