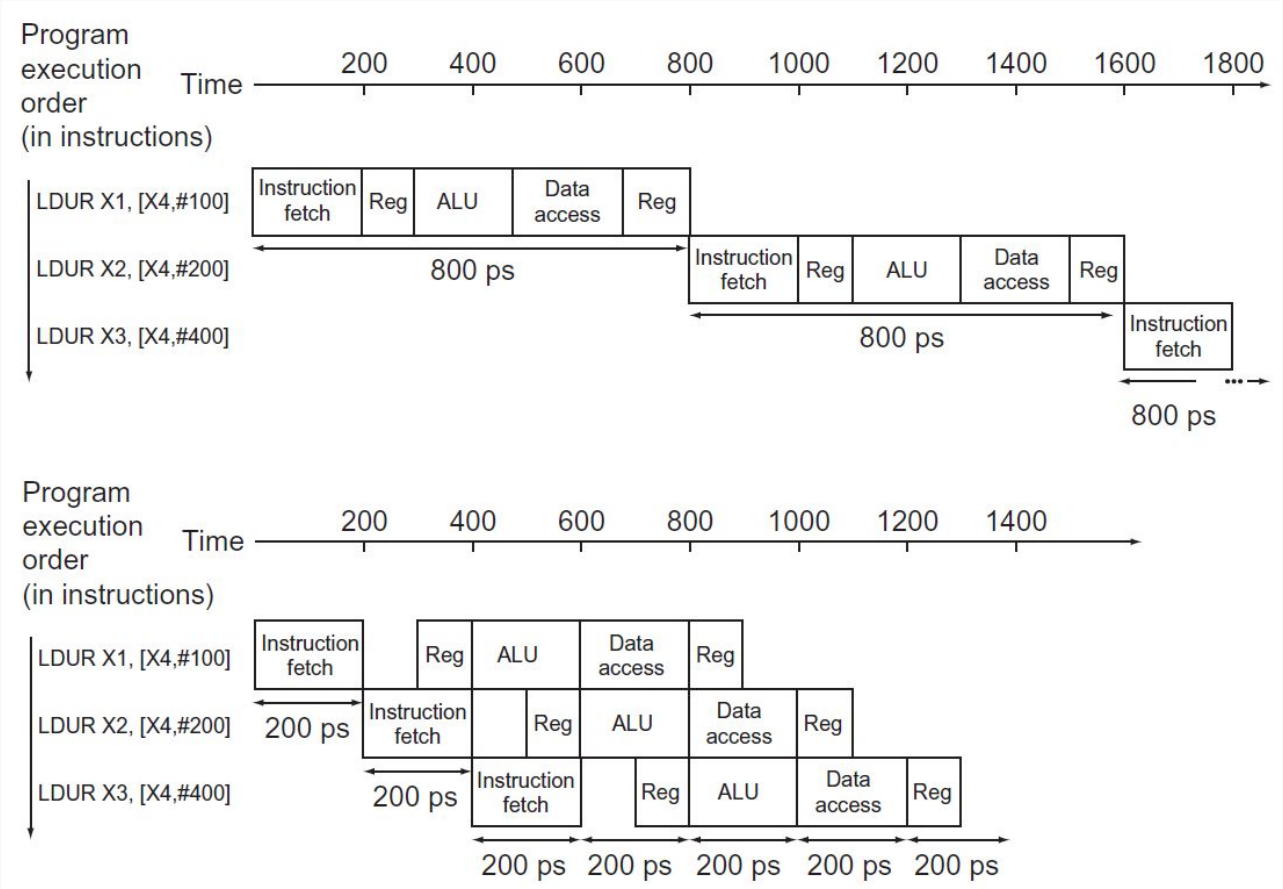
Práctico N° 5: Técnicas de mejora de rendimiento

Arquitectura de Computadoras II - 2021

Dr. Ing. Agustín Laprovitta (alaprovitta@ucc.edu.ar)
Ing. Delfina Velez Ibarra (0703883@ucc.edu.ar)

Single-cycle, nonpipelined execution (top) versus pipelined execution (bottom)



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

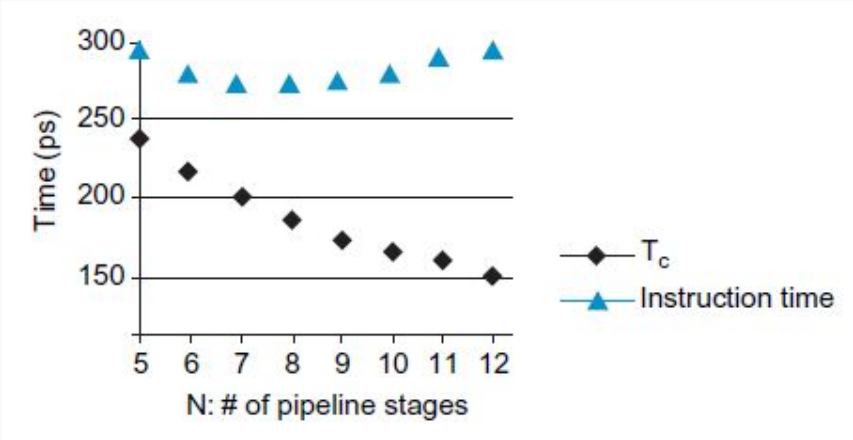
- Latencia de una etapa = Tiempo de ciclo (Tc) = (740/N+ 90) ps
- CPI = 1.23 + 0.1(N−5)
- Tiempo por instrucción (Ti) = Tc * CPI

N	Tc [ps]	CPI	Ti [ps]
5	238	1.23	292.74
6			
7			
8	182.5	1.53	279.225
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



Ejercicio 2 - Branch Prediction

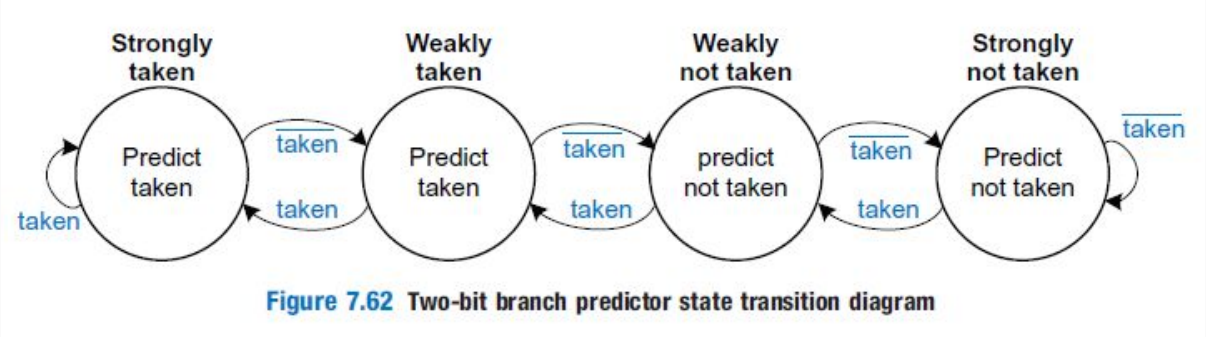


Figure 7.62 Two-bit branch predictor state transition diagram

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?

Ejercicio 2-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 2-b

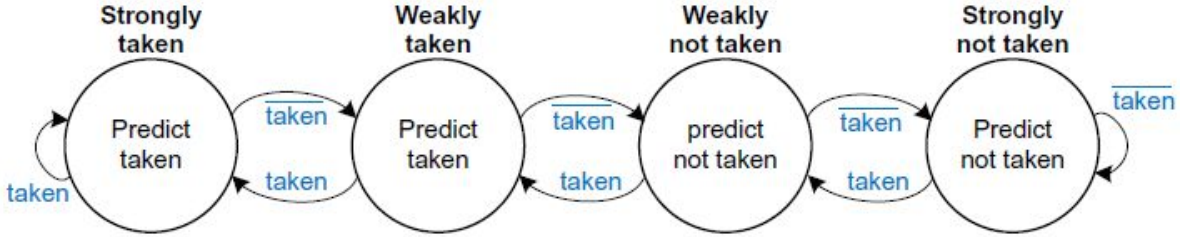


Figure 7.62 Two-bit branch predictor state transition diagram

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

Precisión del predictor = ¼ = 25%

Ejercicio 2-c

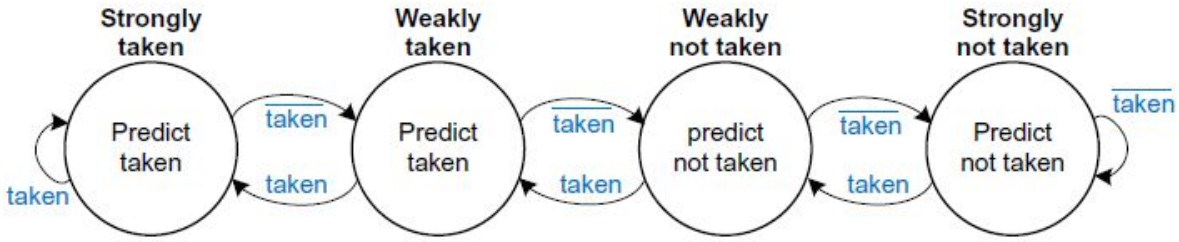


Figure 7.62 Two-bit branch predictor state transition diagram

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%

LEGv8 Static Two-Issue Datapath

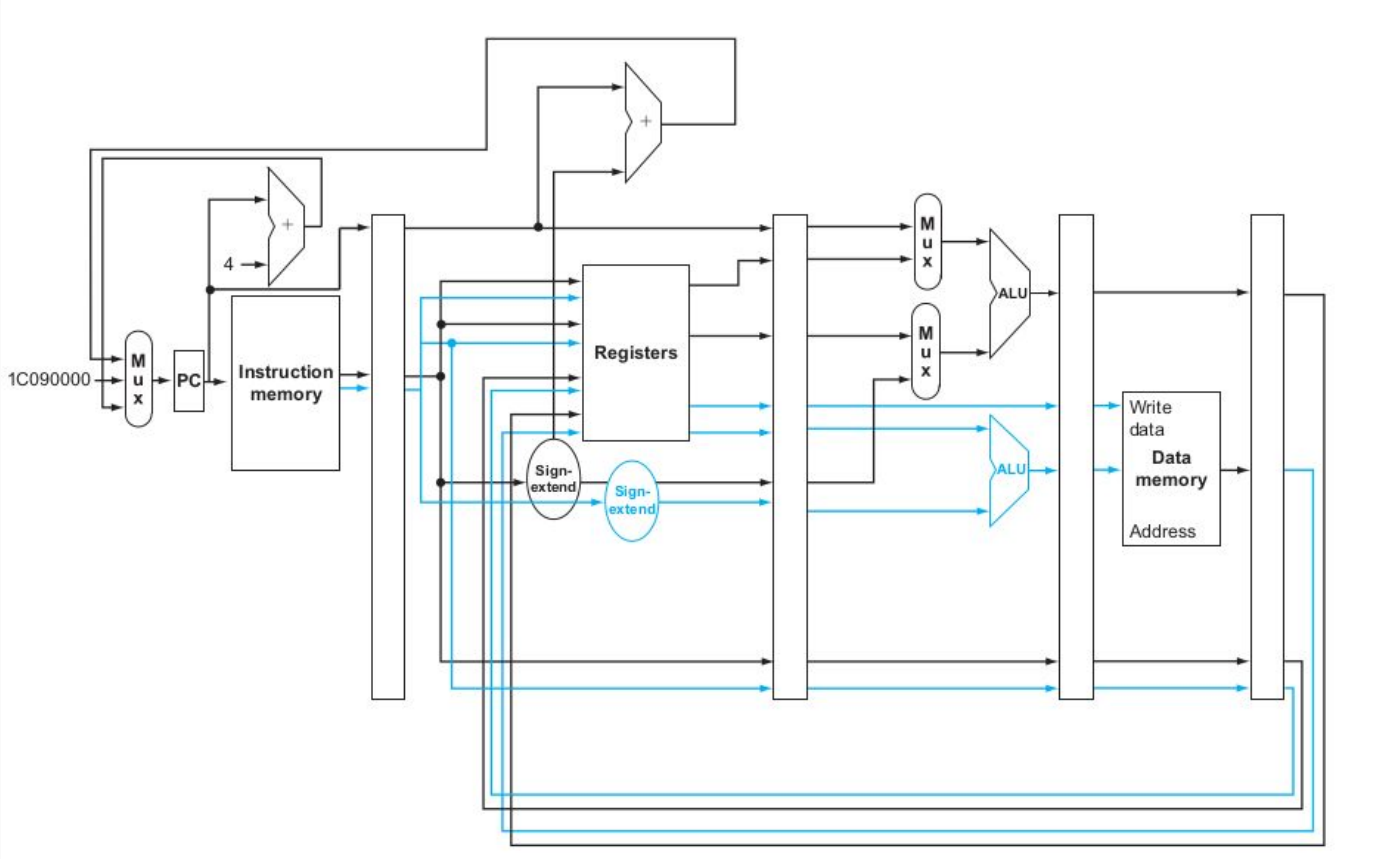


Figura 4.67 - Computer Organization and Design, Arm Edition - Patterson and Hennessy

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 hazard de datos y control 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

Dependencia de datos [1/3]

Las dependencias son un propiedad del programa.
El hecho de que esta dependencia de datos se detecte 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 cuánto 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.

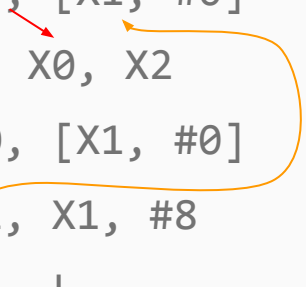
Registros: Relativamente sencillo de detectar, es transparente verlo desde las instrucciones.

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

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*.

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

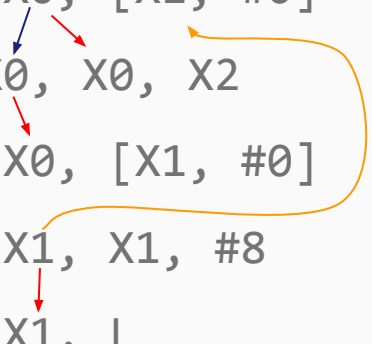
Dos instrucciones usan el mismo registro o posición de memoria, pero no hay flujo real de datos.

- Dependencia de salida: Ocurre cuando las instrucciones *i* y *j* escriben la misma posición de memoria o registro. Si la instrucción *i* precede a la instrucción *j* en el orden del programa, se debe preservar el orden original 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 X3, X4, X0
```

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

- **WAW**:

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

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

Dependencia condicional (de control)

Una dependencia condicional determina el ordenamiento de una instrucción cualquiera respecto a una de salto. De forma en que dicha instrucción se ejecuta en el correcto orden de programa y sólo cuando debe ser.

Restricciones impuestas por la dependencia condicional

Dado el siguiente código de ejemplo:

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

- Una instrucción que tiene una dependencia condicional sobre un salto no puede moverse antes que el salto (o agruparse al mismo) 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.

Ejercicio 3 - 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.

Ejercicio 3 - 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 3-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:	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
			5
			6

Ejercicio 3-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 3-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 3-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]       // X1=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 3-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]      // X1=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 3-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]      // X1=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 3-c

```
1> Loop:    LDUR X1, [   ,#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]   // X1=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:		LDUR X1, [X20,#0]	1
		LDUR X0, [X20,#-8]	2
	ADD X1,X1,X21		3
	ADD X0,X0,X21	STUR X1, [X20,#0]	4
	SUBI X20,X20,#16	STUR X0, [X20,#-8]	5
	CMP X20,X22		6
	B.GT Loop		7

Ejercicio 3-c

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

	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