# Uitwerking tentamen Computerarchitectuur
**18 januari 2017**

---

**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 gegeven frequentie in. Dus

$$\text{CPI} = 0.30 \times 1 + 0.1 \times 4 + 0.20 \times 3 + 0.30 \times 2 + 0.10 \times 2 = 2.1$$

**b.** Met een klokfrequentie van 2.4 GHz vinden er $2.4 \times 10^9$ cycles per seconde plaats. We hebben per instructie gemiddeld 2.1 cycles nodig, ofwel $2.1^{-1}$ cycles per instructie. Er kunnen door deze machine dus gemiddeld $1.143 \times 10^9$ ofwel 1143 miljoen instructies per seconde worden uitgevoerd.

**c.** Berekening nieuwe CPI:

$$\text{CPI} = 0.30 \times 1 + 0.1 \times 2 + 0.20 \times 3 + 0.30 \times 3 + 0.10 \times 3 = 2.3$$

De CPI neemt toe en de performance zal dus afnemen. Dit komt doordat er meer loads/stores plaatsvinden dan FP instructies en load/stores worden duurder gemaakt. Het advies is om in dit geval geen gebruik te maken van pipelining.

**II.a.** Integer Unit 1 en 2, System Register Unit, Load/Store Unit en Floating-Point Unit. In totaal dus 5.
Eventueel mag ook de Branch Processing Unit worden meegeteld.

**b.** Ja, dit is een superscalar architectuur omdat er meerdere functional units aanwezig zijn die tegelijkertijd een instructie kunnen uitvoeren.

**c.** Fetch: 4 instructies per klokperiode, want er is een 128-bit bus (4 instructies) tussen de i-cache en de Instruction Unit.
Issue: 2 instructies per klokperiode, er is een 64-bit bus vanuit de dispatch unit.

**d.** Een precise exception model houdt in dat de pipeline de mogelijkheid heeft om de pipeline precies zo stop te zetten dat de instructies voor de "fault instruction" worden afgerond en alle instructies daarna worden afgebroken. De exception wordt dus op een precieze manier afgehandeld – net als zou gebeuren als het programma in de juiste volgorde zou worden uitgevoerd.
Wanneer er sprake is van een imprecise model, dan kan het bijvoorbeeld voorkomen dat instructies *na* de "fault instruction" al resultaten hebben weggeschreven op het moment dat de exception optreedt. Deze resultaten worden niet automatisch ongedaan gemaakt en het systeem wordt na een exception dus in een niet-gedefinieerde toestand achtergelaten.

**III.a.** Compulsory: cache miss voor eerste geheugenreferentie van een bepaald adres; Capacity: cache miss omdat een blok opnieuw moet worden ingeladen, omdat het vanwege een volle cache uit de cache was gehaald (working set is te groot); Conflict: cache miss omdat een blok opnieuw moet worden ingeladen vanwege een volle set, maar de cache was niet per se vol (te veel blokken beelden af op dezelfde set).

**b.** De cache is 8-way set associative, wat betekent dat elke set uit 8 blokken bestaat. We kunnen de cache vergelijking invullen:

$$\begin{aligned} 2^{\text{index}} &= \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} \\ &= \frac{32 \times 1024}{32 \times 8} = 128 \\ \text{index} &= 7 \end{aligned}$$

en zien dan dat de cache bestaat uit 128 sets en er 7 bits nodig zijn om de cache te kunnen indexeren.

**c.**

| Blokadres | Hit | Set | Opmerking |
|---|---|---|---|
| 0 | N | 0 | |
| 1 | N | 1 | |
| 3 | N | 3 | |
| 128 | N | 0 | |
| 256 | N | 0 | |
| 5 | N | 5 | |
| 512 | N | 0 | |
| 1024 | N | 0 | |
| 2048 | N | 0 | |
| 516 | N | 4 | |
| 4096 | N | 0 | |
| 8192 | N | 0 | |
| 16384 | N | 0 | Set 0 is vol, dus entry ”0”wordt gewist. |
| 3 | Y | 3 | |
| 1 | N | 0 | Opnieuw inladen, was de vorige keer gewist. Entry ”128”wordt gewist. |
| 5 | Y | 5 | |

**d.** We berekenen voor alle gevallen: eerst de “average memory access time”:

$$\begin{aligned}
\text{avg access time}_{16} &= \text{Hit time} + \text{Miss rate} \times \text{Miss penalty} \\
&= 2 + (0.0402 \times 103) = 6.1406 \text{ cycles} \\
\text{avg access time}_{32} &= 2 + (0.0256 \times 106) = 4.7136 \text{ cycles} \\
\text{avg access time}_{64} &= 2 + (0.0249 \times 112) = 4.7888 \text{ cycles}
\end{aligned}$$

De cache line size van 32 bytes lijkt dus toch het meest geschikt.

**e.** De omschreven optimalisatie is “critical word first”. In geval van een cache miss moet de gehele cache line uit het geheugen worden geladen. Echter heeft de processor maar 1 word uit deze cache line direct nodig. De hardware heeft in dit geval de mogelijkheid om eerst het kritieke word te laten en deze direct door te sturen naar de processor zodat deze verder kan. In de achtergrond wordt dan de rest van de cache line in de cache geladen.

**IV.a.** Binnen de pipeline is forwarding geïmplementeerd, dus in veel gevallen is het mogelijk om stalls te vermijden. Belangrijk om op te merken is dat de branch instructie alleen maar stages IF en ID uitvoert. Voor de bepaling of een branch wel of niet moet worden genomen, moet de waarde van het te controleren register dus uiterlijk in de eerste helft van de ID stage van de branch instructie moeten zijn weggeschreven.

| | **Klokperiode** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Instruction** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| `ADD R4,R2,R0` | IF | ID | EX | MEM | WB | | | | | |
| `LW R2,0(R1)` | | IF | ID | EX | MEM | WB | | | | |
| `ADD R2,R2,#10` | | | IF | ID | Stall(1) | EX | M | WB | | |
| `SW R4,0(R1)` | | | | IF | Stall(2) | ID | EX | M | WB | |
| `SUB R1,R1,#4` | | | | | Stall(3) | IF | ID | EX | M | WB |
| `BNEZ R1,Loop` | | | | | | Stall(3) | IF | ID | Stall(4) | |

Redenen voor stalls:

1. `ADD` heeft het resultaat van de load-instructie (`R2`) nodig en moet daarom wachten totdat de load-instructie de MEM stage heeft uitgevoerd. Tussen de MEM en EX stages vindt een forward plaats.
2. Omdat de voorgaande instructie in stall zit, kan de instructie niet verder naar de `ID` stage.
3. Omdat de voorgaande instructie in stall in IF zit, kan de instructie niet verder naar de `IF` stage.
4. De branch instructie heeft het resultaat van de `SUB` instructie nodig en moet daarom wachten totdat de `SUB` instructie de EX stage heeft uitgevoerd. Middels de stall wordt de ID stage van de branch instructie herhaald.

**b.**

- `ADD R4,R2,R0` en `LW R2,0(R1)`: WAR op R2.
- `LW R2,0(R1)` en `ADD R2,R2,#10`: WAW op R2.
- `LW R2,0(R1)` en `SUB R1,R1,#4`: WAR op R1.
- `SW R4,0(R1)` en `SUB R1,R1,#4`: WAR op R1.

**c.**

```
ADD   T0,R2,R0
LW    T1,0(R1)
ADD   T2,T1,#10
SW    T0,0(R1)
SUB   T3,R1,#4
BNEZ  T3,Loop

ADD   T4,T2,R0
LW    T5,0(T3)
ADD   T6,T5,#10
SW    T4,0(T3)
SUB   T7,T3,#4
BNEZ  T7,Loop

ADD   T8,T6,R0
LW    T9,0(T7)
ADD   T10,T9,#10
SW    T8,0(T7)
SUB   T11,T7,#4
BNEZ  T11,Loop

ADD   T12,T10,R0
LW    T13,0(T11)
ADD   T14,T13,#10
SW    T12,0(T11)
SUB   T15,T11,#4
BNEZ  T15,Loop
```

**d.** Neem bijvoorbeeld de eerste hazard tussen `ADD` en `LW`. Het probleem is hier dat `LW` mogelijk R2 eerder overschrijft dan `ADD` de oude waarde kan lezen. Register renaming lost dit op door `LW` te laten schrijven naar een ander register `T1`, waardoor de waarde van R2 die `ADD` mogelijkerwijs nog later zou moeten lezen onaangetast blijft.

**e.** In geval van loop unroll door de compiler, zal de compiler expliciet de code van het programma aanpassen door de loop body meerdere keren achter elkaar te plaatsen. Bij deze kopien zal de compiler uiteraard gebruikmaken van andere registers. Hier vindt dus ook register renaming plaats, maar niet naar schaduwregisters of reservation stations, maar naar geldige registers uit de ISA.
Nadeel loop unroll: de code van het programma neemt toe (dit heeft o.a. implicaties voor de instructiecache), daarnaast is de compiler qua register aantal begrensd tot het aantal registers beschikbaar in de ISA (vaak 32). De processor kan intern veel meer schaduwregisters ter beschikking hebben.
Nadeel dynamic scheduling: maakt de hardware vele malen complexer, power requirement neemt toe. Daarom in embedded processoren vaak niet geimplementeerd.

**V.a.** De functional units worden tussen de hardware-matige threads gedeeld en deze hoeven dus niet te worden gedupliceerd. Binnen de processor is al scheduling en register renaming geïmplementeerd. Het is simpelweg mogelijk om op instructies uit meerdere instructiestromen register renaming toe te passen en deze dan op de gedeelde set function units uit te voeren. Er is dus alleen maar een extra instruction fetch unit nodig en een bijbehorende register file.

**b.** Met de ter beschikking gestelde machine kan een speedup van een factor $4.6/1.5 = 3.067$ worden behaald voor 55% van het programma. Om de maximum speedup te berekenen, gebruiken we Amdahl's law:

$$\begin{aligned} \text{Speedup} &= \frac{1}{(1 - \text{Frac}_{\text{enh}}) + \frac{\text{Frac}_{\text{enh}}}{\text{Speedup}_{\text{enh}}}} \\ &= \frac{1}{(1 - 0.55) + \frac{0.55}{3.067}} \\ &= 1.59 \end{aligned}$$

De theoretisch maximum behaalde speedup is dus een factor 1.59

**c.** We bekijken eerst hoeveel operaties er per klokperiode kunnen worden uitgevoerd. Elke SIMD instructie kan 8 floating-point operaties uitvoeren in 3 klokperioden, dus $2\frac{2}{3}$ per klokperiode. Er zijn 8 cores, dus $21\frac{1}{3}$ FLOP per klokperiode. In een seconde zitten $2.8 \times 10^9$ klokperiodes, dus in een seconde kunnen $59.73 \times 10^9$ operaties worden uitgevoerd, ofwel een piekcapaciteit van 59.73 GFLOPS per seconde.

**d.** Het probleem van cache coherency is dat hetzelfde geheugenadres zich in meerdere caches kan bevinden. Het probleem treedt op wanneer de waarde in één van de caches wordt veranderd. De andere caches en ook het geheugen zijn hiervan dan niet op de hoogte.
Voorbeeld:

- Processor 1 leest waarde X uit het geheugen.
- Processor 1 schrijft naar waarde X. Alleen de cache in processor 1 wordt aangepast.
- Processor 2 leest waarde X uit het geheugen. Processor 2 leest de oude waarde en niet de nieuwe!

Een mogelijke oplossing is snooping. Hierbij zijn alle processoren aangesloten op een shared bus. Elke processor houdt al het verkeer op de bus in de gaten. Voordat een processor zal schrijven naar een geheugenadres, zal het een invalidate op de bus zetten, zodat de waarde uit alle andere caches wordt verwijderd. Wanneer de processor op de bus ziet dat er wordt gelezen vanuit een geheugenlocatie die lokaal is gecached (en dirty is!), dan zal de processor deze meest recente waarde op de bus zetten zodat niet de oude waarde uit het geheugen wordt genomen.