Uitwerking oefententamen Computerarchitectuur
December 2016

I.a. De overall CPI kan worden berekend met de formule

$$\text{CPI} = \sum_{i=1}^{n} \frac{IC_i}{\text{Instruction count}} \times \text{CPI}_\text{i}$$

voor de ratio vullen we telkens de geven frequentie in. Dus

$$\text{CPI} = 0.39 \times 1 + 0.13 \times 8 + 0.25 \times 2 + 0.11 \times 2 + 0.12 \times 3 = 2.51$$

b. In dit geval blijft het aantal FP operaties gelijk. De gemiddelde CPI voor de FP operaties zakt naar 4.8. Hiermee berekenen we de nieuwe overall CPI:

$$\text{CPI}_\text{new} = 0.39 \times 1 + 0.13 \times 4.8 + 0.25 \times 2 + 0.11 \times 2 + 0.12 \times 3 = 2.094$$

en de speedup:

$$\text{Speedup} = \frac{\text{CPI}_\text{orig}}{\text{CPU}_\text{new}} = \frac{2.51}{2.094} = 1.19$$

De wellicht tegenvallende speedup kan worden verklaard door het feit dat de FP operaties maar 13% van het totaal uitmaken.

c. Het grootste deel van het energiegebruik komt voort uit dynamic power nodig voor het schakelen van de transistoren. Dynamic power is direct proportioneel aan de klokfrequentie. Het verhogen van de klokfrequentie leidt dus tot een hogere dynamic power. Gezien de kleine oppervlakken van microchips en het feit dat luchtkoeling nog steeds de meest gebruikte vorm van koeling is, is er een limiet aan het vermogen dat chips kunnen gebruiken (power wall). Dit limiet leidt er dus toe dat het verder verhogen van de klokfrequentie boven een bepaalde grens zonder complexe maatregelen voor koeling niet mogelijk is.

II.a. Feitelijk is een instructie in de "execute stage" wanneer deze een reservation station entry toegekend heeft gekregen. We zien in deze architectuur 20 integer entries en 15 FP entries, dus 35 instructies zouden zich volgens deze definitie tegelijkertijd in de "execute stage" kunnen bevinden.
Je zou ook mogen zeggen dat deze architectuur is uitgerust met 6 functional units en dat er derhalve 6 instructies tegelijkertijd "echt" in uitvoering kunnen zijn, aannemende dat deze functional units non-pipelined zijn.
Tussen de decode unit en de instruction queues zien we een bus van 4 instructies, dus per klokperiode kunnen er maximaal 4 instructies worden ge-issued.

b. Om de piekcapaciteit van de processor te kunnen behalen moeten alle functional units elke klokperiode bezet zijn.
Een programma waarmee de piekcapaciteit niet wordt behaald is op verschillende manieren te construeren. Bijvoorbeeld door geen FP instructies uit te voeren, waardoor die functional units niet worden gebruikt. Of nog eenvoudiger: door een programma met dermate veel data hazards te construeren waardoor er zich veel instructies in de integer queue bevinden, maar niet alle integer functional units kunnen worden gebruikt.

c. Dual-ported houdt in dat de cache twee (lees) poorten heeft. We zien in het diagram dat er tussen de register files en de cache twee 64-bit bussen lopen. Er kunnen dus twee waarden tegelijkertijd uit de cache worden gelezen. (Tevens merken we op dat er twee Addr ALUs met de cache zijn verbonden die ieder een adres kunnen doorgeven). Wanneer de cache single-ported was geweest ipv dual-ported, zou er per klokperiode steeds één 64-bit waarde uit de cache kunnen worden gelezen, wat de memory hierarchy performance niet ten goede komt.

III.a. Bij een direct-mapped cache is het zo dat een blok uit het geheugen maar in één bepaalde cache line kan worden geplaatst. Deze cache line wordt vastgelegd aan de hand van het geheugenadres van dat bepaalde blok. Bij een set-associative cache hebben we te maken met een cache die is opgedeeld in sets, welke bestaan uit een bepaald aantal blokken. Een gegeven geheugenadres kan alleen worden geplaatst binnen één bepaalde set, maar dan wel op elk van de blokken binnen deze set.
Om de vergelijking compleet te maken: een direct-mapped cache is in feite een set-associative cache met sets ter grootte van één blok.

b. De cache is two-way set associative, wat betekent dat elke set uit twee blokken bestaat. We kunnen de cache vergelijking invullen:

$$\begin{aligned} 2^{\text{index}} &= \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} \\ &= \frac{64 \times 1024}{64 \times 2} = 512 \\ \text{index} &= 9 \end{aligned}$$

en zien dan dat de cache bestaat uit 512 sets en er 9 bits nodig zijn om de cache te kunnen indexeren.

c. Beide suggesties leiden tot een betere miss rate. Het probleem is echter dat de grootte van een L1 cache wordt gelimiteerd door de kloksnelheid van de processor. Grotere caches hebben een langere "hit time" omdat er meer entries moeten worden vergeleken (meer indexerings-bits). Omdat er is gegeven dat de verbetering niet ten koste mag gaan van de kloksnelheid kiezen we dus voor het verhogen van de associativity.

d. We berekenen voor beide gevallen (miss rate 3.4% en 1.5%) eerst de "average memory access time":

$$\begin{aligned} \text{avg access time}_{3.4\%} &= \text{Hit time} + \text{Miss rate} \times \text{Miss penalty} \\ &= 0.4\text{ns} + (0.034 \times 60\text{ns}) = 2.44\text{ns} \\ \text{avg access time}_{1.5\%} &= 0.4\text{ns} + (0.015 \times 60\text{ns}) = 1.30\text{ns} \end{aligned}$$

Om de speedup in CPI-tijd te berekenen hebben we de gegevens nodig die aanduiden hoeveel geheugenoperaties er in de instructiemix zitten. We gebruiken de processor performance equation:

$$\begin{aligned} \text{CPU time} &= \text{IC} \times \left(\text{CPI} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\right) \times \text{Clock cyle time} \\ &= \text{IC} \times \left((\text{CPI} \times \text{Clock cycle time}) + (\text{Miss Rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \times \text{Clock cyle time})\right) \end{aligned}$$

en vullen deze in voor beide gevallen:

$$\begin{aligned} \text{CPU time}_{3.4\%} &= \text{IC} \times (2 \times 0.4 + (0.034 + 1.6 \times 60)) = \text{IC} \times 4.064 \\ \text{CPU time}_{1.5\%} &= \text{IC} \times (2 \times 0.4 + (0.015 + 1.6 \times 60)) = \text{IC} \times 2.24 \\ \text{Speedup} &= \frac{\text{CPU time } 3.4\%}{\text{CPU time } 1.5\%} = \frac{\text{IC} \times 4.064}{\text{IC} \times 2.24} = 1.81 \end{aligned}$$

We vinden een speedup van een factor 1.81.

e. Een non-blocking cache houdt in dat wanneer de cache bezig is met de afhandeling van een cache miss, de cache niet blokkeert en nog steeds kan worden geraadpleegd. De miss queue van 8 geeft aan dat de cache 8 misses tegelijkertijd kan afhandelen ("hit under multiple miss"). Als alle 8 slots bezet zijn en er nog een miss optreedt, dan zal de cache wel blokkeren tot er een slot vrij komt.
De voordelen van dit mechanisme zijn dat de effectieve cache miss penalty omlaag gaat en de cache bandbreedte wordt vergroot.

IV.a. Het is belangrijk om op te merken dat er is gegeven dat er in de microarchitectuur *geen* forwarding is geïmplementeerd. Dit betekent dat wanneer een volgende instructie een register nodig heeft dat nog moet worden geschreven, deze volgende instructie moet wachten tot de voorgaande instructie de WB-stage heeft bereikt.

	Klokperiode																		
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
`LW R1,0(R2)`	IF	ID	EX	MEM	WB														
`ADDI R1,R1,#1`		IF	ID	Stall(1)	Stall	EX	MEM	WB											
`SW R1,0(R2)`			IF	Stall(2)	Stall	ID	Stall(3)	Stall	EX	MEM	WB								
`LW R3,0(R5)`				Stall(4)	Stall	IF	Stall(5)	Stall	ID	EX	MEM	WB							
`ADDI R2,R2,#4`					Stall(4)	Stall	Stall	Stall	IF	ID	EX	MEM	WB						
`SUB R4,R3,R2`						Stall(4)	Stall	Stall	Stall	IF	ID	Stall(6)	Stall	EX	MEM	WB			
`BNZ R4,Loop`							Stall(4)	Stall	Stall	Stall	IF	Stall(5)	Stall	ID	Stall(7)	Stall	EX	MEM	WB

Redenen voor stalls:

1. We moeten wachten tot `LW` het `R1` register heeft geschreven. We maken gebruik van het feit dat er in de eerste helft van de klokperiode wordt geschreven naar de register file, zodat we de waarden in de tweede helft van cycle 5 kunnen lezen (herhalen ID stage terwijl de stall aanhoudt).
2. De `SW` instructie moet wachten in IF totdat de voorgaande instructie in de stage na ID (EX) is aangekomen.
3. De `SW` instructie moet wachten op het resultaat van `ADDI`. Merk op dat de instructie wordt vastgehouden in de ID stage totdat de hazard is verdwenen, ook al is de waarde `R1` in de EX stage niet nodig, omdat de ID stage de waarde moet inlezen uit de register file.
4. De instructie kan pas de IF stage betreden wanneer deze beschikbaar is (en niet bezet is door een andere instructie).
5. De instructie kan pas de ID stage betreden wanneer deze beschikbaar is.
6. `SUB` moet wachten totdat het resultaat van `addi` in `R2` is geschreven.
7. `BNZ` heeft het resultaat van `SUB` nodig en moet derhalve wachten tot de writeback van `SUB`.

b. In feite zijn alleen de eerste drie instructies de essentiële instructies van de loop. De tweede load en verder dienen alleen maar voor het bijhouden en controleren van de loop bounds. In totaal kost het uitvoeren van de loopiteratie 19 kloktikken. De eerste drie instructies zijn in 11 kloktikken klaar. De 8 kloktikken die overblijven zijn dus in ieder geval overhead.

c. Het is voor deze opgave belangrijk om een kladblaadje te gebruiken waarop je bijhoudt welke reservation station slots en functional units elke cycle in gebruik zijn. Er komt contention op de CDB voor (omdat er per klokperiode maar 1 resultaat op de CDB kan worden gezet) en ook moet er soms worden gewacht op een vrij reservation station slot voor de integer unit (voor de tweede `BNZ` instructie gebeurt dit voor het eerst).

Iteratie	Instructie	IS	EX	WB
1	`LW R1,0(R2)`	3	4	7
1	`ADDI R1,R1,#1`	4	8	9
1	`SW R1,0(R2)`	5	10	13
1	`LW R3,0(R5)`	6	7	10
1	`ADDI R2,R2,#4`	7	9	11
1	`SUB R4,R3,R2`	8	12	14
1	`BNZ R4,Loop`	9	15	16
2	`LW R1,0(R2)`	10	13	17
2	`ADDI R1,R1,#1`	11	18	19
2	`SW R1,0(R2)`	12	22	25
2	`LW R3,0(R5)`	13	16	20
2	`ADDI R2,R2,#4`	14	16	18
2	`SUB R4,R3,R2`	15	21	22
2	`BNZ R4,Loop`	17	23	24
3	`LW R1,0(R2)`	18	19	23
3	`ADDI R1,R1,#1`	19	24	26
3	`SW R1,0(R2)`	20	28	31
3	`LW R3,0(R5)`	21	25	28
3	`ADDI R2,R2,#4`	22	25	27
3	`SUB R4,R3,R2`	23	29	30
3	`BNZ R4,Loop`	25	31	32

d. Het totaal aantal kloktikken voor 3 iteraties is 32. Dus gemiddeld per iteratie is dat $10\frac{2}{3}$. (Een stuk beter dan de single-issue pipeline zonder forwarding).

e. Wanneer er wordt besloten om een branch te nemen terwijl dat eigenlijk nog niet zeker is spreken we van "speculation". Als later blijkt dat de branch niet mocht worden genomen, moeten alle instructies die (speculatief) zijn uitgevoerd na die branch instructie worden afgebroken en eventuele weggeschreven resultaten moeten ongedaan worden gemaakt. Omdat het ongedaan maken van weggeschreven resultaten erg lastig is, wordt er in de praktijk vaak gebruik gemaakt van een reorder buffer (ROB). In de reorder buffer worden resultaten opgeslagen totdat zeker is dat de bijbehorende instructie mocht worden uitgevoerd. Pas dan worden de resultaten echt weggeschreven in register file of memory (instruction commit).

V.a. In het geval van SIMD wordt er gerekend op één core, maar worden er instructies uitgevoerd die telkens op meerdere data elementen tegelijkertijd opereren. Dus één instructie, maar meerdere data elementen: Single Instruction Multiple Data (SIMD).
In het geval van multi-core processing wordt er gerekend op meer dan één core. Op elke core kunnen verschillende instructies worden uitgevoerd die opereren op verschillende data elementen: Multiple Instruction Multiple Data (MIMD).

b. Met de ter beschikking gestelde machine kan een speedup van een factor 40 worden behaald voor 70% van het programma. Om de maximum speedup te berekenen, gebruiken we Amdahl's law:

$$\begin{aligned} \text{Speedup} &= \frac{1}{(1-\text{Frac}_{\text{enh}}) + \frac{\text{Frac}_{\text{enh}}}{\text{Speedup}_{\text{enh}}}} \\ &= \frac{1}{(1-0.7) + \frac{0.7}{40}} \\ &= 3.15 \end{aligned}$$

De theoretisch maximum behaalde speedup is dus een factor 3.15.

c. We bekijken eerst hoeveel operaties er per klokperiode kunnen worden uitgevoerd. Elke SIMD processor kan 32 floating-point operaties per periode uitvoeren. Er zijn 10 van deze processoren, dus dat maakt 320 operaties per periode. In een seconde zitten 1.5×10^9 klokperiodes, dus in een seconde kunnen 480×10^9 operaties worden uitgevoerd, ofwel een piekcapaciteit van 480 GFLOPS per seconde.

d. De latency van load en store operaties wordt op GPUs verborgen middels hardwarematige threading. Zodra een thread een load/store operatie uitvoert, wordt deze stilgezet terwijl de load/store wordt afgehandeld door het geheugen. Terwijl deze thread stilstaat, kan een andere thread door gaan met het uitvoeren van instructies. Nadat verschillende threads instructies hebben uitgevoerd, zal de load/store operatie van de eerste thread compleet zijn, zodat deze thread door kan gaan met het uitvoeren van instructies terwijl de andere threads zijn stilgezet in afwachting van het afronden van load/store operaties.