Práctico 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?

→La idea de deep pipelines es reducir el periodo del clock al seguir subdividiendo las etapas. Además, hace más fácil, balancear las etapas. Es decir, la performance más alta, se obtiene cuando tengo las etapas balanceadas, por ende, deep pipeline ayuda a esto.
→CPI = ciclos por instrucción. En este caso tiene un promedio de 1.23
→Cuando existe un hazard o un cómputo erróneo, el CPI aumenta en 0.1

- Latencia de una etapa = Tiempo de ciclo (Tc) = (740/N + 90) ps → Tiempo que tarda en ejecutarse una instrucción en un procesador de un ciclo, dividido “N” etapas. Siempre hay que sumarle 90 ya que es el costo de agregar una etapa de pipeline.
- CPI = 1.23 + 0.1(N - 5) → El 5 se corresponde a las etapas.
- Tiempo por instrucción (Ti) = Tc * CPI

N	Tc [ps]	CPI	Ti [ps]
5	238	1.23	292.74
6	213.33	1.33	283.73
7	195.71	1.43	279.87
8	182.5	1.53	279.23
9	172.22	1.63	280.72
10	164	1.73	283.72

Si observamos la tabla, podemos concluir que:

- A medida que aumentan las etapas, el tiempo de ciclo se reduce.
- A medida que aumentan las etapas, aumenta la CPI.
- A medida que aumentan las etapas, se ve una mejora en el Ti pero luego se estanca, y empieza a empeorar.

Práctico 6 - Técnicas de mejora de rendimiento

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 está 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 en un código de 500 instrucciones? Considerando predictores con las siguientes precisiones: 100%, 99%, 90%, 60%.

Problema a resolver: **identificar cual es la próxima instrucción a ejecutar** , para hacer esto, debemos saber:

- Si la instrucción anterior es un salto.
- En caso de ser un branch condicional, si el salto se toma o no. →Sencillo
- Y en caso de que se tome, cuál es la dirección de salto. →Muy difícil

Datos:

- 20 etapas de pipeline.
- El microprocesador trae 5 instrucciones por cada fetch.
- Tenemos bloques de 5 instrucciones, siendo la última un salto.
- El micro se da cuenta de que el salto tomado fue incorrecto en la última etapa, penalidad de 20.

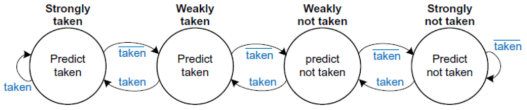
Con precisión de 100%, necesitamos un total de 100 ciclos para hacer fetch de las 500 instrucciones.
Con precisión de 99%, necesitamos un total de 100 ciclos para hacer fetch de las 500 instrucciones, más 20 ciclos por la penalidad por haber errado en una instrucción.
Con precisión de 90%, necesitamos un total de 100 ciclos para hacer fetch de las 500 instrucciones, más 20 * 10 de penalidad por haber errado en 10 instrucciones.
Con precisión de 60%, necesitamos un total de 100 ciclos para hacer fetch de las 500 instrucciones, más 20 * 40 de penalidad por haber errado en 40 instrucciones.

Entonces, en términos de ciclos, tendríamos estos resultados:

- **Con precisión de 100%** = 100 ciclos.
- **Con precisión de 99%** = 100 + 20 * 1 = 120 ciclos.
- **Con precisión de 90%** = 100 + 20 * 10 = 300 ciclos.
- **Con precisión de 60%** = 100 + 20 * 40 = 900 ciclos.

Ejercicio 3: Predictores de saltos

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?

a. **Precision del predictor always-taken** = 3 / 5 = 60%.
Precision del predictor always-not taken = 2 / 5 = 40 %.

b.

Patrón de salto	T	NT	T	T	NT
Predicción	NT	NT	NT	NT	T
F o A	F	A	F	F	F

- Acertó un total de ⅕ = 0.2 → 0.2 * 100 = **20%**.

c.

Patrón de salto	T	NT	T	T	NT
Loop 1	F	A	F	F	F
Loop 2	F	F	F	A	F
Loop 3	A	F	A	A	F
Loop 4	A	F	A	A	F

- Acertó un total de ⅗ = 0.6 → 0.6 * 100 = **60%.**
- **Históricamente,** acertó un total de 8/20 instrucciones → **40%**.

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?

a. Always taken falla un 55% de las veces, y tenemos un total de 30% de frecuencia en las instrucciones de salto (25% CBZ/CBNZ). Luego, habrá un total de (55% * 25%) = 13.75% fallos en el acierto de los saltos.
   → 13.75 * 3 (Clocks penalty) = 41.25% → +**0.4125 CPI.**
b. Always not taken falla un 45% de las veces, y tenemos un total de 30% de frecuencia en las instrucciones de salto (25% CBZ/CBNZ + 5% B). Luego, habrá un total de (45% * 30%) = 11.25% fallos en el acierto de los saltos.
   → 13.5 * 3 (Clocks penalty) = 40.5% → +**0.405 CPI.**
c. 2-Bit falla un 15% de las veces, y tenemos un total de 30% de frecuencia en las instrucciones de salto (25% CBZ/CBNZ + 5% B). Luego, habrá un total de (15% * 30%) = 4.5% fallos en el acierto de los saltos.
   → 4.5 * 3 (Clocks penalty) = 13.5% → +**0.135 CPI.**

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 con n = 4 y m = 4.
b. Comparar la precisión de este predictor con uno de 2-bits (despreciando los primeros ciclos de iniciación).

Tenemos que los bits “n” de GR son 4, y los “m” bits del PC son 4 también. Lo que tiene este predictor, es que para cada salto, tenemos varios patrones posibles. Esta concatenación nos indica a qué memoria debo acceder.

- Por ende, tenemos 2^ (n + m) address para la memoria.

Veamos que pasa para los distintos valores de “j”:

- j = 0 → T
- j = 1 → T
- j = 2 → T

- j = 3 → NT

Veamos que pasa para los valores de "i":
i = 0 →T
i = 1 →T
i = 2 →T
…
i = 99 → T
i = 100 →NT
i → 99 aciertos y 1 fallo
j → 4 aciertos

Evaluación	Valor	GR	Resultado
j < 3	j = 0	1110	Taken
j < 3	j = 1	1101	Taken
j < 3	j = 2	1011	Taken
j < 3	j = 3	0111	Not Taken
i < 100	i = 10	1110	Taken

GHR	PC - B menos sig.	Resultado
1101	0100 (0x14)	11 (T)
1011	0100 (0x14)	11 (T)
0111	0100 (0x14)	00 (NT)
1110	0000 (0x20)	11 (T)

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.

a. Dado que el número obtenido es aleatorio, es imposible predecir localmente los saltos B2, B3 y B4 Sin embargo, el salto B1 si puede ser predecido localmente.
b. B4 está correlacionado con B2 y B3. Si B2 y B3 son taken, B4 también lo va a ser.

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
        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.

Práctico 6 - Técnicas de mejora de rendimiento

a.

Dependencia tipo	Instrucciones	Registros	Hazard
Datos	1 y 2	X0	RAW
Datos	1 y 2	X0	WAW
Datos	2 y 3	X0	RAW
Datos (Cond)	3 y 4	X20	RAW
Datos	4 y 5	X20	RAW
Datos (Cond)	1 y 4	X20	RAW
Datos (2 it.)	4 y 4	X20	RAW

b.

Label	ALU or branch instr.	Data transfer instruction	clk
Loop:	nop	LDUR **X0**, [**X20**,#0]	1
	nop	nop	2
	ADD **X0**,**X0**,X21	nop	3
	SUBI **X20**,**X20**,#8	STUR **X0**, [**X20**,#0]	4
	CMP **X20**,X22	nop	5
	B.GT Loop	nop	6

Modificamos para que sea más eficiente:

Label	ALU or branch instr.	Data transfer instruction	clk
Loop:	SUBI **X20**,**X20**,#8	LDUR **X0**, [**X20**,#0]	1
	CMP **X20**,X22	nop	2
	ADD **X0**,**X0**,X21	nop	3
	B.GT Loop	STUR **X0**, [**X20**,#8]	4