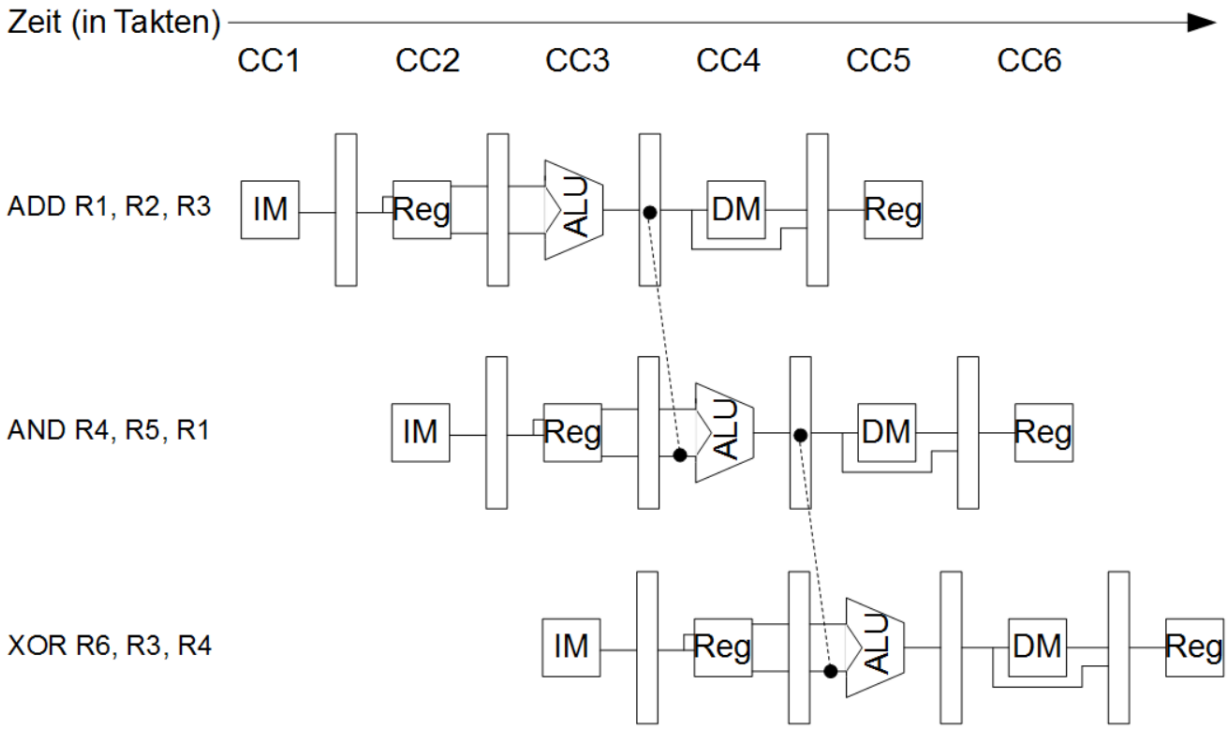
Aufgabe 1

- Das Problem bei dem Programmsegment besteht darin, dass der Befehl AND das Ergebnis von ADD benötigt. Der Befehl ADD schreibt das Ergebnis allerdings erst in der WB-Phase (5. Takt) nach R1, während AND den Wert in R1 bereits in der ID-Phase benötigt. Wenn man berücksichtigt, dass der Befehl AND einen Takt später startet, findet dessen ID-Phase im 3. Takt statt, was bedeutet, dass das Ergebnis noch nicht in R1 steht und AND somit mit dem alten Wert rechnet. Das gleiche Problem ergibt sich bei dem Befehl XOR, der das Ergebnis von AND benötigt, bevor es in R4 steht.
  Anstatt nun Staus in Kauf zu nehmen, ist es möglich, das Problem mit Hilfe von forwarding zu lösen. Dabei wird genutzt, dass das Ergebnis der ALU auch an den ALU-Eingang zurückgeführt wird. Eine spezielle Forwarding Logik, die solche Situationen erkennt, leitet dieses Ergebnis nun als Operanden für den nächsten Befehl an die ALU, anstatt des Wertes aus dem Register.
  In diesem Beispiel erzeugt die ALU das Ergebnis des ADD Befehls in Takt 3, für den AND Befehl wird dieses Ergebnis erst in Takt 4 benötigt, sodass dieser fehlerfrei ausgeführt werden kann. Analog funktioniert es bei dem XOR Befehl.
  In der folgenden Grafik wird dargestellt, wie die Werte zwischen den Befehlen übergeben werden:



- Wird nun der Befehl ADD R1,R2,R3 durch LW R1,1000(R2) ersetzt, lässt sich ein Stau nicht mehr vermeiden. Der Befehl LW hat den Wert für R1 nämlich erst nach der MEM-Phase, also nach dem 4. Takt. Der Befehl AND braucht diesen Wert allerdings schon am Anfang des 4. Taktes, um mit diesem rechnen zu können. Aus diesem Grund müssen die Befehle AND und XOR während des 4. Taktes gestaut werden, damit keine Fehler entstehen.

Aufgabe 2

- Bei Anwendung von Predict-taken wird jeden Branch-Befehl behandelt, als würde er ausgeführt. Nachdem das Sprungziel berechnet wurde, fährt das Programm fort, die Befehle ab dem Sprungziel zu holen und zu bearbeiten, bis das Ergebnis des Branch-Befehls vorliegt. Wenn ein Sprung ausgeführt werden sollte, fährt das Programm fort, ansonsten werden die fälschlich durchgeführten Befehle verworfen und das Programm fährt mit den Befehlen nach dem Branch-Befehl fort. Bei den Branches, bei denen gesprungen wird, reduziert sich der Stau so auf einen Takt. Es ergibt sich für die CPI:

$$CPI = 80\% \cdot 1{,}3 + 20\% \cdot (70\% \cdot (1+3) + 30\% \cdot (1+1)) = 1{,}72$$

- Bei Anwendung von Predict-not-taken wird jeder Branch-Befehl zunächst so behandelt, als würde er nicht ausgeführt. Das Programm fährt fort, die neuen Befehle zu holen und auszuführen, bis das Ergebnis des Branch-Befehls vorliegt. Wenn kein Sprung ausgeführt werden sollte, arbeitet das Programm normal weiter, ansonsten werden die fälschlich durchgeführten Befehle verworfen und der Sprung wird ausgeführt. Da bei 30% der Branches gesprungen wird, führen 30% der Branch-Befehle zu einem Stau. Es ergibt sich folglich:

$$CPI = 80\% \cdot 1{,}3 + 20\% \cdot (70\% \cdot 1 + 30\% \cdot (1+3)) = 1{,}42$$

- Bei Anwendung der freeze-Strategie wird das Programm gestaut, bis das Ergebnis des Branch-Befehls bekannt ist. Jeder Branch-Befehl führt also dazu, dass das Programm um drei Takte gestaut wird. Mit dem Anteil von 20% Branches und einer CPI der restlichen Befehle von 1,3, ergibt sich für die gesamt-CPI:

$$CPI = 80\% \cdot 1{,}3 + 20\% \cdot (1+3) = 1{,}84$$