Winmips – RISC parte 1

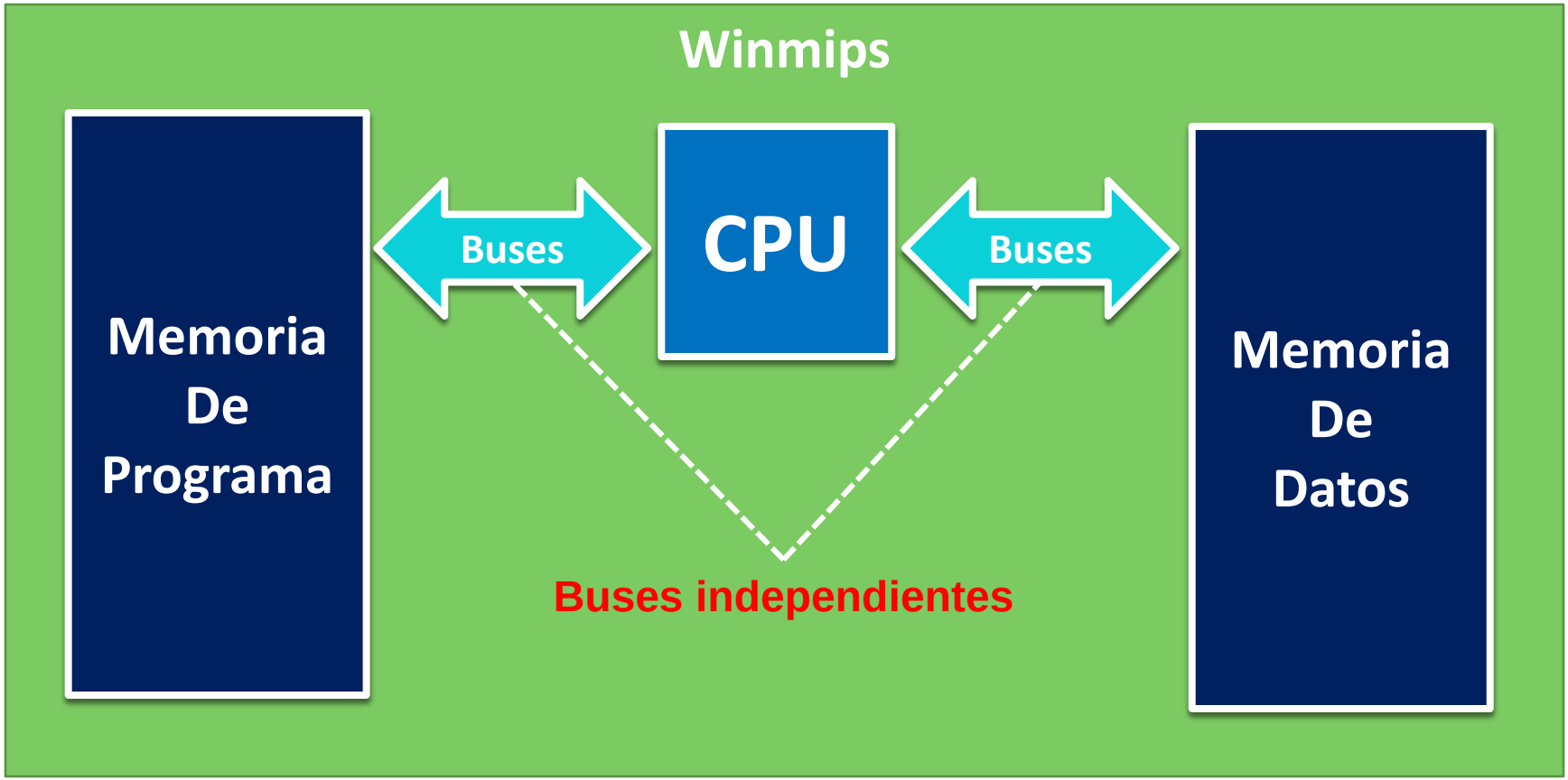
Arquitectura de computadoras

Winmips - Características

- Procesador RISC:
  - Conjunto reducido de instrucciones
  - El poder y las instrucciones del WinMips no es comparable con el MSX88 (1999 vs 1979)
- Arquitectura Harvard
- Instrucciones de 32 bits:
  - Multiplica y divide números enteros
  - Multiplica y divide números flotantes
- Registros de 64 bits
- Memoria 1G de programa 2^{30} palabras
- no hay flags de estado

Winmips - Características

- Arquitectura Harvard



Arquitectura Harvard

- Ventajas :
  - El tamaño de las instrucciones no depende del tamaño de la memoria de datos (no es , necesariamente, múltiplo de 8).
  - Electrónica mas simple
  - Facilita el acceso simultaneo a memoria sin conflictos al tener buses separados

- Desventajas:
  - Menos flexibilidad para almacenar datos y programas:
    - Un programa podría no entrar en memoria aunque la suma de las memorias de datos y programa fuera suficiente

Winmips - Registros

- 32 registros de enteros de propósito general:
  - r{i} con i nro de registro
  - r0 siempre es 0, ignora modificaciones
- 32 registros de punto flotante:
  - f{i} donde i es el nro de registro
- No hay registro de flags de estado
- No tiene soporte de pila

Directivas de compilador

- .code o .text :
  - Indica que el texto que sigue es código fuente
  - El compilador asigna automáticamente la posición en memoria de programa
- .data :
  - Indica al compilador que el texto que sigue son datos
  - El compilador asigna automáticamente la posición en memoria de datos
- .org {dirección específica}
  - Ubica el texto que sigue a partir de una dirección específica. Se puede usar con código o con datos.
- .space {número de bytes}
  - Deja un espacio de bytes entre lo anterior y lo siguiente

Directivas de compilador - Datos

- .ascii {string} :
  - Almacena una cadena ASCII
  - Ejemplo: `mi_variable: .ascii "hola" ;4 bytes`
- .asciiz {string} :
  - Almacena una cadena ASCII terminada en 0
  - Ejemplo: `mi_variable: .asciiz "hola" ;5 bytes`
- .word {n} :
  - almacena número(s) de 64-bits
  - Ejemplo: `mi_variable: .word 1234 ;8 bytes`
- .word 32 {n}
  - almacena número(s) de 32-bits
  - Ejemplo: `mi_variable: .word32 1234 ;4 bytes`

Directivas de compilador - Datos

- .word 16 {n}
  - almacena número(s) de 16-bits
  - Ejemplo: `mi_variable: .word16 1234 ;2 bytes`
- .byte {n}
  - almacena bytes
  - Ejemplo: `mi_variable: .byte 123 ;1 byte`
- .double{n}
  - almacena número(s) en punto flotante
  - Ejemplo: `mi_variable: .double 3.14159 ;8 bytes`

Winmips - Instrucciones

- Instrucciones de 3 operandos:
  - {Operación } {Destino} , {Operando 1}, {Operando 2}
  - Ejemplo:
    - `DADDI R1, R2, 5 ; R1 ← R2 + 5`
    - `DMUL R2, R3, R3 ; R2 ← R3 * R3`
    - `DDIV R25, R4, R3 ; R25 ← R4 / R3`

Instrucciones – Transferencia de datos

- Solo hay 2 instrucciones para acceder a memoria de datos
- Las demás instrucciones utilizan registros y/o valores inmediatos
- **LD** {reg. Dest.} **,** {memoria} **(** {reg. Despl.} **)** :
  - Load Data: asignar un registro con un valor de memoria
  - {reg. Dest} ← [{memoria} + {reg. Despl.}]
  - Ejemplo: `LD R1, mi_var(R2)` `R1 ← [mi_var+R2]`
- **SD** {reg. Orig.} **,** {memoria} **(** {reg. Despl.} **)** :
  - Store Data: copia en memoria el valor de un registro
  - [{memoria} + {reg. Despl.}] ← {reg. Dest}
  - Ejemplo: `SD R1, mi_var(R2)` `[mi_var+R2] ← R1`

Instrucciones – Transferencia de datos

- ¿Porque sumar siempre un registro a una dirección de memoria?
  - ¿ y por que no?
  - Si "mi_var" es un entero de 64 bits y no quiero sumarle a su **dirección** nada, puedo usar R0 que siempre está en cero:

```
LD R1, mi_var(R0)        R1 ← [mi_var]
```

  - Pero si "mi_var" fuera un arreglo (ej: de enteros de 64 bits), sería interesante sumarle a la dirección un desplazamiento:

```
Bucle: LD R1, mi_var(R2)   ;recupera elemento
       ;…
       ;instrucciones para procesar elemento
       ;…
       ADD R2, R2, 8   ;desplaza a prox. Elemento
       J Bucle         ;repite proceso
```

Instrucciones – Aritméticas

- **DADDI** {reg. Dest.} , {reg. Oper. 1} , {inmediato 16 bits} :
  - Sumar un registro y un valor inmediato
  - {reg. Dest} ← {reg. Oper. 1 } + {valor inmediato}
  - Ejemplo:

```
DADDI R1, R2, -1 ;suma valor negativo
DADDI R1, R0, 5  ;asignación valor inmediato
```

- **DADD** {reg. Dest.} , {reg. Oper. 1} , {reg. Oper. 2} :
  - Sumar dos registros
  - {reg. Dest} ← {reg. Oper. 1 } + {reg. Oper. 2 }
  - Ejemplo:

```
DADD R1, R2, R3 ;suma de registros
DADD R1, R2, R0 ;asignación de registro
```

- **Instrucciones similares:**
  - **DSUB, DMUL, DDIV**

Instrucciones – Lógicas

- **ANDI** {reg. Dest.} , {reg. Oper. 1} , {inmediato 16 bits} :
  - Realizar "Y" entre un registro y valor inmediato
  - {reg. Dest} ← {reg. Oper. 1} AND {valor inmediato}
  - Ejemplo:

```
ANDI R1, R2, 5 ; R1 = R2 AND 5
```

- **AND** {reg. Dest.} , {reg. Oper. 1} , {reg. Oper. 2} :
  - Realizar "Y" entre dos registros
  - {reg. Dest} ← {reg. Oper. 1} AND {valor reg. Oper. 2}
  - Ejemplo:

```
AND R1, R2, R3 ; R1 = R2 AND R3
```

- **Instrucciones similares:**
  - **"O": OR, ORI**
  - **"O exclusivo": XOR, XORI**

Instrucciones – Desplazamientos

- **DSLL** {reg. Dest.} , {reg. Oper. 1} , {inmediato 5 bits} :
  - Desplazar a izquierda un registro según un valor inmediato
  - {reg. Dest} ← {reg. Oper. 1} << {valor inmediato}
  - Ejemplo:

```
DSLL R1, R2, 1 ; R1 = R2 << 1
```

- **DSLLV** {reg. Dest.} , {reg. Oper. 1} , {reg. Oper. 2} :
  - Realizar "Y" entre dos registros
  - {reg. Dest} ← {reg. Oper. 1} << {valor reg. Oper. 2}
  - Ejemplo:

```
DSLLV R1, R2, R3 ; R1 = R2 << R3
```

- **Instrucciones similares:**
  - **Desplazamiento Aritmético a derecha:** DSRA, DSRAV

Instrucciones – Comparaciones

- **SLTI** {reg. Dest.} , {reg. Oper. 1} , {inmediato 16 bits} :
  - comparar por menor un registro y valor inmediato
  - {reg. Dest} ← {reg. Oper. 1} < {valor inmediato}
  - Ejemplo:

```
SLTI R1, R2, 2 ; R1=1 si R2<2, R1=0 si R2>=2
```

- **SLT** {reg. Dest.} , {reg. Oper. 1} , {reg. Oper. 2} :
  - comparar por menor dos registros
  - {reg. Dest} ← {reg. Oper. 1} < {reg. Oper. 2}
  - Ejemplo:

```
SLT R1, R2, R3 ; R1=1 si R2<R3, R1=0 si R2>=R3
```

Instrucciones – Transferencia de Control

- **J** {valor inmediato 26 bits o etiqueta} :
  - Transferir control a una dirección (JMP en MSX88)
  - {IP} ← {valor inmediato o etiqueta}
  - Ejemplo: `J seguir`
- **JAL** {valor inmediato o etiqueta} :
  - Transferir control a una dirección y guardar dirección de retorno (CALL en MSX88)
  - {IP} ← {valor inmediato o etiqueta}
  - {R31} ← {dirección de retorno}
  - Ejemplo: `JAL subrutina`
- **JR** {registro} :
  - Transferir control a una dirección contenida en un registro
  - {IP} ← {registro}

Instrucciones – Transferencia de Control

- **BEQ** {reg. Oper. 1} , {reg. Oper. 2} , {etiqueta 16 bits} :
  - Transferir control a una dirección si los registros son iguales
  - {si reg. Oper 1 = reg. Oper 2} → {IP} ← {IP+desp. etiqueta}
  - Ejemplo: `BEQ R1, R2, son_iguales`
- **BEQ Z** {reg. Oper} , {etiqueta 16 bits} :
  - Transferir control a una dirección si un registro es cero
  - {si reg. Oper = 0} → {IP} ← {IP+ desp. etiqueta}
  - Ejemplo: `BEQZ R1, es_cero`

- **Complementos:**
  - **BNE para BEQ**
  - **BNEZ para BEQZ**

Cauce

- Una instrucción debe pasar por cada bloque o etapa
- En todo momento hay 4 etapas ociosas (inactivas)
- En cada etapa, una instrucción tarda 1 ciclo de reloj
- Los tiempos de ejecución de las instrucciones son:
  - 1 instrucción → 5 ciclos
  - 2 instrucciones → 10 ciclos
  - 3 instrucciones → 15 ciclos
  - …
  - N instrucciones → 5*N ciclos
- Ciclos por instrucción (CPI):
  - total de ciclos/ nº instrucciones: 5 *N/N → 5

Cauce - Mejora

- Si el hardware se modifica para permitir que una instrucción aproveche la etapa que deja otra instrucción, mejoramos el tiempo
- En todo momento hay 5 etapas activas
- Los tiempos de ejecución de las instrucciones son:
  - 1 instrucción → 5 ciclos = 4 + 1
  - 2 instrucciones → 6 ciclos = 4 + 2
  - 3 instrucciones → 7 ciclos = 4 + 3
  - …
  - N instrucciones → 4 + N ciclos
- Ciclos por instrucción (CPI): (4+N) / N

Cauce - Mejora

- Los tiempos de ejecución y ciclos por instrucción para 10, 100 y 1000 instrucciones son:
  - 10 instrucciones → CPI = (4 + 10) / 10 = 1,4
  - 100 instrucciones → CPI = (4 + 100) / 100 = 1,04
  - 1000 instrucciones → CPI = (4 + 1000) / 1000 = 1,004
  - Para un ciclo continuo de ejecución donde N es muy grande → CPI = (4+N) / N = aprox. 1

La mejora nos permite ejecutar 5 veces más rápido, es decir 1 instrucción por cada ciclo

Cauce

- **IF**: Búsqueda de instrucción:
  - Recupera instrucción de memoria de programa.
  - Incrementa PC.
- **ID**: Decodificación de instrucción:
  - Si hay operandos, se accede al banco de registros para recuperarlo.
  - Si es operando inmediato (16 bits), se calcula el valor (extiende a 64 bits).
  - Si es un salto se calcula el destino y si se toma o no.
- **EX**: Ejecución de instrucción:
  - Para instrucción de proceso (aritmetica, lógica, etc.), se ejecuta en la ALU.
  - Para acceso a memoria, se calcula la dirección efectiva.
  - Para saltos, se almacena la dirección en PC.
- **MEM**: Solo para instrucciones que acceden a memoria de datos.
- **WB**: si se produjo algún resultado, se almacena en registro destino.

Cauce – Ejemplo de instrucciones

- LD R2, variable(R1): cargar R2 desde memoria

IF	Recupera instrucción de la memoria de programa
ID	Decodifica instrucción. Accede al registro R1 y lo almacena en un registro temporal
EX	Calcula dirección efectiva [variable+R1]
MEM	Recupera valor de la dirección [variable+R1] de memoria de datos y lo almacena en un registro temporal
WB	Guarda en R2 el valor leído de la memoria (lo transfiere desde el registro temporal)

Cauce – Ejemplo de instrucciones

- SD R2, variable(R1): guardar R2 en memoria

IF	Recupera instrucción de la memoria de programa
ID	Decodifica instrucción. Recupera valor de R2 yR1 y los guarda en registros temporales
EX	Calcula dirección efectiva [variable+R1]
MEM	Guarda valor del registro R2 en la dirección [variable+R1] de la memoria de datos
WB	No se necesita escribir registro. No hace nada

Cauce – Ejemplo de instrucciones

- DADD R3, R2, R1: sumar 2 registros

IF	Recupera instrucción de la memoria de programa
ID	Decodifica instrucción. Recupera valor de R1 y R2 y los guarda en registros temporales
EX	Calcula suma de R1 con R2 y guarda el resultado en un registro temporal
MEM	No se necesita acceder a memoria para guardar ningún registro. No hace nada
WB	Guarda resultado de suma en R3 (lo transfiere desde el registro temporal)

Cauce – Ejemplo de instrucciones

- BEQ R1,R2, Etiqueta

IF	Recupera instrucción de la memoria de programa
ID	Decodifica instrucción. Recupera valor de R2 yR1. Determina si salta o no. Calcula la dirección de salto.
EX	Si determino que debe saltar en la etapa ID, Ejecuta el salto
MEM	No requiere acceso a memoria. No hace nada
WB	No se necesita escribir registro. No hace nada

Cauce - Problemas

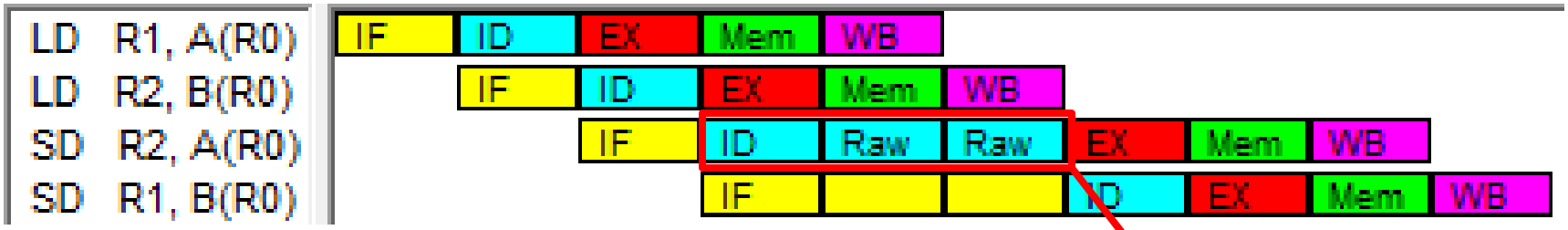
- Si bien en teoría podríamos ejecutar un programa 5 veces mas rápido, en la práctica no es así.
- Puede pasar que una instrucción no pueda avanzar porque su ejecución depende de que termine la ejecución de otra.
- Esta condición donde una instrucción no puede avanzar en el cauce (pipeline) es denominada atasco o “stall”.
- Existe 3 tipos de atascos:
  - **Dependencia de datos:** cuando una instrucción necesita acceder a un registro pendiente de lectura o escritura por otra instrucción .
  - **Dependencia de salto**: cuando una instrucción de salto determina que la siguiente del cauce no es la que se debe ejecutar.
  - **Dependencia estructural**: cuando dos instrucciones intentan acceder a un mismo recurso simultáneamente (próxima práctica)

Dependencia de datos

Los atascos por dependencia de datos son 3:

- RAW (Read After Write): una instrucción necesita leer un registro que otra instrucción aún no escribió.
- WAR (Write After Read): una instrucción necesita escribir un registro que otra instrucción aún no leyó.
- WAW (Write After Write): una instrucción necesita escribir un registro que otra instrucción aún no escribió.

Dependencia de datos - Ejemplo



La instrucción LD escribe el registro en la etapa WB. Cualquier instrucción que intente leer o escribir el mismo registro antes de WB producirá un “atasco”

La instrucción SD lee el registro en la etapa ID. Si hay una instrucción que tiene pendiente una lectura o escritura sobre el mismo registro producirá un “atasco”

La instrucción SD intenta acceder a R2 en ID en el momento que la instrucción LD esta en EX. Se demora la ejecución en esta etapa hasta LD termine la ejecución de la etapa WB.
Se producen 2 “raw stall”

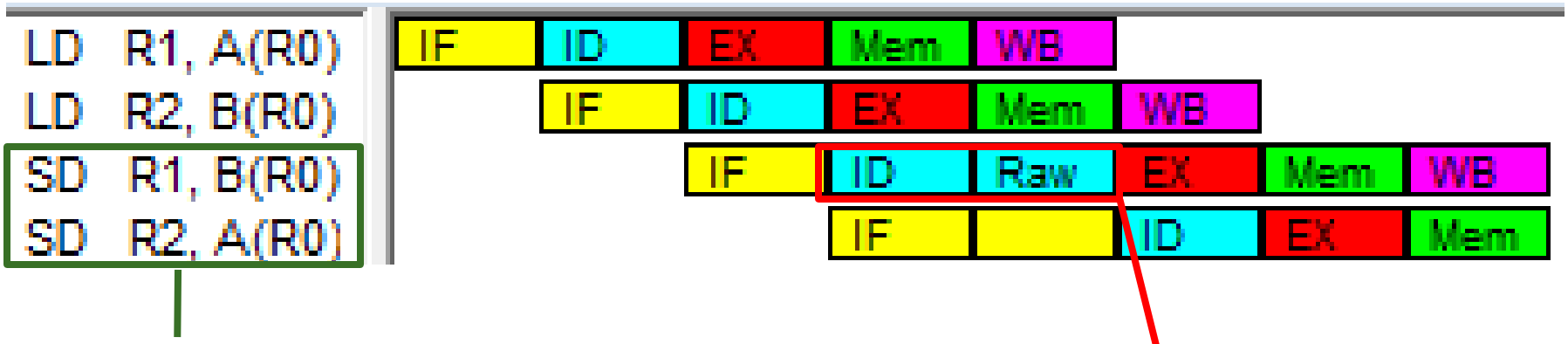
En WB se escribe el registro al principio del ciclo y queda disponible para las demás instrucciones (no se espera a que la etapa concluya)

Dependencia de datos

Estos atascos en general se pueden resolver de 2 maneras:

- Software:
  - El programador o el compilador reordena las instrucciones para minimizar el número de atascos.
- Hardware:
  - Tenemos algún tipo de soporte de hardware que permite resolver algunas situaciones de dependencia de datos.

RAW – Resolución por software



Intercambiamos de posición estas instrucciones para separar los accesos a R2

La instrucción SD intenta acceder a R1 en ID en el momento que la instrucción LD esta en MEM. Se demora la ejecución en esta etapa hasta que LD termine la ejecución de la etapa WB. Se produce 1 "raw stall"

Es importante notar que si bien hay un atasco en la 3º instrucción (SD), las 1º y 2º instrucciones (LD) avanzan 1 etapa mas antes de que se produzca el atasco.

Si bien la relación entre las instrucciones 1º y 3º es igual a la 2º y 4º, una vez que se resuelve el atasco no se vuelve a generar. Esto es porque el retraso que produjo la instrucción 3º "sincronizo" las instrucciones 2º y 4º

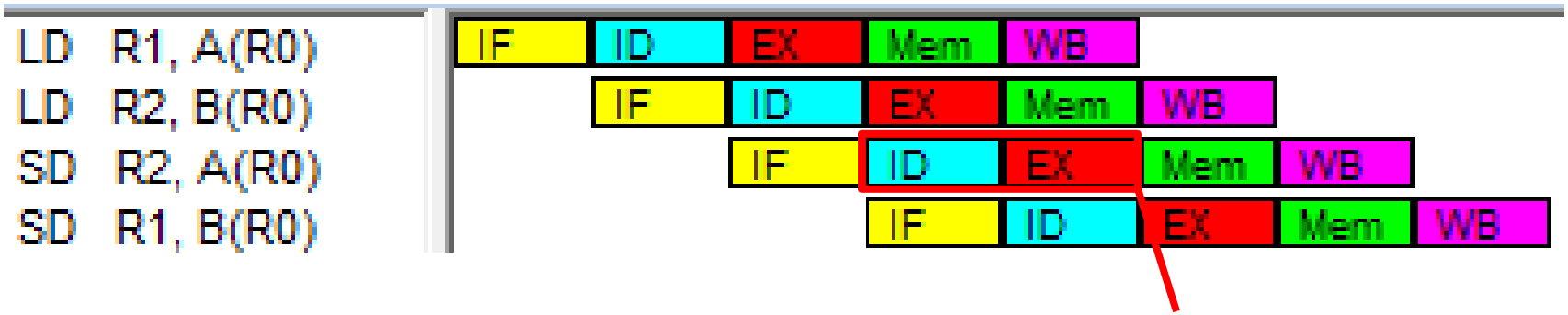
Raw – Resolución por Hardware

- Como muchas veces los valores de los registros de las instrucciones se encuentran en algún registro temporal esperando la etapa WB para escribirlos efectivamente, se podría modificar el hardware para ser utilizados o “adelantados” a las instrucciones que lo necesiten.
- Ejemplos:
  - La instrucción LD deja disponible el valor de un registro al final de la etapa MEM
  - Las instrucciones aritméticas y lógicas dejan el valor disponible a partir de la etapa EX
- El “**Forwarding** “ o adelanto de operandos brinda soporte por **hardware** para evitar en muchos casos los **atascos** por **dependencia de datos**.

Raw – Forwarding

- El forwarding o adelantamiento de registros modifica el cause para que funcione de la siguiente manera:
  - Facilita valores en registros temporales, antes que estén en registros efectivos. Si el valor esta disponible en alguna etapa, no se atasca.
  - “Empuja” a las instrucciones a través del cauce hasta que no sea posible avanzar mas. De esta manera permite que entren otras instrucciones en el cauce. Por ejemplo:
    - las instrucciones aritméticas y lógicas acceden en ID a los operandos pero realmente los necesitan en EX. Puede pasar a EX donde se atascaría si aún no están disponibles.
    - La instrucción SD accede en ID al operando pero realmente lo necesita en la etapa MEM. Puede pasar a EX y luego a MEM donde se atascaría si aún no están disponibles.

RAW – Resolución con Forwarding



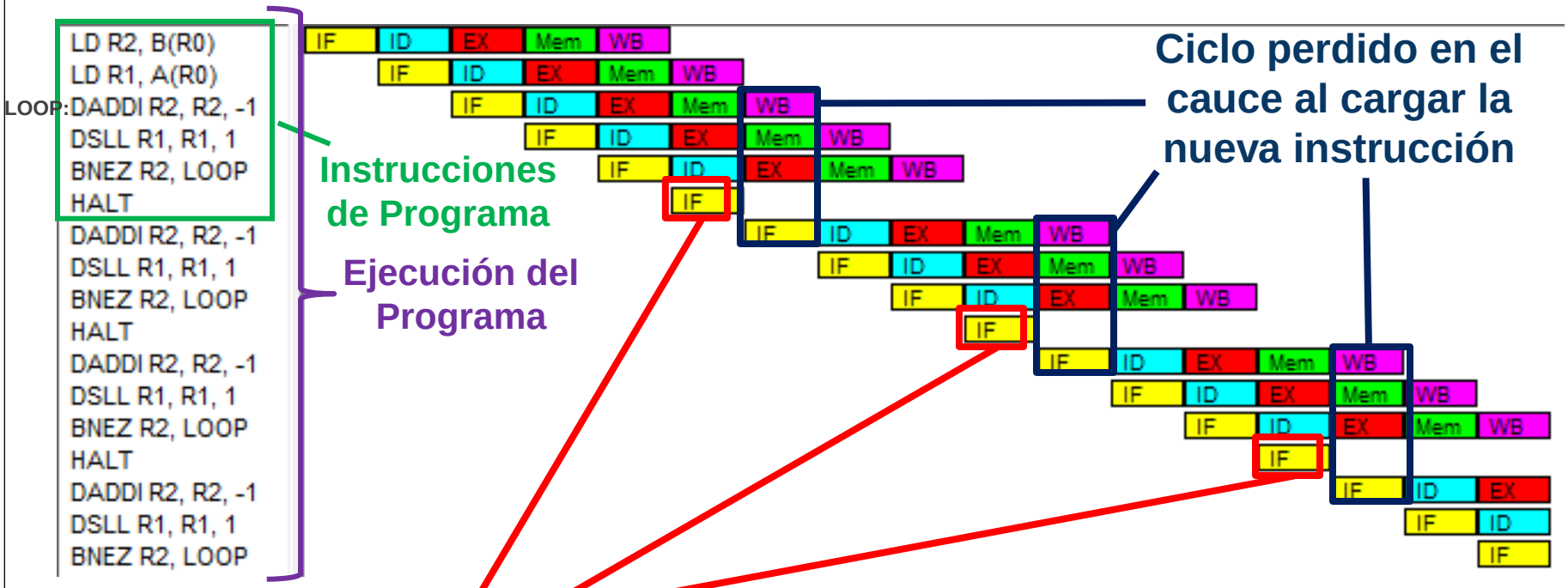
Notar que este programa se ejecuta con la opcion "Enabling forwarding" del simulador

La instrucción SD (R2) requiere el operando en ID pero no esta disponible. Igualmente pasa a la siguiente etapa para resolver dir(A)+R0.
La instrucción LD (R2) deja disponible el valor al final de la etapa MEM.
Cuando SD (R2) entra en la etapa MEM ya tiene disponible el registro, por lo que tampoco se produce un atasco

Atascos por Saltos

- Cuando una instrucción de salto pasa a la etapa ID una nueva instrucción se carga en IF
- En la etapa ID se evalúa si la condición de un salto es verdadera o no
- Si la condición de salto se cumple, hay que descartar la instrucción anterior y cargar la de la instrucción de salto ("Branch taken stall")
- Si tenemos un bucle esto puede suceder siempre, salvo la última vez. Se pierde un ciclo por cada iteración.
- Hay dos mecanismos por hardware para minimizar este problema:
  - Branch Target buffer: utiliza una tabla para "predecir" el salto
  - Delay slot: retarda el salto 1 instrucción , permitiendo la ejecución de la instrucción siguiente al salto.

Atascos por Saltos



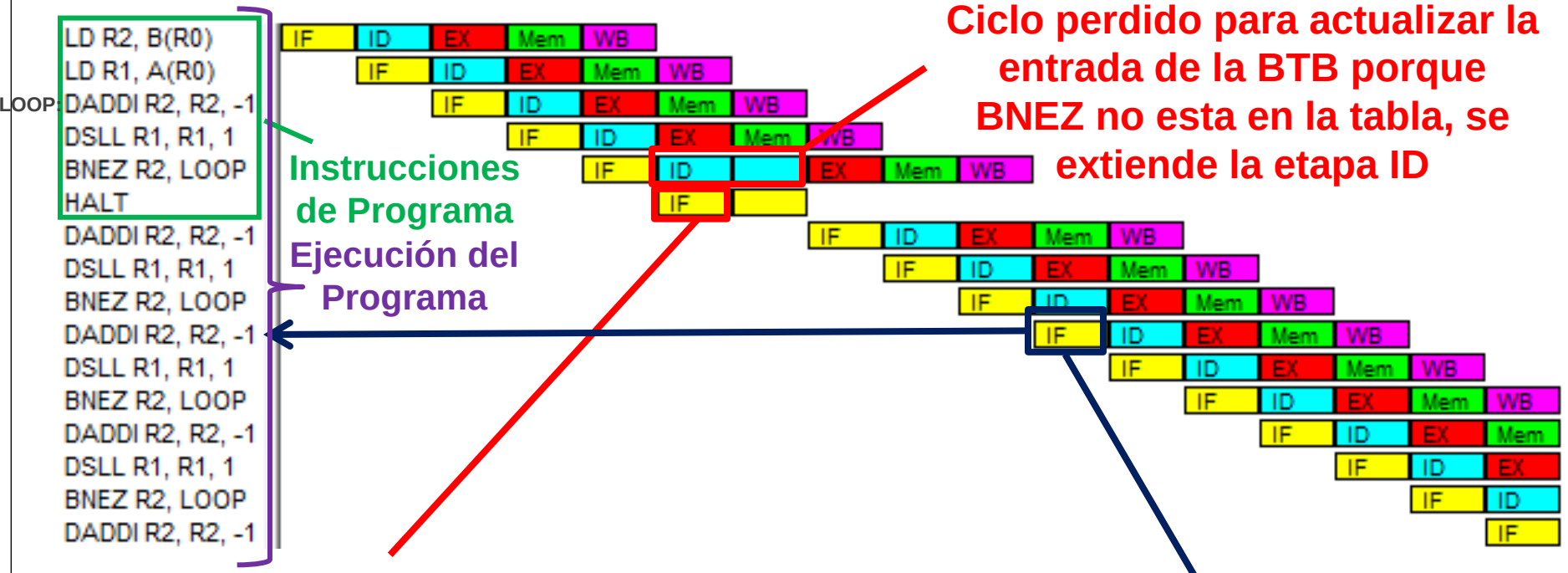
Siempre carga la instrucción HALT . Cuando evalúa BNEZ y falla la condición hay que descartar HALT y cargar DADDI

El simulador tiene la opción “Forwarding” habilitada

Atascos por Salto – **B**ranch **T**arget **B**uffer

- Soporte de hardware que construye una tabla de saltos con:
  - Dirección de la instrucción de salto (para saber si una instrucción esta guardada en la tabla)
  - Dirección de la instrucción a cargar en IF
  - Resultado de la condición del salto del último salto
- Funcionamiento:
  - Cuando una instrucción de salto pasa a la etapa ID se carga en IF la instrucción que figura en la tabla. Si no figura en la tabla se carga en IF la siguiente instrucción al salto
  - Cuando la dirección del salto no está o no coincide la evaluación anterior:
    - Se descarta la instrucción cargada (se pierde un ciclo)
    - Se actualiza la tabla (se pierde otro ciclo)
    - Se carga la instrucción correspondiente
- Mejora: en un loop, solo se pierden 4 ciclos (2 al inicio y 2 al final), luego carga la dirección correcta siempre.

Atascos por Salto – **B**ranch **T**arget **B**uffer



Instrucciones de Programa

Ejecución del Programa

Ciclo perdido para actualizar la entrada de la BTB porque BNEZ no esta en la tabla, se extiende la etapa ID

Se carga HALT en vez de carga la instrucción DADDI. Se perderá este ciclo porque se cargo la instrucción equivocada, pero en este momento no es posible saberlo

Observar que se cargo la instrucción DADDI porque la dirección de BNEZ ya estaba en la BTB y la evaluación del salto coincidió con la última

El simulador tiene la opción "Forwarding" y "Branch Target Buffer" habilitada

Delay Slot o Salto retardado

- Es una **alternativa** a la BTB (Branch Target Buffer) para evitar la pérdida de ciclos en las instrucciones de salto.
- Consiste en retardar las instrucciones de salto 1 ciclo para ejecutar la instrucción siguiente y no descartarla del cauce.
- El ciclo no se pierde se cumpla o no la condición de salto.
- Esta forma de trabajar requiere reestructurar el código para asegurar que el programa se comporte normalmente:
  - Solución simple: agregar NOP luego de cada instrucción de salto. Siempre perdemos un ciclo pero el programa funcionará normalmente.
  - Solución “eficiente”: agregar alguna instrucción anterior a la de salto (en lo posible sin dependencia) para aprovechar el ciclo

Delay Slot o Salto retardado

```
.data
cant: .word 5
datos: .word 1,2,3,4,5
res: .word 0
.code
DADD R1, R0, R0
LD R2, cant(R0)
LOOP: LD R3, datos(R1)
DADDI R2, R2, -1
DSLL R3, R3, 1
SD R3, res(R1)
DADDI R1, R1, 8
BNEZ R2, LOOP
NOP
HALT
; Ejecución en 42 ciclos
```

Observar que para que el programa funcione normalmente es importante NOP. Si no el programa termina!

```
.data
cant: .word 5
datos: .word 1,2,3,4,5
res: .word 0
.code
DADD R1, R0, R0
LD R2, cant(R0)
LOOP: LD R3, datos(R1)
DADDI R2, R2, -1
DSLL R3, R3, 1
SD R3, res(R1)
BNEZ R2, LOOP
DADDI R1, R1, 8
HALT
; Ejecución en 37 ciclos
```

Selección de DADDI R1, R1, 8 para reemplazar NOP, ya que no tiene dependencias en el bucle