Práctico N° 6: Técnicas de mejora de rendimiento

Arquitectura de Computadoras 2023

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?

Ejercicio 1

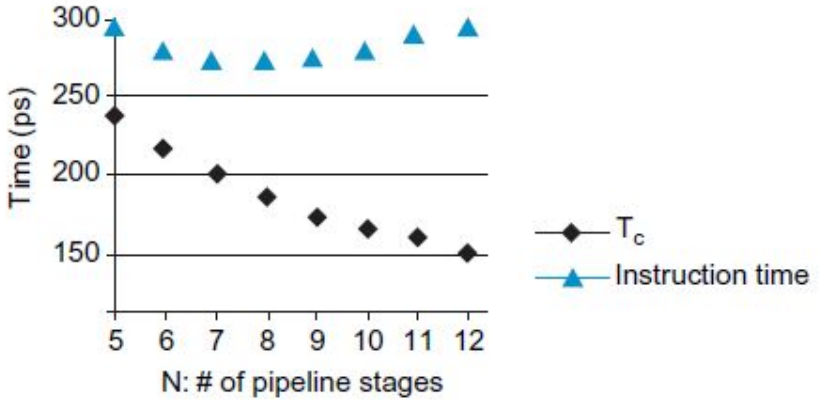
- Latencia de una etapa = Tiempo de ciclo (Tc) = (740/N+ 90) ps
- CPI = 1.23 + 0.1(N–5)
- Tiempo por instrucción (Ti) = Tc * CPI

N	Tc [ps]	CPI	Ti [ps]
5	238	1.23	292.74
6			
7			
8			
9			
10			

Ejercicio 1

- Latencia de una etapa = Tiempo de ciclo (Tc) = (740/N+ 90) ps
- CPI = 1.23 + 0.1(N–5)
- Tiempo por instrucción (Ti) = Tc * CPI

N	Tc [ps]	CPI	Ti [ps]
5	238	1.23	292.74
6	213.33	1.33	283.73
7	195.71	1.43	279.87
8	182.5	1.53	279.23
9	172.22	1.63	280.72
10	164	1.73	283.72



Predicción de saltos

Sección 3.3 y Apéndice C: Computer Architecture - A quantitative Approach, Hennessy y Patterson

El problema a resolver

¿Cuál es la próxima instrucción a hacer fetch?

La instrucción aún no se decodificó (no sabemos si es un salto)

Para levantar la próxima instrucción correcta debemos saber:

1) Si la instrucción anterior es un salto
2) En el caso de un salto condicional, si el salto se toma o no
3) En caso de que se tome, cuál es la dirección de salto.

La predicción de saltos intenta predecir la siguiente instrucción a ejecutar, no solo si es necesario saltar o no, también a qué dirección saltar

Ejercicio 2

- Asuma N = 20 (20 etapas de pipeline), W = 5 (5 instrucciones leídas por fetch)
- Asuma: bloques de 5 instrucciones donde la última es un salto
- El procesador se da cuenta que el salto tomado fue incorrecto en la última etapa (penalidad de 20 ciclos)

¿Cuántos ciclos de reloj toma hacer fetch de todas las instrucciones de un código de 500 instrucciones? Considerando predictores con las siguientes precisiones: 100%, 99%, 90%, 60%.

Precisión del 100%:
100 ciclos. No se hace ningún fetch incorrecto.

Precisión del 99%:
100 (camino de ejecución correcto) + 20 (penalidad) = 120 ciclos
20% instrucciones extras levantadas.

Precisión del 90%:
100 (camino de ejecución correcto) + 20 * 10 (penalidad) = 300 ciclos
200% instrucciones extras levantadas.

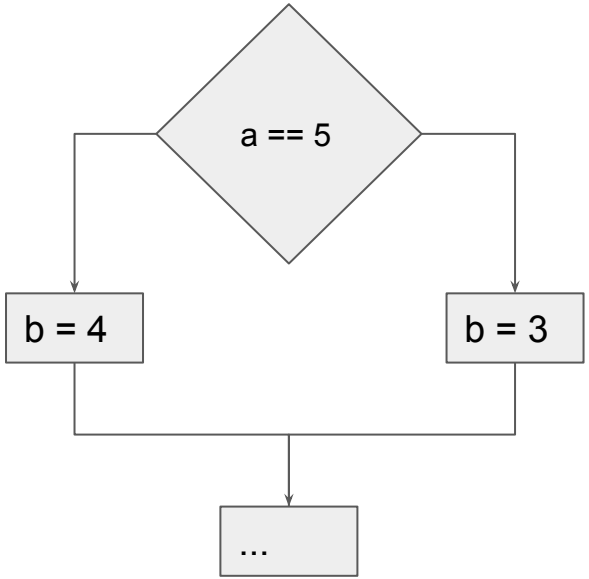
Precisión del 60%:
100 (camino de ejecución correcto) + 20 * 40 (penalidad) = 900 ciclos
800% instrucciones extras levantadas.

¿Es posible evitar el salto?

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

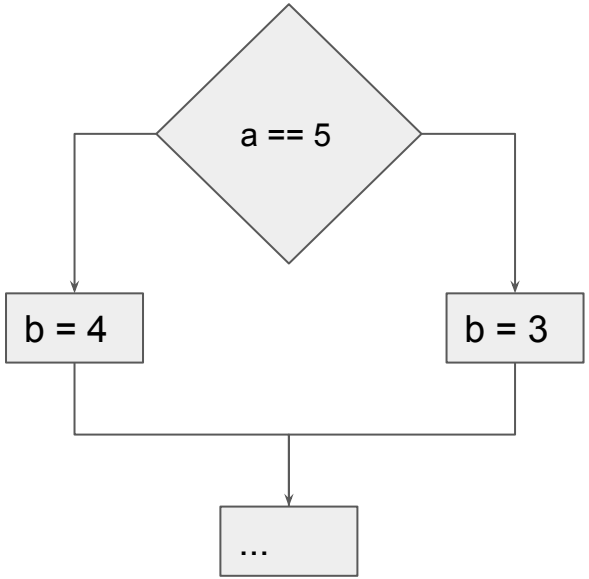
Alternativa 1:

```
     cmp   x0, 5
     b.ne  L2
     mov   x1, 4
     b     L3
L2:
     mov   x1, 3
L3:
```



¿Es posible evitar el salto?

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



Alternativa 2:

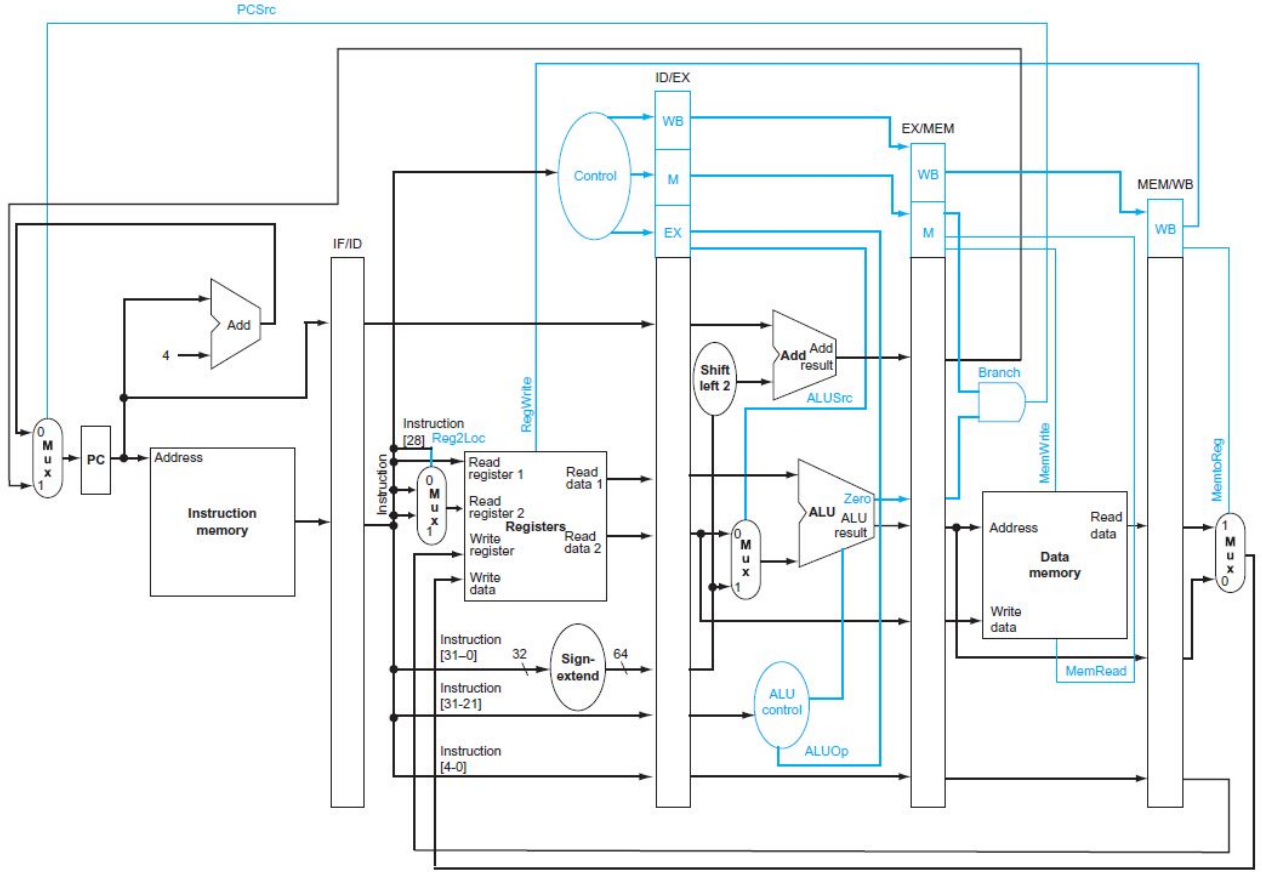
```
cmpi  x0, 5
mov   x0, 4
mov   x1, 3
csel  x1, x0, x1, eq
```

```
CSEL   rd, rn, rm, cc //  if(cc) rd = rn; else rd = rm
```

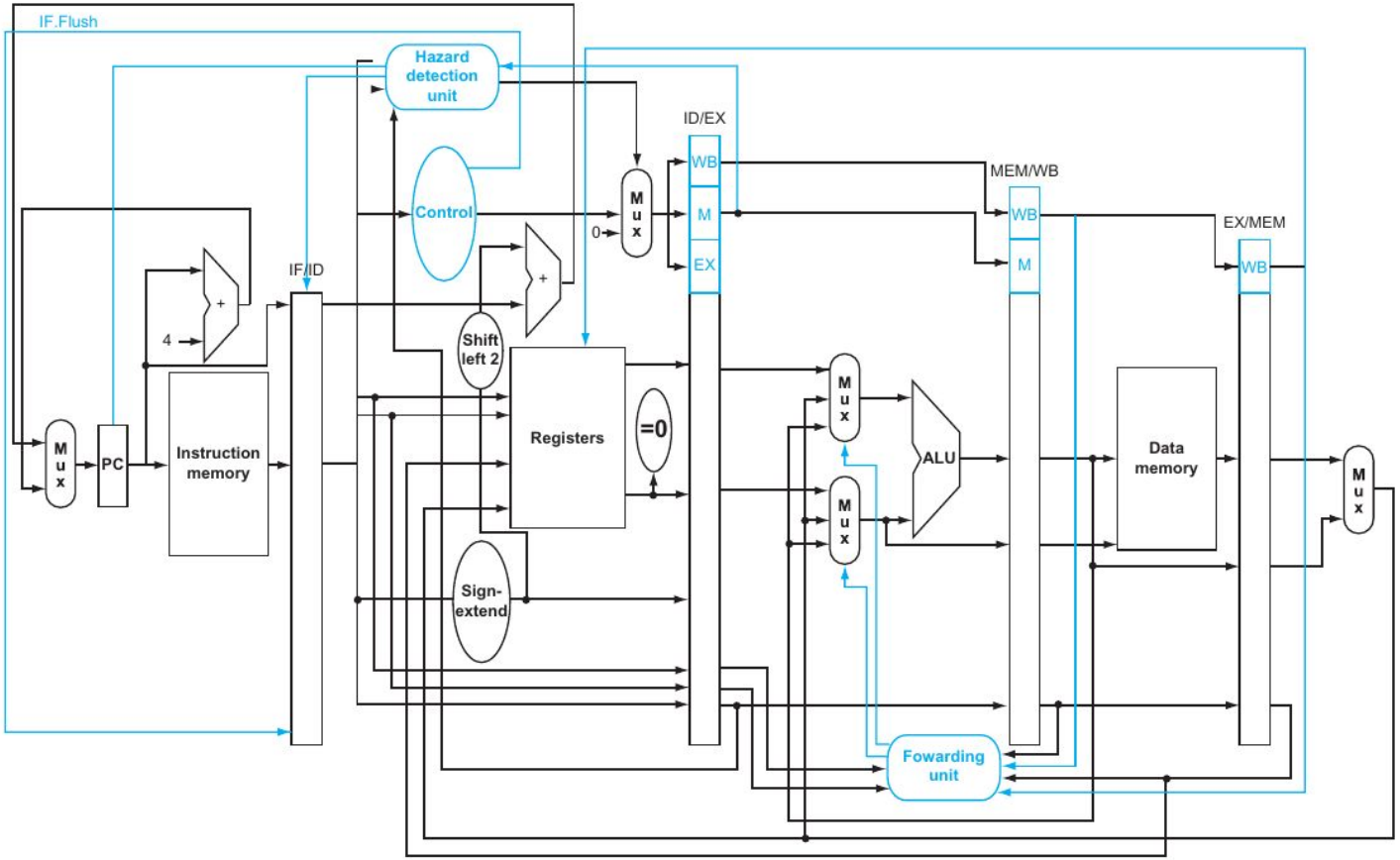
¿Es posible evitar el salto?

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	

El problema a resolver



El problema a resolver

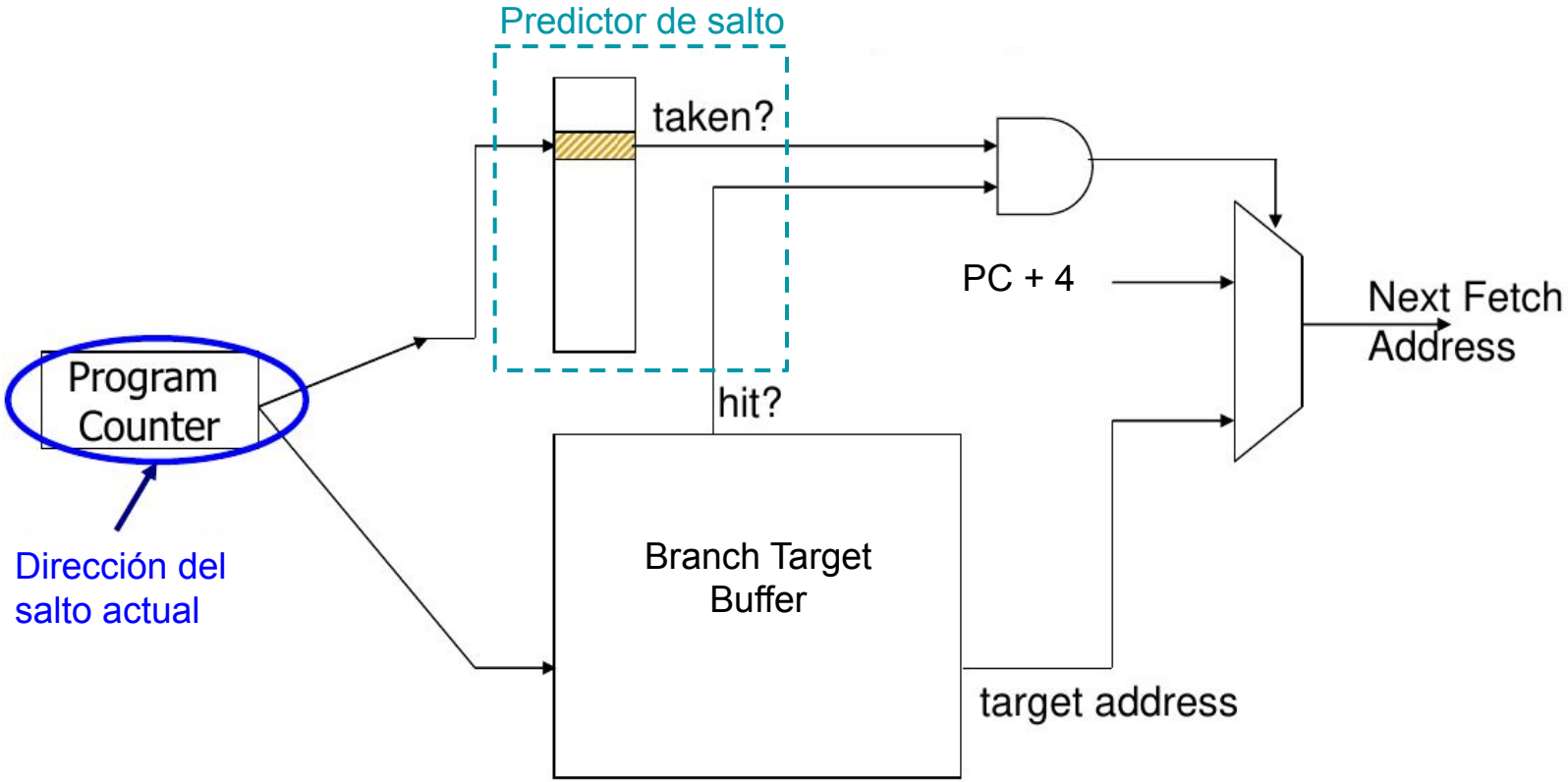


Predicción de la dirección de salto

Branch Target Buffer (BTB): Es una pequeña memoria asociativa que almacena la dirección donde se encuentra un salto y el destino.

PC	Target

Predicción de dirección de salto



Tipo de saltos

Tipo de salto	¿Sabemos la dirección al hacer fetch?	Número de posibles direcciones de salto	En qué etapa se resuelve la dirección del salto
Condicional (CBZ, CBNZ, B.cond)	No (PC + offset o PC + 4)	2	Ex o Dec (ver mod)
Incondicional - B	Si (PC + offset)	1	Dec (PC + offset)
Salto con registro - BR	No (PC = Reg)	Muchos	Ex

Predicción de saltos

Estáticos

- Not taken
- Taken
- BTFNT (backward taken, forward not taken)

Dinámicos

- Predictor local (2 bit)
- Predictor de dos niveles (2 bits mejorado)
- Predictor gshared
- Predictor por torneos (Tournament predictors)
- Tagged Hybrid predictors

Not taken

Ventajas:

- Implementación más sencilla
- No necesita predecir la dirección de salto
- Si acierta no tiene penalidad

Desventajas:

- Baja precisión (30-40% para saltos condicionales)
- Si se equivoca la penalidad es de 3 ciclos de reloj

Taken

Ventajas:

- Mejor precisión en la predicción salto o no salto (60-70% para saltos condicionales)

Desventajas:

- Implementación más compleja
- Siempre tiene penalidad de 1 ciclo

BTFNT

BTFNT: Backward Taken, Forward Not Taken

- Es sencillo saber si el salto es hacia adelante o hacia atrás (offset >0 o <0).

- Cuando un salto es hacia atrás sabemos que es un Loop, la mejor precisión en estos casos se obtiene con la predicción Taken.

- Cuando el salto es hacia adelante, es igualmente probable que sea tomado o no el salto. Los compiladores intentan predecir y organizar el código para que sea el salto no se tome en la mayoría de los casos.

Predicción dinámica de saltos

A medida que una instrucción de saltos es ejecutada, se almacena información sobre el resultado.

Esta información luego es utilizada para intentar predecir el resultado de ejecuciones posteriores de este salto.

Tipos de predictores dinámicos

Predictores locales: Basan su predicción en la información de ejecuciones anteriores del mismo salto.

Predictores globales: Realizan la predicción utilizando la información de otros saltos del programa a ejecutar.

Ej

```
if (cond1)
...
if (cond1 AND cond2)
...
```

```
if (cond1)
   a = 2;
if (a == 0)
...
```

```
if (aa == 2)       //B1
   aa = 0;
if (bb == 2)       //B2
   bb = 0;
if (aa != bb) {    //B3
...
}
```

Predictor de 2 bits (local)

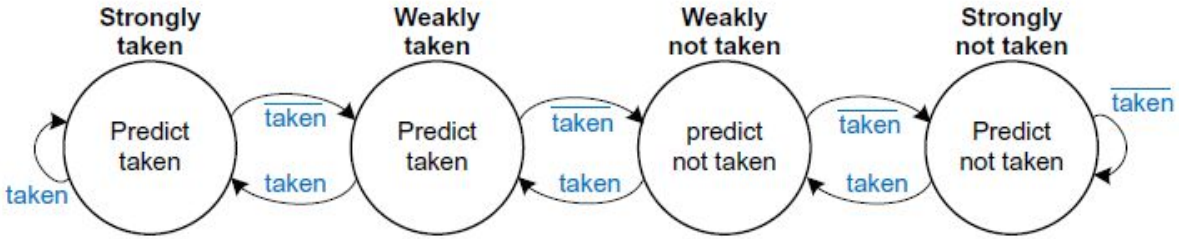
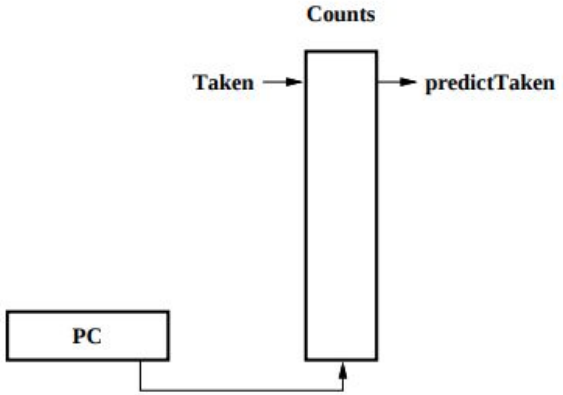


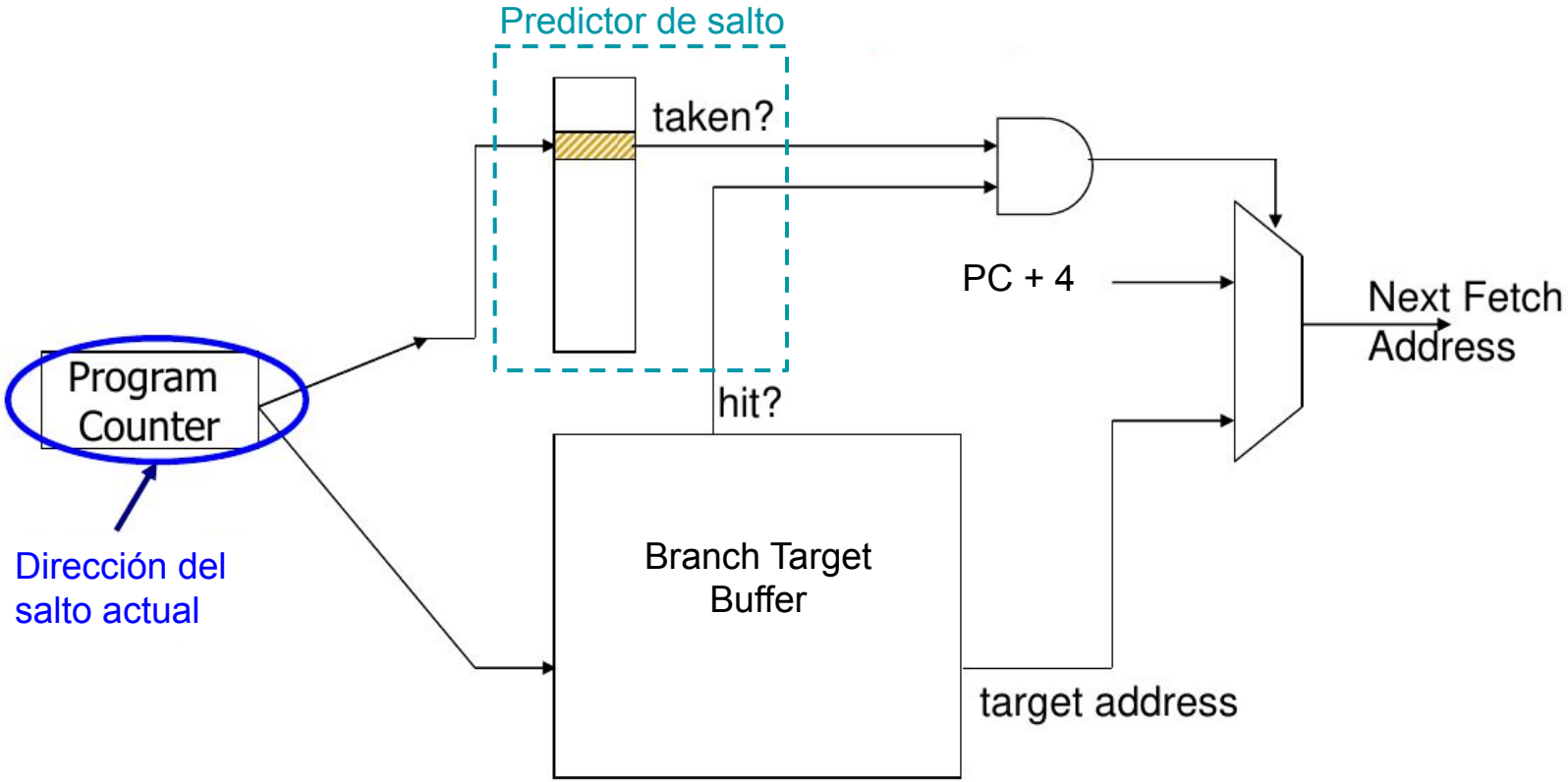
Figure 7.62 Two-bit branch predictor state transition diagram



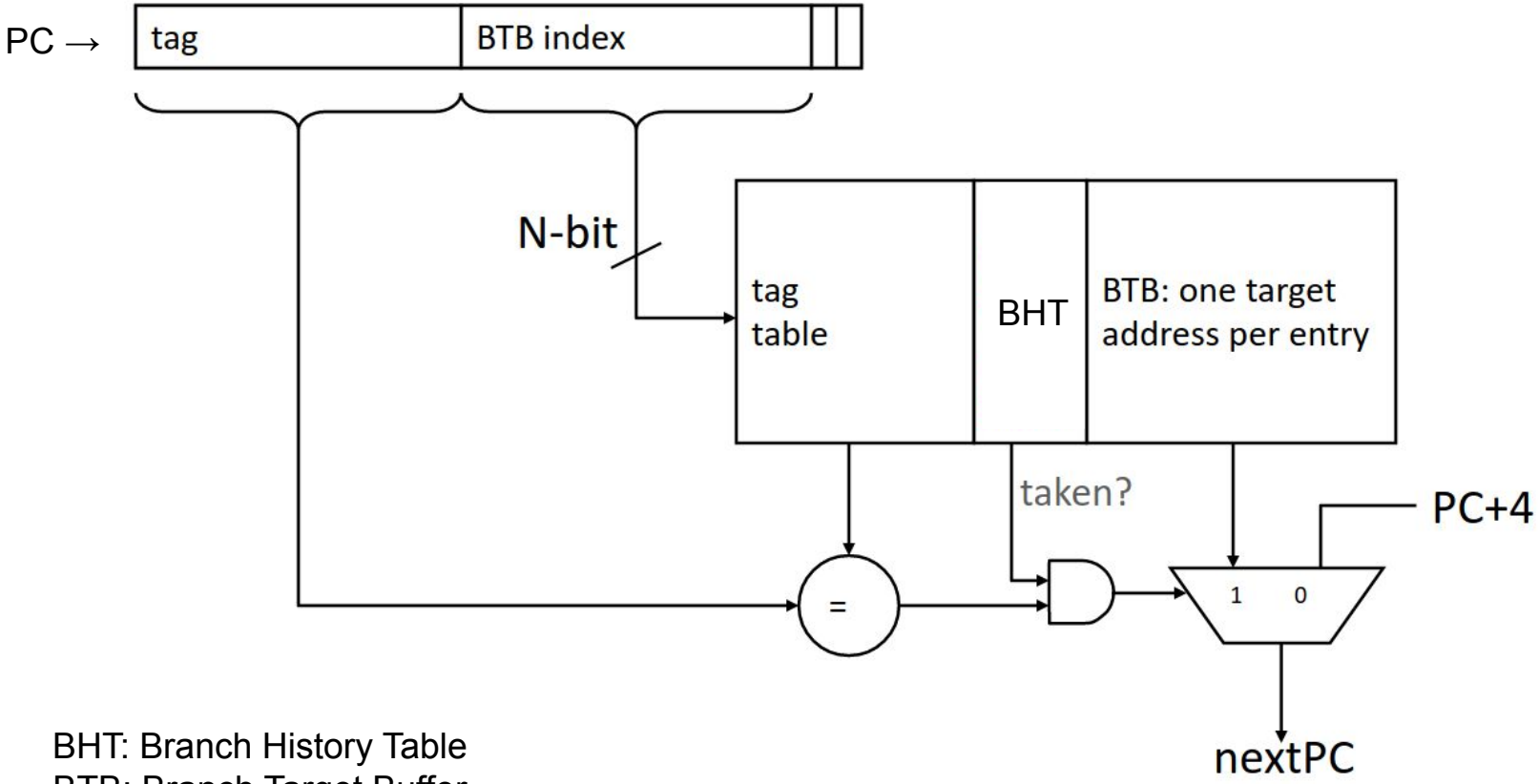
Se implementa una pequeña memoria asociativa (BHT) donde se almacena:

- Los últimos bits de la dirección de la instrucción de salto (a modo de tag)
- 2 Bits indicando el estado en el esquema de salto

Predictor de 2 bits (local)

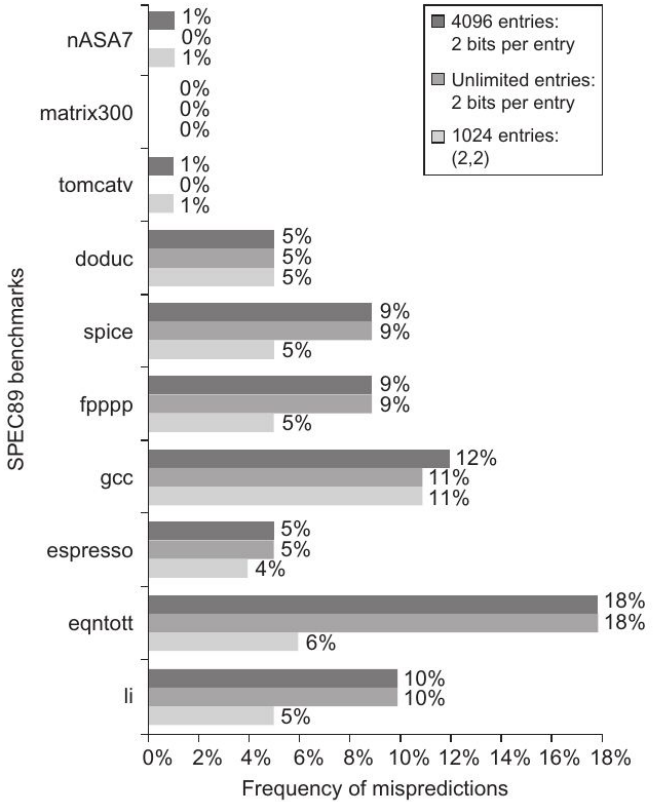


Predictor de 2 bits (local)



BHT: Branch History Table
BTB: Branch Target Buffer

Desventajas del predictor de 2 bits



- El tamaño de la memoria asociativa es un criterio de diseño, en un punto, seguir agrandandola no mejora el rendimiento.

- Este predictor no tiene en cuenta el resultado de otros saltos:

```
     addi x3,x1,-2
     cbz x3,L1       //branch b1 (a!=2)
     add x1,x0,x0    //a=0
L1:  addi x3,x2,-2
     cbz  x3,L2      //branch b2 (b!=2)
     add x2,x0,x0    //b=0
L2:  sub x3,x1,x2    //x3=a-b
     cbz  x3,L3      //branch b3 (a==b)
```

Ejercicio 3 - Branch Prediction

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?

Ejercicio 3-a

Patrón de saltos: **Taken - Not Taken - Taken - Taken - Not Taken**

Precisión del predictor always-taken = ⅗ = 60%

Precisión del predictor always-not taken = ⅖ = 40%

Ejercicio 3-b

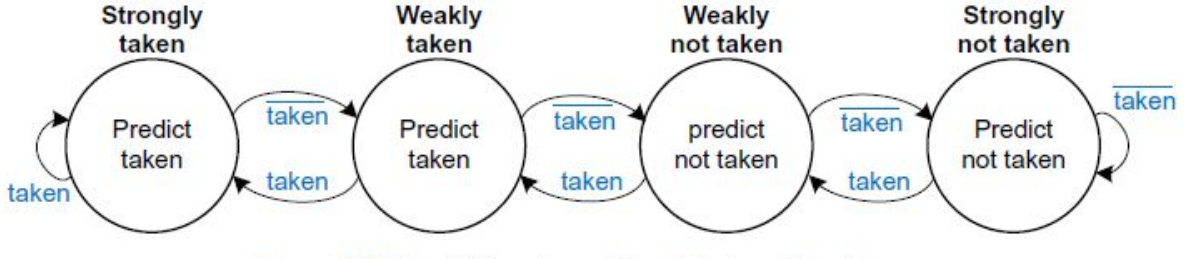


Figure 7.62 Two-bit branch predictor state transition diagram

Patrón de salto	**T**	**NT**	**T**	**T**	**NT**
Predicción	**NT**	**NT**	**NT**	**NT**	**T**
Falla o Acierta	F	A	F	F	F

Precisión del predictor = ⅕ = 20%

Ejercicio 3-c

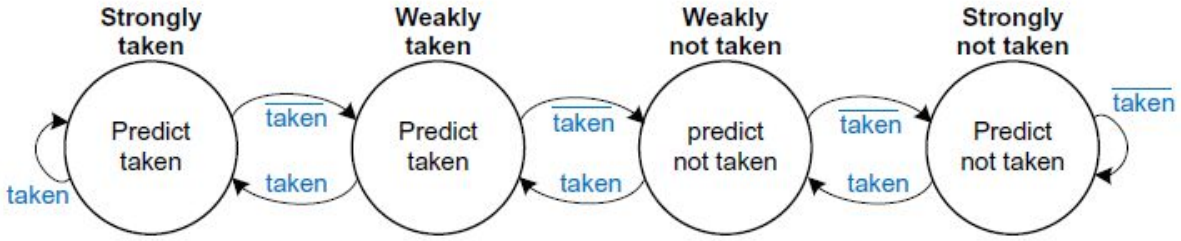


Figure 7.62 Two-bit branch predictor state transition diagram

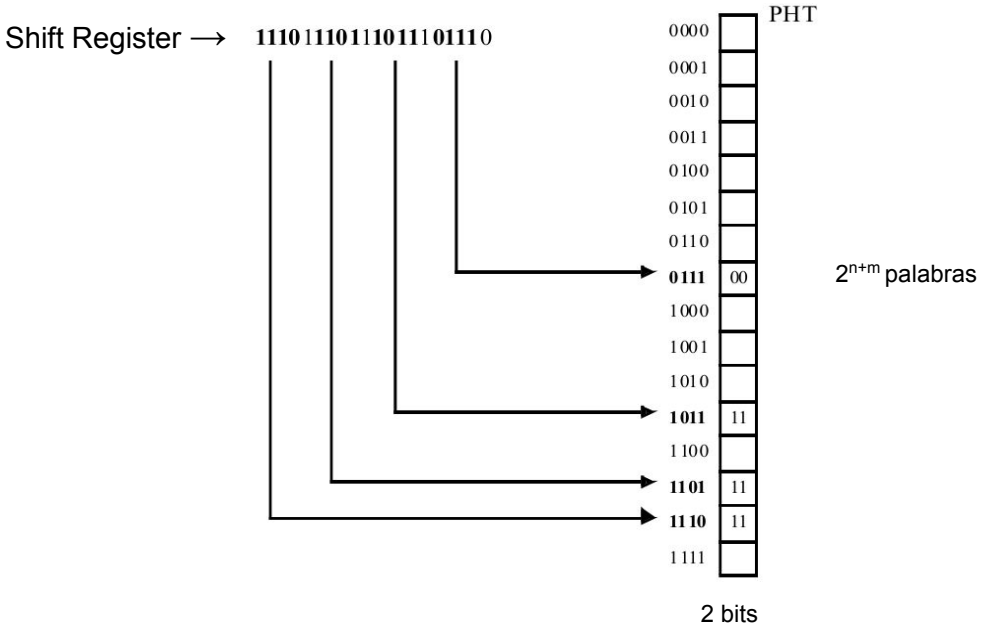
Patrón de salto	**T**	**NT**	**T**	**T**	**NT**
Loop 1	F	A	F	F	F
Loop 2	F	F	F	A	F
Loop 3	A	F	A	A	F

Precisión del predictor = ⅗ = 60%

Predictor global

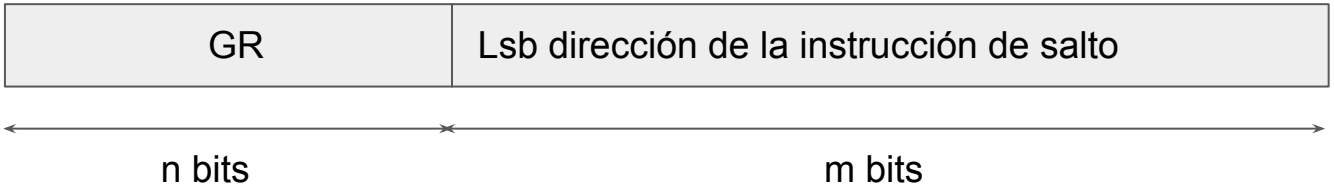
- Se crea un shift register de n bits donde se va almacenando el resultado de los últimos branchs (1 taken - 0 not taken)

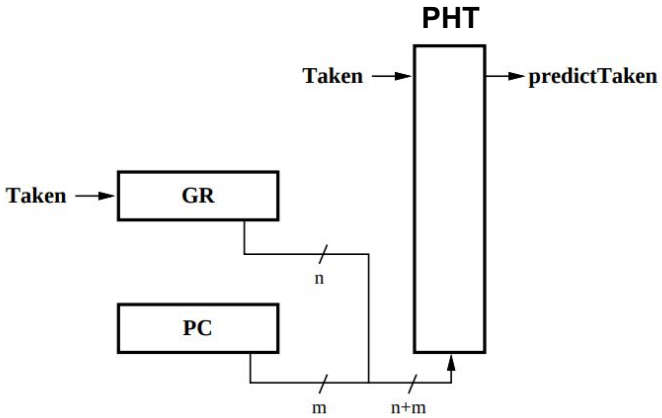
- El resultado se almacena en una tabla (PHT - pattern history table) de 2^{n+m} palabras de dos bits .



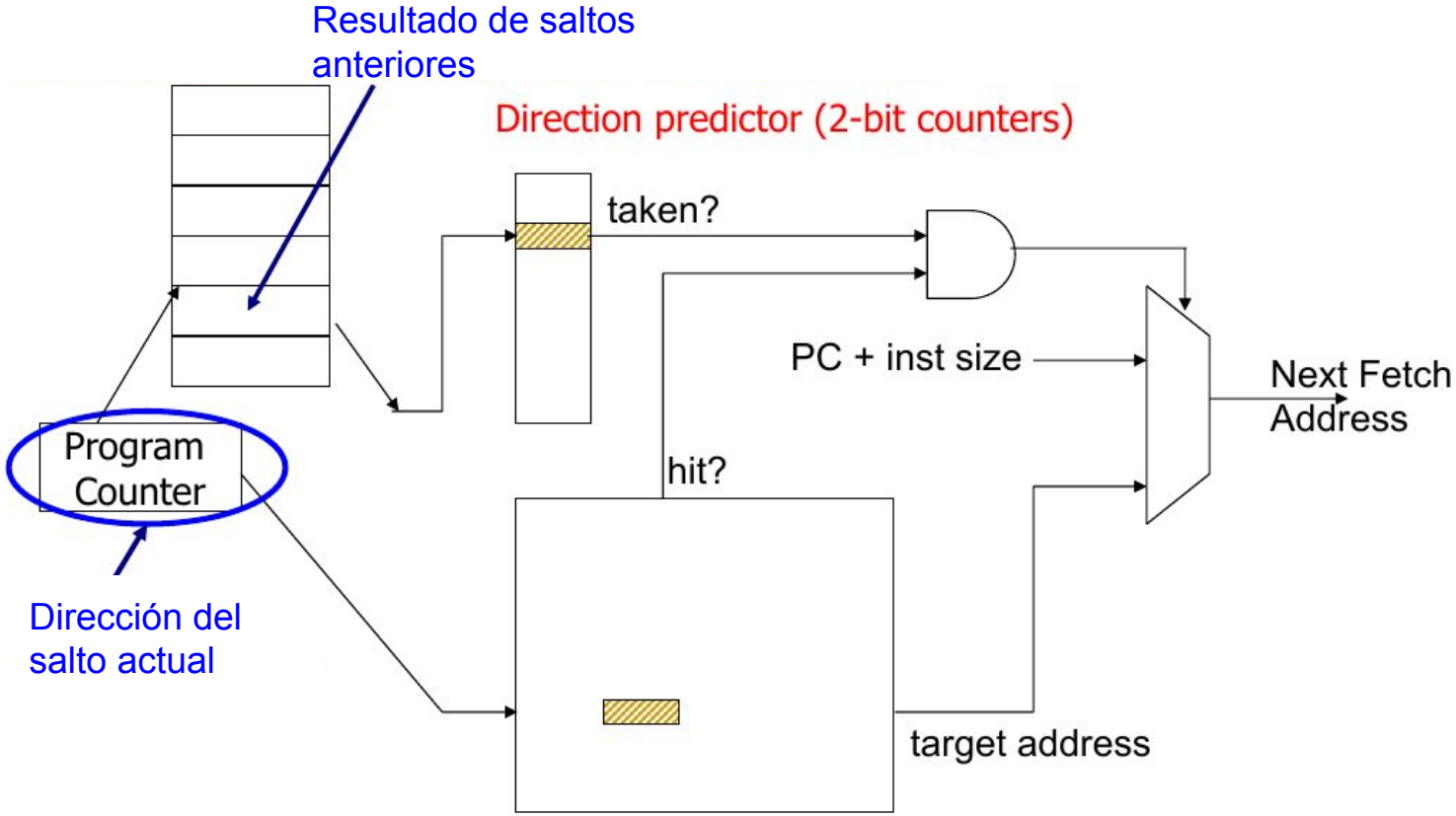
Predictor de dos niveles (local)

- Si se concatena el registro GR con los bits menos significactivos del PC se obtiene un predictor de dos niveles local.





Predictor de dos niveles



Ejercicio 7

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: LE:      add x1, xzr, xzr
0x08: LI:      …
0x0C:          addi x1, x1, 1
0x10:          cmpi x1, 3
0x14:          b.lt LI
0x18:          addi x0, x0, 1
0x1C:          cmpi x0, 100
0x20:          b.lt LE
```

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 del predictor al ejecutar este código con uno de 2-bits (despreciando los primeros ciclos de iniciación).

Ejercicio 3

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

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 con n=4 y m=4.

Evaluación	**Valor**	**GR**	**Resultado**
j < 3	j = 1	1101 (TTNT)	Taken
j < 3	j = 2	1011 (TNTT)	Taken
j < 3	j = 3	0111 (NTTT)	Not Taken
i < 100	i = 10	1110 (TTTN)	Taken

Ejercicio 3

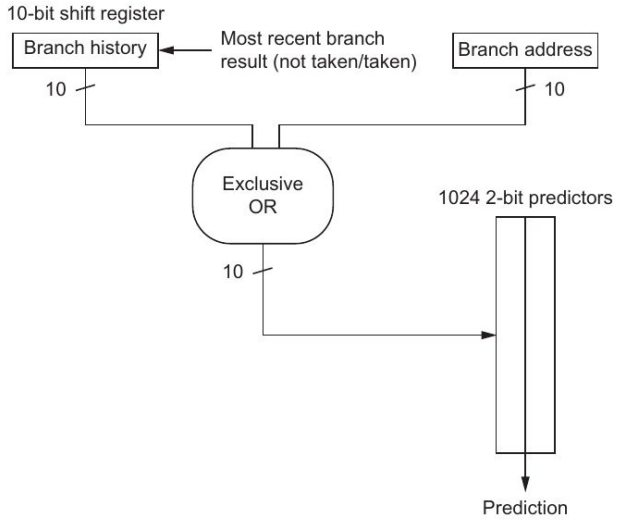
```
for (i = 0; i < 100; i++) {
   for (j = 0; j < 3; j++) {
      …
   }
}
```

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 con n=4 y m=4.

GHR	**PC**	**Resultado**
1101	0100	11
1011	0100	11
0111	0100	00
1110	0000	11

Predictor gshared

- Se utiliza un registro GR y una tabla PHT como en el predictor de dos niveles.
- Se hace una or exclusiva (hash) entre los últimos 10 bits de la dirección de la instrucción de salto y el shift register.



Ejercicio 4

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.

Ejercicio 4

```
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.

Dado que el número obtenido es aleatorio, es imposible predecir localmente los saltos B2, B3 y B4
Sin embargo, el salto B1 si puede ser predecido localmente.

Ejercicio 4

```
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 */
   }
}
```

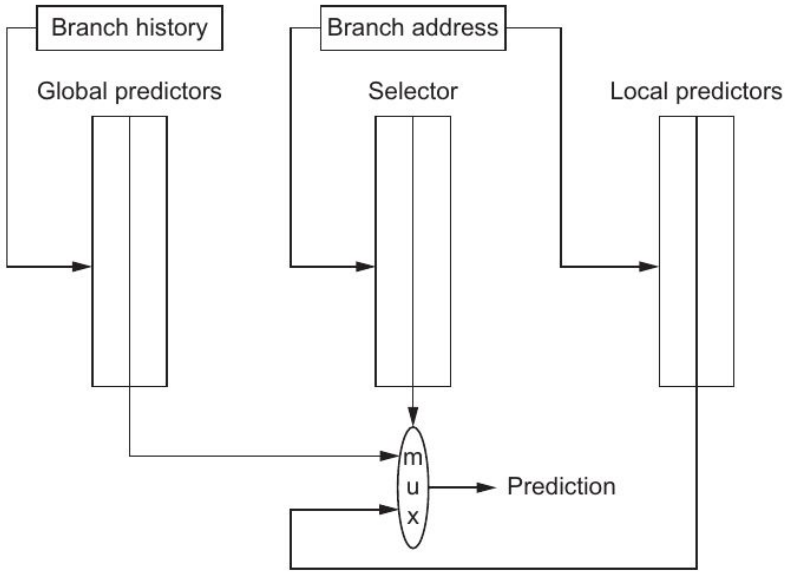
b) ¿Existe correlación global entre algunos de los saltos? Explicar.

B4 está correlacionado con B2 y B3.

Si B2 y B3 son taken, B4 tambien lo va a ser.

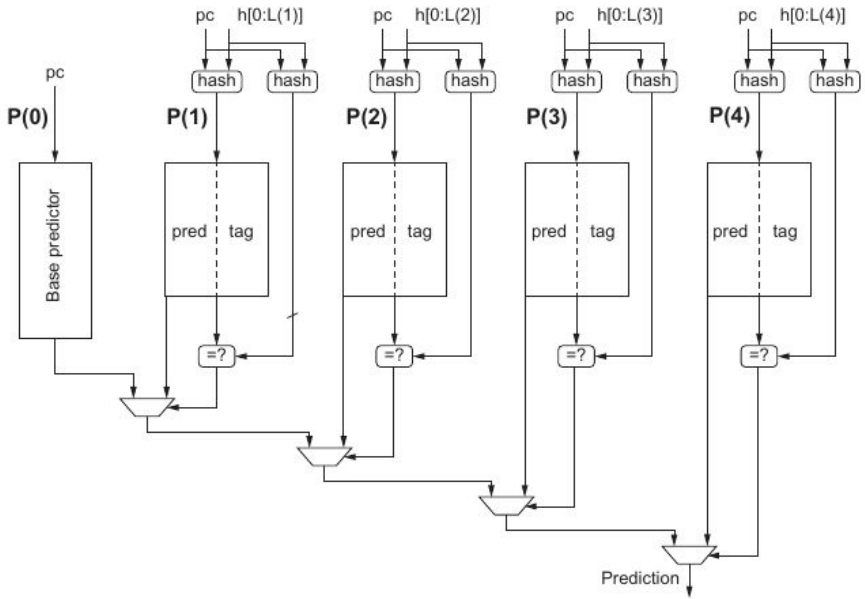
Predictores por torneo

- Se combinan predictores locales (predictor de 2 niveles) y globales
- La ventaja es que elige el tipo de predictor que mejor funciona para cada salto



Tagged Hybrid Predictors

- Basado en un algoritmo de compresión llamado PPM
- Utiliza muchos predictores de 2 niveles con distintos tamaños de historial



Predictores en los procesadores intel i7

Core i7 920 (2008):
- 2 predictores uno rápido (cumple con los límites temporales de ciclo) y uno mayor de backup.
- Cada uno, es un predictor de torneo de 3 predicciones: Predictor local de 2 bits, un predictor global, y un predictor de fin de ciclo.

En los procesadores i7 más nuevos se utiliza un procesador tagged hybrid.

Static Multiple Issue Processor

LEGv8 Static Two-Issue Datapath

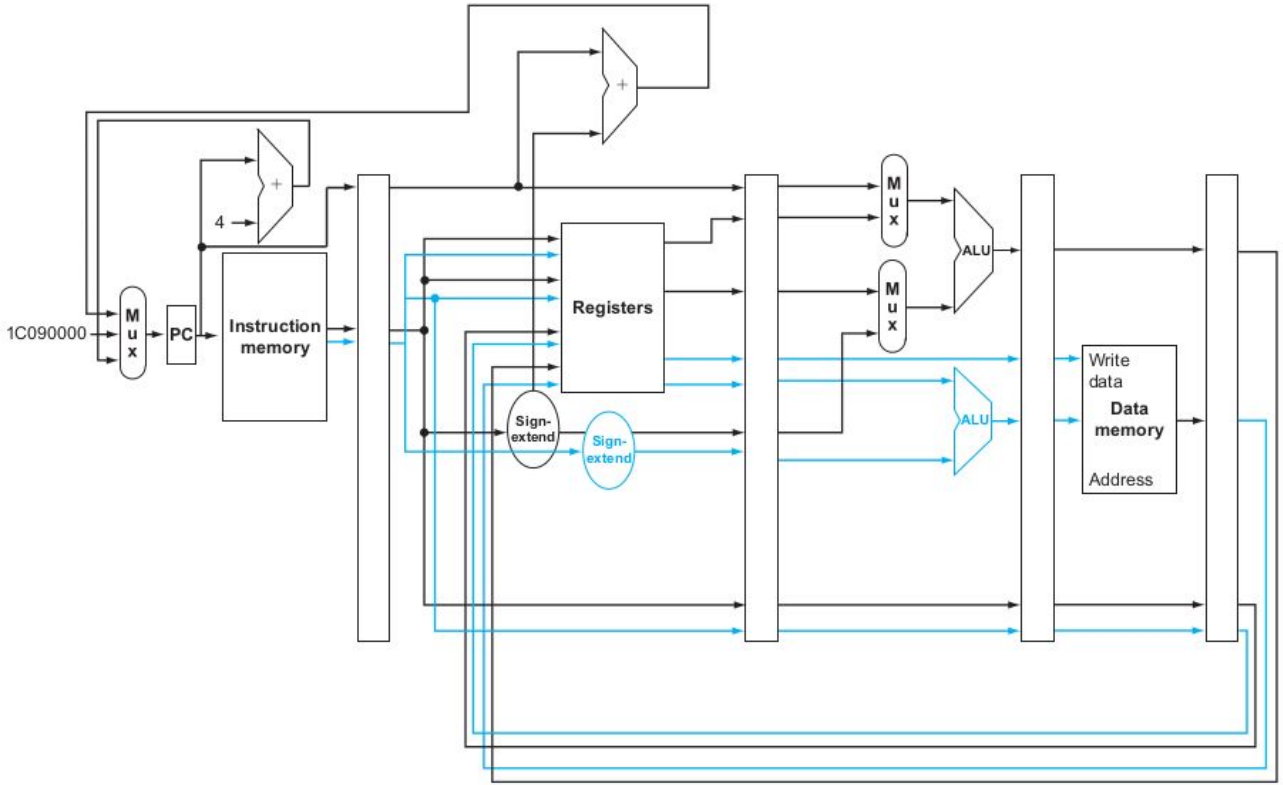


Figura 4.67 - Computer Organization and Design, Arm Edition - Patterson and Hennessy

Static Multiple Issue Processor

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.
- La mayoría de los procesadores relegan la responsabilidad de manejar ciertos data y control hazard al compilador. Ya sea para prevenir los hazards o para reducirlos.

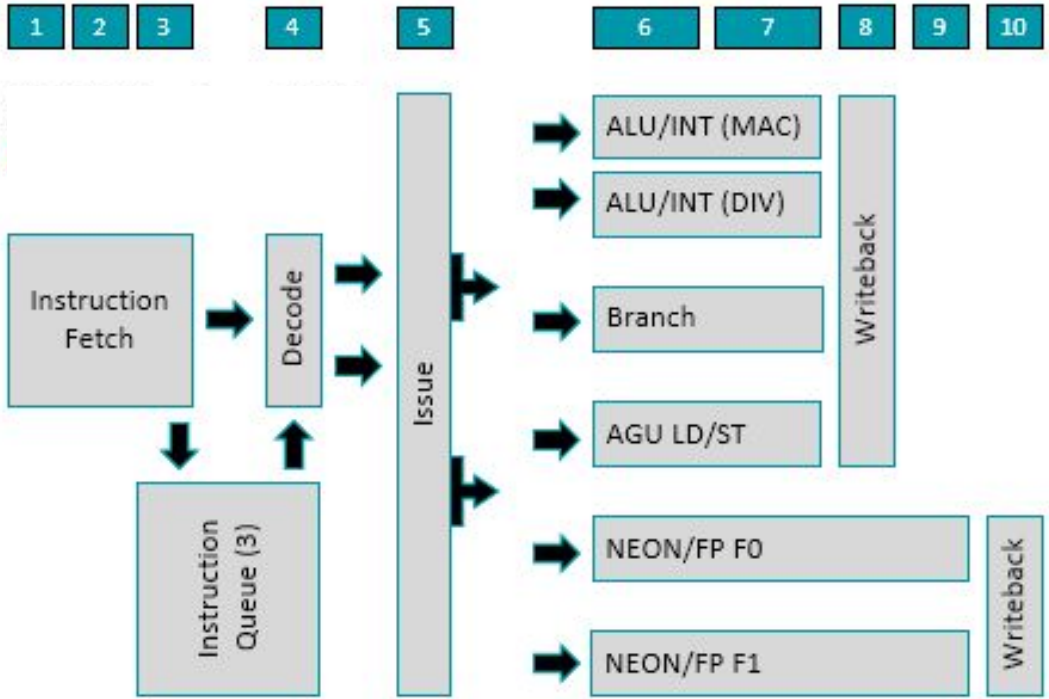
LEGv8 Two-Issue processor

- En cada *Issue Packet* debe haber una instrucción tipo R o branch y una instrucción de acceso a memoria.
- Ejecutar dos instrucciones por ciclo requiere hacer *fetch* y *decode* de instrucciones de 64 bits (el PC se incrementa de a 8).
- Si una de los *issue* no puede utilizarse, se la debe acompañar de un nop.

LEGv8 Static Two-Issue Pipe Stages

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

ARM Cortex-A53 Microarchitecture



https://www.anandtech.com/show/11441/dynamiq-and-arms-new-cpus-cortex-a75-a55/4

Dependencia de datos [1/3]

Las dependencias son un propiedad del programa.
El hecho de que esta dependencia de datos se detecta como *hazard* y si genera o no un *stall,* es propiedad de la organización del *pipeline*.

La dependencia de datos implica:

- La **posibilidad** de un *hazard.*
- El orden en que se debe calcular el resultado.
- Un límite superior en cuanto se puede explotar el paralelismo.

Una dependencia de datos puede superarse:

- Manteniendo la dependencia pero evitando el *hazard*.
- Eliminando la dependencia al modificar el código.

Dependencia de datos [2/3]

Los datos pueden fluir entre instrucciones mediante memoria o registros.

Registros: Relativamente sencillo de detectar, es transparente verlo desde las instrucciones.

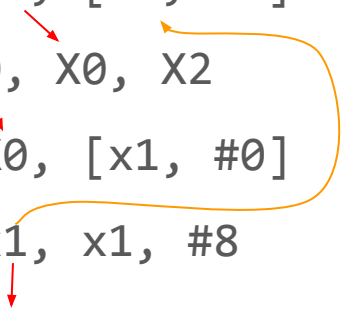
Memoria: Difícil de detectar. Dos instrucciones pueden referirse a la misma posición pero parecer diferente: [x4,100] y [x6,20]. La misma instrucción ejecutada en distintas partes del código puede apuntar a posiciones distintas. Ej: {stur x0, [x4,100]; ldur x1, [x6,20]}

Dependencia real de datos: La instrucción i es dependiente en datos con la instrucción j cuando:

- La instrucción *i* produce un resultado que debe ser utilizado por la instrucción *j*.
- La instrucción *j* es dependiente de datos con la instrucción *k*, y la instrucción *k* es dependiente de datos con la instrucción *i*.

Ejemplo dependencia de datos

```
1>  L:  ldur X0, [X1, #0]     //X0=array element, X1=element addr.
2>      add X0, X0, X2        //add scalar in X2 and save in X0
3>      stur X0, [x1, #0]     //store result
4>      subi x1, x1, #8       //decrement pointer 8 bytes
5>      cbz  x1, L            //branch si x1 es cero
```



Dependencia de datos [3/3]

Dependencia de nombre

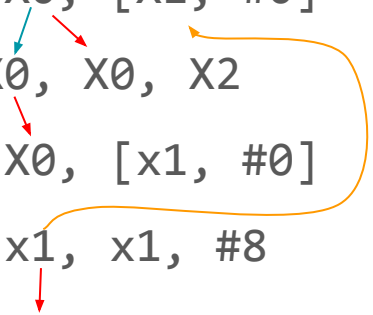
Dos instrucciones usan el mismo registro o posición de memoria, pero no hay flujo real de datos.

- Dependencia de salida: Ocurre cuando las instrucciones *i* y *j* escriben la misma posición de memoria o registro. El orden original se debe preservar para asegurar que el valor final se corresponda con el valor de *j*.

***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 (del registro o de la posición de memoria) se cambia para no generar conflictos.

Ejemplo dependencia de datos

```
1>  L:  ldur X0, [X1, #0]     //X0=array element, X1=element addr.
2>      add X0, X0, X2        //add scalar in X2 and save in X0
3>      stur X0, [x1, #0]     //store result
4>      subi x1, x1, #8       //decrement pointer 8 bytes
5>      cbz  x1, L            //branch si x1 es cero
```



Hazards de datos [1/2]

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 modifique el orden de acceso a los operandos involucrados en la dependencia.

Tipos de *Hazard* de datos:

- **RAW**: j intenta leer un dato antes de que i lo escriba, j obtiene el valor desactualizado (dependencia real de datos).

- **WAW**: j intenta escribir un operando antes de que lo escriba i. Las escrituras se terminan realizando en el orden incorrecto, dejando como valor final el de i, en lugar de j. (Dependencia de salida)

Hazards de datos [2/2]

Ejemplo de *Hazard* de datos:

- **RAW**:

```
add X0, X1, X2          |  add X0, X1, X2
add X3, X4, X0          |  stur X3, [X0, #0]
```

- **WAW**:

```
add X0, X1, X2          |  add X0, X1, X2
add X0, X3, X4          |  ldur X0, [X3, #0]
```

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.

Restricciones impuestas por la dependencia condicional:

- 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 el salto.

- Una instrucción que no tiene una dependencia condicional sobre un salto no puede moverse después del salto de forma en que su ejecución esté controlada por el salto.

```
1>        sub x2,x3,x4
2>        cbnz x2,L1
3>        ldur x1, [X2, #0]
4>   L1:  add x3, x2, x5
```

LEGv8 Static Two-Issue Datapath

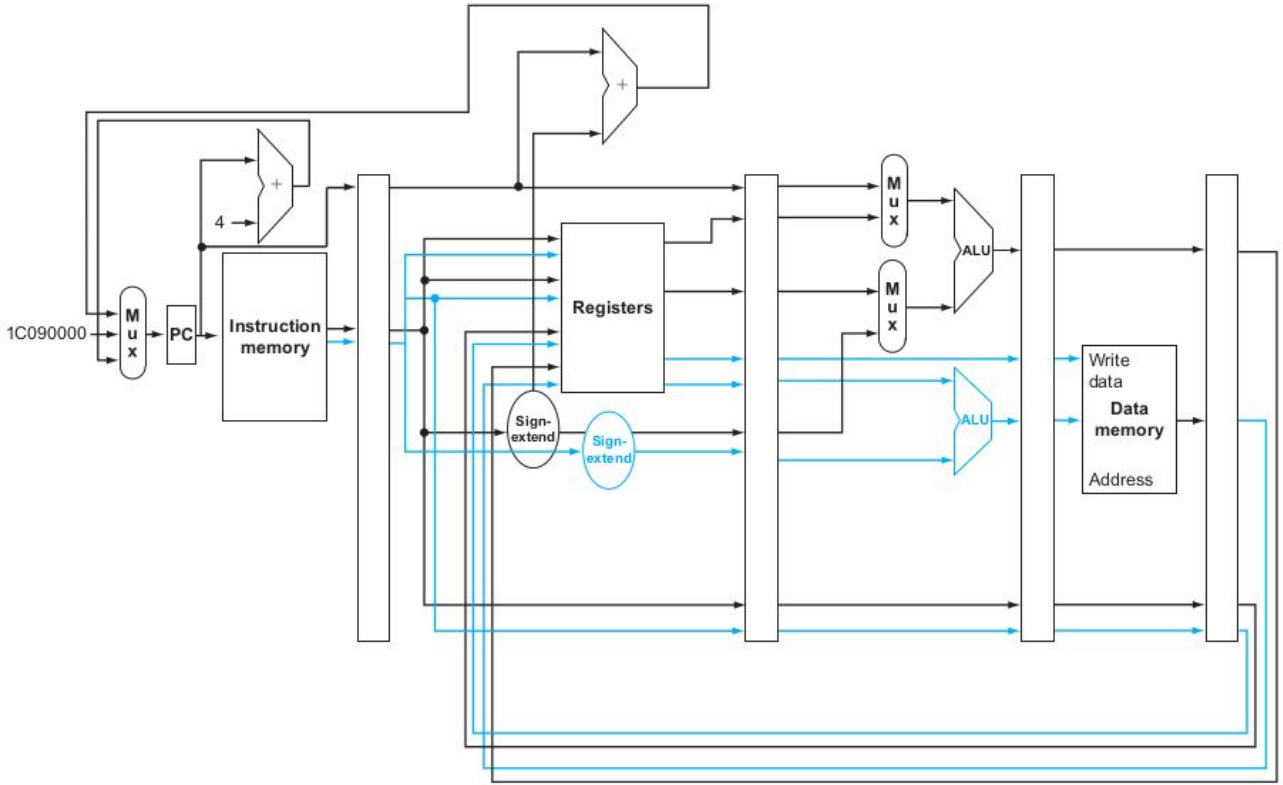


Figura 4.67 - Computer Organization and Design, Arm Edition - Patterson and Hennessy

Ejercicio 6 - Static Multiple Issue Processor

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

Dependencias de datos:

- X0 - 1y2 - RAW
- X0 - 1y2 - WAW (nunca genera hazard en 1-issue)
- X0 - 2y3 - RAW
- X20 - 4y5 - RAW
- X20 - 4y1 - RAW (condicional)
- X20 - 4y3 - RAW (condicional)
- X20 - 4 y 4 en 2 iteraciones distintas - RAW (condicional)

Ejercicio 6-b

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

	ALU or branch instruction	Data transfer instruction	clk
Loop:			1
			2
			3
			4
			5
			6

Ejercicio 6-b en el libro

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		LDUR X0,[X20,#0]	1
	SUBI X20, X20, #8		2
	ADD X0, X0, X21		3
	CMP X20, X22		4
	BGT Loop	STUR X0,[X20,#8]	5

FIGURE 4.68 The scheduled code as it would look on a two-issue LEGv8 pipeline. The empty slots are no-ops.

Ejercicio 6-c Loop unrolling

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

1b>          LDUR X0, [X20,#0]   // X0=array element
2b>          ADD X0,X0,X21       // add scalar in X21
3b>          STUR X0, [X20,#0]   // store result
4b>          SUBI X20,X20,#8     // decrement pointer

5>           CMP X20,X22         // compare to loop limit
6>           B.GT Loop           // branch if X20 > X22
```

Ejercicio 6-c Register renaming

```
1> Loop:     LDUR X1, [X20,#0]   // X0=array element
2>           ADD X1,X1,X21       // add scalar in X21
3>           STUR X1, [X20,#0]   // store result
4>           SUBI X20,X20,#8     // decrement pointer

1b>          LDUR X0, [X20,#0]   // X0=array element
2b>          ADD X0,X0,X21       // add scalar in X21
3b>          STUR X0, [X20,#0]   // store result
4b>          SUBI X20,X20,#8     // decrement pointer

5>           CMP X20,X22         // compare to loop limit
6>           B.GT Loop           // branch if X20 > X22
```

Ejercicio 6-c Eliminar instrucciones innecesarias

```
1> Loop:    LDUR X1, [X20,#0]   // X0=array element
2>          ADD X1,X1,X21       // add scalar in X21
3>          STUR X1, [X20,#0]   // store result
4>          SUBI X20,X20,#8     // decrement pointer

1b>         LDUR X0, [X20,#-8]  // X0=array element
2b>         ADD X0,X0,X21       // add scalar in X21
3b>         STUR X0, [X20,#-8]  // store result
4b>         SUBI X20,X20,#16    // decrement pointer

5>          CMP X20,X22         // compare to loop limit
6>          B.GT Loop           // branch if X20 > X22
```

Ejercicio 6-c Eliminar instrucciones innecesarias

```
1> Loop:     LDUR X1, [X20,#0]   // X0=array element
2>           ADD X1,X1,X21       // add scalar in X21
3>           STUR X1, [X20,#0]   // store result
4>           LDUR X0, [X20,#-8]  // X0=array element
5>           ADD X0,X0,X21       // add scalar in X21
6>           STUR X0, [X20,#-8]  // store result
7>           SUBI X20,X20,#16    // decrement pointer
8>           CMP X20,X22         // compare to loop limit
9>           B.GT Loop           // branch if X20 > X22
```

Ejercicio 6-c

```
1> Loop:    LDUR X1, [X20,#0]   // X0=array element
2>          ADD X1,X1,X21       // add scalar in X21
3>          STUR X1, [X20,#0]   // store result
4>          LDUR X0, [X20,#-8]  // X0=array element
5>          ADD X0,X0,X21       // add scalar in X21
6>          STUR X0, [X20,#-8]  // store result
7>          SUBI X20,X20,#16    // decrement pointer
8>          CMP X20,X22         // compare to loop limit
9>          B.GT Loop           // branch if X20 > X22
```

	ALU or branch instruction	Data transfer instruction	clk
Loop:			1
			2
			3
			4
			5
			6
			7

Ejercicio 6-c

	ALU or branch instruction	Data transfer instruction
Loop:	nop	LDUR X1, [X20,#0]
	nop	LDUR X0, [X20,#-8]
	ADD X1,X1,X21	nop
	ADD X0,X0,X21	STUR X1, [X20,#0]
	SUBI X20,X20,#16	STUR X0, [X20,#-8]
	CMP X20,X22	nop
	B.GT Loop	nop

	ALU or branch instruction	Data transfer instruction	clk
Loop:	SUBI X20,X20,#16	LDUR X1, [X20,#0]	1
	CMP X20,X22	LDUR X0, [X20,#8]	2
	ADD X1,X1,X21	nop	3
	ADD X0,X0,X21	STUR X1, [X20,#16]	4
	B.GT Loop	STUR X0, [X20,#8]	5