ARQUITECTURA DE COMPUTADORES II – Curso 2018

PRÁCTICA 4: Procesadores superescalares y VLIW

Bibliografía de referencia:

[1] Capítulo 2 "*Instruction-Level Parallelism and Its Exploitation*" del libro "*Computer Architecture – A Quantitative Approach*" de J. Hennessy y D. Patterson, quinta edición.

[2] Apéndice H "*Hardware and Software for VLIW and EPIC*" del libro "*Computer Architecture – A Quantitative Approach*" de J. Hennessy y D. Patterson, quinta edición. Descarga: http://booksite.elsevier.com/9780123838728/references.php

[3] Capítulo 3 "*Superscalar Processors*" del libro "*Microprocessor Architecture – From Simple Pipelines to Chip Multiprocessors*" (primera edición), por Jean-Loup Baer.

[4] Libro de apuntes "*MIPS Assembly Language Programming*", por Daniel J. Ellard.

[5] Libro "*ARM System-on-Chip Architecture*" (segunda edición), por Steve Furber.

[6] Paper "*The Microarchitecture of Superscalar Processors*", por J.E. Smith y G.S. Sohi.

[7] Artículo "*ULTRASPARC-III: Expanding the Boundaries of a System on a Chip*", autores varios.

[8] Artículo "*IBM POWER5 Chip: A Dual-core Multithreaded Processor*", autores varios.

[9] Artículo "*Introducing the IA-64 Architecture*", autores varios.

1) Con el fin de optimizar la ejecución de un programa el compilador intentará reordenar las instrucciones del mismo para facilitar la explotación del paralelismo a nivel de instrucciones (ILP) por parte del procesador. Vimos que las dependencias (verdaderas, de salida, o antidependencias) existentes en el programa pueden imponer limitaciones a esta intención, y lo analizamos para fragmentos de código donde estas dependencias se manifestaban en el uso de registros.

Veremos ahora que existen también dependencias a través de memoria que son mucho más difíciles de detectar debido a que las direcciones de memoria en muchos casos son calculadas en tiempo real durante la ejecución del programa, y esto limita severamente la capacidad de reordenamiento del compilador. También veremos que dependiendo del nivel de sofisticación del compilador es posible detectar algunas oportunidades de paralelismo adicionales.

a) Considere un bucle como el siguiente:

```
for (i = 1; i <= 100; i++) {
       A[i+1] = A[i] + C[i];         /* I1 */
       B[i+1] = B[i] + A[i+1];       /* I2 */
}
```

Asumiendo que A, B y C son tres arreglos distintos que no se superponen memoria, analice las dependencias de datos entre las instrucciones I1 e I2 dentro de cada iteración y entre iteraciones consecutivas. ¿Son paralelizables diferentes iteraciones del bucle?

b) Considere ahora el fragmento de código que se muestra a continuación:

```
for (i = 1; i <= 100; i++) {
       A[i] = A[i] + B[i];           /* I1 */
       B[i + 1] = C[i] + D[i];       /* I2 */
}
```

¿Cuáles son ahora las dependencias de datos entre I1 e I2? ¿son paralelizables diferentes iteraciones del bucle? Si la respuesta es negativa muestre cómo puede modificarse para que lo sea.

c) Evalúe el grado de paralelismo de los siguientes tres casos recurrentes:

```
for (i = 2; i <= 100; i++){
       A[i] = A[i - 1] + A[i];
}

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

```
for (i = 1; i <= 100; i++){
       A[2 * i + 3] = A[2 * i] * 5.0;
}
```

2) El siguiente código tiene múltiples tipos de dependencias. Localizarlas, clasificarlas e indicar cuáles pueden ser eliminadas por medio de la técnica de *renombrado de registros*.

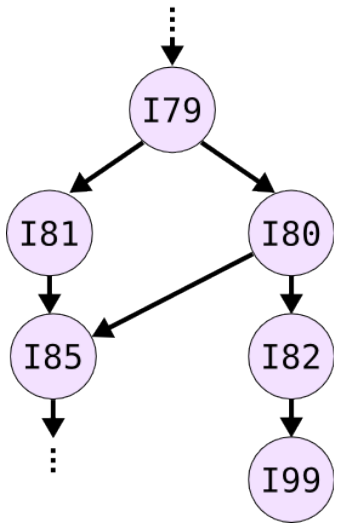
```
Y = X / c;            /* I1 */
X = X + c;            /* I2 */
Z = Y + c;            /* I3 */
Y = c – Y;            /* I4 */
```

3) Considere la siguiente secuencia de instrucciones:

```
; Extraer los primeros tres dígitos decimales del número
; almacenado en NUM
I01     ld  R2,NUM(R8)    ; carga el número en el registro R2
I02     div R3,R2,#10     ; calcula el divisor en R3
I03     mul R4,R3,#10
I04     sub R4,R2,R4      ; calcula el resto y lo deja en R4
I05     st  R4,DIGIT0(R8) ; guarda el primer dígito en memoria
I06     add R2,R3,R0      ; comienza el cálculo del siguiente dígito
I07     div R3,R2,#10     ; calcula el divisor en R3
I08     mul R4,R3,#10     ;
I09     sub R4,R2,R4      ; calcula el resto y lo deja en R4
I10     st  R4,DIGIT1(R8) ; guarda el segundo dígito en memoria
I11     st  R3,DIGIT2(R8) ; guarda el tercer dígito en memoria
```

Suponga que dispone de un procesador superescalar ideal que dispone de una cantidad infinita de unidades funcionales de los siguientes tipos:

- ALUs para sumas, restas, operaciones lógicas, corrimientos de bit, etc. (1 ciclo/instr).
- Unidades de memoria para cargas y almacenamientos en memoria (4 ciclos/instr).
- Unidades de multiplicación/división entera (2 ciclos/instr.).



El procesador también tiene emisión y finalización totalmente desordenada con renombrado de registros sin limitaciones por recursos, y una ventana de entrada de ancho infinito (es decir, puede analizar todas las instrucciones de entrada simultáneamente). En otras palabras, es un procesador superescalar de recursos de todo tipo infinitos. Sin embargo, la superescalaridad no todo lo puede. Responda:

a) ¿Qué tipos de dependencias de datos existen en el fragmento de código? ¿Cuáles son reales y cuáles de nombre? ¿Cuales se resuelven por renombrado de registros?
b) Utilice un grafo orientado como el del ejemplo de la derecha para representar las dependencias reales presentes en el fragmento.
c) Utilizando esta representación, responda: ¿Cuál es la máxima cantidad de instrucciones que se ejecutarán simultáneamente? ¿cuántos ciclos le toma a este procesador superescalar ideal completar la ejecución del fragmento de programa? ¿es posible reducir el tiempo de ejecución por debajo de esta cifra? Justifique.

4) Considere la siguiente secuencia de instrucciones:

```
I1:     ADDF    R12,R13,R14    ; suma en punto flotante R12 = R13 + R14
I2:     ADD     R1,R8,R9       ; suma entera R1 = R8 + R9
I3:     MUL     R4,R2,R3       ; multiplicación entera R4 = R2 * R3
I4:     MUL     R5,R6,R7
I5:     ADD     R10,R5,R7
I6:     ADD     R11,R2,R3
```

Suponga que dispone de un procesador superescalar que tiene las siguientes características:

i) Puede captar, decodificar y emitir dos instrucciones simultáneamente en un ciclo de reloj.
ii) Puede finalizar (*commit*) dos instrucciones simultáneamente.

iii) Dispone de las siguientes unidades funcionales:
1. Un sumador de punto flotante (dos ciclos de reloj).
2. Un sumador entero (un ciclo de reloj).
3. Un multiplicador entero (un ciclo de reloj).

En este contexto, tenga en cuenta que I3 e I4 están en conflicto por la misma unidad funcional, al igual que I2, I5 e I6. Además I5 depende del valor generado por I4 (dependencia de datos verdadera).

a) Calcule la cantidad de ciclos de reloj que insume la ejecución del segmento de código presentado si se emplea la estrategia de emisión y finalización en orden.
b) Muestre cómo se reduce el tiempo de ejecución si las instrucciones se ordenan: I1, I2, I6, I4, I5, I3.
c) Repita el cálculo si se emplea la estrategia de emisión y finalización desordenada con renombrado de registros.

5) Considere un procesador VLIW con las siguientes características:
i) Arquitectura de carga-almacenamiento.
ii) Dispone de 5 unidades funcionales, tal que en cada ciclo de reloj puede emitir simultáneamente:
2 referencias a memoria
2 operaciones de punto flotante
1 operación entera o 1 salto
iii) Las operaciones con enteros se realizan en un ciclo de reloj.
iv) Necesita de un ciclo adicional de reloj para la carga de un *double* y de dos ciclos adicionales para cada operación en punto flotante. Ambas operaciones están segmentadas.

Suponga el siguiente fragmento de código en C:

```
for (i = 479; i >= 0; i--) {
       x[i] = x[i] + s;
}
```

Donde "*x*" es un arreglo y "*s*" una constante, ambos en punto flotante. Este fragmento se compilaría, en un procesador estándar, al siguiente código:

```
       ADD    R1,R0,3832     ; Inicializo el registro índice.
Loop:                        ; Comienzo del bucle
       LDD    RF0,(R1)       ; Cargo el elemento del vector en RF0
       ADDF   RF4,RF0,RF2    ; Hago la suma. RF2 contiene el valor
                              ; de la variable "s"
       STD    (R1),RF4       ; Almaceno el resultado en el vector
       SUB    R1,R1,#8       ; Actualizo el registro índice R1=R1 - 8
       BGEZ   R1,Loop        ; Fin del bucle.
```

Supongo que R1 contiene inicialmente la dirección del primer elemento de *x* y que el registro de punto flotante RF2 contiene el valor de *s*.
a) Muestre una primera distribución de las instrucciones para el procesador descripto. Calcule la cantidad de ciclos que necesita el bucle completo para ejecutarse (no considere la inicialización del bucle).
b) Considere la siguiente versión modificada del código que produce el mismo resultado:

```
for (i = 479; i >= 0; i -= 2) {
       x[i] = x[i] + s;
       x[i + 1] = x[i + 1] + s;
}
```

¿Cómo sería el código resultante para un procesador estándar? ¿Cuál sería la distribución mejorada de instrucciones en el procesador VLIW? Calcule la cantidad de ciclos que requiere este nueva versión del código.
c) ¿Disminuye la cantidad de ciclos si desenrollo una vez más el bucle? ¿Hasta cuántas iteraciones se pueden desenrollar si se dispone de 32 registros?

6) Para la arquitectura del ejercicio anterior, sin tener en cuenta los retardos adicionales (todas las operaciones se realizan en un ciclo de reloj):

a) Escriba un bucle de suma de dos vectores en punto flotante para un procesador estándar.

b) Desenrolle el bucle tantas veces como sea necesario para optimizar la utilización de la arquitectura VLIW.

c) Muestre las instrucciones VLIW resultantes.

7) Considere el siguiente fragmento de código:

```
if (A == 0) { S = T; };
```

a) Si los registros R1, R2 y R3 contienen los valores de A, S y T respectivamente, muestre el código resultante para una arquitectura estándar (MIPS), comparado con el que se obtendría en una arquitectura que incorpore instrucciones condicionales (ARM). Se recomienda tomar como base las referencias [4] y [5]. En la página 65 de [5] puede ver un ejemplo de ejecución condicional, y en la 113 un listado de los códigos de condición disponibles.

b) Repita para el siguiente fragmento de código:

```
if (A > 0)      { A = T; }
else            { A = -T; };
```

c) Responda:

1. ¿Qué ventajas presenta la ejecución condicional para una arquitectura convencional (no superescalar ni VLIW)?
2. ¿Qué beneficios adicionales tiene para una arquitectura capaz de ejecutar múltiples instrucciones simultáneamente (superescalar o VLIW)?
3. ¿Qué diferencias tiene la ejecución predicada de *Itanium* respecto de la ejecución condicional de ARM?

8) Evalúe similitudes y diferencias entre las características superescalares del AMD K5, el Power5, el MIPS R10000 y el UltraSPARC-II. Tome en cuenta los siguientes elementos: cantidad de instrucciones que pueden ejecutar simultáneamente, tipo de emisión (ordenada, desordenada), tipo de finalización (ordenada, desordenada), capacidad de *multithreading*, etc.

9) En pocas palabras (dos o tres renglones debieran ser suficientes) resuma los siguientes conceptos clave y podrá lucirse con ellos en la próxima fiesta a la que asista:

1. ¿Cuál es la diferencia fundamental entre los enfoques superescalar y VLIW/EPIC que explotan el ILP de un programa?
2. ¿Qué dificultades presenta la aproximación VLIW?
3. ¿Qué diferencia la arquitectura EPIC de Itanium de una arquitectura VLIW “pura”?
4. ¿Cómo se compara una arquitectura superescalar respecto de una arquitectura VLIW en términos de compatibilidad binaria?