Arquitectura de Computadoras 2024

Práctico N° 6: Técnicas de mejora de rendimiento

Ejercicio 1: Deep Pipelines

Considere construir un procesador con pipeline dividiendo el procesador de un solo ciclo en N etapas. El procesador de ciclo único tiene un retardo de propagación a través de la lógica combinacional de 740ps. La penalidad por agregar un registro de pipeline es de 90ps. Suponga que el retardo de la lógica combinacional se puede dividir arbitrariamente en cualquier número de etapas y que la lógica de hazard del pipeline no aumenta el retardo.

Asumiendo que un pipeline de cinco etapas tiene un CPI de 1.23 y que cada etapa adicional aumenta el CPI en 0.1 debido a las predicciones de salto erróneas y otros hazard. ¿Cuántas etapas de pipeline deberían usarse para hacer que el procesador ejecute los programas lo más rápido posible?

Ejercicio 2: Predictores de saltos

Asuma un microprocesador con 20 etapas de pipeline, con un fetch que levanta 5 instrucciones por ciclo. Este procesador ejecuta un código donde 1 de cada 5 instrucciones es un salto y esta conformado por bloques de 5 instrucciones donde la última es un salto. La penalidad por una mala predicción es de 20 ciclos. ¿Cuántos ciclos de instrucción toma hacer fetch de todas las instrucciones? Considerando predictores con las siguientes precisiones: 100%, 99%, 90%, 60%.

Ejercicio 3: Predictores de saltos

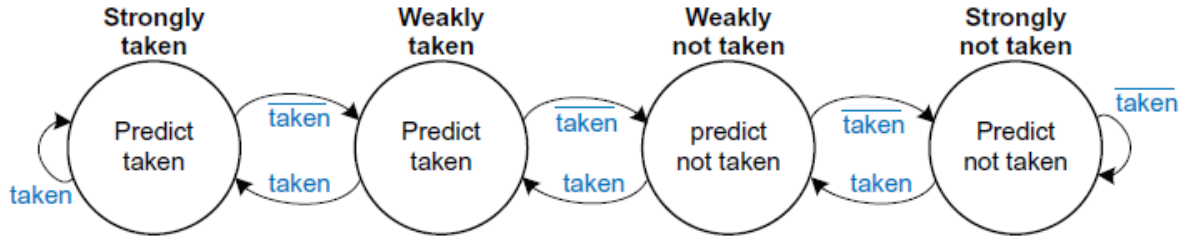

Figure 7.62 Two-bit branch predictor state transition diagram

Este ejercicio analiza la precisión de varios predictores de saltos para el siguiente patrón repetitivo (ej, en un loop) donde los saltos resultaron: **Taken - Not Taken - Taken - Taken - Not Taken.**

a) ¿Cuál es la precisión de los predictores *always-taken* y *always-not taken* para el patrón dado?
b) ¿Cuál es la precisión del predictor de *2-bits* para los primeros 4 saltos de este patrón? Asumir que el predictor arranca en *Strongly not taken*.
c) ¿Cuál es la precisión de este predictor de *2-bits* si el patrón completo se repite infinitamente?

Ejercicio 4: Predictores de saltos

Asumiendo que la distribución de instrucciones dinámicas se divide en las siguientes categorías:

R-Type	CBZ/CBNZ	B	LDUR	STUR
40%	25%	5%	25%	5%

y las siguientes precisiones en los métodos de predicción de salto:

Always-Taken	Always-Not-Taken	2-Bit
45%	55%	85%

Considerando que el resultado y la dirección del salto se determinan en la etapa de decodificación (ID) y se aplican en la etapa de ejecución (EX) y que no hay hazards de datos.

a) ¿Cuántos CPI (Ciclos por instrucción) extras se producen debido a los fallos de predicción del método *Always-Taken*?

b) ¿Cuántos CPI (Ciclos por instrucción) extras se producen debido a los fallos de predicción del método *Always-Not-Taken*?

c) ¿Cuántos CPI (Ciclos por instrucción) extras se producen debido a los fallos de predicción del método *2-Bit*?

Ejercicio 5: Predictores de saltos

El siguiente código en C puede escribirse en ARMv8 de la siguiente forma:

```
for (i = 0; i < 100; i++) {
   for (j = 0; j < 3; j++) {
      …
   }
}
```

```
0x00:        add x0, xzr, xzr
0x04: L2:    add x1, xzr, xzr
0x08: L1:    …
0x0C:        addi x1, x1, 1
0x10:        cmpi x1, 3
0x14:        b.lt L1
0x18:        addi x0, x0, 1
0x1C:        cmpi x0, 99
0x20:        b.lt L2
```

a) Mostrar como queda la tabla de historial de patrones (PHT) considerando que el procesador que ejecuta este código, cuenta con un predictor de saltos local de dos niveles.
b) Comparar la precisión de este predictor con uno de 2-bits (despreciando los primeros ciclos de iniciación).

Ejercicio 6: Predictores de saltos

Asuma que el siguiente código itera en un array largo y lleno de números enteros positivos aleatorios. El código cuenta con 4 saltos, etiquetados B1, B2, B3 y B4. Cuando decimos que un salto es Taken, nos referimos a que el código dentro de las llaves es ejecutado.

```
for (int i=0; i<N; i++) {        /* B1 */
   val = array[i];               /* TAKEN PATH for B1 */
   if (val % 2 == 0) {           /* B2 */
      sum += val;                /* TAKEN PATH for B2 */
   }
   if (val % 3 == 0) {           /* B3 */
      sum += val;                /* TAKEN PATH for B3 */
   }
   if (val % 6 == 0) {           /* B4 */
      sum += val;                 /* TAKEN PATH for B4 */
   }
}
```

a) Determinar cuál de los cuatro saltos muestra una correlación local.
b) ¿Existe correlación global entre algunos de los saltos? Explicar.

Ejercicio 7: Static Multiple Issue Processor

Considere un procesador LEGv8 two-issue, donde en cada "issue packet" una de las instrucciones puede ser una operación de la ALU o un salto y la otra puede ser un *load* o *store*, tal como se muestra en la figura. El compilador asume toda la responsabilidad de eliminar los hazard, organizar el código e insertar operaciones tipo "nop" para que el código se ejecute sin necesidad de detección de hazard o generación de stalls.

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Considere el siguiente bucle:

```
Loop:   LDUR X0, [X20,#0]   // X0=array element
        ADD X0,X0,X21       // add scalar in X21
        STUR X0, [X20,#0]   // store result
        SUBI X20,X20,#8     // decrement pointer
        CMP X20,X22         // compare to loop limit
        B.GT Loop           // branch if X20 > X22
```

a) Analice en el código las dependencias de datos y determine cuales generan data hazards en nuestro procesador one-issue, **sin** forwarding-stall. En cada caso indique: el tipo de hazard, el operando en conflicto y los números de las instrucciones involucradas. Suponga que los saltos están perfectamente predichos, de modo que los *control hazard* son manejados por hardware.

b) ¿Cómo se organizaría el siguiente código en un procesador two-issue con pipeline y forwarding-stall para evitar la mayor cantidad posible de stalls?.

c) Suponga que el compilador es capaz de determinar que la cantidad de iteraciones del bucle se da en múltiplos de 2. Utilice la técnica estática "*loop unrolling*" para re-ordenar la ejecución del código. Determine mejora en el tiempo de ejecución respecto al punto (b). Se calcula como: *Tiempo de ejecución sin mejora/ Tiempo de ejecución con mejora*

Ejercicio 8: Static Multiple Issue Processor

En este ejercicio se compara el rendimiento de los procesadores de 1-issue y 2-issue, teniendo en cuenta las transformaciones que se pueden realizar en un programa para optimizar la ejecución de 2-issue.

Los problemas en este ejercicio se refieren al siguiente bucle (escrito en C):

```
for(i=0;i!=j;i+=2)
    b[i]=a[i]-a[i+1];
```

El código utiliza los siguientes registros:

i	j	a	b	Temporary values
X5	X6	X1	X2	X10–X15

Un compilador con poca o ninguna optimización podría generar el siguiente código de assembler LEGv8:

```
       ADD X5, XZR, XZR
       B ENT
TOP:   LSL X10, X5, #3
       ADD X11, X1, X10
       LDUR X12, [X11, #0]
       LDUR X13, [X11, #8]
       SUB X14, X12, X13
       ADD X15, X2, X10
       STUR X14, [X15, #0]
       ADDI X5, X5, #2
ENT:   CMP X5, X6
       B.NE TOP
```

Asumiendo que el procesador de 2-issue tiene las siguientes propiedades:

1. En cada *issue packet* una instrucción debe ser una operación de memoria y la otra una de tipo aritmética/lógica o un salto.
2. El procesador tiene todos los caminos de forwarding posibles entre las etapas (incluyendo los caminos a la etapa ID para la resolución de saltos).
3. El procesador predice los saltos perfectamente.
4. Dos instrucciones no pueden procesarse juntas en un paquete si una depende de la otra.
5. Si se requiere un stall, ambas instrucciones en el paquete deben volverse stall.

a) Analice en el código las dependencias de datos y determine cuales generan data hazards en nuestro procesador one-issue, **sin** forwarding-stall. En cada caso indique: el tipo de hazard, el operando en conflicto y los números de las instrucciones involucradas. Suponga que los saltos están perfectamente predichos, de modo que los *control hazard* son manejados por hardware.

b) Dibuje un diagrama de pipeline que muestre cómo se ejecuta el código LEGv8 dado anteriormente en el procesador de 2-issue con pipeline y forwarding-stall. (Suponga que sale del bucle después de dos iteraciones.)

c) ¿Cuál es el aumento de velocidad al pasar de un procesador de 1-issue a un procesador de 2-issue, ambos con pipeline y forwarding-stall? (Suponga que el bucle ejecuta miles de iteraciones).

d) Reorganice/reescriba el código LEGv8 dado anteriormente para lograr un mejor rendimiento en el procesador de 2-issue. (No utilice la técnica de *loop unrolling*).

e) Repita el inciso (b), pero usando el código optimizado en el inciso (d).

f) Aplique la técnica de *loop unrolling* al código LEGv8 del inciso (d) para que cada iteración del nuevo bucle se corresponda con dos iteraciones del bucle original. Luego, reorganice/reescriba su nuevo código para lograr un mejor rendimiento en el procesador de 2-issue. Está permitido utilizar otros registros si se considera necesario. Puede asumir que **j** es un múltiplo de 4. (Sugerencia: reorganice el bucle para que algunos cálculos aparezcan fuera del bucle y al final del bucle. Puede suponer que los valores de los registros temporales no son necesarios después del bucle).

g) Repita el inciso (f), pero esta vez suponga que el procesador de 2-issue puede ejecutar dos instrucciones aritméticas/lógicas juntas. (En otras palabras, la primera instrucción en un paquete puede ser cualquier tipo de instrucción, pero la segunda debe ser una instrucción aritmética o lógica. No se pueden programar dos operaciones de memoria al mismo tiempo).

h) Comparar el aumento de velocidad de los incisos (d), (f) y (g) respecto al (b).

Ejercicio 9: Static Multiple Issue Processor

Para los siguientes fragmentos de código LEGv8:

A	B

`LDUR X8, [X0, #40]`
`ADD X9, X1, X2`
`SUB X10, X1, X3`
`AND X11, X3, X4`
`ORR X12, X1, X5`
`STUR X5, [X0, #80]`
	`loop: LDUR X0, [X20,#0]`
	`LDUR X1, [X21,#0]`
	`ADD X0,X0,X1`
	`STUR X0,[X20,#0]`
	`ADDI X20, X20, #8`
	`ADDI X21, X21, #8`
	`SUBI X2, X2, #1`
	`CBZ X2, loop`

a) Analizar las dependencias de datos y determinar cuales generan data hazards en nuestro procesador one-issue, **sin** forwarding-stall. En cada caso indicar: el tipo de hazard, el operando en conflicto y los números de las instrucciones involucradas. (Los saltos están perfectamente predichos, de modo que los *control hazard* son manejados por hardware).

b) Mostrar el orden de ejecución para un microprocesador con pipeline, de una vía, con *forwarding-stall* y que predice los saltos perfectamente.

c) Mostrar el orden de ejecución para un microprocesador multiple-issue de dos vías, con todos los caminos de forwarding posibles entre las etapas y que predice los saltos perfectamente. En cada *issue packet* la primera instrucción puede ser cualquier tipo y la segunda debe ser una instrucción aritmética o lógica.

d) En el fragmento (B): aplicar la técnica de *loop unrolling* para que cada iteración del nuevo bucle se corresponda con dos iteraciones del bucle original (X2 es múltiplo de 2). Luego reorganizar/reescribir el código para lograr un mejor rendimiento en el procesador de 2-issue del inciso (c). Está permitido utilizar otros registros si se considera necesario.

e) Calcular la mejora en eficiencia para cada técnica.

Ejercicio tipo parcial:

Un procesador 2-issue de arquitectura LEGv8 posee las siguientes propiedades:

1. En cada *issue packet* una instrucción debe ser una operación de acceso a memoria y la otra de tipo aritmética/lógica o un salto.
2. El procesador tiene todos los caminos de forwarding posibles entre las etapas (incluyendo caminos a la etapa ID para la resolución de saltos).
3. El procesador predice los saltos perfectamente.
4. Dos instrucciones no pueden procesarse juntas en un paquete si una requiere el resultado de la otra.
5. El compilador asume toda la responsabilidad de eliminar los hazard, organizar el código e insertar instrucciones “nop” para que el código se ejecute sin necesidad de generación de stalls.

Para el siguiente fragmento de código LEGv8 (donde X2 = 0):

```
      1>  ADDI X0, XZR, #0x100
      2>  ADDI X10, XZR, #50
loop: 3>  LDUR X1, [X0,#0]
      4>  ADD X2, X2, X1
      5>  LDUR X1, [X0,#8]
      6>  SUBI X10, X10, #1
      7>  ADD X2, X2, X1
      8>  STUR X2, [X0,#8]
      9>  ADDI X0, X0, #16
      10> CBNZ X10, loop
      ...
```

a. Dibuje un diagrama de pipeline que muestre cómo se ejecuta el código LEGv8 dado en el procesador de 2-issue (sólo hasta completar una iteración del bucle). Sin modificar el orden de ejecución, organice el código para evitar la mayor cantidad posible de *stalls*. Deje indicados los caminos de *forwarding* utilizados. (Completar en la tabla dada al final del ejercicio).

b. Suponiendo que no es económicamente viable integrar los multiplexores de tres entradas que son necesarios para implementar todos los caminos de *forwarding*, analice el código dado y determine si es mejor reenviar solo desde el registro de pipeline EX / MEM (EX → EX) o solo desde el registro MEM / WB (MEM → EX).

c. Indique el aumento de velocidad en la ejecución del código dado al pasar de un procesador de 1-issue a un procesador de 2-issue, ambos con pipeline y forwarding-stall. Considere la totalidad de las iteraciones realizadas por el código.

d. Cuántas instrucciones LDUR se ejecutarán por lazo si se aplica la técnica del loop-unrolling al código dado, con el fin de minimizar la cantidad de iteraciones del lazo? Asuma que X10 se inicializa en la instrucción <2> con un valor múltiplo de 4.