Lösningar till tentamen i kursen EDA330

Datorsystemteknik

28/8 1997

Följande är skisser till lösningar av uppgifterna. Full poäng på en uppgift kräver i de flesta fall en något fylligare motivering. I en del fall är alternativa lösningar möjliga.

1.

a. $T = I * CPI * T_C + T_{I/O}$
$CPI = CPI_0 + P_{mem} * P_{miss} * T_{penalty} = 1,3 + 0,2*0,1*10 = 1,3 + 0,2 = 1,5$
$I = 2*10^9$
$Tc = 1/100\ MHz = 10 * 10^{-9}\ s$
$T_{I/O} = 1000 * 10\ ms = 10\ s$
$T = 2*10^9 * 1,5 * 10 * 10^{-9} + 10 = 30 + 10 =$ **40 s**

b. Halverad miss penalty: CPI = 1,3 + 0,1 = 1,4, T = 28 + 10 = 38 s
Halverad I/O-väntan: T = 30 + 5 = 35 s
Välj halverad väntetid!

2.

a. op = 101011 = 43 = sw
rs = 00010 = 2 = $2
rt = 00000 = 0 = $0
address= 0000000000000000 = 0
Instruktionen är sw $0, 0($2), vilken lagrar värdet noll på den adress som är lagrad i register 2.

b. sign = 1 (negativt)
exponent field = $01011000_2 = 88_{10}$ = exponent +127 => exponent = -39
significand field = $10000000000000000000000_2$ => significand = 1.1_2
Flyttalet är $-1.1_2 * 2^{-39} = -11_2 * 2^{-40} =$ **$-3 * 2^{-40}$**

c. Tvåkomplementsform, och talet är negativt. Ta fram det positiva talet genom att ta tvåkomplementet:
Invertera: 01010011101111111111111111111111
Addera 1: 01010011110000000000000000000000
Detta tal kan skrivas $2^{30} + 2^{28} + 2^{25} + 2^{24} + 2^{23} + 2^{22} =$
$(2^8 + 2^6 + 2^3 + 2^2 + 2^1 + 2^0) * 2^{22} =$
$(256 + 64 + 8 + 4 + 2 + 1) * 2^{22} = 335 * 2^{22}$
Det sökta talet är alltså **$-335 * 2^{22}$**.

3.

a. **Se avsnitt 6.7 i kursboken** och nedan.

b. Observation: snurran löper två varv. 1+ 2*4 + 1 = 10 instruktioner ska exekveras. Inga stalls pga datakonflikter. Om rätt instruktion alltid hämtas efter hoppinstruktionen så fås inte heller några styrkonflikter. Första instruktionen utförs då i WB-steget i cykel 5, och sista instruktionen i cykel 5+9 = 14. Utan styrkonflikter tar programmet alltså 14 cykler, och det återstår bara att för varje metod beräkna hur många extra cykler (t.ex pga stalls) som krävs.

Always stall:
I detta fall görs en stall till hoppinstruktionen utförts i MEM-steget. Det blir alltså ett tillägg av tre cykler för varje gång hoppinstruktionen utförs, dvs 2*3 = 6 extra cykler. **Vid always stall tar programmet 20 cykler**.

Assume not taken:
I detta fall gissar man att hoppet inte ska tas och hämtar därför instruktioner direkt efter hoppinstruktionen. Om det är en felaktig gissning, så upptäcks detta efter tre cykler som alltså blir bortkastade. I detta fall tas hoppet en gång, och inte en gång. Det blir alltså en felaktig gissning som kostar tre extra cykler. **Vid assume not taken tar programmet 17 cykler**.

Assume taken:
Detta är raka motsatsen till föregående fall, och eftersom hoppet utförs en gång och passeras en gång, så blir effekten densamma. Det blir alltså en felaktig gissning som kostar tre extra cykler. **Vid assume taken tar programmet 17 cykler**.

(Kommentar. Vad skulle hända om man använde delayed branch:
I detta fall läggs utgår man från att instruktionerna direkt efter hoppinstruktionen kommer att utföras, och man planerar därför koden efter detta. Utfallet här blir alltså beroende av om man bedömer att instruktionerna direkt efter hoppinstruktionen är “ofarliga” eller ej. För att vara på den säkra sidan kan man alltid lägg in nop-instruktioner, i detta fall tre stycken, direkt efter hoppinstruktionen. Här skulle kostnaden bli densamma som vid always stall. Det är också möjligt att man bedömer att sw-instruktionen och de därpå närmast följande två instruktionerna inte gör någon skada om de utförs även om hoppet tas. I så fall skulle man inte få någon extra kostnad alls. Mellanting mellan dessa båda lösningar är naturligtvis också möjliga, så vid delayed branch tar programmet 14, 16, 18, eller 20 cykler beroende på vad som antas om instruktionerna efter hopp-instruktionen.)

4.

Set/block					0		1		2		3		4		5		6		7	
Byte	Block	Tag	Set	Hit	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
12	3	0	3	n							**0**									
30	7	0	7	n							0								**0**	
13	3	0	3	y							**0**								0	
76	19	2	3	n							0	**2**							0	
5	1	0	1	n			**0**				0	2							0	
14	3	0	3	y			0				**0**	2							0	
44	11	1	3	n			0				0	**1**							0	
116	29	3	5	n			0				0	1			**3**				0	
15	3	0	3	y			0				**0**	1			3				0	
44	11	1	3	y			0				0	**1**			3				0	
101	25	3	1	n			0	**3**			0	1			3				0	
76	19	2	3	n			0	3			**2**	1			3				0	
6	1	0	1	y			**0**	3			2	1			3				0	
40	10	1	2	n			0	3	**1**		2	1			3				0	
12	3	0	3	n			0	3	1		2	**0**			3				0	

Tabellen ovan ger innehållet i varje cacheblock efter varje referens genom att ange tag för de block som ligger i cacheminnet. De fyra första kolumnerna ger byte-adressen, motsvarande block-adress (i detta fall byte-adress/4), motsvarande set-nummer (i detta fall block-adress modulo 8), och tag (i detta fall block-adress/8). Tag för senast refererade cache-block är markerad med fetstil.

Hit rate = 5/15 = 33%

5. **Se kursboken s 469-473**.

a. Ökande blockstorlek utnyttjar rumslokaliteten bättre.

b. Ju större block, desto färre block ryms i cache-minnet, vilket ökar konkurrensen om platserna i cache-minnet. Vid mycket stora block kan denna effekt ta ut effekten av att rumslokaliteten utnyttjas bättre.

c. Kostnaden för missar i form av extra klockcykler är $1,2 * P_{miss} * T_{penalty}$. För respektive blockstorlek blir denna kostnad:
b = 4: 1,2 * 0,38 * (4+4/4) = 2,28
b = 16: 1,2 * 0,22 * (4+16/4) = 2,112
b = 64: 1,2 * 0,19 * (4+64/4) = 3,8
b = 256: 1,2 * 0,27 * (4+256/4) = 22,032
Exekveringstiden bestäms av $I*CPI*T_c$. I detta fall ändras dock endast CPI

genom de extra klockcyklerna. **Blockstorleken 16B bör väljas**.

6. Se kursboken s. 559-562.

7.

Deluppgift	1	X	2
a	1		
b	1		
c		X	
d			2
e			2
f		X	
g		X	
h	1		
i	1		
j	1		
k			2
l			2