# Aceleración de una Tarea

Rodrigo Baessa

Figuras: "Computer Organization and Design" 3 Ed., Patterson, D. y Henessy, J. Morgan Kaufmann 2005

# Aceleración de una Tarea

No importando que tarea se este realizando, en muchas ocasiones es deseable que esta se realice más rápido.

Intuitivamente, pensamos en asignar más recursos a la tarea para acelerar su ejecución.

También, pensamos en realizar trabajo simultáneamente, *en paralelo*.

# Paralelismo

El *paralelismo* (parallelism) consiste en realizar trabajo en paralelo, al mismo tiempo.

Hay, típicamente dos acercamientos al paralelismo:

a) realizar la misma tarea al mismo tiempo ó,
b) dividir la tarea en sub-tareas y realizar estas al mismo tiempo.

# Pipelining

El *pipelining* es una forma de paralelismo que consiste en dividir la tarea en sub-tareas y realizar estas al mismo tiempo.

Es el concepto detrás de la línea de ensamble.

# Pipelining

Un ejemplo de la aplicación de pipelining para la aceleración de la ejecución de una tarea es el cuidado de la ropa.

El cuidado de la ropa implica: Lavar la ropa, Secarla, Plancharla y/o doblarla, Guardar la ropa.

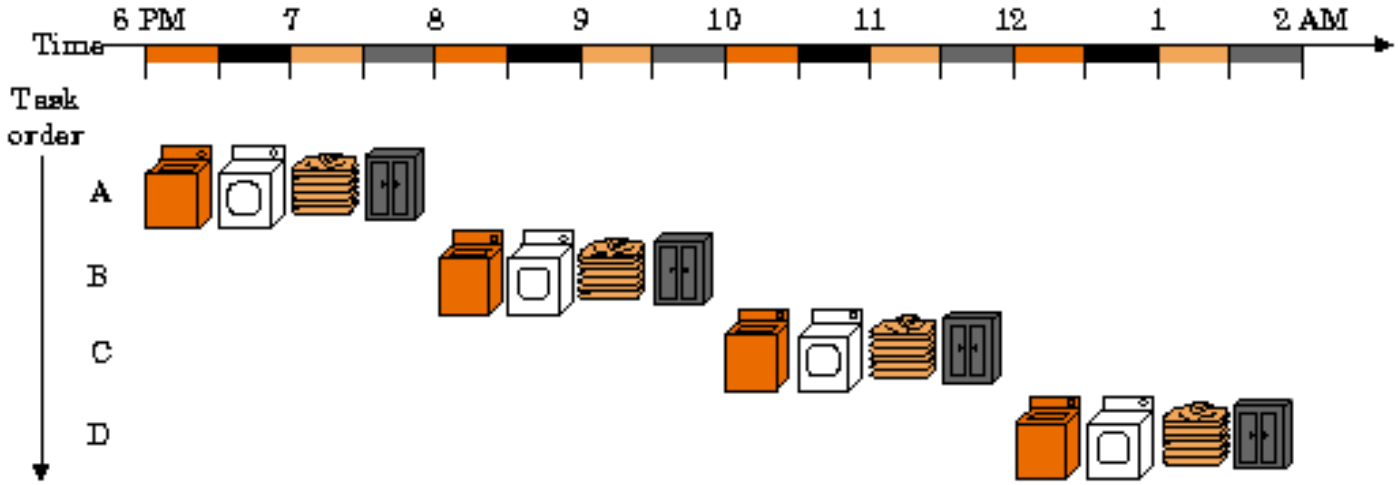
# Pipelining

Entonces, la tarea del cuidado de la ropa, puede dividirse en 4 sub-tareas:

1) Lavar la ropa
2) Secarla
3) Plancharla y/o doblarla
4) Guardar la ropa

# Pipelining

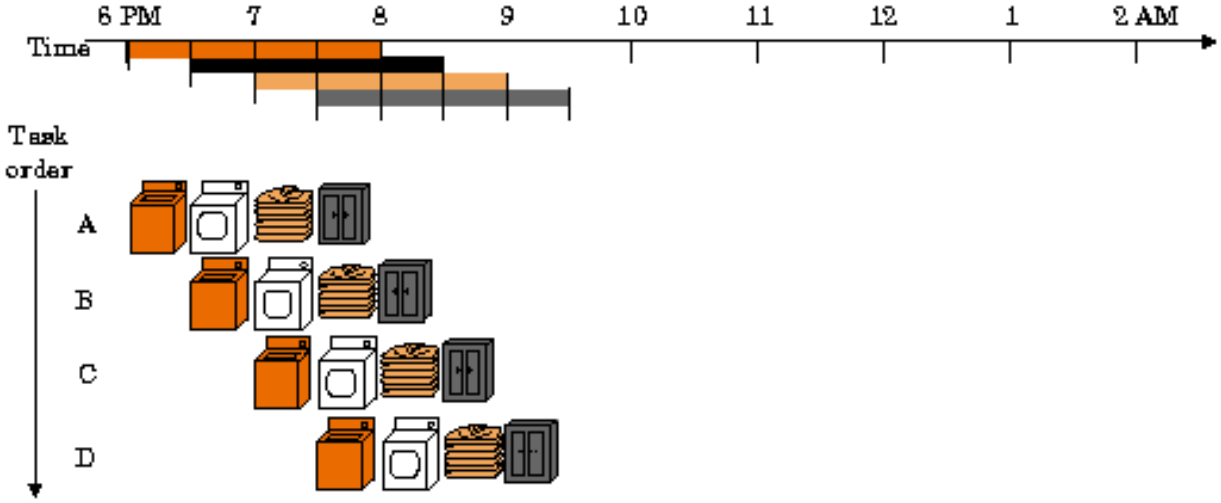
Si yo tengo 4 tandas de ropa, el cuidado de la ropa sin utilizar pipelining sería de la siguiente manera:




Aquí, se asume que cada sub-tarea toma 30 minutos.

# Pipelining

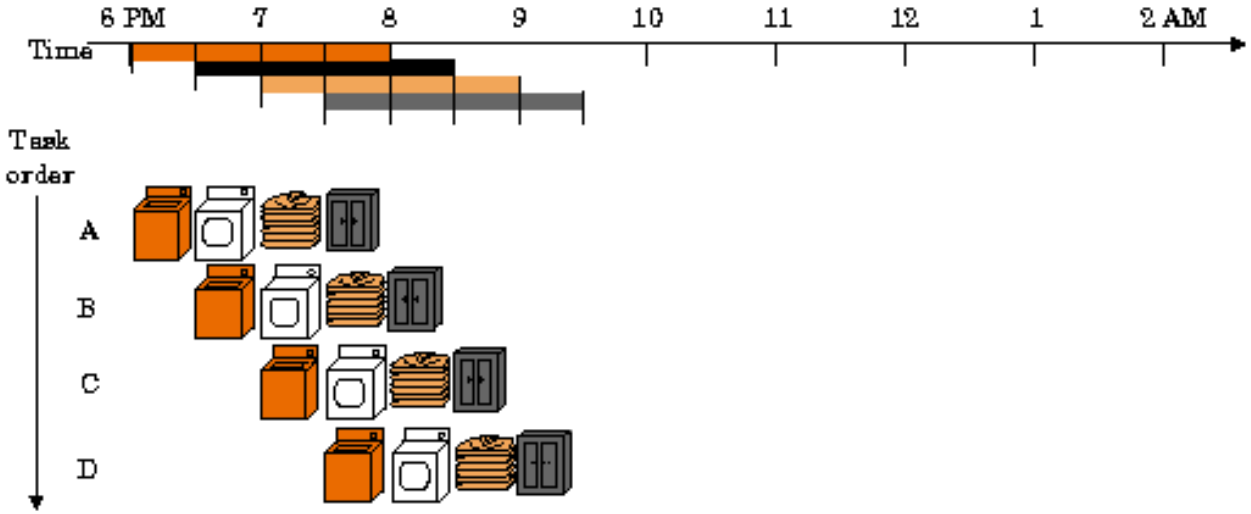
Sin embargo, si le pido ayuda a 3 amigos para implementar pipelining, sería de la siguiente manera:




Noten que la primera tanda de ropa tarda lo mismo que antes en terminar, pero después se completan cada 30 minutos.

# Pipelining

El tiempo en que el pipeline tarda en llenarse se conoce como *pipeline latency*.




Una vez se llena, se aprecia el aumento de desempeño (*speed-up*).

# Pipelining

Si se logra dividir la tarea en *n* sub-tareas que toman el mismo tiempo el speed-up que se alcanza una vez se llena el pipeline es de *n* veces. O sea, que se ejecuta la tarea *n* veces más rápido.

# Pipelining

Sin embargo, es muy difícil que las sub-tareas (etapas) se puedan dividir en tiempos iguales. Siempre hay sub-tareas que toman más tiempo que otras.

# Pipelining

La sub-tarea que toma más tiempo es la que domina la velocidad del pipeline. Así, si la sub-tarea más tardada se demora 45 minutos, el pipeline procesará las tareas cada 45 minutos una vez se llene.

# Pipelining

Otro problema que nos encontramos que hace imposible llegar al speed-up máximo teórico es que al dividir en sub-tareas se van agregando tiempos adicionales (*overhead*), relacionados con la misma división y control adicional necesarios.

# Pipelining

Entonces, la suma del tiempo de las sub-tareas siempre es mayor que el de la tarea original.

Mientras más sub-tareas, mayor es el diferencial de tiempo.

# Pipelining

Para acelerar la ejecución de una tarea en la computación podemos aplicar pipelining a la ejecución de las instrucciones. Esto se conoce como *instruction level parallelism* (paralelismo a nivel de instrucción).

# Pipelining

En la ejecución de una instrucción siempre se realizan ciertos pasos: leer la instrucción de memoria, decodificar la instrucción (qué debe hacer la instrucción), ejecutar la instrucción y a veces, guardar un resultado.

# Pipelining

Así, se puede dividir la ejecución de una instrucción en las siguientes sub-tareas:

1) Instruction Fetch (IF)
2) Instruction Decode (ID)
3) Instruction Execute (IE)
4) Write-Back (WB)

Formando entonces un pipeline de 4 etapas.

# Pipeline Hazards

Adicional a los problemas que describimos antes que afectan el desempeño del pipeline, existen problemas adicionales, propios de la naturaleza secuencial de la ejecución de los programas.

Estos se denominan *pipeline hazards*.

# Pipeline Hazards

Los pipeline hazards se dividen en tres tipos:

1) Structural Hazards
2) Control Hazards
3) Data Hazards

# Structural Hazards

Los Structural Hazards ocurren cuando dos o más etapas del pipeline tratan de utilizar el mismo recurso único. Esto es reconocido como un error de diseño.

Ej. Bus de Datos en SPARC 1

# Ejemplo de Structural Hazards

```
        .data
array1: .space  12     # 12 bytes (array of 3 integers)
        .text
__start:
       la $t0, array1 #  load base address of array into $t0
       li $t1, 5      #  $t1 = 5
       sw $t1, ($t0)  #  first array element set to 5
       li $t1, 13     #   $t1 = 13
       sw $t1, 4($t0) #  second array element set to 13
       li $t1, -7     #   $t1 = -7
       sw $t1, 8($t0) #  third array element set to -7
```

# Ejemplo de Structural Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| Fetch | `la $t0, array1` | **Usa el bus de datos para cargar la instrucción** |
| | `li $t1, 5` | |
| | `sw $t1, ($t0)` | |
| | `li $t1, 13` | |
| | `sw $t1, 4($t0)` | |
| | `li $t1, -7` | |
| | `sw $t1, 8($t0)` | |

# Ejemplo de Structural Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| Decode | `la $t0, array1` | **Usa la lógica de control** |
| Fetch | `li $t1, 5` | **Usa el bus de datos para cargar la instrucción** |
| | `sw $t1, ($t0)` | |
| | `li $t1, 13` | |
| | `sw $t1, 4($t0)` | |
| | `li $t1, -7` | |
| | `sw $t1, 8($t0)` | |

# Ejemplo de Structural Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| Execute | `la $t0, array1` | **Usa el ALU/Lógica de Control** |
| Decode | `li $t1, 5` | **Usa la lógica de control** |
| Fetch | `sw $t1, ($t0)` | **Usa el bus de datos para cargar la instrucción** |
| | `li $t1, 13` | |
| | `sw $t1, 4($t0)` | |
| | `li $t1, -7` | |
| | `sw $t1, 8($t0)` | |

# Ejemplo de Structural Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| Write Back | `la $t0, array1` | **Usa el Datapath para escribir en `$t0`** |
| Execute | `li $t1, 5` | **Usa el ALU/Lógica de Control** |
| Decode | `sw $t1, ($t0)` | **Usa la lógica de control** |
| Fetch | `li $t1, 13` | **Usa el bus de datos para cargar la instrucción** |
| | `sw $t1, 4($t0)` | |
| | `li $t1, -7` | |
| | `sw $t1, 8($t0)` | |

# Ejemplo de Structural Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| | `la $t0, array1` | |
| Write Back | `li $t1, 5` | **Usa el Datapath para escribir en `$t1`** |
| Execute | `sw $t1, ($t0)` | **Usa el ALU/Lógica de Control** |
| Decode | `li $t1, 13` | **Usa la lógica de control** |
| Fetch | `sw $t1, 4($t0)` | **Usa el bus de datos para cargar la instrucción** |
| | `li $t1, -7` | |
| | `sw $t1, 8($t0)` | |

# Ejemplo de Structural Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| | `la $t0, array1` | |
| | `li $t1, 5` | |
| Write Back | `sw $t1, ($t0)` | **Usa el bus de datos para escribir en `($t0)`** |
| Execute | `li $t1, 13` | **Usa el ALU/Lógica de Control** |
| Decode | `sw $t1, 4($t0)` | **Usa la lógica de control** |
| Fetch | `li $t1, -7` | **Usa el bus de datos para cargar la instrucción** |
| | `sw $t1, 8($t0)` | |

# Control Hazards

Los Control Hazards ocurren cuando se altera el flujo normal secuencial de un programa debido a saltos (branches) y llamadas a subrutinas (calls).

# Control Hazards

El procesador reconoce que una instrucción ($i$) es un branch hasta la etapa de Instruction Decode. Para esto, ya fue a buscar a memoria la siguiente instrucción ($i+1$). En el caso de un salto incondicional, esto causa siempre un problema ya que se debe eliminar del pipeline la siguiente instrucción ($i+1$) e ir a buscar la instrucción del destino del salto (target).

# Control Hazards

En el caso de los saltos condicionales se vuelve más complicado ya que la condición comúnmente se revisa en la etapa de Instruction Execute y el salto puede o no tomarse. De igual forma, si se toma, ya hay adentro del pipeline dos instrucciones que no deben de ejecutarse.

# Ejemplo #1 de Control Hazards

```
main:
1)  addi $a0, $0, 5       # $a0 = 5
2)  addi $a1, $0, -20     # $a1 = -20
3)  addi $a2, $0, 13      # $a2 = 13
4)  add  $a3, $a0, $a1    # $a3 = $a0 + $a1
5)  b    next             #saltar a next
rutina:
6)  sub  $s0, $a3, $a1
7)  add  $s0, $0, $v0
.......................................................
next:
8)  add $a1, $a3, $a2
9)  sw  $t1, 8($sp)     #save register $t1
10) sw  $t0, 4($sp)     #save register $t0
11) sw  $s0, 0($sp)     #save register $s0
```

# Ejemplo #1 de Control Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `4) add $a3,$a0,$a1` | |
| **Decode** | `3) addi $a2,$0,13` | |
| **Execute** | `2) addi $a1,$0,-20` | |
| **Write Back** | `1) addi $a0,$0,5` | |

# Ejemplo #1 de Control Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `5) b next` | **Aún no sabemos que es un salto incondicional** |
| **Decode** | `4) add $a3,$a0,$a1` | |
| **Execute** | `3) addi $a2,$0,13` | |
| **Write Back** | `2) addi $a1,$0,-20` | |

# Ejemplo #1 de Control Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| Fetch | `6) sub $s0,$a3,$a1` | Instrucción equivocada (no es el target) |
| Decode | `5) b next` | Aquí nos enteramos de que es un salto incondicional |
| Execute | `4) add $a3,$a0,$a1` | |
| Write Back | `3) addi $a2,$0,13` | |

# Ejemplo #1 de Control Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| Fetch | `8) add $a1,$a3,$a2` | Instrucción correcta (target en la etiqueta "next") |
| Decode | ~~`6) sub $s0,$a3,$a1`~~ | Debemos reemplazar por NOP (Burbuja) |
| Execute | `5) b next` | El salto lo hace la etapa de Fetch |
| Write Back | `4) add $a3,$a0,$a1` | |

# Ejemplo #1 de Control Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `9) sw $t1,8($sp)` | |
| **Decode** | `8) add $a1,$a3,$a2` | |
| **Execute** | ~~`6) sub $s0,$a3,$a1`~~ | **Burbuja que ocupa un ciclo del pipeline** |
| **Write Back** | `5) b next` | |

# Ejemplo #1 de Control Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `10) sw $t0,4($sp)` | |
| **Decode** | `9) sw $t1,8($sp)` | |
| **Execute** | `8) add $a1,$a3,$a2` | |
| **Write Back** | ~~`6) sub $s0,$a3,$a1`~~ | **Burbuja que ocupa un ciclo del pipeline** |

# Ejemplo #2 de Control Hazards

```
main:
1)  addi $a1, $0, 40      # $a1 = 40
2)  addi $t5, $0, 3       # $t5 = 3
3)  add  $a3, $a2, $t4    # $a3 = $a2 + $t4
4)  bne  $t6, $t7, loop   # saltar a loop si no son iguales
rutina:
5)  sub  $s0, $a3, $a1
6)  add  $s0, $t0, $t1
7)  add  $s2, $t2, $t3
.......................................................
loop:
8)  add $a1, $a3, $a2
9)  sw  $t1, 8($sp)     #save register $t1
10) sw  $t0, 4($sp)     #save register $t0
```

# Ejemplo #2 de Control Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `4) bne $t6,$t7,loop` | **Aún no sabemos que es un salto condicional** |
| **Decode** | `3) add $a3,$a2,$t4` | |
| **Execute** | `2) addi $t5,$0,3` | |
| **Write Back** | `1) addi $a1,$0,40` | |

# Ejemplo #2 de Control Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `5) sub $s0,$a3,$a1` | |
| **Decode** | `4) bne $t6,$t7,loop` | **Aquí nos enteramos de que es un salto condicional, pero debemos comparar para saber si saltar o no.** |
| **Execute** | `3) add $a3,$a2,$t4` | |
| **Write Back** | `2) addi $t5,$0,3` | |

# Ejemplo #2 de Control Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `6) add $s0,$0,$v0` | |
| **Decode** | `5) sub $s0,$a3,$a1` | |
| **Execute** | `4) bne $t6,$t7,loop` | **Comparamos...** |
| **Write Back** | `3) add $a3,$a2,$t4` | |

(Continuar)

Condición Verdadera
(Tomar el Salto)

Condición Falsa
(No Tomar el Salto)

# Si se toma el salto...

| Etapa | Instrucción | Comentarios |
|---|---|---|
| Fetch | `8) add $a1,$a3,$a2` | **Destino del salto** |
| Decode | ~~`6) add $s0,$0,$v0`~~ | **BURBUJA** |
| Execute | ~~`5) sub $s0,$a3,$a1`~~ | |
| Write Back | `4) bne $t6,$t7,loop` | |

Regresar

# Si no se toma el salto...

| Etapa | Instrucción | Comentarios |
|---|---|---|
| Fetch | `7) add $s2,$t2,$t3` | |
| Decode | `6) add $s0,$0,$v0` | **No hay problema!** |
| Execute | `5) sub $s0,$a3,$a1` | |
| Write Back | `4) bne $t6,$t7,loop` | |

Regresar

# Control Hazards

El impacto de los control hazards se hace evidente al saber que el promedio de branches y calls en la ejecución de un programa es de aprox. 25-30% del total de instrucciones.

# Control Hazards

Para tratar de disminuir este impacto hay dos tipos de soluciones:

a) Estrategias Estáticas
b) Estrategias Dinámicas

Las estrategias estáticas son las que se implementan durante la compilación.

Las estrategias dinámicas son las que se implementan durante la ejecución del código.

# Control Hazards

Estrategias Estáticas

Dentro de las estrategias estáticas tenemos las siguientes:

a) Loop Un-rolling
b) Delayed Branching

# Control Hazards

Estrategias Estáticas

a) Loop Un-rolling: Consiste en copiar varias veces el cuerpo de un loop para que el número total de saltos sea menor.

# Control Hazards

Estrategias Estáticas

b) Delayed Branching: Un procesador que implementa delayed branching siempre ejecuta las instrucciones que ya ingresaron al pipeline, se tome ó no el branch.

# Control Hazards

Estrategias Estáticas

Delayed Branching: Es trabajo del compilador el encontrar una(s) instrucción(es) útil(es) que pueda(n) insertarse después del branch (en el *branch delay slot*) para que no se desperdicie el tiempo en caso de tomarse el branch.

# Control Hazards

Estrategias Estáticas

Delayed Branching (cont):

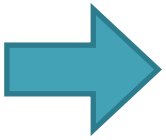
Por eso se llama delayed branching, por que no se salta en el momento sino que después de las instrucciones que ya ingresaron al pipeline.

Si no encuentra ninguna, inserta un NOP (no operation).

Así, se puede dar reordenamiento de las instrucciones durante la compilación!

# Ejemplo de Loop Unrolling

```
  addi $t3,$t0,0
loop:
  lw   $t1, Array($t3)
  add  $t4, $t4, $t1
  addi  $t3, $t3, 4
  blt   $t3, 120, loop
```



```
  addi $t3,$t0,0
loop:
  lw   $t1, Array($t3)
  add  $t4, $t4, $t1
  addi  $t3, $t3, 4

  lw   $t1, Array($t3)
  add  $t4, $t4, $t1
  addi  $t3, $t3, 4

  lw   $t1, Array($t3)
  add  $t4, $t4, $t1
  addi  $t3, $t3, 4

  blt   $t3, 120, loop
```

# Ejemplo #1 con Delayed Branching

```
main:
1)   addi $a0, $0, 5
2)   addi $a1, $0, -20
3)   addi $a2, $0, 13
4)   add  $a3, $a0, $a1
5)   b    next
rutina:
6)   sub  $s0, $a3, $a1
7)   add  $s0, $0, $v0
.......................
next:
8)   add $a1, $a3, $a2
9)   sw  $t1, 8($sp)
10)  sw  $t0, 4($sp)
11)  sw  $s0, 0($sp)
```

```
main:
1)   addi $a0, $0, 5
2)   addi $a1, $0, -20
3)   addi $a2, $0, 13
5)   b    next
4)   add  $a3, $a0, $a1
rutina:
6)   sub  $s0, $a3, $a1
7)   add  $s0, $0, $v0
.......................
next:
8)   add $a1, $a3, $a2
9)   sw  $t1, 8($sp)
10)  sw  $t0, 4($sp)
11)  sw  $s0, 0($sp)
```

# Ejemplo #1 con Delayed Branching

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `5) b next` | **Aún no sabemos que es un salto incondicional** |
| **Decode** | `3) addi $a2,$0,13` | |
| **Execute** | `2) addi $a1,$0,-20` | |
| **Write Back** | `1) addi $a0,$0,5` | |

# Ejemplo #1 con Delayed Branching

| Etapa | Instrucción | Comentarios |
|---|---|---|
| Fetch | `4) add $a3,$a0,$a1` | De todos modos queríamos ejecutar esta instrucción |
| Decode | `5) b next` | Nos enteramos de que es un salto incondicional |
| Execute | `3) addi $a2,$0,13` | |
| Write Back | `2) addi $a1,$0,-20` | |

# Ejemplo #1 con Delayed Branching

| Etapa | Instrucción | Comentarios |
|---|---|---|
| Fetch | `8) add $a1,$a3,$a2` | Instrucción correcta (target en la etiqueta "next") |
| Decode | `4) add $a3,$a0,$a1` | |
| Execute | `5) b next` | El salto lo hace la etapa de Fetch |
| Write Back | `3) addi $a2,$0,13` | |

# Ejemplo #1 con Delayed Branching

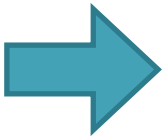
| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `9) sw $t1,8($sp)` | |
| **Decode** | `8) add $a1,$a3,$a2` | **Note que no hubo burbuja** |
| **Execute** | `4) add $a3,$a0,$a1` | |
| **Write Back** | `5) b next` | |

# Ejemplo #1 con Delayed Branching

| Etapa | Instrucción | Comentarios |
|---|---|---|
| Fetch | `10) sw $t0,4($sp)` | |
| Decode | `9) sw $t1,8($sp)` | |
| Execute | `8) add $a1,$a3,$a2` | |
| Write Back | `4) add $a3,$a0,$a1` | |

# Ejemplo #2 con Delayed Branching

```
main:
1)   addi $a1, $0, 40
2)   addi $t5, $0, 3
3)   add  $a3, $a2, $t4
4)   bne  $t6, $t7, loop
rutina:
5)   sub  $s0, $a3, $a1
6)   add  $s0, $t0, $t1
7)   add  $s2, $t2, $t3
.......................
   loop:
8)   add $a1, $a3, $a2
9)   sw  $t1, 8($sp)
10)  sw  $t0, 4($sp)
```



```
main:
1)   addi $a1, $0, 40
4)   bne  $t6, $t7, loop
2)   addi $t5, $0, 3
3)   add  $a3, $a2, $t4
rutina:
5)   sub  $s0, $a3, $a1
6)   add  $s0, $t0, $t1
7)   add  $s2, $t2, $t3
.......................
   loop:
8)   add $a1, $a3, $a2
9)   sw  $t1, 8($sp)
10)  sw  $t0, 4($sp)
```

# Ejemplo #2 con Delayed Branching

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `4) bne $t6,$t7,loop` | **Aún no sabemos que es un salto condicional** |
| **Decode** | `1) addi $a1,$0,40` | |
| **Execute** | | |
| **Write Back** | | |

# Ejemplo #2 con Delayed Branching

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `2) addi $t5, $0, 3` | **Instrucción que de todos modos queremos ejecutar** |
| **Decode** | `4) bne $t6,$t7,loop` | **Aquí nos enteramos de que es un salto condicional, pero debemos comparar para saber si saltar o no.** |
| **Execute** | `1) addi $a1,$0,40` | |
| **Write Back** | | |

# Ejemplo #2 con Delayed Branching

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `3) add $a3,$a2,$t4` | **Instrucción que de todos modos queremos ejecutar** |
| **Decode** | `2) addi $t5,$0,3` | **Instrucción que de todos modos queremos ejecutar** |
| **Execute** | `4) bne $t6,$t7,loop` | **Comparamos...** |
| **Write Back** | `1) addi $a1,$0,40` | |

(Continuar)

Condición Verdadera
(Tomar el Salto)

Condición Falsa
(No Tomar el Salto)

# Si se toma el salto...

| Etapa | Instrucción | Comentarios |
|---|---|---|
| Fetch | 8) add $a1,$a3,$a2 | Destino del salto |
| Decode | 3) add $a3,$a2,$t4 | No hay problema! |
| Execute | 2) addi $t5,$0,3 | |
| Write Back | 4) bne $t6,$t7,loop | Comparamos... |

Regresar

# Si no se toma el salto...

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `5) sub $s0,$a3,$a1` | **Instrucción siguiente** |
| **Decode** | `3) add $a3,$a2,$t4` | **No hay problema!** |
| **Execute** | `2) addi $t5,$0,3` | |
| **Write Back** | `4) bne $t6,$t7,loop` | **Comparamos...** |

Regresar

# Control Hazards

Estrategias Dinámicas

Branch Prediction Buffer: El *branch prediction buffer* es una tabla de dos columnas (dirección y predicción) en la cual el procesador va tomando nota de la dirección de las instrucciones que son branches junto con uno o más bits de predicción (P). Si se usa un bit de predicción, solo se indica si la última vez el branch se tomo (P=1) o no (P=0).

# Control Hazards

Estrategias Dinámicas

Branch Prediction Buffer: Si se utilizan dos bits, se puede incrementar P cada vez que se toma el branch y decrementar cada vez que no se tome.

Dependiendo del valor de P, el procesador toma una acción prediciendo que ocurrirá con el branch. Al tener información histórica se reduce la probabilidad de cometer errores.

# Control Hazards

Estrategias Dinámicas

Branch Target Buffer: El *branch target buffer* es un branch prediction buffer al cual se le agrega una columna con la dirección del destino (target) del branch, así, se tiene disponible de inmediato.

Branch Folding: *Branch Folding* se conoce como la técnica de guardar en un branch target buffer la instrucción del destino (target) del branch, y no solamente la dirección.

# Data Hazards

Los Data Hazards ocurren cuando una etapa del pipeline trata de accesar un dato que aún no esta disponible, ya que esta siendo actualizado por una instrucción anterior. Esto ocurre cuando existe una dependencia de datos (*data dependency*) entre dos instrucciones y estas están separadas por la distancia correcta.

# Data Dependencies

Hay dos tipos de Data Dependencies:

a) True o Real Data Dependencies
b) Name Dependencies

# Data Dependencies

<u>True Data Dependencies</u>:

Esto es cuando hay flujo de información entre dos instrucciones *i* y *j*. La instrucción *i* (que esta antes de la instrucción *j*) escribe un resultado que la instrucción *j* utiliza como operando. Esto se conoce como Read After Write (RAW). Si existe una dependencia de datos entre las instrucciones *i* y *j* y, entre las instrucciones *j* y *k* también hay una dependencia entre *i* y *k*.

# Data Dependencies

Name Dependencies:

Esto es cuando no hay flujo de información entre dos instrucciones que utilizan el mismo *Name*. "Name" es el conjunto que incluye tanto a los registros como a las posiciones de memoria. Hay de dos tipos: Write After Write (WAW) y Write After Read (WAR). En ambos casos se pueden presentar problemas cuando hay reordenamiento de instrucciones. (Algo que ya vimos que ocurre.)

# Data Dependencies

Name Dependencies:

En Write After Write (WAW), dos instrucciones escriben al mismo *name*. Si estas se reordenan y terminan en orden invertido las instrucciones que accesan al *name* obtendrán un valor incorrecto.

# Data Dependencies

Name Dependencies:

En el Write After Read (WAR), una instrucción lee y luego otra escribe al mismo name. Si estas se reordenan y terminan en orden invertido la instrucción que lee al name obtendrá un valor incorrecto.

Para ambos casos, lo que procede es utilizar dos names separados, esto se conoce como *name scheduling*.

# Ejemplo de Data Hazards (RAW)

```
main:
1)   addi $a0, $0, 5       # $a0 = 5
2)   addi $a1, $0, -20     # $a1 = -20
3)   addi $a2, $0, 13      # $a2 = 13
4)   add  $a3, $a2, $a1    # $a3 = $a2 + $a1
5)   sub  $s0, $a3, $a1
6)   add  $s0, $0, $v0
```

# Ejemplo de Data Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `4) add $a3,$a2,$a1` | |
| **Decode** | `3) addi $a2,$0,13` | |
| **Execute** | `2) addi $a1,$0,-20` | |
| **Write Back** | `1) addi $a0,$0,5` | |

# Ejemplo de Data Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| Fetch | `5) sub $s0,$a3,$a1` | |
| Decode | `4) add $a3,$a2,$a1` | |
| Execute | `3) addi $a2,$0,13` | |
| Write Back | `2) addi $a1,$0,-20` | |

# Ejemplo de Data Hazards

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `6) add $s0,$0,$v0` | |
| **Decode** | `5) sub $s0,$a3,$a1` | |
| **Execute** | `4) add $a3,$a2,$a1` | **Necesitamos el valor de $a2** |
| **Write Back** | `3) addi $a2,$0,13` | **Estamos guardando el valor de $a2** |

# Data Hazards

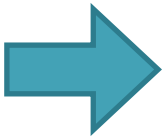
Los métodos para reducir o eliminar los data hazards son:

a) Incorporar hardware adicional al pipeline para que estén disponibles los resultados antes (en registros temporales).

b) Reordenar las instrucciones para cambiar la separación entre data dependencies (esto lo hace el compilador).

# Ejemplo de Reordenamiento

```
main:
1)   addi $a0, $0, 5
2)   addi $a1, $0, -20
3)   addi $a2, $0, 13
4)   add  $a3, $a2, $a1
5)   sub  $s0, $a3, $a1
6)   add  $s0, $0, $v0
```



```
main:
2)   addi $a1, $0, -20
3)   addi $a2, $0, 13
1)   addi $a0, $0, 5
4)   add  $a3, $a2, $a1
5)   sub  $s0, $a3, $a1
6)   add  $s0, $0, $v0
```

# Después del Reordenamiento

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `4) add $a3,$a2,$a1` | |
| **Decode** | `1) addi $a0,$0,5` | |
| **Execute** | `3) addi $a2,$0,13` | |
| **Write Back** | `2) addi $a1,$0,-20` | |

# Después del Reordenamiento

| Etapa | Instrucción | Comentarios |
|---|---|---|
| **Fetch** | `5) sub $s0,$a3,$a1` | |
| **Decode** | `4) add $a3,$a2,$a1` | |
| **Execute** | `1) addi $a0,$0,5` | |
| **Write Back** | `3) addi $a2,$0,13` | **Aquí se guarda el resultado de $a2** |

# Después del Reordenamiento

| Etapa | Instrucción | Comentarios |
|---|---|---|
| Fetch | 6) add $s0,$0,$v0 | |
| Decode | 5) sub $s0,$a3,$a1 | |
| Execute | 4) add $a3,$a2,$a1 | Necesitamos el valor de $a2, y lo tenemos listo |
| Write Back | 1) addi $a0,$0,5 | Aquí se guarda el resultado de $a0 |