Arquitectura y Tecnología de Computadores. Curso 08/09

Arquitecturas Paralelas. **Practica V**

RIESGOS EN PROCESADORES SEGMENTADOS (II)

Ejecución real sobre el MIPS R3000

1. Objetivo:

Evaluar la incidencia de los **riesgos estructurales**, los **riesgos por dependencias de datos** y los **riesgos de control** sobre el rendimiento de un procesador segmentado real mediante la ejecución de programas sobre dicho procesador.

2. Análisis a realizar:

La presente práctica se va a realizar sobre una máquina basada en un procesador **RISC** segmentado y mas concretamente sobre una estación de trabajo que incorpora el procesador **MIPS R3000** de 32 bit. El S.O. de dicha máquina es Ultrix 4.2A (Unix).

Los Ingenieros de la firma MIPS utilizaron un diseño segmentado de cinco etapas para el MIPS R3000. Esta estructura segmentada permite la ejecución concurrente de hasta un máximo de cinco instrucciones, cada un de ellas en una etapa diferente de su cauce, siendo posible obtener un **ritmo ideal de ejecución de una instrucción por ciclo**. Las etapas del cauce de ejecución son: **Búsqueda de Instrucción** (IF), **Decodificación de Instrucción y Lectura de Operandos** (ID), **Ejecución** (EX), **Acceso de Datos en Memoria** (MEM) y **Escritura de Registro** (WB). Otras características importantes del R3000 son la existencia de memorias CACHE de Instrucciones y Datos independientes en el propio chip y la avanzada **optimización de código** que permiten sus compiladores.

Los análisis a realizar con objeto de poner de manifiesto la influencia de los riesgos aludidos sobre el rendimiento del procesador segmentado son los siguientes:

1. Analizar la **influencia de los problemas estructurales** sobre el ritmo de ejecución en el cauce, tanto los debidos a la escasez de recursos (vías de acceso a memoria), como los debidos a la falta de segmentación en alguna de sus unidades funcionales (unidades de punto flotante).

2. Analizar la **influencia de las dependencias de datos** sobre el ritmo de ejecución en el cauce (una de las instrucciones establece un nuevo valor del contenido de un registro y alguna de las posteriores utiliza el mismo como uno de sus operandos).

3. Analizar la **influencia de las instrucciones de control** sobre el ritmo de ejecución en el cauce (recordar que la instrucción de salto genera un hueco de retardo de salto que se tratará de rellenar de acuerdo con una determinada estrategia de planificación).

Los análisis se efectuarán sobre los mismos programas de prueba sintéticos utilizados en la práctica anterior (ligeramente modificados para adaptarlos al lenguaje ensamblador del procesador real).

3. Pasos a seguir en la práctica:

- Todos los programas sintéticos que simulamos en la práctica anterior, los ejecutaremos ahora sobre el procesador MIPS R3000 de la maquina OBOE *(oboe.epsig.uniovi.es).* Preguntar al profesor los datos de la cuenta para acceso a la maquina.

- Todas las conclusiones respecto al comportamiento de los programas y a los posibles ciclos perdidos por problemas que puedan acontecer se deducirán a partir de los tiempos de ejecución medidos para los programas y se validarán teniendo en cuenta las simulaciones realizadas en la practica anterior.

- Como norma general se deberán medir tiempos significativos (del orden de segundos) y habrá que hacerlo varias veces (3 o 5) para quedarse con los valores mas frecuentes. Además, hay que tener en cuenta que variaciones en las medidas de solo 0,1 segundos se pueden achacar a la precisión de la mismas, y que por lo tanto diferencias entre programas de esa magnitud se pueden considerar despreciables.

Análisis de Riesgos Estructurales

- Edita y compara los programas **est1-1.s** y **est1-2.s** (figura 1) correspondientes al análisis de la influencia de los problemas estructurales debidos a escasez de recursos (vías de acceso a memoria). Sustituye la constante NUM_ITER en el código por el valor potencia de 10 que creas necesario para que los programas se ejecuten en un tiempo del orden de pocos segundos.

  ¿En cuál de los programas crees que puede haber problemas estructurales derivados del acceso a memoria?

  Compila cada uno de los programas desactivando las optimizaciones efectuadas por el compilador (-O0) y ejecútalos con el comando **time** para medir su tiempo de ejecución (el tiempo de cpu de usuario).

cc -O0 est1-1.s -o est1-1.out	cc -O0 est1-2.s -o est1-2.out
time est1-1.out	time est1-2.out

  Repite el proceso si no has acertado en la predicción del valor de NUM_ITER.

  ¿Difieren finalmente los tiempos de ejecución? ¿a que crees que se debe? (ver Anexo II).

est1-1.s

```
        .data   0
dato:   .word   1

        .text
        .globl  main
main:
        la   $20  dato
        add  $30, $0,  NUM_ITER

loop:   add  $21, $21, $21
        xor  $2,  $2,  $2
        and  $3,  $4,  $5
        subu $30, $30, 1
        add  $6,  $7,  $8
        add  $9,  $10, $11
        bne  $30, 0,   loop
        j    $31
```

est1-2.s

```
        .data   0
dato:   .word   1

        .text
        .globl  main
main:
        la   $20  dato
        add  $30, $0,  NUM_ITER

loop:   add  $21, $21, $21
        lw   $2,  ($20)
        and  $3,  $4,  $5
        subu $30, $30, 1
        add  $6,  $7,  $8
        add  $9,  $10, $11
        bne  $30, 0,   loop
         j    $31
```

Fig. 1. Influencia de los problemas estructurales (I)

- Edita y compara los programas **est2-1.s** y **est2-2.s** (figura 2) correspondientes al análisis de la influencia de los problemas debidos a la falta de segmentación en alguna de las unidades funcionales (unidades de punto flotante) ¿en cuál de ellos crees que puede haber problemas estructurales?

  Compila cada uno de los programas desactivando las optimizaciones efectuadas por el compilador y ejecútalos con el comando **time** para medir su tiempo de ejecución:

cc -O0 est2-1.s -o est2-1.out	cc -O0 est2-2.s -o est2-2.out
time est2-1.out	time est2-2.out

  ¿Difieren los tiempos de ejecución? ¿cuál es entonces el tiempo de detención? ¿a que crees que se debe?

est2-1.s

```
        .data   0
dato:   .word   1

        .text
        .globl main
main:
        la    $20  dato
        add   $30, $0, NUM_ITER

loop:   xor   $2,  $2,  $2
        and   $3,  $4,  $5
        subu  $30, $30, 1
        add   $6,  $7,  $8
        add.d $f0, $f2, $f4
        sub.d $f6, $f8, $f10
        bne   $30, 0,   loop
         j    $31
```

est2-2.s

```
        .data  0
dato:   .word  1

        .text
        .globl main
main:
        la    $20  dato
        add   $30, $0, NUM_ITER

loop:   xor   $2,  $2,  $2
        and   $3,  $4,  $5
        subu  $30, $30, 1
        add.d $f0, $f2, $f4
        add   $6,  $7,  $8
        sub.d $f6, $f8, $f10
        bne   $30, 0,   loop
        j     $31
```

Fig. 2. Influencia de los problemas estructurales (II)

A la vista de los resultados ¿dirías que la unidad de suma/resta flotante esta segmentada?

Teniendo en cuenta las simulaciones de la práctica anterior ¿cuántos ciclos adicionales por iteración consume el programa **est2-1.s** respecto a **est2-2.s** debido a la detención? Recordando que la simulación de **est2-1.s** variaba dependiendo del simulador utilizado ¿cuál de los dos simuladores utilizados reproduce mas fielmente el comportamiento del procesador en este apartado?

Compila ahora normalmente cada uno de los programas (la optimización de código esta activada por defecto) y ejecútalos con el comando **time** para medir su tiempo de ejecución:

cc est2-1.s -o est2-1op.out	cc est2-2.s -o est2-2op.out
time est2-1op.out	time est2-2op.out

¿Difieren ahora los tiempos de ejecución? ¿a que crees que se debe?

Análisis de Riesgos por Dependencias de Datos

- Edita y compara los programas **dep1-1.s** y **dep1-2.s** (figura 3) correspondientes al análisis de la influencia de las dependencias de datos NO provocadas por instrucciones de carga ¿en cual de ellos crees que se da realmente una dependencia de datos?

  Compila cada uno de los programas desactivando las optimizaciones efectuadas por el compilador y ejecútalos con el comando **time** para medir su tiempo de ejecución ¿difieren los tiempos de ejecución? ¿a que crees que se debe?

  ¿Corresponde el comportamiento observado en el procesador al simulado en la práctica anterior?

dep1-1.s

```
        .data   0
dato:   .word   1

        .text
        .globl main
main:
        la   $20  dato
        add  $30, $0,  NUM_ITER

loop:   add  $21, $21, $21
        xor  $2,  $2,  $2
        and  $3,  $4,  $5
        subu $30, $30, 1
        add  $6,  $7,  $8
        add  $9,  $10, $11
        bne  $30, 0,   loop
        j    $31
```

dep1-2.s

```
        .data   0
dato:   .word   1

        .text
        .globl main
main:
        la   $20  dato
        add  $30, $0,  NUM_ITER

loop:   add  $21, $21, $21
        xor  $2,  $2,  $2
        and  $3,  $2,  $4
        subu $30, $30, 1
        add  $6,  $7,  $8
        add  $9,  $10, $11
        bne  $30, 0,   loop
        j    $31
```

Fig. 3. Influencia de las dependencias de datos (I)

- Edita y compara los programas **dep2-1.s** y **dep2-2.s** (figura 4) correspondientes al análisis de la influencia de las dependencias de datos provocadas por instrucciones de carga ¿en cuál de ellos crees que se da realmente una dependencia de datos?

  Compila cada uno de los programas desactivando las optimizaciones efectuadas por el compilador y ejecútalos con el comando **time** para medir su tiempo de ejecución ¿difieren ahora los tiempos de ejecución? ¿cuál es entonces el tiempo de detención? ¿a que crees que se debe?

  ¿Corresponde el comportamiento observado en el procesador al simulado en la práctica anterior? Teniendo en cuenta las simulaciones ¿cuántos ciclos se pierden debido a la dependencia?, o lo que es lo mismo, ¿qué número de ciclos de procesador supone el *"hueco de retardo de carga"*?

  Edita el programa **dep2-3.s** y compáralo con el **dep2-2.s** ¿cuál es la solución que propone este programa al problema de la dependencia de datos? Compila el nuevo programa desactivando las optimizaciones efectuadas por el compilador y ejecútalo con el comando **time** para medir su tiempo de ejecución ¿difiere su tiempo de ejecución respecto al del programa **dep2-1.s**? ¿a que crees que se debe?

  Compila ahora normalmente cada uno de los dos primeros programas (la optimización de código esta activada por defecto) y ejecútalos con el comando **time** para medir su tiempo de ejecución ¿difieren ahora los tiempos de ejecución? ¿a que crees que se debe?

dep2-1.s

```
        .data   0
dato:   .word   1

        .text
        .globl main
main:
        la  $20  dato
        add $30, $0, NUM_IT

loop:   add  $21, $21, $21
        lw   $2,  ($20)
        and  $3,  $4,  $5
        subu $30, $30, 1
        add  $6,  $7,  $8
        add  $9,  $10, $11
        bne  $30, 0,   loop
        j    $31
```

dep2-2.s

```
        .data   0
dato:   .word   1

        .text
        .globl main
main:
        la  $20  dato
        add $30, $0, NUM_IT

loop:   add  $21, $21, $21
        lw   $2,  ($20)
        and  $3,  $2,  $4
        subu $30, $30, 1
        add  $6,  $7,  $8
        add  $9,  $10, $11
        bne  $30, 0,   loop
        j    $31
```

dep2-3.s

```
        .data   0
dato:   .word   1

        .text
        .globl main
main:
        la  $20  dato
        add $30, $0, NUM_IT

loop:   add  $21, $21, $21
        lw   $2,  ($20)
        subu $30, $30, 1
        and  $3,  $2,  $4
        add  $6,  $7,  $8
        add  $9,  $10, $11
        bne  $30, 0,   loop
        j    $31
```

Fig. 4. Influencia de las dependencias de datos (II)

Análisis de Riesgos de Control

- Edita y compara los programas **con1.s** y **con2.s** (figura 5) correspondientes al análisis de la influencia de los problemas de control. Antes de nada, fíjate que NO se ha necesitado insertar una instrucción **nop** en los programas detrás de la instrucción de salto ¿a qué crees que se debe? ¿cuál de los simuladores utilizados reproduce mejor entonces el comportamiento del procesador en este apartado?

  Compila cada uno de los programas desactivando las optimizaciones efectuadas por el compilador y ejecútalos con el comando time para medir su tiempo de ejecución. Comprobarás que el programa **con2.s** tarda menos en ejecutarse ¿a qué crees que se debe?

con1.s

```
        .data   0
dato:   .word   1

        .text
        .globl main
main:
       la  $20 dato
       add $30, $0, NUM_ITER
       xor $21, $21, $21

loop:  xor  $2,  $2,  $2
       and  $3,  $4,  $5
       subu $30, $30, 1
       add  $6,  $7,  $8
       add  $9,  $10, $11
       addi $21, $21, 1
       bne  $30, 0,   loop
       j    $31
```

con2.s

```
        .data   0
dato:   .word   1

        .text
        .globl main
main:
       la  $20 dato
       add $30, $0, NUM_ITER
       xor $21, $21, $21

loop:  xor  $2,  $2,  $2
       and  $3,  $4,  $5
       subu $30, $30, 1
       add  $6,  $7,  $8
       add  $9,  $10, $11
       bne  $30, 0,   loop
       addi $21, $21, 1
       j    $31
```

Fig. 5. Influencia de los problemas de control

Compila ahora normalmente el programa **con1.s** (la optimización de código esta activada por defecto) y ejecútalo con el comando **time** para medir su tiempo de ejecución. Compara el tiempo medido con el del programa **con2.s** compilado sin optimización ¿difieren? ¿a que crees que se debe?

En función de lo anterior ¿crees que el compilador intenta rellenar el "hueco de retardo de salto" al compilar con optimización? ¿podría ello justificar las diferencias de tiempo observadas durante toda la práctica para las versiones más rápidas de los programas de cada apartado entre las ejecuciones con y sin optimización? ¿cuál es entonces el tiempo de detención o retardo de salto?

Teniendo en cuenta las simulaciones ¿qué número de ciclos de procesador supone el *"hueco de retardo de salto"*? Comprueba que la proporción entre el número de ciclos del *"hueco de retardo de carga"* y el número de ciclos del *"hueco de retardo de salto"* obtenidos en las simulaciones coincide con la proporción entre los tiempos de retardo de carga y de salto medidos en los programas.

Otra forma de comprobar que efectivamente la optimización del compilador activa el salto retardado (lo que implica: 1- reordenación del código y 2- que no se aborta la instrucción posterior al salto) consiste en incluir la directiva "**. set noreorder**" justo después de la directiva ". text" en el primero de los programas. Con esta directiva evitamos que el compilador reordene el código durante la optimización y como consecuencia el propio compilador nos indicará que es necesario incluir una instrucción **nop** justo después del salto (ya que en caso contrario el programa terminaría antes de tiempo al ejecutar la instrucción "j $31" durante el hueco de retardo de salto).

Deducción de la frecuencia de reloj del procesador

Durante el transcurso de la practica se ha planteado varias veces la cuestión de cuantos ciclos suponen cada uno de los tiempos de detención o *huecos de retardo* producidos y para ello se remitía a las simulaciones efectuadas en la practica anterior.

Una forma alternativa de deducir el numero de ciclos es hacerlo directamente a partir de los tiempos de ejecución medidos. Para ello basta recordar que considerando un ritmo de ejecución ideal en el cauce, en cada nuevo ciclo finaliza una nueva instrucción. Así pues, añadir una instrucción mas a un programa supone un ciclo más en su tiempo de ejecución. Conociendo esto, la idea es ir añadiendo un número creciente de instrucciones replicadas en cualquiera de los programas que NO provoquen detención y midiendo sus nuevos tiempos de ejecución (1 ciclo mas por cada réplica). En el momento que el tiempo de ejecución se iguale al del programa que provoca detención, deduciremos que el tiempo de detención equivale a tantos ciclos como réplicas hayamos insertado.

Experimenta lo anterior con los programas **dep2-1.**s y **dep2-2.s** y verifica que el número de ciclos de detención deducido coincide con el obtenido en las simulaciones.

Una vez conocido a qué equivale un determinado tiempo de detención en numero de ciclos, podremos calcular fácilmente el tiempo de ciclo y por tanto la frecuencia de reloj (aproximada) del procesador. Deduce por tanto la frecuencia de reloj y comprueba en el Anexo II que efectivamente esa frecuencia esté en el rango de las frecuencias disponibles para el procesador.

4. Trabajo a entregar:

Todas los resultados y comentarios se resumirán en una nueva hoja de calculo del documento Excel de la asignatura, completando el modelo utilizado en la práctica anterior según se indica en el Anexo I.

Anexo I: *Tabla resumen de resultados (Excel)*

Programa	Tiempos (seg./ciclos)			Comentarios (2 líneas máximo)		
	Oboe	WinDLX	DLXView	**Oboe**	**WinDLX**	**DLXView**
est1-1						
est1-2						
est2-1						
est2-2						
est2-1 (ss)	-	-				
est2-1op		-	-			
est2-2op		-	-			
dep1-1						
dep1-2						
dep1-3	-					
dep1-2 (fw)	-		-			
dep1-3 (fw)	-		-			
dep2-1						
dep2-2						
dep2-3						
dep2-2 (fw)	-		-			
dep2-3 (fw)	-		-			
dep2-1op		-	-			
dep2-2op		-	-			
con1						
con2						
con1op		-	-			
con3	-					

Anexo II: El Procesador MIPS R3000

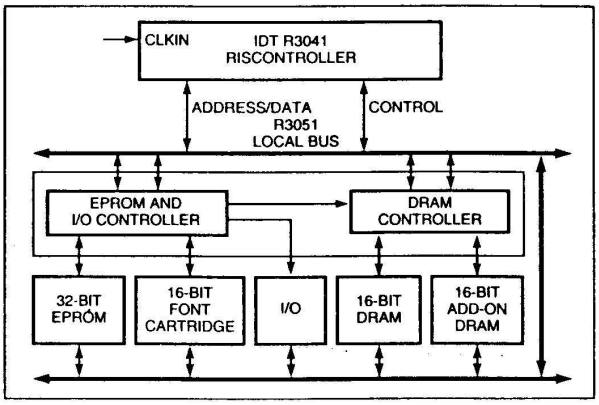
- **Load/store architecture with five-stage pipeline**
- **Compilers achieve high level of code optimization**
- **Compact RISC core for integrated controllers**
- **Dual caches, external (in R3000) or internal for derivatives**
- **Some derivatives contain on-chip FPU**

Developed initially at Stanford University, the 32-bit Mips RISC architecture was one of the first commercial RISC µPs. A classic RISC design, Mips µPs were the speed demons of first-generation RISC processors but burdened system designers with a complex off-chip dual-caching scheme. These µPs are known for their sophisticated design and advanced code optimization that their compilers achieve. Licensed chip vendors have used the Mips core and architecture as a base for embedded processors, many adding on-chip caches and reducing pinouts. R2000/3000 Mips CPUs generally use an off-chip FPU; some R3000 versions have an on-chip FPU.

Mips processors are built around a set of 32-bit, general-purpose registers in a central register file. To minimize control logic, the instruction set is reduced to 73 instructions, addressing options are limited, and the chip has a three-address, load/store architecture. Similarly, instruction sizes are fixed to one 32-bit word to minimize decoding and speed processing.

Like many early RISC µPs, Mips CPUs balanced throughput against complex instructions, including multiply and divide. The R2000 had limited multiply and divide capabilities. Later Mips CPUs added a full multiply but had limited divide capability. The CPUs generated all addresses and handled the memory interface control for up to three additional external coprocessors. Initially, Mips chips used an external FPU but later R3000 versions have brought the FPU on chip.

Mips engineers used a five-stage pipeline for the R3000. The pipeline lets up to five instructions execute concurrently—each at a different stage of its instruction cycle, thus giving the effect of single-cycle execution. The pipeline stages are instruction fetch (IF), read operands and decode instruction (RD), execute (ALU), access data memory (MEM), and write-back results (WB). A branch-delay slot minimizes branch effects. The compiler fills the instruction slot following the branch with an NOP or an instruction from the current thread that can be executed before the branch takes effect.



This example system, typical of a laser printer, demonstrates the flexible bus width where a 32-bit PROM interface mixes with a 16-bit font cartridge and 16- or 32-bit DRAM.

Even though the original Mips R3000 processor lacked an on-chip cache, all of the R3000 derivatives include on-chip instruction and data cache. The on-chip caches help exploit the high-speed pipeline while maintaining a cost-effective DRAM-based memory system. Instruction caches vary from 2 to 16 kbytes. Data caches vary from 512 bytes to 8 kbytes. The new R4400 extends this concept with 16-kbyte instruction and 16-kbyte data caches.

■ VARIATIONS/SPECIAL FEATURES

Mips Technologies Inc, now part of Silicon Graphics, develops Mips chips and licenses them for manufacturing. Licensed vendors include IDT, LSI Logic, NEC, NKK, Siemens, and Toshiba. Some vendors have architectural licenses, allowing them to modify chip designs.

LSI Logic sells **R3000** as an embedded processor or core along with other library cores, targeting low-power or multiple core designs. Library includes basic CPU core and building blocks: configurable-instruction and data cache (directed mapped or two-way set associative); four-deep write buffer, DRAM controller (supports synchronous DRAM and interleaving); timers/counters; and wait-state generator. LSI also introduced a family for the ATM marketplace, called "Atomizer."

LR33300/33310: Embedded R3000; static design; 20 to 50 MHz, 4- to 8-kbyte instruction cache, 2- to 4-kbyte data cache, three counter/timers, DRAM controller (supports synchronous DRAM and interleaving), four-deep write buffer, configurable DRAM or SRAM, lockable instruction cache entries, nonmultiplexed bus. $30 to $75.

LR33020/33120: Embedded R3000 for X-terminals; reimplemented core; static design; 25 to 40 MHz; 4-kbyte instruction cache; 1-kbyte data cache; graphics coprocessor with bitblt processor and DMA channel. $55 to $90.

LR33050: Embedded R3000, on-chip FPU, 25 to 33 MHz; 4-kbyte instruction cache, 1-kbyte data cache, timer/counters, DRAM controller, four-word write buffer, burst mode, nonmultiplexed external bus. $129 to $135.

Integrated Device Technology's R3041: Embedded R3000, 16 to 33 MHz, programmable memory interface and PROM boot options (8, 16, or 32 bit), 2-kbyte instruction cache, 512-byte data cache; multiplexed bus. $12 to $22.

R3051/52: Embedded R3000, 20 to 40 MHz, 4/8-kbyte instruction cache, 2-kbyte data cache, four-deep R/W buffer to memory, multiplexed external bus. $28 to $52.

R3071/81: Embedded R3000, 20 to 50 MHz, configurable cache (16-kbyte instruction/4-kbyte data or 8-kbyte each), four-deep R/W buffer, multiplexed bus, floating point unit (81 only). $46 to $99.

SUPPORT

■ **HARDWARE** The Mips design is a straightforward minimal architecture with bare-bones test features. Hardware tools include ICEs or logic analyzers for most Mips chips. Most chip vendors sell evaluation boards for test and development.

■ **SOFTWARE** Mips provides native- and cross-development tools, including a system simulator, cache-design and optimization tools, and host/target debugger software. Mips compilers effectively optimize code execution on the RISC CPUs. Unix and real-time kernels are available for Mips processors. Application software for embedded designs includes page-description languages, X-servers, and PROM monitors. LSI Logic provides a Coreware program.