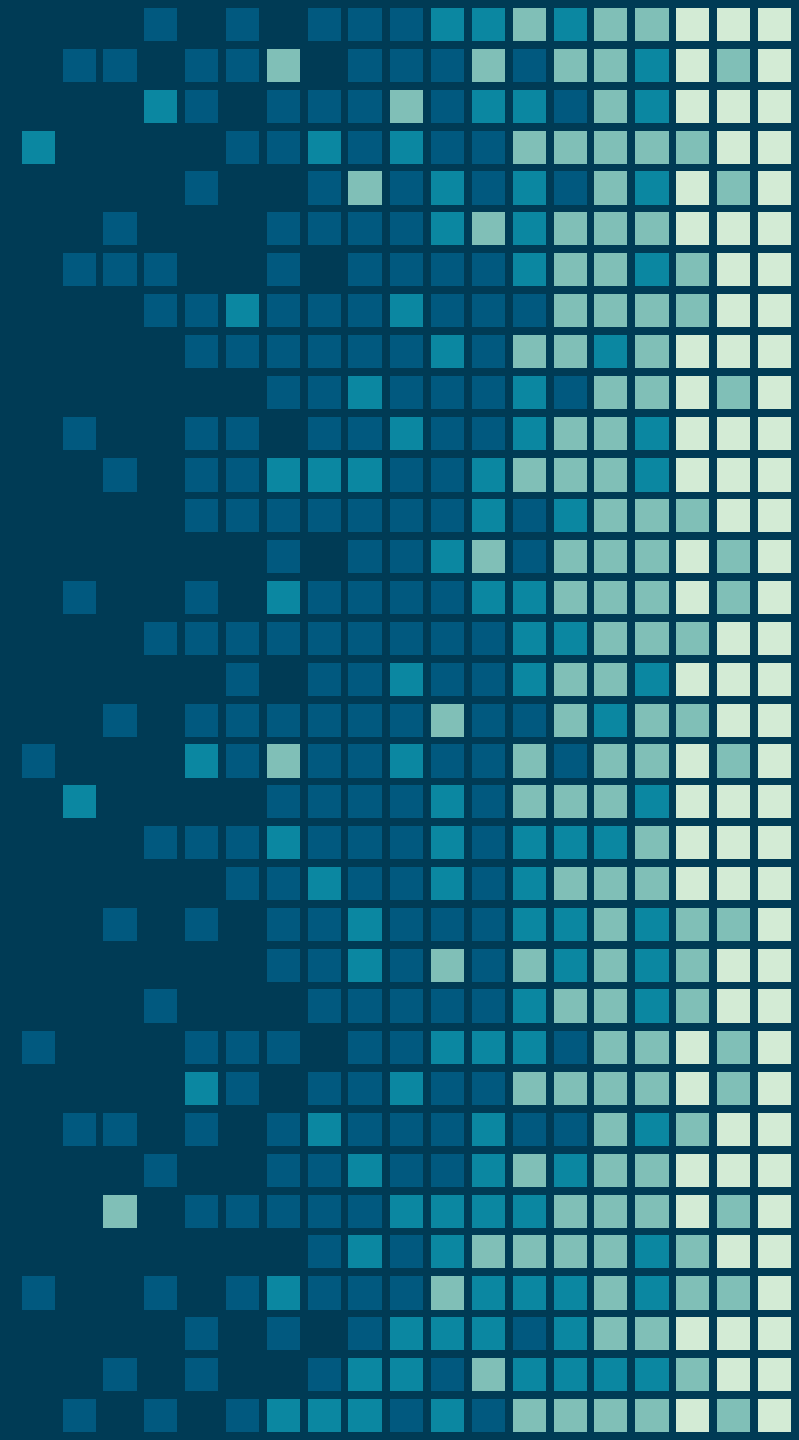
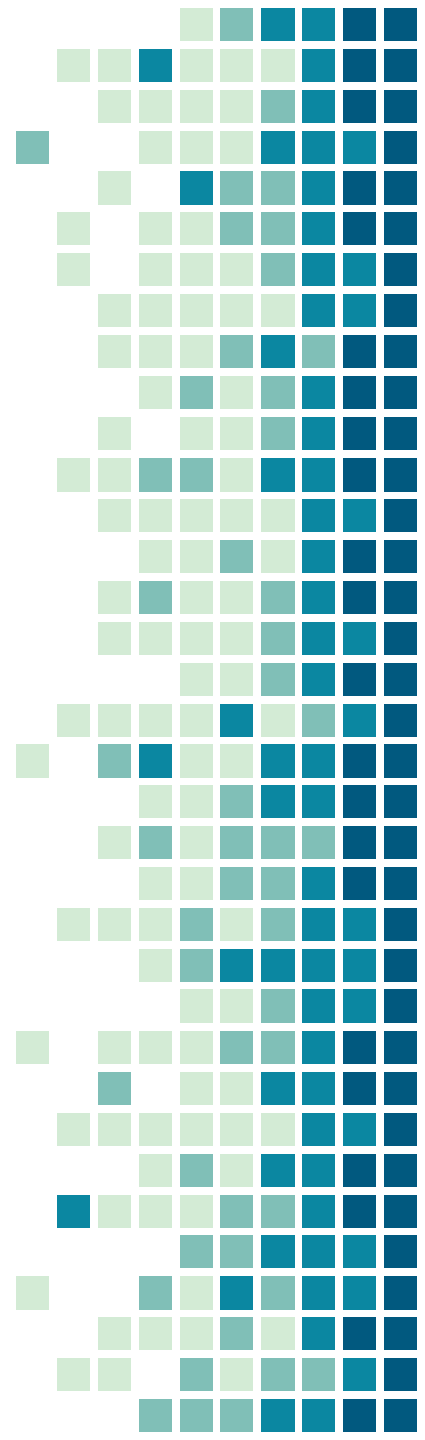


Introducción a VLIW II



Agenda

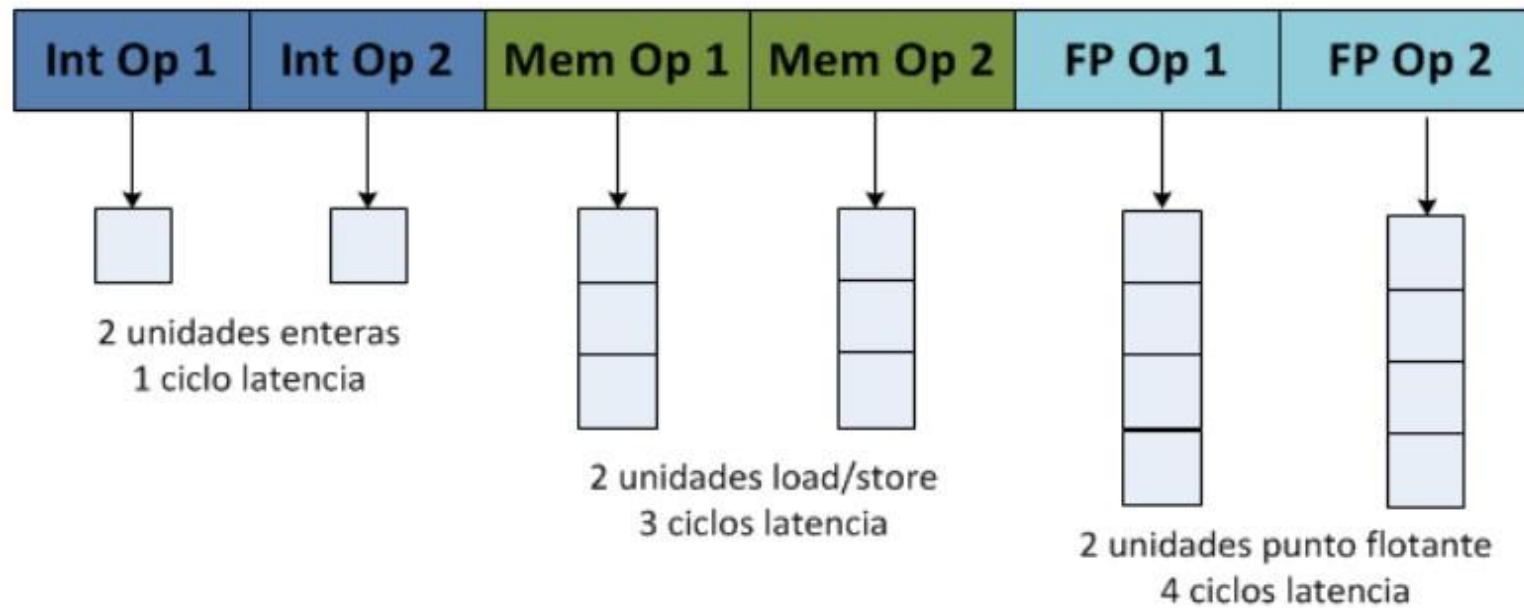
- **Repaso VLIW.**
- **Loop unrolling.**
- **Ejecución condicional**
- **Especulación**



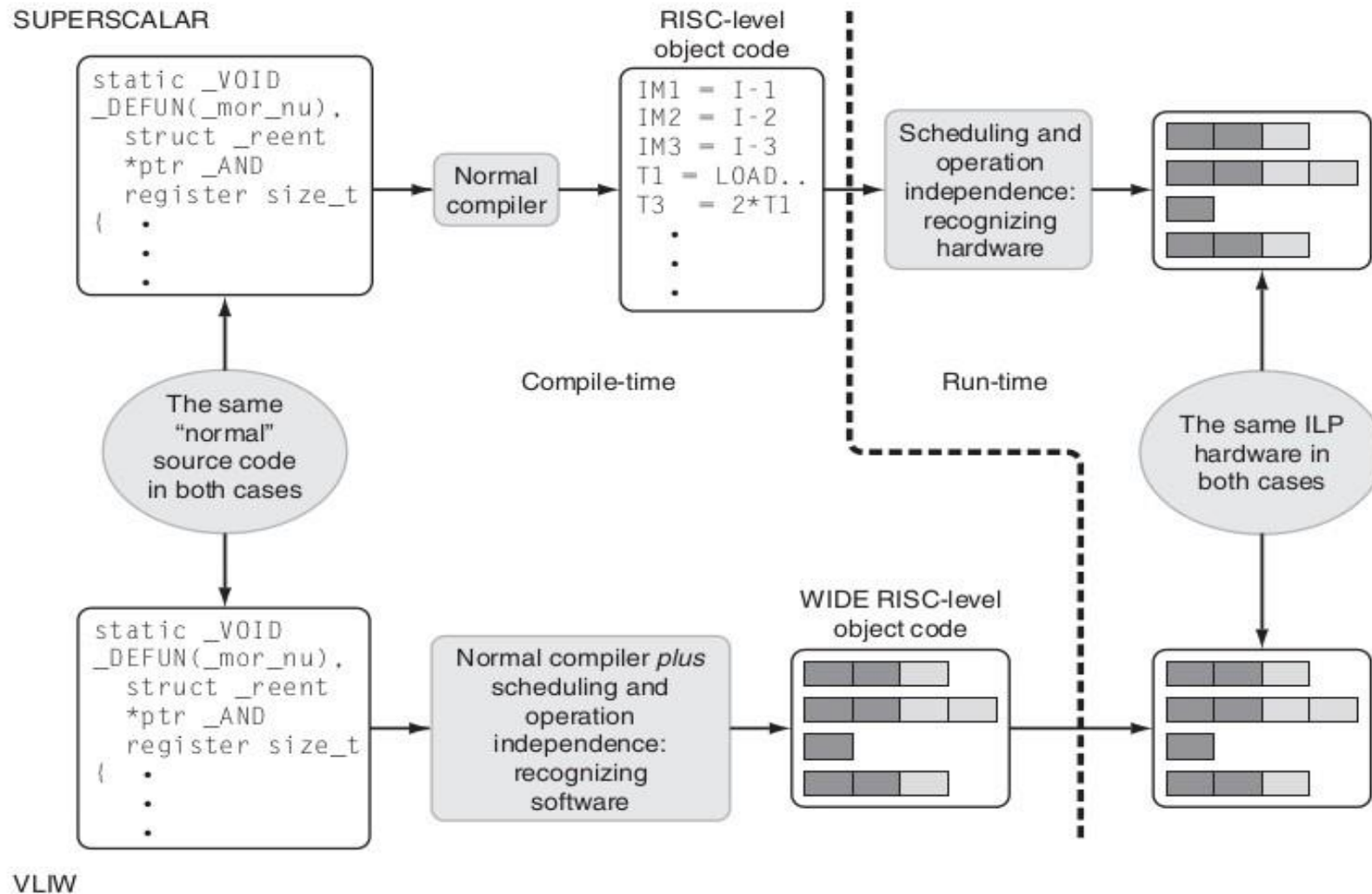
Repaso VLIW



- ❑ Múltiples operaciones en una sola instrucción. Unidades funcionales replicadas.
- ❑ La arquitectura que implemente VLIW requiere:
 - Paralelismo dentro de una misma instrucción (segmentación paralela + diversificada)
 - Los datos no son utilizados antes de que estén listos



Superscalar vs. VLIW * [Fisher, J. A]

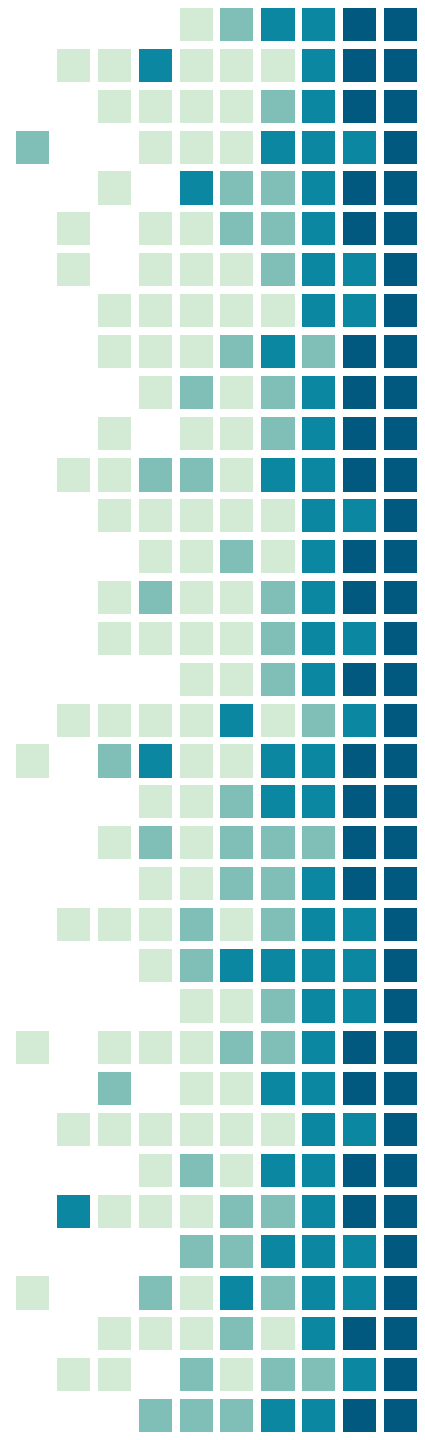


Loop unrolling



Loop unrolling

- **Técnica para incrementar el paralelismo, desde el compilador:**
 - ✓ El "cuerpo" del loop es replicado múltiples veces (factor de unrolling) modificando el código dentro del loop, aprovechando los recursos de Hw.
 - ✓ El compilador debe verificar que las iteraciones del loop sean independientes.



Loop unrolling



- Extiende longitud de bloque básico (se replica el código dentro del loop). Reduce stalls y aumenta el paralelismo.
- Incrementa el número de registros
- Incrementa el tamaño del código



Ejemplo alto nivel

```
for
(i=1;i<=100;i++)
{
A[i]=B[i]+C[i]
}
```

Con un **loop unrolling de 4**, se puede convertir en:

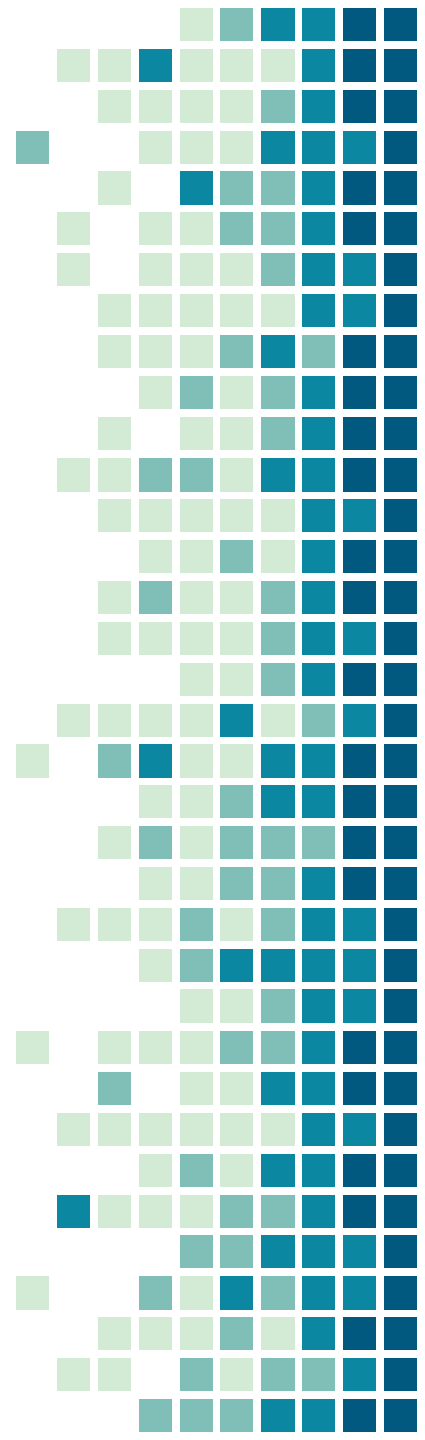
```
for (i=1;i<=97;i=i+4)
{
A[i]=B[i]+C[i]
A[i+1]=B[i+1]+C[i+1]
A[i+2]=B[i+2]+C[i+2]
A[i+3]=B[i+3]+C[i+3]
}
```

Loop unrolling

Iteraciones del loop deben ser independientes una de la otra.

```
—  
LOOP: LD      F0, 0(R1)  
      ADD     F4, F0, F2  
      SD      F4, 0(R1)  
      SUBI    R1, R1, #8  
      BNE     R1, R2, LOOP
```

Pueden existir dependencias dentro del loop, que deben considerarse



Loop unrolling

Se considera:

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

Dependencias de control y
datos

Salto resuelto en etapa
de decodificación

Latencia en operaciones:
(ej.)

Loop unrolling

```
—  
LOOP:  LD      F0, 0(R1)  
       ADD     F4, F0, F2  
       SD      F4, 0(R1)  
       SUBI    R1, R1, #8  
       BNE     R1, R2, LOOP
```

El código anterior se convierte en:

```
—  
LOOP:  LD      F0, 0(R1)  
       NOP  
       ADD     F4, F0, F2  
       NOP  
       NOP  
       SD      F4, 0(R1)  
       SUBI    R1, R1, #8  
       NOP  
       BNE     R1, R2, LOOP  
       NOP
```

Sin loop unrolling = 10 ciclos por iteración

Loop unrolling

Con un loop unrolling de 4:

```
LOOP: LD    F0,0(R1)
      ADD   F4,F0,F2
      SD    F4,0(R1)
      LD    F0,-8(R1)
      ADD   F4,F0,F2
      SD    F4,-8(R1)
      LD    F0,-16(R1)
      ADD   F4,F0,F2
      SD    F4,-16(R1)
      LD    F0,-24(R1)
      ADD   F4,F0,F2
      SD    F4,-24(R1)
      SUBI  R1,R1,#32
      BNE   R1,R2,LOOP
```

Existen dependencias de datos y nombre. Las de nombre son solucionadas por **renombramiento** desde el compilador.

Loop unrolling

Solo permanecen las dependencias **reales** (dentro de cada cuerpo). La calendarización se encarga de determinar el orden correcto de forma que se minimicen los NOPS.

Luego del renombramiento:

```
LOOP: LD      F0,0(R1)
      ADD     F4,F0,F2
      SD      F4,0(R1)
      LD      F6,-8(R1)
      ADD     F8,F6,F2
      SD      F8,-8(R1)
      LD      F10,-16(R1)
      ADD     F12,F10,F2
      SD      F12,-16(R1)
      LD      F14,-24(R1)
      ADD     F16,F14,F2
      SD      F16,-24(R1)
      SUBI    R1,R1,#32
      BNE     R1,R2,LOOP
```

Loop unrolling

Bajo la calendarización (orden):

```
LOOP: LD      F0,0(R1)
      LD      F6,-8(R1)
      LD      F10,-16(R1)
      LD      F14,-24(R1)
      ADDD    F4,F0,F2
      ADDD    F8,F0,F2
      ADDD    F12,F0,F2
      ADDD    F16,F0,F2
      SD      F4,0(R1)
      SD      F8,-8(R1)
      SUBI    R1,R1,#32
      SD      F12,16(R1)
      BNE     R1,R2,LOOP
      SD      F16,8(R1)
```

14 ciclos para **4** iteraciones.

Pasa de 10 (sin loop unrolling) a 3.5 ciclos por iteración



Loop unrolling en VLIW

Si se calendariza en arquitectura VLIW-5, para $Lu=7$:

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer op/ Branch
LD F0,0(R1)	LD F6,-8(R1)	NOP	NOP	NOP
LD F10,-16(R1)	LD F14,-24(R1)	NOP	NOP	NOP
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F23	NOP
LD F26,-48(R1)	NOP	ADDD F12,F10,F2	ADDD F16,F14,F2	NOP
NOP	NOP	ADDD F20,F18,F2	ADDD F24,F22,F2	NOP
SD F4,0(R1)	SD F8,-8(R1)	ADDD F28,F26,F2	NOP	NOP
SD F12,-16(R1)	SD F16,-24(R1)	NOP	NOP	NOP
SD F20,-32(R1)	SD F24,-40(R1)	NOP	NOP	SUBI R1,R1,#5
SD F28,8(R1)	NOP	NOP	NOP	BNEZ R1,LOOP

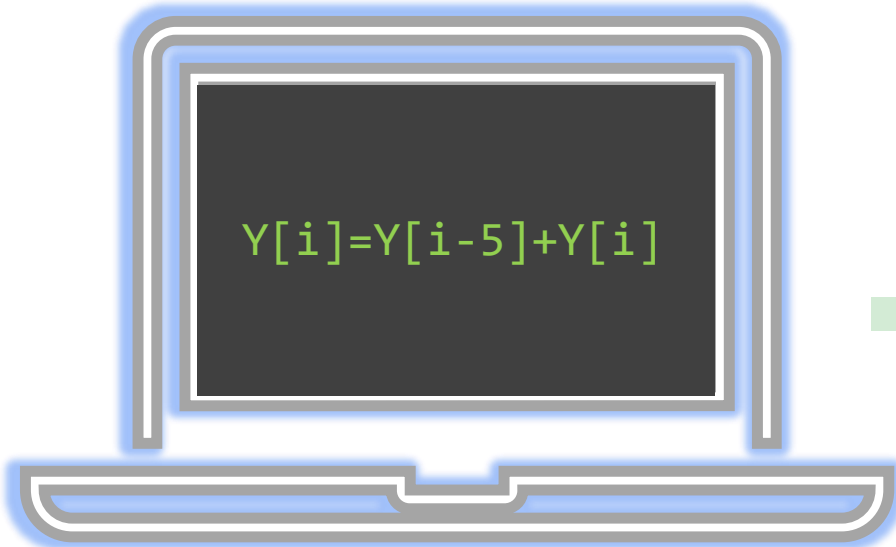
9 ciclos por 7 iteraciones (loop unrolling =7)

1.28 ciclos por iteración.

Loop-carried dependencies

Es común que iteraciones de ciclos dependan de iteraciones anteriores.

- **Loop-carried dependence:** Acceso a datos en iteraciones actuales dependen de valores producidos en iteraciones anteriores.

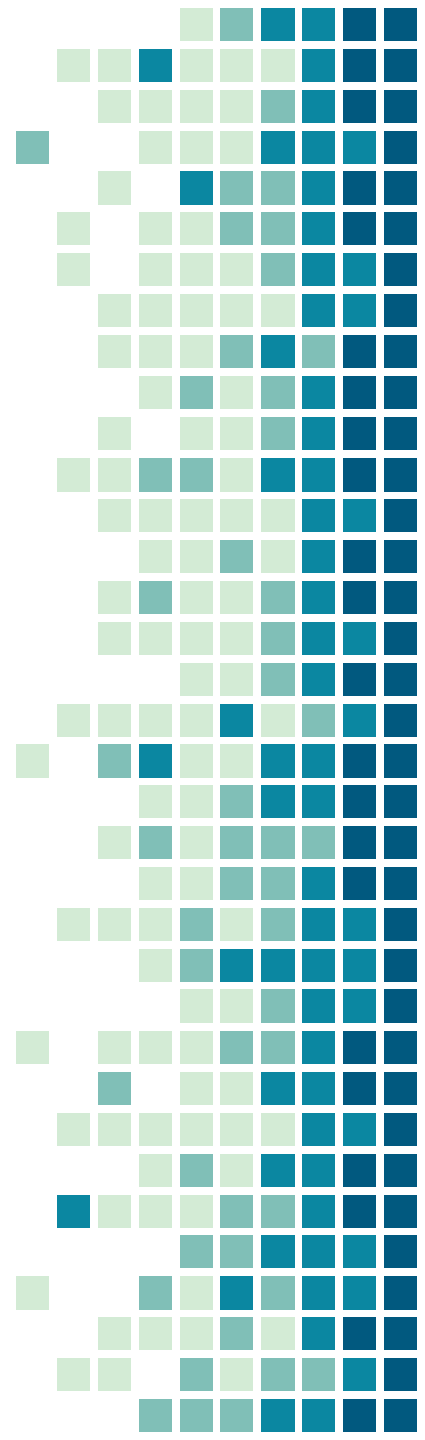


La iteración actual depende de una iteración anterior ($k-5$). Existe un límite máximo para unrolling.

Loop-carried dependencies

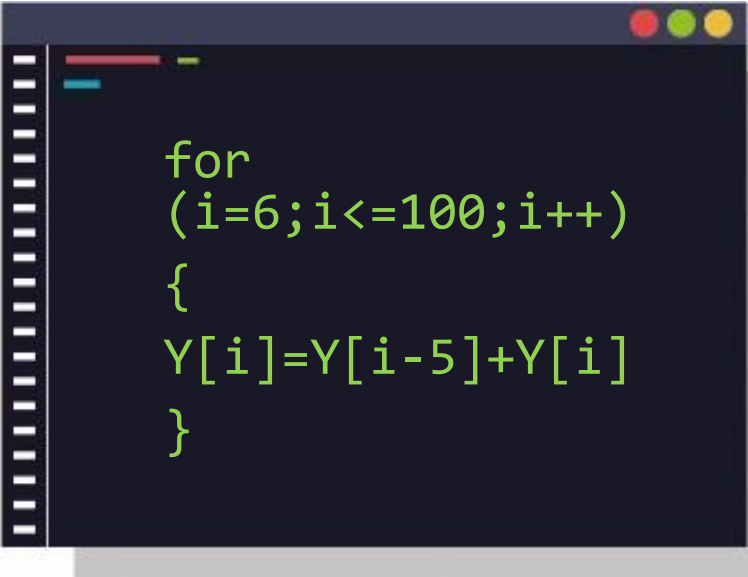
```
for (i=1;i<=97;i=i+4)
{
A[i]=B[i]+C[i]
A[i+1]=B[i+1]+C[i+1]
A[i+2]=B[i+2]+C[i+2]
A[i+3]=B[i+3]+C[i+3]
}
```

En el código anterior, no hay dependencias entre iteraciones, por lo que el factor de unrolling (solamente considerando código) puede ser teóricamente hasta **100**.



Loop-carried dependencies

Cada iteración depende de $i-5$.
Iteraciones i , $i+1$, $i+2$, $i+3$, $i+4$
son independientes ($i+5$
depende de i , por lo que no se puede
incluir dentro de la
misma iteración). **Máximo factor de
unrolling=5**



```
for
(i=6;i<=100;i++)
{
Y[i]=Y[i-5]+Y[i]
}
```

Loop-carried dependencies

```
for
(i=6;i<=100;i++)
{
Y[i]=Y[i-5]+Y[i]
}
```

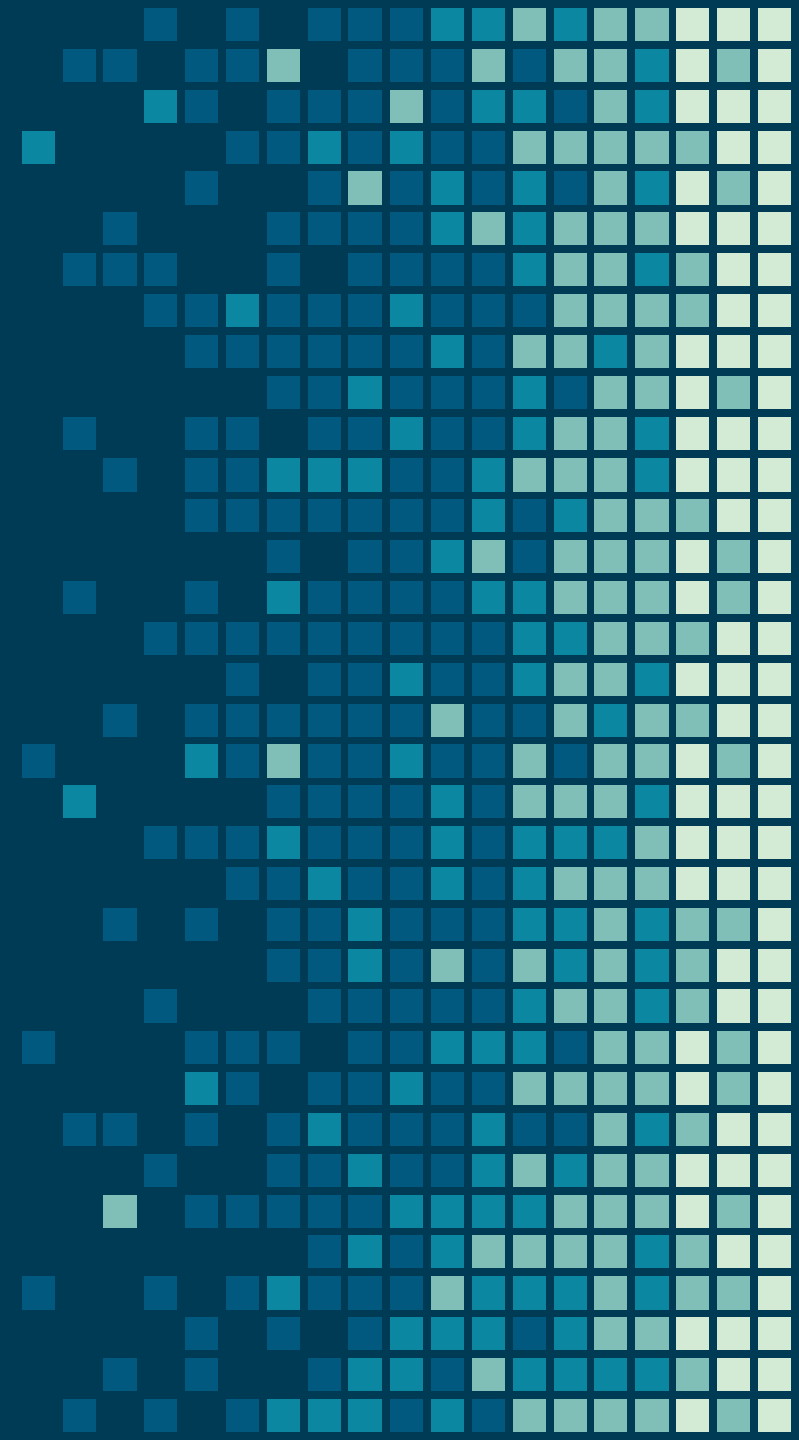
Cada iteración depende de $i-5$.

Iteraciones i , $i+1$, $i+2$, $i+3$, $i+4$ son independientes ($i+5$ depende de i , por lo que no se puede incluir dentro de la misma iteración).

Máximo factor de unrolling=5

```
for (i=6;i<=96;i=i+5) {
Y[i]=Y[i-5]+Y[i]
Y[i+1]=Y[i-4]+Y[i+1]
Y[i+2]=Y[i-3]+Y[i+2]
Y[i+3]=Y[i-2]+Y[i+3]
Y[i+4]=Y[i-1]+Y[i+4]
}
```

Ejecución condicional



Dependencias de control

Las dependencias de control pueden limitar significativamente el nivel de paralelismo que se puede explotar. Posibles soluciones

- Extender el set de instrucciones para incluir instrucciones
- condicionales (ejecución condicionales) o con predicado
- Utilizar especulación de saltos a nivel del compilador (con el soporte de HW requerido).

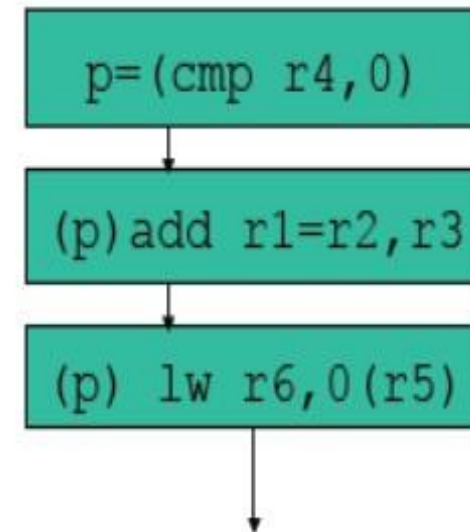
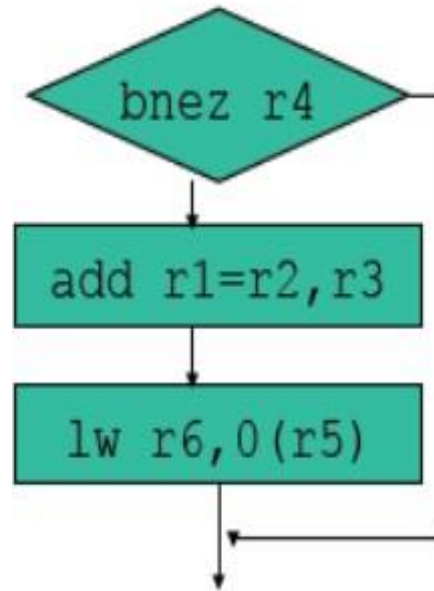
Ejecución condicional (o con predicado)

Una instrucción con predicado contiene un registro booleano (0-1) para el predicado, dentro del campo de la codificación de la instrucción.

- ➡ (p) op Rd, R1, R2.
- ➡ Si el set tiene soporte para predicado, la instrucción solo se ejecuta si p es verdadero
- ➡ Si p es falso, la operación se convierte en un NOP.
- ➡ Los branches deben convertirse a instrucciones con predicado.

Conversión if

Proceso que convierte un salto condicional en una secuencia de instrucciones con predicado.



Ejecución condicional (o con predicado)

1

Se eliminan las dependencias de control, se sustituyen por dependencias de datos (en ejemplo anterior, la instrucción con predicado depende del resultado p de la comparación)

2

Se eliminan los saltos.

3

No requiere hacer "flush" del pipeline. No requiere técnicas de predicción de salto.

5

Instrucciones pueden incluir un operando más, en caso de utilizar predicado.

4

El set de instrucciones se vuelve más complejo.

6

Permite crear bloques básicos más grandes.

7

Si el bloque con predicado es grande y es ejecutado con menos frecuencia, se disminuye el desempeño (la mayoría de instrucciones se convierten en NOPs)

Ejecución condicional

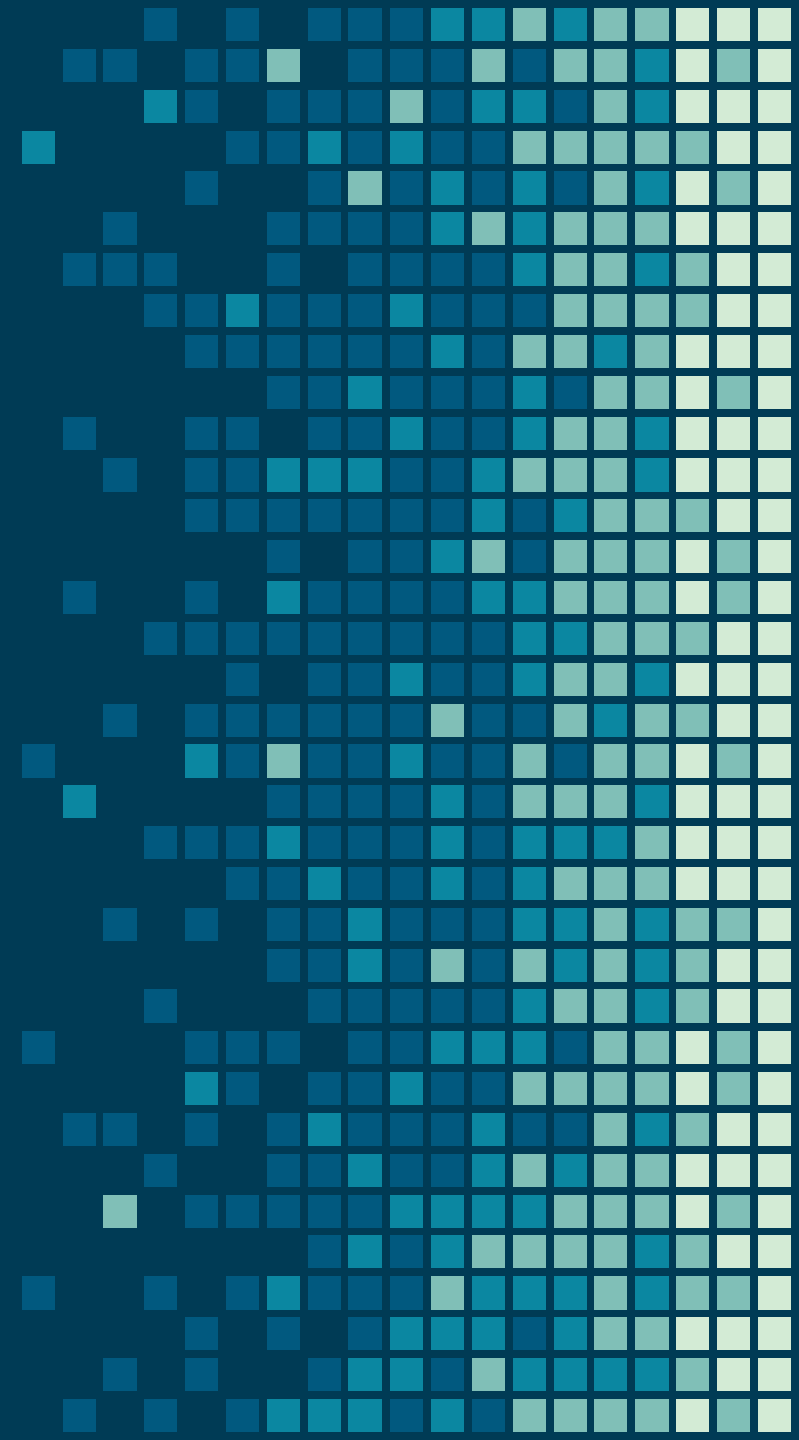
```
### Here we extend VEX to support full predication
### The notation "($p) op" means "if (p) op;"
###
    cmpgt $p1 = $r5, 0
::
    ($p1) mpy $r3 = $r1, $r2
    xnop 1 ### (assuming a 2-cycle mpy latency)
::
    stw 0[$r10] = $r3
::
### if p1 is TRUE, the mpy executes normally
### if p1 is FALSE, the mpy is treated as a nop
```

Predicado

Efectivo cuando:

- Predicciones erróneas son críticas y la penalidad es alta.
- La ruta más larga es ejecutada más frecuentemente. Saltos desbalanceados.

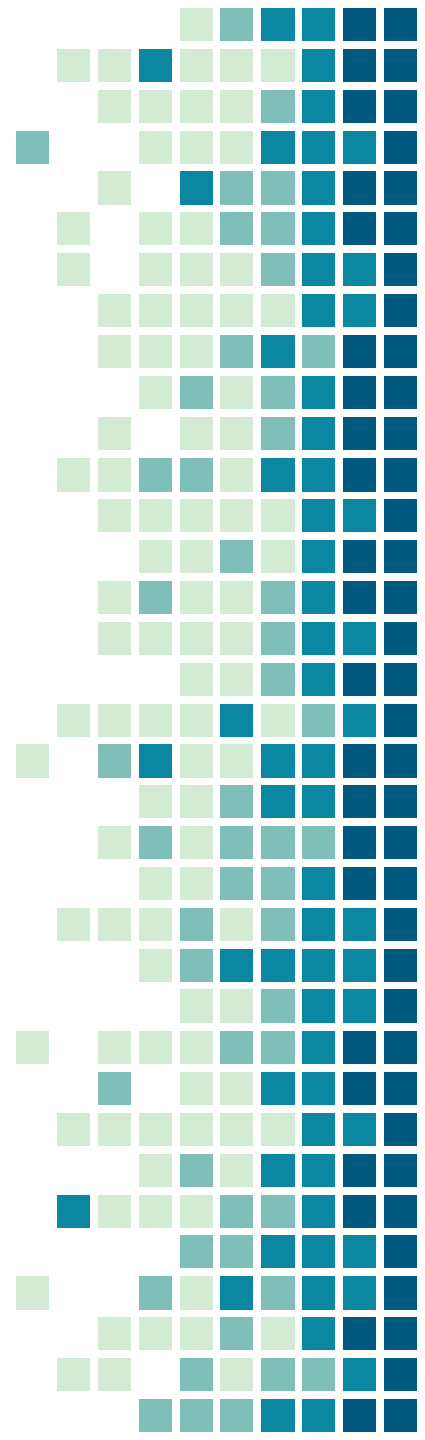
Especulación



Especulación

Cuando no se tiene predicción de saltos, el paralelismo es limitado

Al superar las limitantes de las dependencias de control, se puede explotar un mayor nivel de ILP, especulando (prediciendo) el resultado de los saltos con una política determinada. Típicamente especulación correcta.

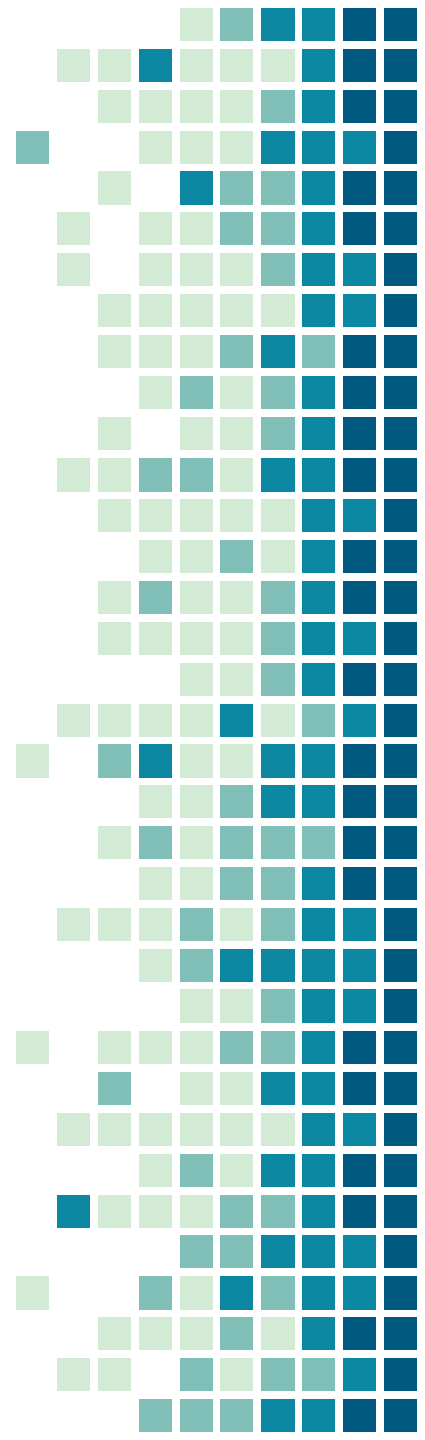


Especulación



Se hace búsqueda, lanzamiento y ejecución de instrucciones como si el resultado de la predicción fuera siempre correcto. Se debe proporcionar un mecanismo para manejar el caso contrario.

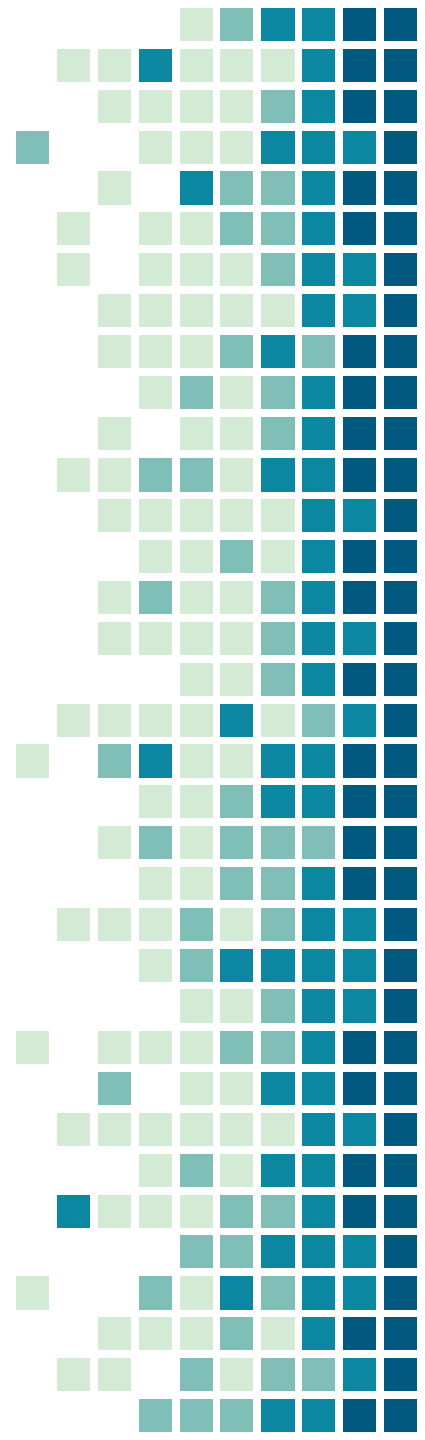
- La especulación puede ser soportada tanto por el **compilador** como por el **hardware**



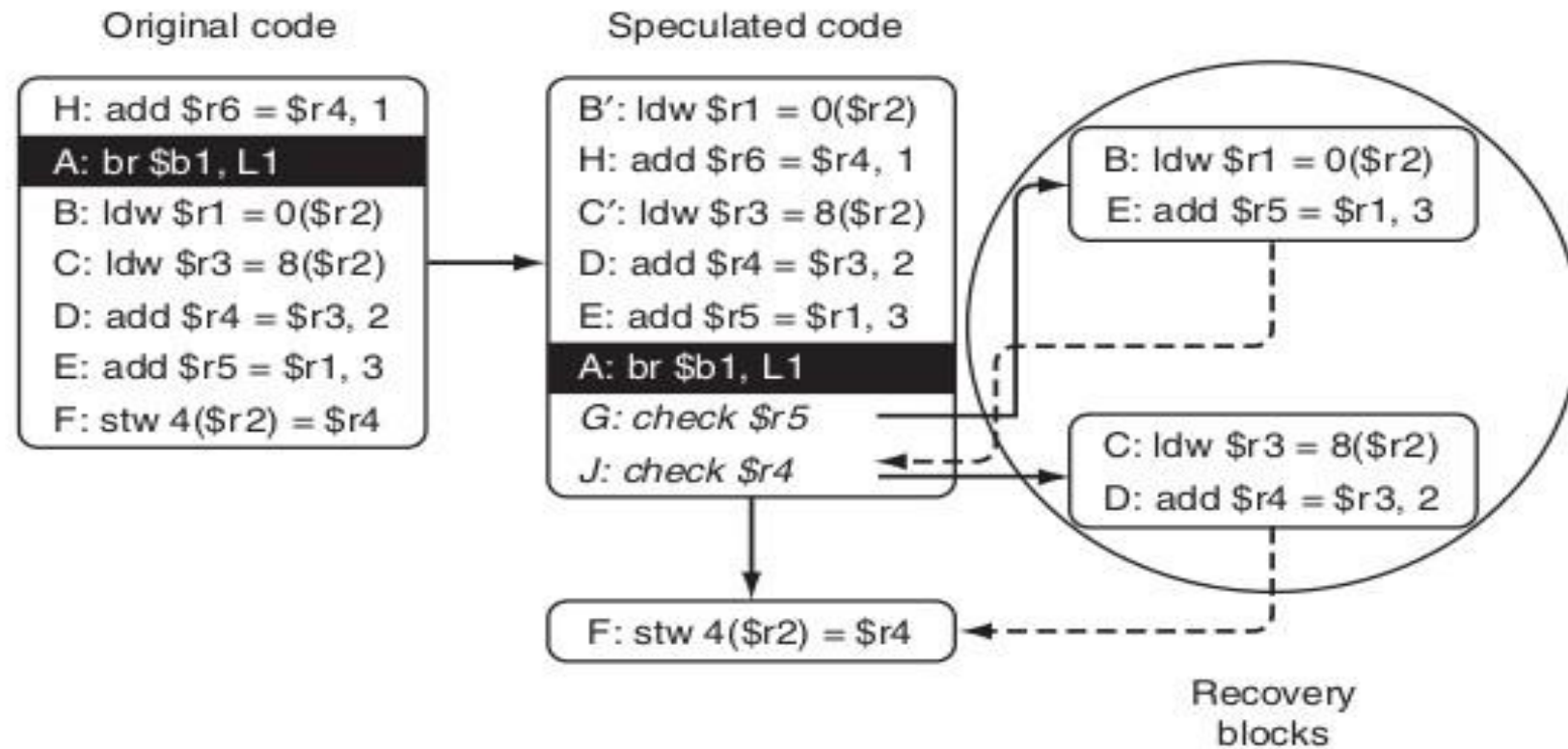
Especulación

La especulación requiere:

- ✓ La habilidad del compilador de encontrar instrucciones que pueden ser movidas sin afectar el flujo de datos (calendarización).
- ✓ Habilidad de ignorar excepciones en instrucciones especuladas o corregir en caso de presentarse una excepción.
- ✓ Habilidad de intercambiar entre loads y stores con conflictos de direcciones.

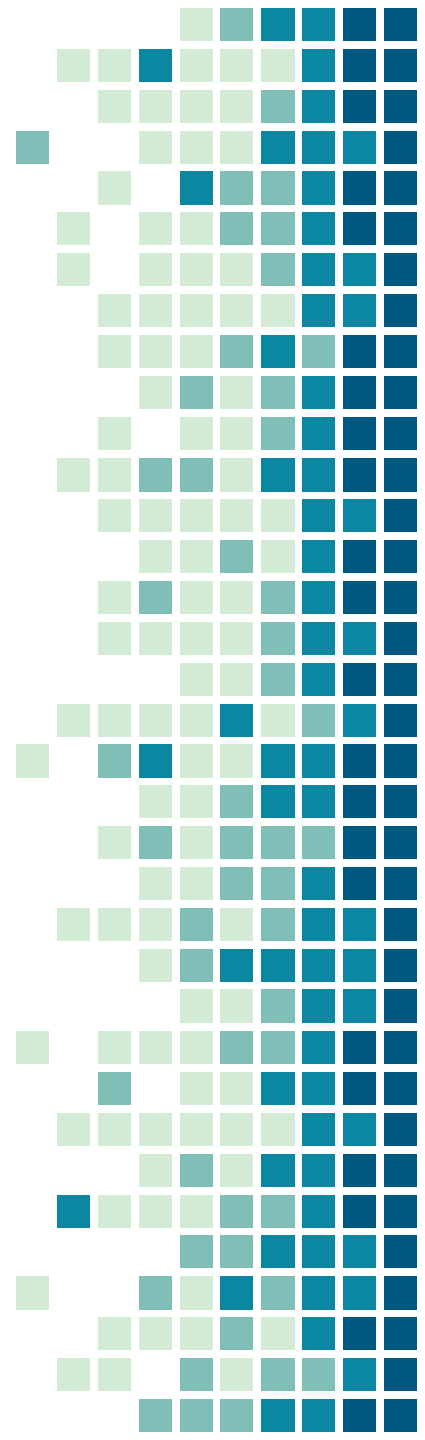


Especulación



Referencias

-► C. Murillo, M. Aguilar (2014)
Arquitectura VLIW
-► J Hennesy and David Patterson (2012)
Computer Architecture: A Quantitative Approach. 5th Edition. Elsevier -
Morgan Kaufmann.
-► Fisher, J. A., Faraboschi, P., & Young, C. (2005).
Embedded computing: a VLIW approach to architecture, compilers and tools.
Elsevier.



¿Preguntas?

Realizado por: Jason Leitón Jiménez.

Tecnológico de Costa Rica

Ingeniería en Computadores

2024

