# EMA | Técnicas de mejora de rendimiento (Unidad 6)



# Material importante

## Clases

- Carpeta de clases: https://drive.google.com/drive/u/1/folders/10weMj2M6tQZ2Lr0OFWIIZoBTb4GMhhU0
  - Unidad 6: Técnicas de mejora de rendimiento
    1. https://drive.google.com/file/d/1pmqM9ueAF1q707C56-68WAtbtWrspCBk/view?usp=sharing
    2. https://drive.google.com/file/d/1MlhCIkXOCrE2DF7h79QdpPGMxLVdeJvF/view?usp=sharing
    3. https://drive.google.com/file/d/1d_qE34jtwmv9Zu7j4WHqmDT856_v-xQA/view?usp=sharing

## Filminas

- Unidad 6: Técnicas de mejora de rendimiento
  - Filmina de profes: https://famaf.aulavirtual.unc.edu.ar/pluginfile.php/28613/mod_resource/content/0/Clase TP6.pdf
  - Práctico: https://famaf.aulavirtual.unc.edu.ar/pluginfile.php/28513/mod_resource/content/0/TP6_Técnicas de mejora de rendimiento.pdf

# Notas

¿En qué se diferencia un micro que está en un celu, con otro que está en un cluster? → Principalmente en las técnicas de mejora de rendimiento que tienen implementados cada uno (haciendo mucho más eficiente, claramente, al del cluster)

## Deep Pipelines

- Mejora de rendimiento enfocada a *profundizar en el pipeline* y agregar muchísimas etapas de estas
- Vimos al principio que tiene 5 etapas (F, D, E, M, WB), pero podríamos considerar con **deep pipelines** que, a esas 5 (o algunas), las subdividamos en otras etapas → Sobretodo la parte del EXECUTE
  - Subdividir en más etapas implica agregar más columnas de registros → Lo cual me suma a la latencia por la de este registro
    - Es importante a tener en cuenta dado que los datos necesitan pasarse por un FLOPR
  - La idea es que el período de clock sea más chico, cosa que me permite aumentar la velocidad de ejecución de las instrucciones
  - Además, hace más fácil el “balanceamiento” de las etapas para que todas manejen una latencia parecida y que ninguna me perjudique a sobre-manera las demás
  - Algo que tiene también el hecho de hacer *pipeline profundo* es que cuando me equivoco al hacer mal una predicción de salto, la penalidad es mucho más alta → **PRINCIPAL DESVENTAJA**
- Cálculos a tener en cuenta
  - $\text{Latencia de una etapa} = \text{Tiempo de ciclo (Tc)} = \frac{\text{Tiempo de ciclo en procesador sin pipeline}}{\text{Cantidad de etapas de pipeline}} + \text{Latencia de agregar registro para pipeline}$
  - $\text{Tiempo por instrucción (Ti)} = Tc \times CPI$ donde CPI es la cantidad de clocks por instrucción.

        - El CPI se obtiene como promedio de diferentes grupos de códigos que se usan para probar el micro. Para cada benchmark, lo que se hace es contar la cnt. de ciclos y dividirla por la cnt. de instrucciones
        - El conjunto de benchmarks prueba distintos casos, totalmente diversos, para el cual se prueba el micro
    - **Lo que buscamos** → Es reducir el *Ti !!!*
        - El *Tc* SIEMPRE mejora, pero el *CPI* va empeorando a medida que se agregan más etapas, ya que aumenta dada la penalidad por una predicción de salto errónea (ya que tenemos más etapas)
- Conclusión → No siempre un pipeline más profundo es mejor. Es una relación de compromiso, lo importante es identificar el punto en el que deja de mejorar y comienza a empeorar (*en términos de Análisis I, encontrar el mínimo local de la función*)

# Predicciones de saltos

- El problema que queremos resolver es → *¿Cuál es la próxima instrucción que debería fetchear?*
    - Es decir, queremos predecir correctamente la próxima instrucción a ejecutar
    - Para hacer esto, debemos saber:
        - ¿La instrucción anterior es un salto?
        - En caso de un salto condicional, ¿se toma o no?
        - En caso que se tome, ¿cuál es la dirección del salto?
- Dado esto, con el *Ejercicio 2* del práctico, podemos ver y notar lo **importante** que son los predictores de salto y su correcto funcionamiento / precisión.
    - Los predictores de salto de hoy en día andan **TODOS** con más del 90% de precisión (el Intel I9 se *estima* que está arriba del 95%)

## EVITAR EL SALTO (para compiladores)

- Algo en lo que se centran los compiladores y diseñadores de arquitecturas es tratar de evitar los saltos en los casos donde se pueda (para evitar cualquier tipo de conflicto con la predicción)
- Para esto, se implementaron y agregaron *nuevas* instrucciones, las cuales permiten obviar muchísimos tipos de saltos, tales como:

```
if (a == 5) b = 4;
else b = 3;

// Lo cual nos queda en
    cmp x0, 5
    b.ne L2
    mov x1, 4
    b L3
L2:
    mov x1, 3
L3:

// Peeeero, con instrucciones nuevas podemos tenerlo como:
    cmp x0, 5
    mov x0, 4
```

```
mov x1, 3
csel x1, x0, x1, eq // --> Usa las flags del micro !!!
```

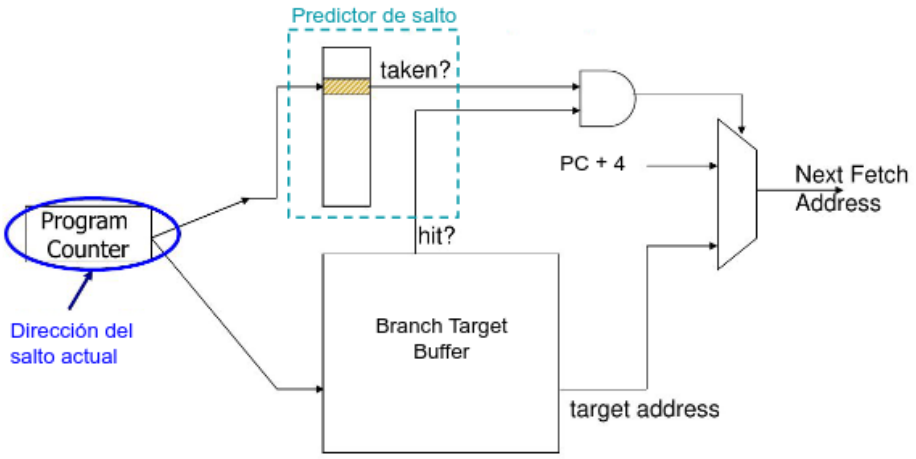
- Estas nuevas instrucciones, se *ahorran* el hecho de algunos saltos, **usando las flags del micro** para saber cómo realizar la asignación

| Conditional Instructions | | |
|---|---|---|
| CCMN | rn, $\#i_5$, $\#f_4$, cc | if(cc) rn + i; else N:Z:C:V = f |
| CCMN | rn, rm, $\#f_4$, cc | if(cc) rn + rm; else N:Z:C:V = f |
| CCMP | rn, $\#i_5$, $\#f_4$, cc | if(cc) rn − i; else N:Z:C:V = f |
| CCMP | rn, rm, $\#f_4$, cc | if(cc) rn − rm; else N:Z:C:V = f |
| *CINC* | rd, rn, cc | if(cc) rd = rn + 1; else rd = rn |
| *CINV* | rd, rn, cc | if(cc) rd = ∼rn; else rd = rn |
| *CNEG* | rd, rn, cc | if(cc) rd = −rn; else rd = rn |
| CSEL | rd, rn, rm, cc | if(cc) rd = rn; else rd = rm |
| *CSET* | rd, cc | if(cc) rd = 1; else rd = 0 |
| *CSETM* | rd, cc | if(cc) rd = ∼0; else rd = 0 |
| CSINC | rd, rn, rm, cc | if(cc) rd = rn; else rd = rm + 1 |
| CSINV | rd, rn, rm, cc | if(cc) rd = rn; else rd = ∼rm |
| CSNEG | rd, rn, rm, cc | if(cc) rd = rn; else rd = −rm |

- Algo **importante** a tener en cuenta, es que para ahorrar la predicción de salto, se ejecutan las instrucciones de las dos partes del condicional, para luego usar la instrucción condicional y elegir cuál es el resultado que queremos
  - Se tiene que hacer un trade-off y una relación de compromiso por parte del compilador, para que vea si es mejor (o abarca menos ciclos) hacer uso de esta instrucción, o ver un condicional con el posible error de predicción de salto

## PREDECIR EL SALTO (<u>para el micro</u>)

- **En caso que salte, ¿cómo saber a dónde?**
  - Dado que nosotros queremos poder predecir el salto, y dado el hecho de que es posible de que muchas veces se tome el *mismo salto* (por ejemplo, en el caso de un for / while), entonces lo que se hace es **agregar una caché** al micro, cosa de que se pueda identificar y saber con exactitud **a dónde debo saltar** para un salto en particular (dado que la instrucción y el inmediato van a ser los mismos) ⇒ Esto hace que, a lo sumo, se falle solo la *primera vez*
    - Guardamos algunos bits del PC y a dónde debería saltar
    - Esto se llama **Branch Target Buffer (BTB)** → memoria asociativa que almacena la dirección donde se encuentra el salto y el destino
  - Motivo de ello, nuestra etapa de FETCH queda del siguiente modo:




- En este diagrama se puede ver que, si el *predictor de salto* da taken y la caché da hit, entonces se pone el valor que resuelve / devuelve la caché. Caso contrario, pasa el PC + 4.

  - Sin embargo, agregar la BTB *no implica* que se resolvió totalmente este problema, sino que la predicción es mejor. ¿Por qué no se resolvió totalmente (y sigue siendo solo una predicción)?
    - Por ello mismo, como sigue siendo una predicción, se siguen ejecutando *todas* las etapas para ver si realmente la predicción fue correcta y, en caso que no, hacer un flush de las instrucciones extra mal cargadas.
    - Para ver porqué pasa esto, podemos ver los *tipos de saltos* que tenemos:
      - Condicionales (CBZ, CBNZ, B.cond)
        - No se sabe la dirección del salto al hacer FETCH, sino que se resuelve en DECODE (micro modificado) o EXECUTE (micro normal que vimos en ODC)
        - Tiene 2 posibles saltos (si se toma o no)
      - Incondicional (B)
        - Se sabe la dirección del salto al hacer FETCH, dado que tiene una sola posible dirección de salto → Se termina de resolver en DECODE
      - Salto con registro (BR)
        - No se sabe la dirección del salto al hacer FETCH, sino que recién se sabe en el EXECUTE
        - Tiene *casi infinitos* lugares de saltos, dado que depende del valor de un registro → Encima, de una iteración a otra se puede cambiar la dirección de salto, por lo que la **BTB fallaría en estos casos (no me lo resuelve bien)**

- **Ahora, ¿cómo predecir si se toma el salto? → Tipos de predictores de saltos** → Vamos a ver cada uno de forma particular a continuación de esta lista.
  - Estáticos → SIEMPRE tomo la misma decisión
    - Not taken
    - Taken
    - BTFNT (backward taken, forward not taken)

- Dinámicos → A medida que una instrucción de saltos es ejecutada, se almacena información sobre el resultado. Esta información es usada luego para intentar predecir el resultado de ejecuciones posteriores a este salto. (Se basa en el uso de la *BTB* para eliminar la penalidad de *Taken*)
  - Locales → Basan su predicción en la información de ejecuciones anteriores *del mismo salto* (intenta predecir muy bien los loops)
    - Predictor local (2 bit)
    - Predictor de dos niveles (2 bit mejorado) → usa una idea de predictores globales
    - Predictor gshared → usa una idea de predictores globales
    - Predictor de fin de ciclo → Se usa para los finales de un loop interno en un esquema de *loops anidados*
      - Si se nota que cada 5000 ciclos se hace Not Taken porque es fin del loop, entonces tiene en cuenta esa predicción para ese salto en particular
  - Globales → Realizan la predicción utilizando la información de *otros saltos* del programa a ejecutar (intenta predecir muy bien las estructuras if / else)
    - Predictor por torneos (Tournament predictors)
    - Tagged Hybrid predictors

## Predictor de salto *Not taken*

- Idea → Se predice que *nunca* se toma el salto
  - Predictor estático
- Ventajas
  - Implementación mucho más sencilla
  - No necesita predecir la dirección del salto
  - Si acierta no tiene penalidad
- Desventajas
  - Baja precisión (30 a 40% para saltos condicionales)
  - Si se equivoca, la penalidad es de 3 ciclos de reloj

## Predictor de salto *Taken*

- Idea → Se predice que *siempre* se toma el salto
  - Predictor estático
- Ventajas
  - Mejor precisión en la predicción de salto o no salto (60 a 70% para saltos condicionales) → Por los casos donde de hay for o while, por ejemplo.
- Desventajas
  - Implementación más compleja → Porque tenés que tratar de predecir la dirección con el BTB

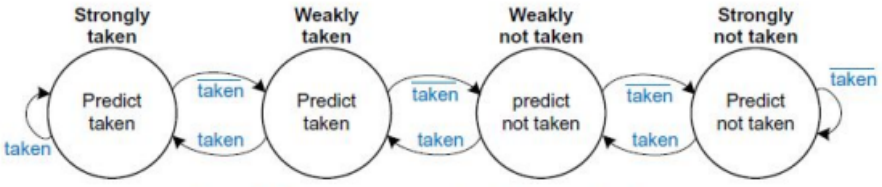
- *Siempre* tiene penalidad de 1 ciclo de reloj (acierte o no) → porque no sabemos la dirección a donde vamos a saltar
    - En caso que se agregue el BTB, entonces no siempre se tiene esta penalidad, sino solo en los casos en donde se falla !!!

## Predictor de salto *BTFNT*

- Idea → Se predice como una solución de compromiso dependiendo de si el *offset* es positivo o negativo (i.e., si se salta para adelante o para atrás respectivamente)
    - Predictor estático
    - Si se salta para atrás (*offset* < *0*), entonces se sabe que es un *loop*, por lo que se usa *Taken*
    - En caso contrario, si se salta para adelante (*offset* > *0*), es igualmente probable que sea tomado como no el salto. Por ello, entonces, se usa *Not Taken* → (por ej., los casos de if / else)
        - Los *compiladores* intentan predecir y organizar el código para que, en estos casos, el salto hacia delante no se tome en la mayoría de situaciones.
- Ventajas
    - Mejora mucho la precisión de la predicción del salto → Trata de eliminar la penalidad del Taken
- Desventajas
    - Implementación mucho más compleja dado que depende del *offset* que nos pasen para el salto la decisión que se tome.

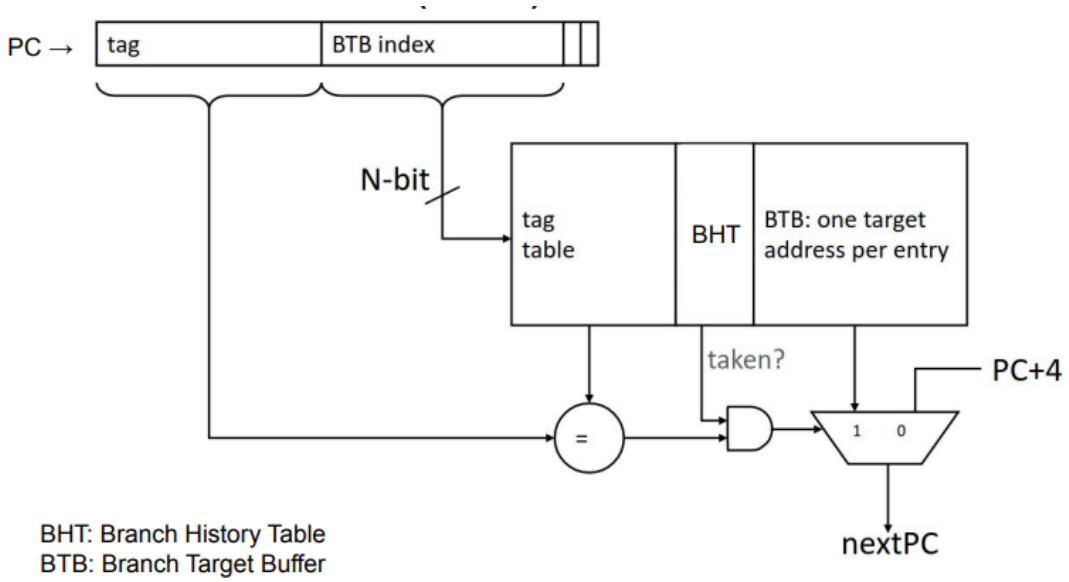
## Predictor de 2 bits (local)

- Predictor dinámico local → de 2 bits
- Sigue este *diagrama de estados* para la predicción




    - Depende de los resultados de las predicciones anteriores para el salto. La idea es que para cambiar la decisión a considerar, debo errarle **dos veces** seguidas
        - Esto nos sirve para cuando tenemos *for anidados* por ejemplo
        - Ayuda a errarle *una sola vez* en la condición de salida, pero después seguir prediciendo bien para los loops anidados
    - Los valores a considerar son:
        - 11 → Strongly taken
        - 10 → Weakly taken
        - 01 → Weakly not taken
        - 00 → Strongly not taken

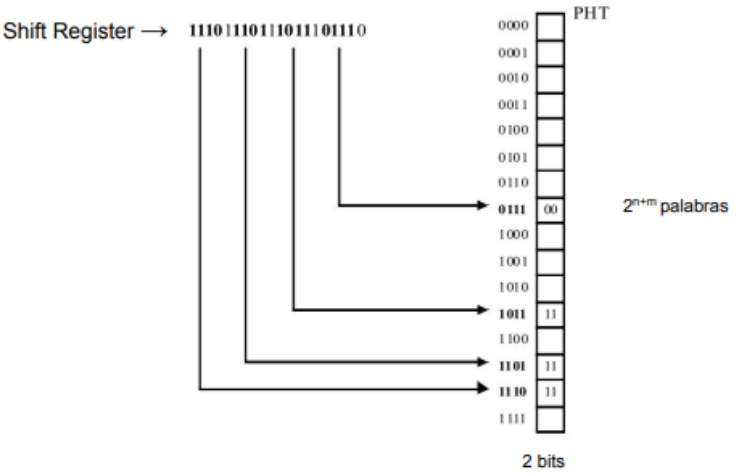
- ¿Como se implementa este tipo de predictor?
  - La máquina local (diagrama de estados) va a funcionar independientemente para cada salto en particular
  - Para ello, se necesita tener una memoria para ir almacenando 2 bits para cada salto, para saber el estado actual en el que está (*tener en cuenta que son 4 estados y, por ello son 2 bits*)
  - Para ello, a partir del PC se indexa en una tabla donde se tiene esta información de 2 bits → Se implementa una pequeña memoria asociativa (BHT, *Branch History Table*) donde esto es almacenado → Se usa y se combina con el *BTB* para que nuestro FETCH quede así:




  - El tamaño de la *BHT* es un criterio de diseño pero, en un punto, seguir agrandándola no mejora el rendimiento. → Poner 4096 entradas a “infinitas”, no mejora el rendimiento que tengamos
    - Tener en cuenta que mientras más filas se agreguen, más “latencia” o tiempo va a tomar, dado que la comparación de los tags va a ser mayor
- **Deventaja** → *Solo* funciona para los loops, no toma los casos condicionales if / else dado que no considera el “contexto” u otros saltos anteriores tomados.

## Predictores globales (en gral.)

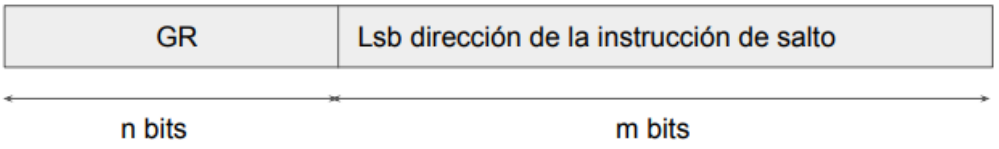
- Se crea un “registro de desplazamiento” (*shift register*) donde vamos almacenando si los saltos anteriores se tomaron o no (1 *taken*, 0 *not taken*)
  - Va a ser nuestro “historial de saltos global”, el cual se va a almacenar en una memoria PHT (*pattern history table*), la cual es una memoria “normal” que está indexada por address y tiene $2^{n+m}$ palabras de 2 bits → Se indexa a esta memoria con el *shift register* que tengamos
  - Según esta indexación, se obtiene el valor de dónde se encuentra en la máquina de estados vista antes (la de 2 bits local), solo que ahora en vez de usar BHT, usamos PHT indexando con el *shift register* (para tener en cuenta el “contexto” de todos lo saltos anteriores)

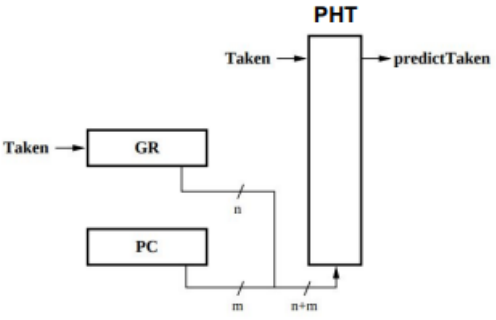



- A diferencia de los *predictores locales*, en vez de centrarse en la predicción de un salto en específico, se centra en la predicción de un salto teniendo en cuenta el patrón con el que llegó (historial de todos los saltos en el *shift register*), siendo súper útil para los casos de if / else, pero no para los loops.

## Predictor de dos niveles (local)

- Se concatena el registro de *GR (shift register)* con los bits menos significativos del PC, usando la tabla PHT mencionada anteriormente (por eso el $2^{n+m}$ de tamaño)
  - n bits del registro de desplazamiento
  - m bits del PC (m bits menos significativos de este)



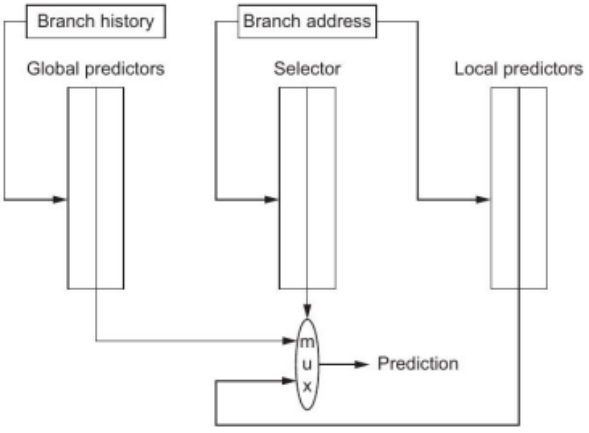

- En definitiva, usa la “mejora de patrones” que da la idea para los predictores globales, pero enfocándola de forma local para un salto en particular (dado que los bits menos significativos del PC hacen que se considere el estado de ese salto en particular)
  - Es decir, *para este salto en particular me voy a fijar el historial de saltos anteriores que tuvo*
  - Y se sigue usando la máquina de estados vista en el predictor de 2 bit local

## Predictor gshared (local con *hashes*) → Mejora al de dos niveles

- Muy parecido al predictor de dos niveles, solo que en vez de concatenar los bits del GR y el PC, hace un *hash* (XOR) entre los últimos 10 bits de cada uno → Hace un mejor uso de la memoria PHT, dado que la *probabilidad* de que colisionen dos valores iguales con distinto PC y GR es muy baja (básicamente, la idea del uso de *hashes* en programación)
    - La PHT va a tener 1024 entradas ($2^{10}$).

### Predictor por torneo (local + global)

- Se usa el *ejercicio 4* de la guía como intro a este tipo de predictores y su potencial → Acá funciona MUY bien
- Se tienen **muchos predictores** y después me “entreno” otro predictor que me dice, para cada caso, cuál es el que me conviene
    - En este caso, vamos a considerar que se combinan predictores locales (de 2 niveles) y globales
    - La ventaja es que elige el tipo de predictor que mejor funciona para cada salto




- **SIEMPRE** se van a ir entrenando los dos tipos de predictores (locales y globales), sólo que se va a agregar uno nuevo que selecciona cuál va a ser el que se va a usar para este salto en particular
    - Para ello, el *Selector* pone dos bits que dice cuál de los dos predictores es mejor → Se usa la misma idea de la *máquina de estados* para saber qué elección considerar para ese salto → Tener en cuenta que es en función del PC (local) la selección y el manejo de esta máquina de estados
        - Se entrena en función de qué predictor se usó y si ese le erró o no
- **Es el predictor de bajo costo más usado !!!** → Es el que toma las *mejores características* de los dos tipos de predictores

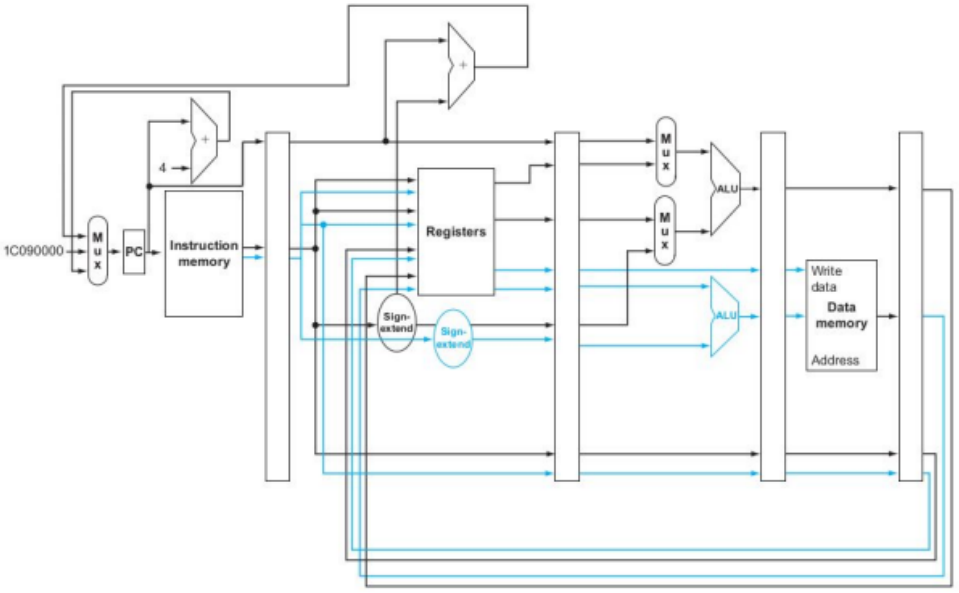
## Static Multiple Issue Processor → (poder ejecutar varias instrucciones en paralelo)

- Tienen **distintos caminos** para ir **ejecutando instrucciones en paralelo.** Es decir, para hacer la ejecución de “bloques” de instrucciones al mismo tiempo (no están corridas una fase como en pipeline, sino que van en paralelo desde el FETCH → Pd.: también se puede combinar con pipeline pero de “bloques de instrucciones”)

- Es un procesador que puede ejecutar varias instrucciones en simultáneo, las cuales deben ser *empaquetadas* por el compilador (*issue packet*)
- En algunos casos, se restringe qué tipo de instrucciones pueden ejecutarse en simultáneo → Depende de la arquitectura que se considere
- Al ser *Static*, relegan la responsabilidad de manejar ciertos data / control hazards al compilador, ya sea para prevenirlos o reducirlos
- **Desventajas** → Hace que aparezcan nuevos tipos de dependencias (dato o control), las cuales hay que saber *manejar*

## LEGv8 Static Two-Issue processor

- Un ejemplo es el esquema del **LEGv8 Static Two-Issue Datapath** → esquema más rudimentario y sin agregar tanto hardware




- Es *Static* porque el PC siempre trae 64 bits (las dos instrucciones, dado que cada una tiene 32 bits en LEGv8)
  - Por ello, entonces ejecutar dos instrucciones por ciclo requiere hacer *fetch* y *decode* de instrucciones de 64 bits *(el PC se incrementa de a 8)*
- Sigue existiendo el *forwarding* + *stall*, pero solo sobre instrucciones de diferentes clocks (no las que se ejecutan en simultáneo !!!)
- En este caso, como la memoria se comparte, **permitiría ejecutar una operación de tipo R o B (no M) con otra de tipo M** de forma paralela, agregando simplemente otro s*ign-extend* y otra *ALU* (en realidad es un sumador nomás, no una ALU, dado que se quiere hacer la suma del valor del registro más el offset para el cálculo del address de la memoria).
  - Dado que el ADD o BRANCH va por “arriba” (negra) mientras que el de memoria (LOAD / STORE) va por “abajo” (celeste) modificando la *Data memory*
  - En definitiva, en cada *Issue Packet* debe haber una instrucción de tipo R o branch y una instrucción de acceso a memoria

        - Si una de los *issue* no puede utilizarse, se la debe acompañar de un ***nop*** *!!!*

## Dependencias a tener en cuenta

- **Dependencia de datos**
    - Es una propiedad de programa (*no tiene nada que ver con el hardware*)
    - El hecho de que se detecte como *hazard* y se genere o no un *stall,* es propiedad de la organización del *pipeline*
    - Implica:
        - Posibilidad de hazard
        - Orden en que se debe calcular el resultado
        - Límite superior en cuanto se puede explotar el paralelismo
    - Puede superarse:
        - Manteniendo la dependencia pero evitando el hazard → micro con *forwarding*
        - Eliminando la dependencia al modificar el código → compilador
    - Tipos:
        - Clasificación por dónde se aplica
            - Por *registro* → relativamente sencillo de detectar, es transparente verlo desde las instrucciones
            - Por *memoria* → implica hacer primero un STORE y luego un LOAD ⇒ difícil de detectar dado que depende de la suma de un registro + un *offset* para saber el address
        - Clasificación por tipo de dependencia
            - *Dependencia real de datos* → La instrucción i es dependiente en datos con la instrucción j cuando
                - i produce un resultado que debe ser usado por j
                - j es dependiente de datos de k, siendo k dependiente de datos con i (i.e., se cumple *transitividad*)
            - *Dependencia de nombre* (nueva a considerar dado que tomamos varias instrucciones en simultáneo !!!) → dos instrucciones usan el mismo registro o posición de memoria, pero no hay flujo real de datos
                - Ocurre cuando las instrucciones i, j escriben la misma posición de memoria o registro → *Dependencia de salida*
                - El orden original se debe preservar para asegurar que el valor final se corresponda con el de j
                - ¿Cómo solucionarlo? → Con **register renaming** ⇒ Dado que las dependencias de nombre no son dependencias reales, las instrucciones se pueden ejecutar simultáneamente o en otro orden, si el nombre se cambia para no generar conflictos.
- **Hazard de datos**

    - Es una dependencia de datos o de nombre entre instrucciones que se ejecutan lo suficientemente cerca en tiempo, de forma en que al superponerse en la ejecución se modifica el orden de acceso a los operandos involucrados en la dependencia
        - Básicamente, cuando están suficientemente cerca de modo que la ejecución de uno rompe la del otro
    - Tipos:
        - **RAW** → j intenta leer un dato antes que i lo escriba,  obteniendo valor desactualizado (dependencia real)
        - **WAW** → j intenta escribir un operando antes que lo escriba i. Las escrituras se realizan en orden incorrecto, dejando como valor final el de i en vez del de j (dependencia de salida)
- **Dependencia de datos condicional**
    - Una dependencia condicional determina el ordenamiento de una instrucción respecto a otra de salto, de forma en que dicha instrucción se ejecuta en el correcto orden de programa y solo cuando debe ser.
        - Básicamente es el hecho de que una dependencia de datos exista o no dependiendo de si el salto se toma o no
        - !!! Nos limita el hecho de cómo se realiza el *empaquetamiento* de las *issues* (instrucciones) ⇒ No se puede poner una de salto *condicional* junto con la siguiente (PC + 4) dado que podría no ser esta la instrucción que se ejecuta. Motivo de ello, se usa la **nop** para armar este *issue packet*.
            - Tener en cuenta que el **nop** va únicamente cuando se quiere poner en el mismo packet el branch *condicional* con la “posible” siguiente instrucción (dado que depende de la predicción)
    - Restricciones impuestas:
        - Una instrucción que tiene una dependencia condicional sobre un salto no puede moverse **antes** que el salto de forma en que su ejecución **no** esté controlada por este
        - Una instrucción que **no** tiene dependencia condicional no puede moverse **después** del salto, de forma en que su ejecución esté controlada por este

## Truquitos de *compilador* para usar de forma más eficiente el Two-Issue processor

- Re-organizar el código (orden de las instrucciones) → Tener en cuenta las *restricciones* para los branchs condicionales
- Eliminar instrucciones que no se usen → O *compactar* instrucciones en una sola cuando se pueda
- Técnica de *renombre de registros* para evitar dependencias → con modificaciones de registros que no afecten la ejecución
- *Loop-unrolling* → Básicamente, si tenemos un loop, para aprovechar el hecho de los *issue packet*, poner más de una iteración para ejecutar en una sola, dividiendo la cantidad de iteraciones según la cantidad que se coloque (https://en.wikipedia.org/wiki/Loop_unrolling)

```
# IDEA CON PYTHON (lo vamos a hacer con assembly pero es para la idea nomás)
# Empezamos con:
```

```
sum = 0
it = 4
while it != 0:
  sum += it
  it -= 1

# Ahora tenemos con loop unrolling:
sum = 0
it = 4
while it != 0:
  sum += it
  it -= 1

  sum += it
  it -= 1
```

# Ejercicios resueltos (Práctico)

## Ejercicio 1

**Ejercicio 1: Deep Pipelines**
Considere construir un procesador con pipeline dividiendo el procesador de un solo ciclo en N etapas. El procesador de ciclo único tiene un retardo de propagación a través de la lógica combinacional de 740ps. La penalidad por agregar un registro de pipeline es de 90ps. Suponga que el retardo de la lógica combinacional se puede dividir arbitrariamente en cualquier número de etapas y que la lógica de hazard del pipeline no aumenta el retardo.
Asumiendo que un pipeline de cinco etapas tiene un CPI de 1.23 y que cada etapa adicional aumenta el CPI en 0.1 debido a las predicciones de salto erróneas y otros hazard. ¿Cuántas etapas de pipeline deberían usarse para hacer que el procesador ejecute los programas lo más rápido posible?

- Datos:
  - Clk de proc. de ciclo único de 740ps
  - Penalidad de agregar reg. a pipeline de 90ps
  - CPI de pipeline con N = 5 de 1.23 → Cada nueva etapa suma 0.1
- Suposiciones:
  - El retardo de la lógica combinacional se puede dividir arbitrariamente en cualquier número de etapas
  - La lógica de hazard del pipeline no aumenta el retardo
- ¿Qué queremos?
  - El mejor N tal que Ti (tiempo de instrucción) sea mínimo

Para esto, notemos que $T_i = T_c \times CPI$ donde $CPI = 1.23 + (N - 5) \times 0.1$ y $T_c = (\frac{740}{N} + 90)ps$. Luego, entonces, veamos la siguiente tabla:

| N | Tc | CPI | Ti |
|---|---|---|---|
| 1 | 830 | 0,83 | 688,9 |
| 2 | 460 | 0,93 | 427,8 |
| 3 | 336,6666667 | 1,03 | 346,7666667 |
| 4 | 275 | 1,13 | 310,75 |
| 5 | 238 | 1,23 | 292,74 |
| 6 | 213,3333333 | 1,33 | 283,7333333 |
| 7 | 195,7142857 | 1,43 | 279,8714286 |
| 8 | 182,5 | 1,53 | 279,225 |
| 9 | 172,2222222 | 1,63 | 280,7222222 |
| 10 | 164 | 1,73 | 283,72 |
| 11 | 157,2727273 | 1,83 | 287,8090909 |
| 12 | 151,6666667 | 1,93 | 292,7166667 |
| 13 | 146,9230769 | 2,03 | 298,2538462 |
| 14 | 142,8571429 | 2,13 | 304,2857143 |

Por lo que, entonces, llegamos a que N = 8 es nuestra mejor elección.

## Ejercicio 2

**Ejercicio 2: Predictores de saltos**
Asuma un microprocesador con 20 etapas de pipeline, con un fetch que levanta 5 instrucciones por ciclo. Este procesador ejecuta un código donde 1 de cada 5 instrucciones es un salto y esta conformado por bloques de 5 instrucciones donde la última es un salto. La penalidad por una mala predicción es de 20 ciclos. ¿Cuántos ciclos de instrucción toma hacer fetch de todas las instrucciones? Considerando predictores con las siguientes precisiones: 100%, 99%, 90%, 60%.

- Datos
  - Tenemos micro con 20 etapas de pipeline
  - Es 5-issue ⇒ el fetch levanta 5 instrucciones por ciclo
  - De las instrucciones que se levantan, solo la última (de cada bloque de 5) es un salto
  - La penalidad por mala predicción es de 20 ciclos ⇒ Se da cuenta que le erró recién en la última etapa del pipeline
- ¿Qué queremos? → Saber cuántos ciclos de instrucción toma hacer fetch de todas las instrucciones teniendo en cuenta las diferentes precisiones de cada predictor.
  - !!! Nota → Queremos saber cuántos ciclos toma hacer todos los fetch, no cuando se ejecutan todas las instrucciones

Dado esto, entonces, podemos notar que la cantidad de ciclos por instrucción dado el porcentaje de error $p$ y la cantidad de instrucciones $N$ (digamos $5|N$), entonces tenemos que la cantidad de ciclos que lleva hacer fetch de todas las instrucciones es de

$$f(N,p) = \frac{N}{5} + p \times \frac{N}{5} \times 20 = \frac{N}{5} \times (1 + 20 \times p)$$

Tener en cuenta que $p$ es un porcentaje.

Luego, entonces, tenemos que nuestra tabla queda en:

| N | % Acierto | % Error | f(N, p) |
|---|---|---|---|
| 500 | 100% | 0% | 100 |
| 500 | 99% | 1% | 120 |
| 500 | 90% | 10% | 300 |
| 500 | 60% | 40% | 900 |
| | | | |
| 3000 | 100% | 0% | 600 |
| 3000 | 99% | 1% | 720 |
| 3000 | 90% | 10% | 1800 |
| 3000 | 60% | 40% | 5400 |

# Ejercicio 3

**Ejercicio 3: Predictores de saltos**

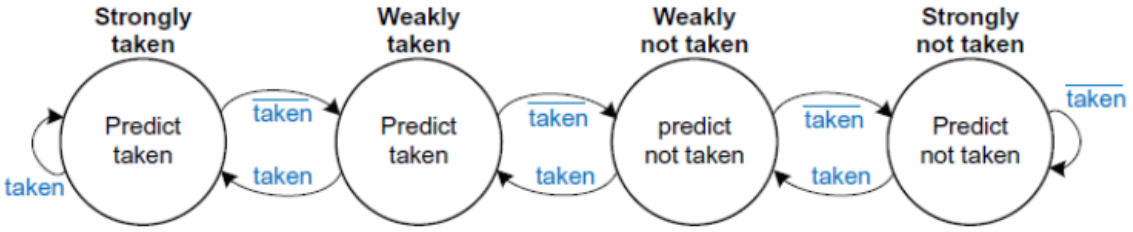



**Figure 7.62 Two-bit branch predictor state transition diagram**

Este ejercicio analiza la precisión de varios predictores de saltos para el siguiente patrón repetitivo (ej, en un loop) donde los saltos resultaron: **Taken - Not Taken - Taken - Taken - Not Taken.**

a) ¿Cuál es la precisión de los predictores *always-taken* y *always-not taken* para el patrón dado?
b) ¿Cuál es la precisión del predictor de *2-bits* para los primeros 4 saltos de este patrón? Asumir que el predictor arranca en *Strongly not taken*.
c) ¿Cuál es la precisión de este predictor de *2-bits* si el patrón completo se repite infinitamente?

## Ítem a

- **Always-taken** tiene un acierto de 3 sobre 5 en una sola iteración de este patrón, por lo que su precisión es de $\frac{3}{5} = 0.6$
- **Always-not taken** tiene un acierto de 2 sobre 5 en una sola iteración de este patrón, por lo que su precisión es de $\frac{2}{5} = 0.4$

## Ítem b

- Suposición → Arrancamos en *Strongly not taken* para el predictor de 2 bits
- Nuestra tabla de los primeros cuatro saltos del patrón es de

| Saltos tomados | Predicciones | Estado al que pasa (en bits) | Estado al que pasa (bits) |
|---|---|---|---|
| Taken | Not Taken | "00" | "01" |
| Not Taken | Not Taken | "01" | "00" |
| Taken | Not Taken | "00" | "01" |

| | | | |
|---|---|---|---|
| Taken | Not Taken | "01" | "10" |

- Luego, la presición para estos primeros cuatro saltos es de $\frac{1}{4} = 0.25$

### Ítem c

- Mismo predictor de (b) pero con patrón infinito
- La tabla nos queda:

| Saltos tomados | Predicciones | Estado al que pasa (en bits) | Estado al que pasa (bits) |
|---|---|---|---|
| Taken | Not Taken | "00" | "01" |
| Not Taken | Not Taken | "01" | "00" |
| Taken | Not Taken | "00" | "01" |
| Taken | Not Taken | "01" | "10" |
| Not Taken | Taken | "10" | "01" |
| | | | |
| Taken | Not Taken | "01" | "10" |
| Not Taken | Taken | "10" | "01" |
| Taken | Not Taken | "01" | "10" |
| Taken | Taken | "10" | "11" |
| Not Taken | Taken | "11" | "10" |
| | | | |
| Taken | Taken | "10" | "11" |
| Not Taken | Taken | "11" | "10" |
| Taken | Taken | "10" | "11" |
| Taken | Taken | "11" | "11" |
| Not Taken | Taken | "11" | "10" |

  - Gracias a la cual sabemos que toda iteración luego de la 3era va a ser igual a esta (dado que termina con estado “10” y empieza con ese mismo).
  - Por ello, entonces, si nuestro patrón es *infinito*, entonces va a converger a $\frac{3}{5} = 0.6$

## Ejercicio 4

**Ejercicio 4: Predictores de saltos**

Asumiendo que la distribución de instrucciones dinámicas se divide en las siguientes categorías:

| R-Type | CBZ/CBNZ | B | LDUR | STUR |
|---|---|---|---|---|
| 40% | 25% | 5% | 25% | 5% |

y las siguientes precisiones en los métodos de predicción de salto:

| Always-Taken | Always-Not-Taken | 2-Bit |
|---|---|---|
| 45% | 55% | 85% |

Considerando que el resultado y la dirección del salto se determinan en la etapa de decodificación (ID) y se aplican en la etapa de ejecución (EX) y que no hay hazards de datos.

a) ¿Cuántos CPI (Ciclos por instrucción) extras se producen debido a los fallos de predicción del método *Always-Taken*?

b) ¿Cuántos CPI (Ciclos por instrucción) extras se producen debido a los fallos de predicción del método *Always-Not-Taken*?

c) ¿Cuántos CPI (Ciclos por instrucción) extras se producen debido a los fallos de predicción del método *2-Bit*?

- Suposiciones:
  - Sin hazard de datos
  - Resultado y dirección de salto se determinan en DECODE y se aplican en EXECUTE ⇒ 1 ciclo de penalidad ante error

Ahora, para resolver el ejercicio, notemos que lo único que nos importa es el porcentaje de instrucciones de tipo branch **condicional** (que son las únicas que debemos predecir), es decir, las de CBZ/CBNZ, el cual es del 25%. Luego, entonces, veamos que:

- *Always-Taken*
  - Tiene presición del $45\%$ para CBZ/CBNZ
  - $0.25 \times (1 - 0.45) = 0.1375$
- *Always-Not-Taken*
  - Tiene presición del $55\%$ para CBZ/CBNZ
  - $0.25 \times (1 - 0.55) = 0.1125$
- *2-Bit*
  - $0.25 \times (1 - 0.85) = 0.0375$

# Ejercicio 5

**Ejercicio 5: Predictores de saltos**
El siguiente código en C puede escribirse en ARMv8 de la siguiente forma:

```
for (i = 0; i < 100; i++) {
   for (j = 0; j < 3; j++) {
      ...
   }
}
```

```
0x00:        add x0, xzr, xzr
0x04: L2:    add x1, xzr, xzr
0x08: L1:    ...
0x0C:        addi x1, x1, 1
0x10:        cmpi x1, 3
0x14:        b.lt L1
0x18:        addi x0, x0, 1
0x1C:        cmpi x0, 99
0x20:        b.lt L2
```

a) Mostrar como queda la tabla de historial de patrones (PHT) considerando que el procesador que ejecuta este código, cuenta con un predictor de saltos local de dos niveles.
b) Comparar la precisión de este predictor con uno de 2-bits (despreciando los primeros ciclos de iniciación).

## Ítem a

- Datos a tener en cuenta

    - Consideramos que usamos un *predictor de saltos local de dos niveles*, con $n = m = 4$ (i.e., tomamos cuatro bits de shift register y los 4 últimos del PC para usar en la PHT).
    - No consideramos los valores iniciales del GHR dado que se quiere saber el valor final de la tabla ⇒ Consideramos solo las combinaciones de condiciones finales que nos van a importar para los valores que quedarían en la PHT
- Veamos primero el valor que debería tomar para cada "combinación importante" de las condiciones de i, j:

| Vista desde el for de | Condición para i | Condición para j | Shift register (GHR) | Resultado del salto | Estado al que pasa |
|---|---|---|---|---|---|
| j | X | j = 1 | "1101" | Taken | "11" |
| j | X | j = 2 | "1011" | Taken | "11" |
| j | i < 100 | j = 3 | "0111" | Not Taken | "00" |
| i | i = 100 | j = 3 | "1110" | Not Taken | "10" |

- Luego, entonces, la PHT nos queda como:

| GHR | PC | Estado |
|---|---|---|
| "1101" | 0x14 -> "0100" | "11" |
| "1011" | 0x14 -> "0100" | "11" |
| "0111" | 0x14 -> "0100" | "00" |
| "1110" | 0x20 -> "0000" | "10" |

### Ítem b

- En este caso, para el predictor de dos bits, tenemos que considerar los mismos casos, pero tener en cuenta que únicamente nos interesa el "historial" del mismo salto.
- Luego, entonces, vemos que para el loop interno *siempre* va a fallar en la iteración de la condición de salida, al igual que con el loop externo. Por ello, entonces, su precisión (en caso promedio de iteración) será de $\frac{3}{4} = 0.75$
- Respecto al predictor de *dos niveles*, tenemos que su precisión va a ser de $\frac{4}{4} = 100\%$ en el caso promedio, luego de *entrenarse* con algunas iteraciones iniciales

> 💡 Estos valores son para el caso promedio de iteración. Las condiciones de salida de i las van a fallar los dos al final de todo por ejemplo, por lo que *ninguno* es 100% preciso.

## Ejercicio 6

**Ejercicio 6: Predictores de saltos**
Asuma que el siguiente código itera en un array largo y lleno de números enteros positivos aleatorios. El código cuenta con 4 saltos, etiquetados B1, B2, B3 y B4. Cuando decimos que un salto es Taken, nos referimos a que el código dentro de las llaves es ejecutado.

```
for (int i=0; i<N; i++) {        /* B1 */
   val = array[i];               /* TAKEN PATH for B1 */
   if (val % 2 == 0) {           /* B2 */
      sum += val;                /* TAKEN PATH for B2 */
   }
   if (val % 3 == 0) {           /* B3 */
      sum += val;                /* TAKEN PATH for B3 */
   }
   if (val % 6 == 0) {           /* B4 */
      sum += val;                 /* TAKEN PATH for B4 */
   }
}
```

a) Determinar cuál de los cuatro saltos muestra una correlación local.
b) ¿Existe correlación global entre algunos de los saltos? Explicar.

### Ítem a

- B2, B3 y B4 no tienen *correlación local* dado que depende de un valor aleatorio en memoria (en el array)
- B1, por su parte, sí tiene correlación local dado que es la condición del *for* y puede ser predecido localmente

### Ítem b

- Respecto a la correlación *global*, esta se encuentra para B4 respecto a B2 y B3
  - Si B2 y B3 se *toman*, entonces B4 también
  - Caso contrario, si alguno *no se toma*, entonces B4 tampoco

## Ejercicio 7

Recomiendo ver la explicación de los profes en la *clase 3* → https://drive.google.com/file/d/1d_qE34jtwmv9Zu7j4WHqmDT856_v-xQA/view?usp=sharing

Ver desde el minuto **1:01:50** al **1:34:12.**

Si se quiere ver explicación más detallada sobre el *camino de ejecución* (lo cual sirve para ejercicios posteriores), ver hasta el minuto **1:39:38**

**Ejercicio 7: Static Multiple Issue Processor**

Considere un procesador LEGv8 two-issue, donde en cada "issue packet" una de las instrucciones puede ser una operación de la ALU o un salto y la otra puede ser un *load* o *store*, tal como se muestra en la figura. El compilador asume toda la responsabilidad de eliminar los hazard, organizar el código e insertar operaciones tipo "nop" para que el código se ejecute sin necesidad de detección de hazard o generación de stalls.

| Instruction type | Pipe stages | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ALU or branch instruction | IF | ID | EX | MEM | WB | | | |
| Load or store instruction | IF | ID | EX | MEM | WB | | | |
| ALU or branch instruction | | IF | ID | EX | MEM | WB | | |
| Load or store instruction | | IF | ID | EX | MEM | WB | | |
| ALU or branch instruction | | | IF | ID | EX | MEM | WB | |
| Load or store instruction | | | IF | ID | EX | MEM | WB | |
| ALU or branch instruction | | | | IF | ID | EX | MEM | WB |
| Load or store instruction | | | | IF | ID | EX | MEM | WB |

Considere el siguiente bucle:

```
Loop:   LDUR X0, [X20,#0]   // X0=array element
        ADD X0,X0,X21       // add scalar in X21
        STUR X0, [X20,#0]   // store result
        SUBI X20,X20,#8     // decrement pointer
        CMP X20,X22         // compare to loop limit
        B.GT Loop           // branch if X20 > X22
```

a) Analice en el código las dependencias de datos y determine cuales generan data hazards en nuestro procesador one-issue, **sin** forwarding-stall. En cada caso indique: el tipo de hazard, el operando en conflicto y los números de las instrucciones involucradas. Suponga que los saltos están perfectamente predichos, de modo que los *control hazard* son manejados por hardware.

b) ¿Cómo se organizaría el siguiente código en un procesador two-issue con pipeline y forwarding-stall para evitar la mayor cantidad posible de stalls?.

c) Suponga que el compilador es capaz de determinar que la cantidad de iteraciones del bucle se da en múltiplos de 2. Utilice la técnica estática "*loop unrolling*" para re-ordenar la ejecución del código. Determine mejora en el tiempo de ejecución respecto al punto (b). Se calcula como: *Tiempo de ejecución sin mejora/ Tiempo de ejecución con mejora*

## Ítem a

| Registro | Instrucciones | Tipo de dependencia | ¿Causa hazard? |
|---|---|---|---|
| x0 | 1, 2 | RAW | Sí |
| x0 | 1, 2 | WAW | No |
| x0 | 2, 3 | RAW | Sí |
| x20 | 4, 5 | RAW | Sí |
| x20 | 4, 1 | RAW (cond.) | No |
| x20 | 4, 3 | RAW (cond.) | No |
| x20 | 4, 4 | RAW (cond.) | No |
| x20 | 4, 4 | WAW (cond.) | WAW (cond.) |

## Ítem b

- Sin usar *loop-unrolling* y simplemente organizando mejor el código de 1 iteración, tenemos la siguiente disposición de los *issue packets* (teniendo en cuenta las dependencias mencionadas en el ítem a):

| Tags | ALU or branch instruction | Data transfer instruction | CLK |
|---|---|---|---|
| Loop: | SUBI X20,X20,#8 | LDUR X0, [X20,#0] | 1 |
| | CMP X20,X22 | nop | 2 |
| | ADD X0,X0,X21 | nop | 3 |
| | B.GT Loop | STUR X0, [X20,**#8**] | 4 |

## Ítem c

- Ahora, si usamos *loop-unrolling* con dos iteraciones. Entonces tenemos el siguiente código a considerar:

```
Loop:   LDUR X0, [X20,#0] // X0=array element
        ADD X0,X0,X21 // add scalar in X21
        STUR X0, [X20,#0] // store result
        SUBI X20,X20,#8 // decrement pointer

        LDUR X0, [X20,#0] // X0=array element
        ADD X0,X0,X21 // add scalar in X21
        STUR X0, [X20,#0] // store result
        SUBI X20,X20,#8 // decrement pointer
        CMP X20,X22 // compare to loop limit
        B.GT Loop // branch if X20 > X22
```

- Ahora, para evitar conflictos con *nombre de registros*, vamos a considerar una pequeña variante cambiando los X0 de la segunda parte por X1:

```
Loop:   LDUR X0, [X20,#0] // X0=array element
        ADD X0,X0,X21 // add scalar in X21
        STUR X0, [X20,#0] // store result
        SUBI X20,X20,#8 // decrement pointer

        LDUR X1, [X20,#0] // X0=array element
        ADD X1,X1,X21 // add scalar in X21
        STUR X1, [X20,#0] // store result
        SUBI X20,X20,#8 // decrement pointer
        CMP X20,X22 // compare to loop limit
        B.GT Loop // branch if X20 > X22
```

- Como último paso antes de armar los *packets*, vamos a *compactar los SUBIS*, quedándonos con lo siguiente:

```
Loop:   LDUR X0, [X20,#0] // X0=array element
        ADD X0,X0,X21 // add scalar in X21
        STUR X0, [X20,#0] // store result

        LDUR X1, [X20,#-8] // X0=array element
        ADD X1,X1,X21 // add scalar in X21
        STUR X1, [X20,#-8] // store result
        SUBI X20,X20,#16 // decrement pointer
        CMP X20,X22 // compare to loop limit
        B.GT Loop // branch if X20 > X22
```

- Luego, entonces, si reorganizamos el código, nos queda que los *issue packets* son:

| Tags | ALU or branch instruction | Data transfer instruction | CLK |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Loop: | SUBI X20,X20,#16 | LDUR X0, [X20,#0] | 1 |
| | CMP X20,X22 | LDUR X1, [X20,#8] | 2 |
| | ADD X0,X0,X21 | nop | 3 |
| | ADD X1,X1,X21 | STUR X0, [X20,#16] | 4 |
| | B.GT Loop | STUR X1, [X20,#8] | 5 |

- Dado esto, ahora tenemos que comparar los tiempos **sin mejora** contra los tiempos **con mejora**. Para ello, comparemos (b) con (c).
  - Notemos que dos iteraciones de (b) implican 8 CLKs
  - Y que dos iteraciones (condensadas en una sola por loop-unrolling) en (c) implican 5 CLKs

  Luego, entonces, la mejora es de $\frac{3N}{2}$CLKs donde $N$ es la cantidad de iteraciones.

# Ejercicio 8 (Falta hacer)

**Ejercicio 8: Static Multiple Issue Processor**

En este ejercicio se compara el rendimiento de los procesadores de 1-issue y 2-issue, teniendo en cuenta las transformaciones que se pueden realizar en un programa para optimizar la ejecución de 2-issue.

Los problemas en este ejercicio se refieren al siguiente bucle (escrito en C):

```
for(i=0;i!=j;i+=2)
    b[i]=a[i]-a[i+1];
```

El código utiliza los siguientes registros:

| i | j | a | b | Temporary values |
|---|---|---|---|---|
| X5 | X6 | X1 | X2 | X10-X15 |

Un compilador con poca o ninguna optimización podría generar el siguiente código de assembler LEGv8:

```
        ADD X5, XZR, XZR
        B ENT
TOP:    LSL X10, X5, #3
        ADD X11, X1, X10
        LDUR X12, [X11, #0]
        LDUR X13, [X11, #8]
        SUB X14, X12, X13
        ADD X15, X2, X10
        STUR X14, [X15, #0]
        ADDI X5, X5, #2
ENT:    CMP X5, X6
        B.NE TOP
```

Asumiendo que el procesador de 2-issue tiene las siguientes propiedades:

1. En cada *issue packet* una instrucción debe ser una operación de memoria y la otra una de tipo aritmética/lógica o un salto.
2. El procesador tiene todos los caminos de forwarding posibles entre las etapas (incluyendo los caminos a la etapa ID para la resolución de saltos).
3. El procesador predice los saltos perfectamente.
4. Dos instrucciones no pueden procesarse juntas en un paquete si una depende de la otra.
5. Si se requiere un stall, ambas instrucciones en el paquete deben volverse stall.

a) Analice en el código las dependencias de datos y determine cuales generan data hazards en nuestro procesador one-issue, **sin** forwarding-stall. En cada caso indique: el tipo de hazard, el operando en conflicto y los números de las instrucciones involucradas. Suponga que los saltos están perfectamente predichos, de modo que los *control hazard* son manejados por hardware.
b) Dibuje un diagrama de pipeline que muestre cómo se ejecuta el código LEGv8 dado anteriormente en el procesador de 2-issue con pipeline y forwarding-stall. (Suponga que sale del bucle después de dos iteraciones.)
c) ¿Cuál es el aumento de velocidad al pasar de un procesador de 1-issue a un procesador de 2-issue, ambos con pipeline y forwarding-stall? (Suponga que el bucle ejecuta miles de iteraciones).
d) Reorganice/reescriba el código LEGv8 dado anteriormente para lograr un mejor rendimiento en el procesador de 2-issue. (No utilice la técnica de *loop unrolling*).
e) Repita el inciso (b), pero usando el código optimizado en el inciso (d).
f) Aplique la técnica de *loop unrolling* al código LEGv8 del inciso (d) para que cada iteración del nuevo bucle se corresponda con dos iteraciones del bucle original. Luego, reorganice/reescriba su nuevo código para lograr un mejor rendimiento en el procesador de 2-issue. Está permitido utilizar otros registros si se considera necesario. Puede asumir que **j** es un múltiplo de 4. (Sugerencia: reorganice el bucle para que algunos cálculos aparezcan fuera del bucle y al final del bucle. Puede suponer que los valores de los registros temporales no son necesarios después del bucle).
g) Repita el inciso (f), pero esta vez suponga que el procesador de 2-issue puede ejecutar dos instrucciones aritméticas/lógicas juntas. (En otras palabras, la primera instrucción en un paquete puede ser cualquier tipo de instrucción, pero la segunda debe ser una instrucción aritmética o lógica. No se pueden programar dos operaciones de memoria al mismo tiempo).
h) Comparar el aumento de velocidad de los incisos (d), (f) y (g) respecto al (b).

# Ejercicio 9 (Falta hacer)

**Ejercicio 9: Static Multiple Issue Processor**
Para los siguientes fragmentos de código LEGv8:

| A | B |
|---|---|
| `LDUR X8, [X0, #40]`<br>`ADD X9, X1, X2`<br>`SUB X10, X1, X3`<br>`AND X11, X3, X4`<br>`ORR X12, X1, X5`<br>`STUR X5, [X0, #80]` | `loop: LDUR X0, [X20,#0]`<br>`LDUR X1, [X21,#0]`<br>`ADD X0,X0,X1`<br>`STUR X0,[X20,#0]`<br>`ADDI X20, X20, #8`<br>`ADDI X21, X21, #8`<br>`SUBI X2, X2, #1`<br>`CBZ X2, loop` |

a) Analizar las dependencias de datos y determinar cuales generan data hazards en nuestro procesador one-issue, **sin** forwarding-stall. En cada caso indicar: el tipo de hazard, el operando en conflicto y los números de las instrucciones involucradas. (Los saltos están perfectamente predichos, de modo que los *control hazard* son manejados por hardware).
b) Mostrar el orden de ejecución para un microprocesador con pipeline, de una vía, con *forwarding-stall* y que predice los saltos perfectamente.
c) Mostrar el orden de ejecución para un microprocesador multiple-issue de dos vías, con todos los caminos de forwarding posibles entre las etapas y que predice los saltos perfectamente. En cada *issue packet* la primera instrucción puede ser cualquier tipo y la segunda debe ser una instrucción aritmética o lógica.
d) En el fragmento (B): aplicar la técnica de *loop unrolling* para que cada iteración del nuevo bucle se corresponda con dos iteraciones del bucle original (X2 es múltiplo de 2). Luego reorganizar/reescribir el código para lograr un mejor rendimiento en el procesador de 2-issue del inciso (c). Está permitido utilizar otros registros si se considera necesario.
e) Calcular la mejora en eficiencia para cada técnica.

# Ejercicio 10 (tipo parcial) (Falta hacer)

**Ejercicio tipo parcial:**

Un procesador 2-issue de arquitectura LEGv8 posee las siguientes propiedades:

1. En cada *issue packet* una instrucción debe ser una operación de acceso a memoria y la otra de tipo aritmética/lógica o un salto.
2. El procesador tiene todos los caminos de forwarding posibles entre las etapas (incluyendo caminos a la etapa ID para la resolución de saltos).
3. El procesador predice los saltos perfectamente.
4. Dos instrucciones no pueden procesarse juntas en un paquete si una requiere el resultado de la otra.
5. El compilador asume toda la responsabilidad de eliminar los hazard, organizar el código e insertar instrucciones "nop" para que el código se ejecute sin necesidad de generación de stalls.

Para el siguiente fragmento de código LEGv8 (donde X2 = 0):

```
       1>  ADDI X0, XZR, #0x100
       2>  ADDI X10, XZR, #50
loop:  3>  LDUR X1, [X0,#0]
       4>  ADD X2, X2, X1
       5>  LDUR X1, [X0,#8]
       6>  SUBI X10, X10, #1
       7>  ADD X2, X2, X1
       8>  STUR X2, [X0,#8]
       9>  ADDI X0, X0, #16
      10> CBNZ X10, loop
      ...
```

**a.** Dibuje un diagrama de pipeline que muestre cómo se ejecuta el código LEGv8 dado en el procesador de 2-issue (sólo hasta completar una iteración del bucle). Sin modificar el orden de ejecución, organice el código para evitar la mayor cantidad posible de *stalls*. Deje indicados los caminos de *forwarding* utilizados. (Completar en la tabla dada al final del ejercicio).

**b.** Suponiendo que no es económicamente viable integrar los multiplexores de tres entradas que son necesarios para implementar todos los caminos de *forwarding*, analice el código dado y determine si es mejor reenviar solo desde el registro de pipeline EX / MEM ( EX → EX) o solo desde el registro MEM / WB (MEM → EX).

**c.** Indique el aumento de velocidad en la ejecución del código dado al pasar de un procesador de 1-issue a un procesador de 2-issue, ambos con pipeline y forwarding-stall. Considere la totalidad de las iteraciones realizadas por el código.

**d.** Cuántas instrucciones LDUR se ejecutarán por lazo si se aplica la técnica del loop-unrolling al código dado, con el fin de minimizar la cantidad de iteraciones del lazo? Asuma que X10 se inicializa en la instrucción <2> con un valor múltiplo de 4.