Procesadores superescalares y aplicaciones con WinMIPS64

Juan Ignacio Goñi Diego Marqués

24 de octubre de 2008

1. Introducción

En este trabajo se tratan diversas cuestiones referentes a procesadores superescalares con pipelining, en particular se utiliza el simulador WinMIPS64 que corresponde a un procesador RISC de 64 bits.

Un pipeline cuenta con 5 etapas básicas, las cuales a lo largo de este trabajo se abrevian como sigue:

- Búsqueda de instrucción (Instruction Fetch): IF
- Decodificación de instrucción / búsqueda de registro (Instruction Decode / Register Fetch): ID
- Ejecución (Execution): EX
- Acceso a memoria (Memory Access): MEM
- Escritura (Register Write Back): WB

A lo largo del trabajo se presentan diversos programas de testeo, cuyos CPI se presentan en la tabla 1, a la cual es útil regresar cuando se mencionan los efectos de las optimizaciones realizadas sobre los códigos.

En la sección 2 se presentan los riesgos estructurales, en la 3 riesgos de datos, en la 4 riesgos de control, en la 5 características superescalares, en la 6 se comenta la optimización de loop unroll y el cambio de la configuración del procesador para lograr un mejor rendimiento, y finalmente algunas conclusiones.

2. Riesgos estructurales

Cuando dos o más instrucciones en el pipeline requieren algún recurso de hardware al mismo tiempo ocurren los riesgos estructurales. Por ejemplo, cuando dos instrucciones pretenden acceder a memoria a la vez o cuando el microprocesador requiere hacer un fetch de una nueva instrucción desde

Programa	CPI
branchstall.s	1.333
control.s	1.714
controlopt.s	1.571
delayopt.s	1.600
forwardingOK.s	2.000
maxOptDelaySlot.s	1.308
RAW.s	2.000
structural.s	2.000
structuralgood1.s	1.667
structuralgood2.s	1.500
superscalar.s	1.778
superscalaropt.s	1.556

Tabla 1: CPI para cada uno de los programas testeados

memoria y una instrucción en la etapa de ejecución requiere de acceder a un dato allí guardado. También aparecen estos riesgos si el uso de una unidad funcional requiere más de un ciclo de reloj y entonces el pipeline pierde el sincronismo.

En la arquitectura que implementa el simulador `winmips64` ocurren cuando a la salida del pipeline de ejecución, dos instrucciones quieren pasar a la etapa de uso de memoria, pero sólo una de ellas puede hacerlo. El siguiente código de la figura 1 muestra un ejemplo.

```
.text
add.d   f7,f7,f3        ; sumamos números aleatorios
nop                     ; salteamos un lugar (FPadder lugar 2)
nop                     ; salteamos un lugar (FPadder lugar 3)
nop                     ; salteamos un lugar (FPadder lugar 4)
halt                    ; Fin (con 1 structural stall)
```

Figura 1: Código del archivo *structural.s*. El objetivo es generar una parada por riesgo estructural.

Cuando el `add.d` de la línea dos termina de pasar por las cuatro etapas de ejecución que tiene, el `nop` de la línea 4 quiere entrar en la misma etapa, la de acceso a memoria (MEM).

Si bien la etapa de instruction fetch (IF) puede llegar a competir por acceso a memoria con las instrucciones de cargado y guardado de datos en memoria, ya que el uso de caches separados para datos e instrucciones evita que esto ocurra. El código de la figura 2 funciona correctamente en el pipeline.

El uso de sumadores no genera competencias para calcular direcciones

de memoria en las distintas etapas; por ejemplo, en el código de la figura 3, que necesita calcular posiciones de datos relativos, no aparece este tipo de competencia y se ejecuta correctamente en el pipeline.

3. Riesgos por dependencia de datos

Los riesgos de datos ocurren cuando entre dos instrucciones sucesivas, la segunda depende del resultado de la primera. Pueden ser de tres tipos: RAW, WAR o WAW. En la figura 4 se observa un ejemplo de riesgo de datos RAW.

```
        .data
DATA:   .word 0x12345
DATA2:  .word 0x08

        .text
        ld r1,DATA2(r0)         ; accedemos a memoria para instrucción y dato
        ld r2,DATA2(r0)         ; varias veces
        ld r3,DATA2(r0)         ;
        ld r4,DATA2(r0)         ;
        ld r5,DATA2(r0)         ;
        halt                    ; Fin (sin stall)
```

Figura 2: Código del archivo *structuralgood1.s*. El objetivo es generar una parada por riesgo estructural, pero no se genera.

```
        .data
DATA:   .word 0x12345
DATA2:  .word 0x08

        .text
        ld      r1, DATA2(r0)   ; cargamos un offset
        dadd    r2, r2, r2      ; instrucción aritmética
        ld      r3, DATA(r1)    ; instrucción de acceso a
                                        ; memoria relativa (suma el offset)
        ld      r4, DATA(r1)    ; instrucción de acceso a
                                        ; memoria relativa (suma el offset)
        ld      r5, DATA(r1)    ; instrucción de acceso a
                                        ; memoria relativa (suma el offset)

        dadd    r6, r6, r6      ; instrucción aritmética
        dadd    r7, r7, r7      ; instrucción aritmética
        halt                    ; Fin (sin stall)
```

Figura 3: Código del archivo *structuralgood2.s*. El objetivo es generar una parada por riesgo estructural, pero no se genera.

El forwarding permite que una instrucción con una dependencia de datos se empiece a ejecutar aún sin que la otra instrucción haya escrito el resultado en el registro. Así, en vez de perder dos ciclos de reloj, se tienen los resultados disponibles inmediatamente. La idea de su funcionamiento es que el hardware "cablea" el resultado directamente a la ALU desde los LATCHES que hay entre las etapas posteriores a la operación. Así, el hardware requiere de una lógica para que si detecta una dependencia de datos, permita el ingreso de los resultados desde el latch correspondiente a la etapa del pipeline que posee el resultado. De todas formas al agregar el forwarding al microprocesador, hay paradas del pipeline que no se pueden evitar, por ejemplo, cuando se manejan datos provenientes de memoria. Como los datos provenientes de memoria sólo están disponibles después de la ejecución de la etapa MEM de la instrucción `ld` se debe parar el pipeline cuando la instrucción siguiente requiere de los datos antes de que dicha instrucción pase a la etapa EX.

En el código de la figura 4, cuando se lo ejecuta sin forwarding, se producen dos paradas y tiene un CPI de 2.2 y cuando se lo ejecuta con forwarding se produce sólo una parada y entonces tiene un CPI de 2.0. Llamativamente, si bien el simulador hace la parada correspondiente para evitar el riesgo de datos asociado a la secuencia `ld dsub`, la realiza en la etapa de ejecución EX, por lo que correspondería que utilizara los valores antiguos de los registros que intervienen en la resta. De todas maneras, el resultado en cantidad de paradas y de la operación es el correcto.

```
                .data
DATA:    .word 0x12345

                .text
                ld      r1, DATA(r2)    ; cargamos 12345 en r1
; esta instrucción utiliza r1 y le cambia el contenido

                dsub    r4, r1, r5      ; le restamos a 12345 el valor de r5
; riesgo de RAW, porque es posible que no se haya terminado de cargar 12345
; al momento de ejecutar la instrucción (se debe leer después de escribir)

                and     r6, r1, r7      ; operación lógica con 12345
; ya no hay riesgo porque la operación dsub paralizó el pipeline y el
; resultado ya está disponible

                or      r8, r1, r9      ; otra operación lógica con 12345
; sin riesgo

HALT:           halt                    ; fin del programa
```

Figura 4: Código del archivo *RAW.s*. El objetivo es generar un riesgo de datos RAW.

4. Riesgos de control

Los riesgos de control ocurren cuando el programa tiene una instrucción de salto condicional o no. Cuando se debe ejecutar dicho tipo de instrucción, el pipeline corre riesgo de quedar desactualizado ya que el cálculo del destino del salto se lleva a cabo en la etapa EX y recién luego de ella se actualiza el registro PC del microprocesador.

El código de la figura 5 posee un salto condicional para realizar un ciclo.

```
                .data
d1:             .word 2
d2:             .word 3

                .text
                ld      r1, d1(r0)      ; cargamos en memoria el primer factor
                ld      r2, d2(r0)      ; cargamos en memoria el segundo factor
                dadd    r3, r0, r0      ; inicializamos el acumulador
suma:
                dadd    r3, r3, r2      ; sumamos el factor al acumulador
                daddi   r1, r1, -1      ; restamos 1 al contador
                bnez    r1, suma        ; repetimos para obtener el resultado
HALT:   halt                            ; fin del programa
```

Figura 5: Código del archivo *control.s*. El objetivo es mostrar un salto de control.

En dicho código, se multiplican 2 valores guardados en memoria y el resultado de la multiplicación queda en el registro r3. Por más que el procesamiento sea paralelo, el procesador hace lo que le indica el programa, por lo que el resultado queda donde se esperaba. La diferencia radica en la cantidad de paradas del pipeline o stalls. Cuando se intercambian los datos, se ejecuta más veces la resta de la condición de corte, cuyo resultado luego utiliza `benz`. Con este código, es más eficiente sumar la menor cantidad de veces, por lo que es conveniente que el operando `d1` sea 2. Una forma de evitar stalls es cambiando el lugar de la resta de la condición de corte y ponerla antes de la suma, como se observa en el código de la figura 6. Así, se evita el RAW si el forwarding está habilitado, como se explica en la sección 3.

Si se habilita el delay slot, el programa 5 no funciona, esto se debe a que el delay slot permite el fetch de más intrucciones después del `bnez` y su consecuente ejecución. La instrucción que sigue después del condicional es un `halt`, por lo que el programa se detiene sin haber ejecutado toda la multiplicación. Se debe resaltar que el simulador marca la aparición de un *branch stall* aún cuando no lo haya. Esto se puede ver al probar el código de la figura 7 con delay slot y forwarding activados, donde el pipeline no se detiene en absoluto y funciona correctamente.

Se puede optimizar el código para el uso del delay slot pero por la simplici-

```
                .data
d1:             .word 2
d2:             .word 3

                .text
                ld      r1, d1(r0)      ; cargamos el contador
                ld      r2, d2(r0)      ; cargamos el factor
                dadd    r3, r0, r0      ; inicializamos el acumulador
suma:
                daddi   r1, r1, -1      ; restamos 1 al contador
                dadd    r3, r3, r2      ; sumamos el factor al acumulador
                bnez    r1, suma        ; repetimos si no terminamos
HALT:           halt                    ; fin del programa
```

Figura 6: Código del archivo *controlopt.s*. El objetivo es evitar un riesgo de datos RAW.

```
                .data
d1:             .word 2
d2:             .word 3

                .text
                ld      r1, d1(r0)      ; cargamos la cantidad de veces a sumar
                ld      r2, d2(r0)      ; cargamos el número a sumar
                dadd    r3, r0, r0      ; inicializamos el acumulador
suma:
                daddi   r1, r1, -1      ; restamos uno a la condición
                dadd    r3, r3, r2      ; sumamos el acumulador con el número
                bnez    r1, suma        ; repetimos si quedan por sumar
                nop                     ; instrucción para el delay slot
; aquí podríamos hacer el dadd pero entonces habría que poner un nop
; entre el daddi y el bnez porque habría dependencia de datos

HALT:           halt                    ; fin del programa
```

Figura 7: Código del archivo *branchstall.s*. El objetivo es ilustrar un caso de branch stall.

dad del problema al tener tan pocas instrucciones, se cae en una dependencia de datos. Una buena optimización para el delay slot es el código de la figura 8.

```
                .data
d1:             .word 2
d2:             .word 3

                .text
                ld      r1, d1(r0)      ; cargamos el factor en el contador
                ld      r2, d2(r0)      ; cargamos el otro factor
                dadd    r3, r0, r0      ; inicializamos el acumulador
suma:
                daddi   r1, r1, -1      ; restamos uno al contador
                bnez    r1, suma        ; repetimos si no es el último
; aquí se optimiza utilizando el delay slot, ya que se realizará el fetch
; del próximo dadd, y se lo ejecutará completamente, pero cuando termine de
; ejecutarse el program counter estará en suma otra vez. Esta optimización
; presenta un RAW entre las instrucciones daddi y bnez ya que ambas
; requieren utilizar el registro r1
                dadd    r3, r3, r2      ; sumamos en el acumulador
HALT:           halt                    ; fin del programa
```

Figura 8: Código del archivo *delayopt.s*. El objetivo es mostrar una optimización utilizando delay slot.

Otra opción, que no tiene paradas a costa de un uso mayor de la memoria, y de los registros es la que se muestra en la figura 9 pero requiere de una operación simple que puede ser inaceptable que es manualmente restar uno a uno de los datos. En términos de tiempo no se gana absolutamente nada, pero mantiene el pipeline siempre ocupado.

Figura 9: Código del archivo *maxOptDelaySlott.s*. El objetivo es mostrar una optimización para mantener el pipeline siempre ocupado.

5. Características superescalares

Las características superescalares del procesador simulado por el `winmips64` se deben al hecho que paraleliza en unidades funcionales distintas las operaciones de punto flotante. El procesador, entonces, muestra 4 unidades de ejecución: una unidad para enteros, una para sumas y restas de punto flotante, otra para multiplicación de punto flotante y otra para división de números en formato IEEE. Así, es posible que mientras se ejecutan instrucciones de

punto flotante se puedan seguir realizando otros tipos de operaciones que utilicen otra unidad funcional como sea una instrucción de punto flotante distinta o una instrucción de descarga de datos a memoria.

Esta separación se debe a que las operaciones de punto flotante no pueden ser realizadas en un único ciclo de reloj sin perjudicar la performance general del sistema o un alto costo de fabricación y por lo tanto se las paraleliza. Como no pueden ser realizadas en un único ciclo, estas unidades también presentan un pipeline, así, si hay una secuencia de instrucciones de punto flotante que las requieran, no se paraliza todo el pipeline general del procesador. Esta solución plantea una nueva problemática debido a que los resultados de las operaciones de punto flotante necesitan de una cantidad distinta de ciclos de reloj para ser completadas, por lo que si bien las instrucciones entran en el orden correcto, no necesariamente saldrán en el orden en que entraron. Por ejemplo, al ejecutar el código de la figura 10, las instrucciones `mul.d` y `add.d` entran en el microprocesador en ese orden, pero salen luego de la operación `ld` y primero el `add.d` y luego el `mul.d`.

```
                .data
a:              .double 12.34
b:              .double 23.45
c:              .double 34.56
d:              .double 45.67
n:              .word 0x1234567

                .text
                l.d     f1, a(r0)       ; cargamos los operandos
                l.d f2, b(r0)   ;
                l.d f3, c(r0)   ;
                l.d f4, d(r0)   ;

                mul.d f5, f1, f2; lanzamos primero una multiplicación
                add.d f6, f3, f4; luego una suma
                ld      r1, n(r0)       ; y finalmente un load y un and
                and r1, r1, r0  ;
; el orden de finalización es exactamente el inverso al indicado en el
; código fuente
HALT:   halt                    ; fin del programa
```

Figura 10: Código del archivo *superscalar.s*. El objetivo es mostrar el funcionamiento superescalar.

Para que un procesador de características superescalares pueda disparar instrucciones fuera de orden, se requiere que al menos tenga una estructura interna que permita el almacenamiento temporal de instrucciones. Allí deben poder esperar hasta que los recursos que requieren se liberen y puedan ser ejecutadas. El simulador no presenta a simple vista ningún esquema de este

tipo, es más, las instrucciones tenedrían que detenerse antes de entrar en la ALU o alguna de sus unidades funcionales paralelas y deberían dispararse de manera tal que terminen según fueron ingresadas en el programa. Además, con un esquema así, el simulador podría evitar paradas por riesgos que no evita. Por lo tanto no es posible el disparo de instrucciones fuera de orden.

Para aprovechar las características superescalares del microprocesador se debe tener en cuenta que el pipeline de punto flotante puede estar parado esperando un resultado sin que se afecte el resto de los pipelines, así, se pueden buscar secuencias de operaciones que hagan el uso indispensable de la ALU de operaciones enteras para poner en funcionamiento las unidades de punto flotante y luego seguir con las demás operaciones de enteros mientras se utiliza la unidad de punto flotante cuando termina con sus tareas. De todas maneras, al tener un pipeline interno, las secuencias de operaciones de punto flotante que no entren en riesgos de datos también pueden ser una buena optimización. Hay que tener en cuenta a la hora de optimizar con estas unidades que, como se muestra en la sección 2, cuando se paraleliza se puede caer en riesgos estructurales, por lo que la mejor optimización es muy complicada de obtener y debería sincronizarse la salida de la unidad de punto flotante con un RAW de la ALU de enteros por datos en la memoria.

Para mostrar un ejemplo de este último concepto se muestra el código de la figura 11

```
        .data
a:      .double 12.34
b:      .double 23.45
c:      .double 34.56
d:      .double 45.67
n:      .word 0x1234567

        .text
        l.d     f1, a(r0)       ; cargamos valores
        l.d     f2, b(r0)       ;
        nop                     ; nops para no detener el pipeline,
        nop                     ; (podrían ser operaciones útiles con la ALU)
        add.d   f3, f2, f1      ; suma en punto flotante en otra unidad
        nop                     ; nop para no detener el pipeline
; (también podría ser una operación útil con la ALU)
        ld      r2, n(r0)       ; carga del parámetro de la operación xor
        xor     r3, r2, r0      ; operación con riesgo RAW
HALT:   halt                    ; fin del programa
```

Figura 11: Código del archivo *superscalaropt.s*. El objetivo es mostrar una optimización del funcionamiento superescalar.

6. Optimización con loop unroll

Al utilizar loop unroll se baja mucho el tiempo debido a una mayor aprovechamiento del pipeline, ya que al realizar el salto condicional se vacia el mismo. El uso de loop unroll se baja el número de saltos y por ende el pipeline se vacia mucho menos, mejorando considerablemente el rendimiento del programa.

Al cambiar la configuración del procesador no se consiguen mejoras en el rendimiento, solo se consigue mantener la performance o se denigra la misma. Se podría mejorar el rendimiento poniendole delay slot pero para poder realizar eso habría que cambiar el código del programa.

```
        .data
d1:     .word 20
d2:     .word 24
        .text
        LD      R1, d1(R0)
        LD      R2, d2(R0)
        DADD    R3, R0, R0
suma:
        DADD    R3, R3, R2
        DADD    R3, R3, R2
        DADD    R3, R3, R2
        DADD    R3, R3, R2
        DADDI   R1, R1, -4
        BNEZ R1, suma
        HALT
```

Figura 12: Código del archivo *loop.s*. El objetivo es mostrar la mejora de la optimización del loop unroll.

7. Conclusiones

El simulador `winmips64` de arquitectura RISC es muy ilustrativo respecto de cómo funciona un pipeline en la realidad. Pese a las inconsistencias que presenta con las definiciones de Hennessy y Patterson, los valores de stalls y CPI que presenta son una buena aproximación de los valores reales de un microprocesador. Por otro lado, la optimización del código de un programa para que funcione sobre un pipeline es una tarea compleja y meticulosa, ya que si bien una optimización puede parecer buena localmente, puede afectar el funcionamiento global del programa.